FORM

v5.0.1-33-gdf7fc94

Generated by Doxygen 1.9.8

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

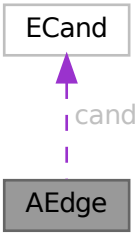
Here is a list of all documented files with brief descriptions:

Chapter 3

Data Structure Documentation

3.1 AEdge Class Reference

Collaboration diagram for AEdge:



Public Member Functions

- AEdge (int n0, int l0, int n1, int l1)

Data Fields

- ECand * cand
- int nodes [2]
- int nlegs [2]
- int ptcl
- int DUMMYPADDING

3.1.1 Detailed Description

Definition at line 1164 of file grcc.h.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 AEdge()

```
AEdge::AEdge (
            int n0,
            int l0,
            int n1,
            int l1 )
```

Definition at line 8275 of file grcc.cc.

3.1.2.2 ~AEdge()

```
AEdge::~AEdge (
            void  )
```

Definition at line 8286 of file grcc.cc.

3.1.3 Field Documentation

3.1.3.1 cand

```
ECand* AEdge::cand
```

Definition at line 1171 of file grcc.h.

3.1.3.2 nodes

```
int AEdge::nodes[2]
```

Definition at line 1172 of file grcc.h.

3.1.3.3 nlegs

```
int AEdge::nlegs[2]
```

Definition at line 1173 of file grcc.h.

3.1.3.4 ptcl

```
int AEdge::ptcl
```

Definition at line 1174 of file grcc.h.

3.1.3.5 DUMMYPADDING

```
int AEdge::DUMMYPADDING
```

Definition at line 1176 of file grcc.h.

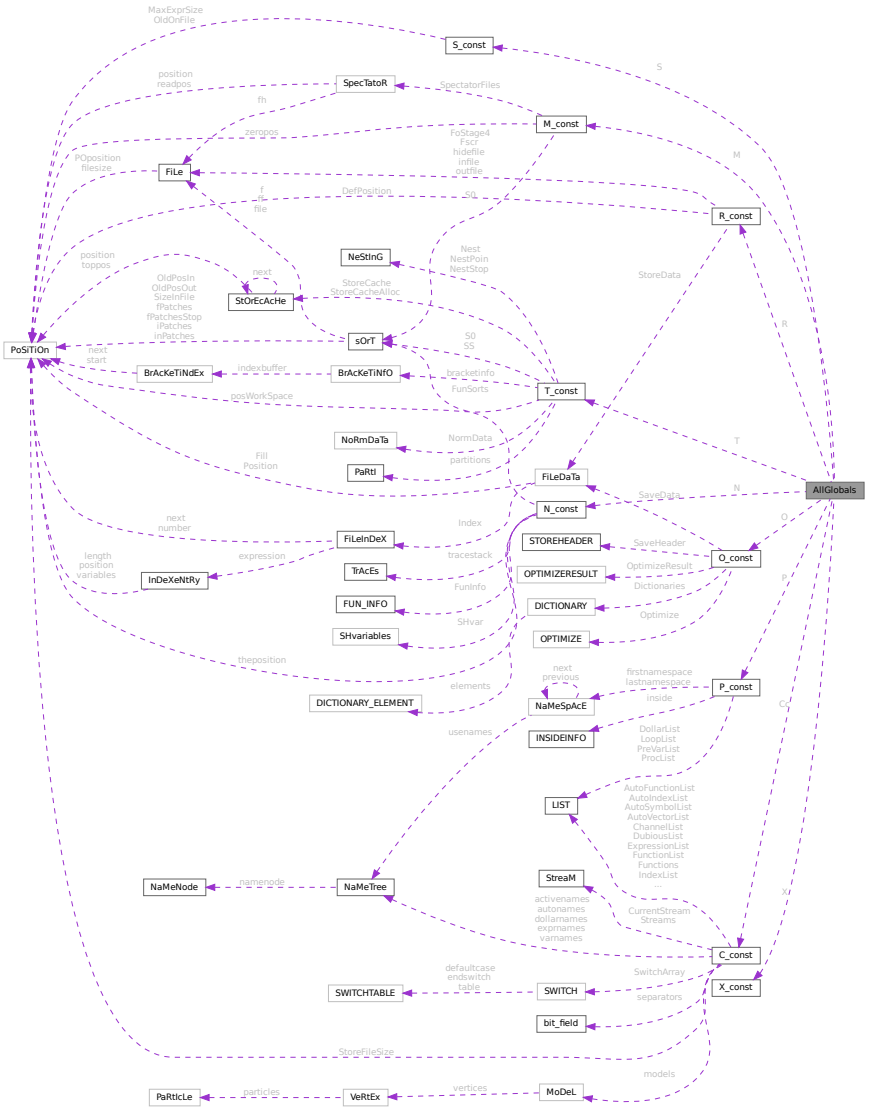
The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.2 AllGlobals Struct Reference

```
#include <structs.h>
```

Collaboration diagram for AllGlobals:



Data Fields

- struct M_const M
- struct C_const Cc
- struct S_const S
- struct R_const R
- struct N_const N
- struct O_const O
- struct P_const P
- struct T_const T
- struct X_const X

3.2.1 Detailed Description

Without pthreads (FORM) the ALLGLOBALS struct has all the global variables

Definition at line 2503 of file structs.h.

3.2.2 Field Documentation

3.2.2.1 M

```
struct M_const AllGlobals::M
```

Definition at line 2504 of file structs.h.

3.2.2.2 Cc

```
struct C_const AllGlobals::Cc
```

Definition at line 2505 of file structs.h.

3.2.2.3 S

```
struct S_const AllGlobals::S
```

Definition at line 2506 of file structs.h.

3.2.2.4 R

```
struct R_const AllGlobals::R
```

Definition at line 2507 of file structs.h.

3.2.2.5 N

```
struct N_const AllGlobals::N
```

Definition at line 2508 of file structs.h.

3.2.2.6 O

```
struct O_const AllGlobals::O
```

Definition at line 2509 of file structs.h.

3.2.2.7 P

```
struct P_const AllGlobals::P
```

Definition at line 2510 of file structs.h.

3.2.2.8 T

```
struct T_const AllGlobals::T
```

Definition at line 2511 of file structs.h.

3.2.2.9 X

```
struct X_const AllGlobals::X
```

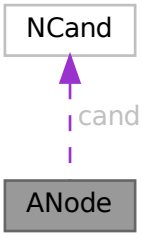
Definition at line 2512 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.3 ANode Class Reference

Collaboration diagram for ANode:



Public Member Functions

- ANode (int dg)
- int newleg (void)

Data Fields

- int deg
- int nlegs
- int * anodes
- int * aedges
- int * aelegs
- NCand * cand

3.3.1 Detailed Description

Definition at line 1140 of file grcc.h.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 ANode()

```
ANode::ANode (
            int dg )
```

Definition at line 8220 of file grcc.cc.

3.3.2.2 ~ANode()

```
ANode::~ANode (
            void  )
```

Definition at line 8238 of file grcc.cc.

3.3.3 Member Function Documentation

3.3.3.1 newleg()

```
int ANode::newleg (
            void  )
```

Definition at line 8255 of file grcc.cc.

3.3.4 Field Documentation

3.3.4.1 deg

```
int ANode::deg
```

Definition at line 1150 of file grcc.h.

3.3.4.2 nlegs

```
int ANode::nlegs
```

Definition at line 1151 of file grcc.h.

3.3.4.3 anodes

```
int* ANode::anodes
```

Definition at line 1152 of file grcc.h.

3.3.4.4 aedges

```
int* ANode::aedges
```

Definition at line 1153 of file grcc.h.

3.3.4.5 aelegs

```
int* ANode::aelegs
```

Definition at line 1154 of file grcc.h.

3.3.4.6 cand

```
NCand* ANode::cand
```

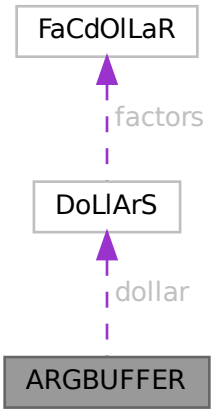
Definition at line 1155 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.4 ARGBUFFER Struct Reference

Collaboration diagram for ARGBUFFER:



Data Fields

- WORD * buffer
- DOLLARS dollar
- LONG size
- int type
- int dummy

3.4.1 Detailed Description

Definition at line 601 of file ratio.c.

3.4.2 Field Documentation

3.4.2.1 buffer

```
WORD* ARGBUFFER::buffer
```

Definition at line 602 of file ratio.c.

3.4.2.2 dollar

```
DOLLARS ARGBUFFER::dollar
```

Definition at line 603 of file ratio.c.

3.4.2.3 size

```
LONG ARGBUFFER::size
```

Definition at line 604 of file ratio.c.

3.4.2.4 type

```
int ARGBUFFER::type
```

Definition at line 605 of file ratio.c.

3.4.2.5 dummy

```
int ARGBUFFER::dummy
```

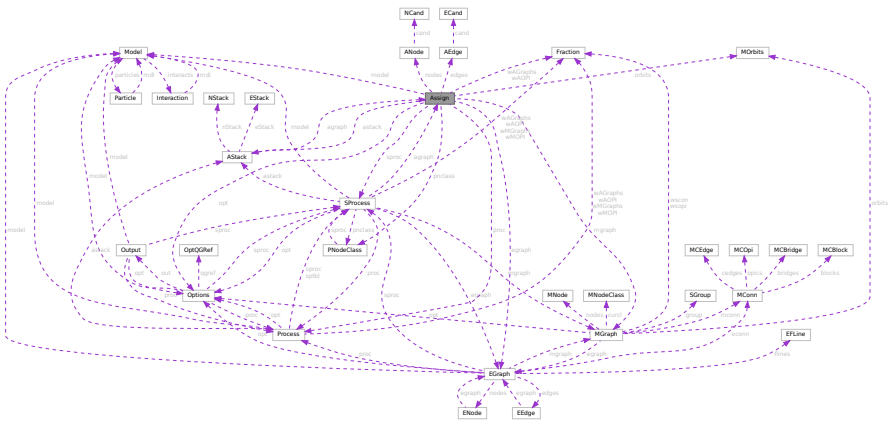
Definition at line 606 of file ratio.c.

The documentation for this struct was generated from the following file:

- ratio.c

3.5 Assign Class Reference

Collaboration diagram for Assign:



Public Member Functions

- Assign (SProcess ∗sprc, MGraph ∗mgr, PNodeClass ∗pnc)
- void prCand (const char ∗msg)
- void checkAG (const char ∗msg)
- Bool assignAllVertices (void)
- Bool selectVertex (void)
- Bool selectVertexSimp (int lastv)
- Bool selectLeg (int v, int lastlg)
- Bool assignVertex (int v)
- Bool allAssigned (void)
- Bool fromMGraph (void)
- void addEdge (int n0, int n1, int nplist, int ∗plist)
- void connect (int n0, int l0, int eg, int el, int n1, int l1)
- Bool fillEGraph (int aid, BigInt nsym, BigInt esym, BigInt nsym1)
- int ∗ reordLeg (int n, int ∗reord, int ∗plist, int ∗used)
- int getLegParticle (int n, int ln)
- int legEdgeParticle (int n, int ln, int pt)
- int legPart (int v, int lg, int nplst, int ∗plst, int ∗rlist, const int size)
- int candPart (int v, int ln, int ∗plist, const int size)
- int selUnAssVertex (void)
- int selUnAssVertexSimp (int lastv)
- int selUnAssLeg (int v, int lastlg)
- NCandSt assignIVertex (int v, int ia)
- Bool assignPLeg (int n, int ln, int pt)
- Bool isOrdPLeg (int n, int ln, int pt)
- Bool detEdge (int e)
- Bool candPartClassify (int v, int ∗npdass, int ∗pdass, int ∗npuass, int ∗puass, const int size)
- Bool updateCandNode (int v)
- Bool checkOrderCpl (void)
- Bool isOrdLegs (void)
- Bool isIsomorphic (MNodeClass ∗cl, BigInt ∗nsym, BigInt ∗esym, BigInt ∗nsym1)
- int cmpPermGraph (int ∗p, MNodeClass ∗cl)
- int cmpNodes (int nd0, int nd1, MNodeClass ∗cn)
- BigInt edgeSym (void)
- void saveCouple (int ∗sav)
- void restoreCouple (int ∗sav)
- Bool subCouple (int ∗cpl)
- Bool checkCand (const char ∗msg)
- void checkNode (int n, const char ∗msg)

Data Fields

- EGraph ∗ egraph
- MGraph ∗ mgraph
- SProcess ∗ sproc
- Process ∗ proc
- Model ∗ model
- Options ∗ opt
- AStack ∗ astack
- PNodeClass ∗ pnclass
- MOrbits ∗ orbits
- int nNodes
- int nEdges

- int nExtern
- int nETotal
- BigInt nAGraphs
- Fraction wAGraphs
- BigInt nAOPI
- Fraction wAOPI
- ANode ** nodes
- AEdge ** edges
- CheckPt checkpoint0
- int cplleft [GRCC_MAXNCPLG]

3.5.1 Detailed Description

Definition at line 1183 of file grcc.h.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 Assign()

```
Assign::Assign (
            SProcess * sprc,
            MGraph * mgr,
            PNodeClass * pnc )
```

Definition at line 8295 of file grcc.cc.

3.5.2.2 ~Assign()

```
Assign::~Assign (
            void  )
```

Definition at line 8371 of file grcc.cc.

3.5.3 Member Function Documentation

3.5.3.1 prCand()

```
void Assign::prCand (
            const char * msg )
```

Definition at line 8391 of file grcc.cc.

3.5.3.2 checkAG()

```
void Assign::checkAG (
            const char * msg )
```

Definition at line 8427 of file grcc.cc.

3.5.3.3 assignAllVertices()

```
Bool Assign::assignAllVertices (
            void  )
```

Definition at line 8451 of file grcc.cc.

3.5.3.4 selectVertex()

```
Bool Assign::selectVertex (
            void  )
```

Definition at line 8514 of file grcc.cc.

3.5.3.5 selectVertexSimp()

```
Bool Assign::selectVertexSimp (
            int lastv )
```

Definition at line 8475 of file grcc.cc.

3.5.3.6 selectLeg()

```
Bool Assign::selectLeg (
            int v,
            int lastlg )
```

Definition at line 8553 of file grcc.cc.

3.5.3.7 assignVertex()

```
Bool Assign::assignVertex (
            int v )
```

Definition at line 8655 of file grcc.cc.

3.5.3.8 allAssigned()

```
Bool Assign::allAssigned (
            void  )
```

Definition at line 8778 of file grcc.cc.

3.5.3.9 fromMGraph()

```
Bool Assign::fromMGraph (
            void  )
```

Definition at line 8858 of file grcc.cc.

3.5.3.10 addEdge()

```
void Assign::addEdge (
            int n0,
            int n1,
            int nplist,
            int * plist )
```

Definition at line 9004 of file grcc.cc.

3.5.3.11 connect()

```
void Assign::connect (
            int n0,
            int l0,
            int eg,
            int el,
            int n1,
            int l1 )
```

Definition at line 9049 of file grcc.cc.

3.5.3.12 fillEGraph()

```
Bool Assign::fillEGraph (
            int aid,
            BigInt nsym,
            BigInt esym,
            BigInt nsym1 )
```

Definition at line 9078 of file grcc.cc.

3.5.3.13 reordLeg()

```
int * Assign::reordLeg (
            int n,
            int * reord,
            int * plist,
            int * used )
```

Definition at line 9218 of file grcc.cc.

3.5.3.14 getLegParticle()

```
int Assign::getLegParticle (
            int n,
            int ln )
```

Definition at line 9293 of file grcc.cc.

3.5.3.15 legEdgeParticle()

```
int Assign::legEdgeParticle (
            int n,
            int ln,
            int pt )
```

Definition at line 9320 of file grcc.cc.

3.5.3.16 legPart()

```
int Assign::legPart (
            int v,
            int lg,
            int nplst,
            int * plst,
            int * rlist,
            const int size )
```

Definition at line 9340 of file grcc.cc.

3.5.3.17 candPart()

```
int Assign::candPart (
            int v,
            int ln,
            int * plist,
            const int size )
```

Definition at line 9356 of file grcc.cc.

3.5.3.18 selUnAssVertex()

```
int Assign::selUnAssVertex (
            void  )
```

Definition at line 9409 of file grcc.cc.

3.5.3.19 selUnAssVertexSimp()

```
int Assign::selUnAssVertexSimp (
            int lastv )
```

Definition at line 9389 of file grcc.cc.

3.5.3.20 selUnAssLeg()

```
int Assign::selUnAssLeg (
            int v,
            int lastlg )
```

Definition at line 9438 of file grcc.cc.

3.5.3.21 assignIVertex()

```
NCandSt Assign::assignIVertex (
            int v,
            int ia )
```

Definition at line 9477 of file grcc.cc.

3.5.3.22 assignPLeg()

```
Bool Assign::assignPLeg (
            int n,
            int ln,
            int pt )
```

Definition at line 9523 of file grcc.cc.

3.5.3.23 isOrdPLeg()

```
Bool Assign::isOrdPLeg (
            int n,
            int ln,
            int pt )
```

Definition at line 9596 of file grcc.cc.

3.5.3.24 detEdge()

```
Bool Assign::detEdge (
            int e )
```

Definition at line 9575 of file grcc.cc.

3.5.3.25 candPartClassify()

```
Bool Assign::candPartClassify (
            int v,
            int * npdass,
            int * pdass,
            int * npuass,
            int * puass,
            const int size )
```

Definition at line 9640 of file grcc.cc.

3.5.3.26 updateCandNode()

```
Bool Assign::updateCandNode (
            int v )
```

Definition at line 9677 of file grcc.cc.

3.5.3.27 checkOrderCpl()

```
Bool Assign::checkOrderCpl (
            void  )
```

Definition at line 9816 of file grcc.cc.

3.5.3.28 isOrdLegs()

```
Bool Assign::isOrdLegs (
            void  )
```

Definition at line 9854 of file grcc.cc.

3.5.3.29 isIsomorphic()

```
Bool Assign::isIsomorphic (
            MNodeClass * cl,
            BigInt * nsym,
            BigInt * esym,
            BigInt * nsym1 )
```

Definition at line 9892 of file grcc.cc.

3.5.3.30 cmpPermGraph()

```
int Assign::cmpPermGraph (
            int * p,
            MNodeClass * cl )
```

Definition at line 9968 of file grcc.cc.

3.5.3.31 cmpNodes()

```
int Assign::cmpNodes (
            int nd0,
            int nd1,
            MNodeClass * cn )
```

Definition at line 10050 of file grcc.cc.

3.5.3.32 edgeSym()

```
BigInt Assign::edgeSym (
            void  )
```

Definition at line 10070 of file grcc.cc.

3.5.3.33 saveCouple()

```
void Assign::saveCouple (
            int * sav )
```

Definition at line 8615 of file grcc.cc.

3.5.3.34 restoreCouple()

```
void Assign::restoreCouple (
            int * sav )
```

Definition at line 8627 of file grcc.cc.

3.5.3.35 subCouple()

```
Bool Assign::subCouple (
            int * cpl )
```

Definition at line 8639 of file grcc.cc.

3.5.3.36 checkCand()

```
Bool Assign::checkCand (
            const char * msg )
```

Definition at line 10129 of file grcc.cc.

3.5.3.37 checkNode()

```
void Assign::checkNode (
            int n,
            const char * msg )
```

Definition at line 10201 of file grcc.cc.

3.5.4 Field Documentation

3.5.4.1 egraph

```
EGraph* Assign::egraph
```

Definition at line 1188 of file grcc.h.

3.5.4.2 mgraph

```
MGraph* Assign::mgraph
```

Definition at line 1189 of file grcc.h.

3.5.4.3 sproc

```
SProcess* Assign::sproc
```

Definition at line 1190 of file grcc.h.

3.5.4.4 proc

```
Process* Assign::proc
```

Definition at line 1191 of file grcc.h.

3.5.4.5 model

```
Model* Assign::model
```

Definition at line 1192 of file grcc.h.

3.5.4.6 opt

```
Options* Assign::opt
```

Definition at line 1193 of file grcc.h.

3.5.4.7 astack

```
AStack* Assign::astack
```

Definition at line 1194 of file grcc.h.

3.5.4.8 pnclass

```
PNodeClass* Assign::pnclass
```

Definition at line 1195 of file grcc.h.

3.5.4.9 orbits

```
MOrbits* Assign::orbits
```

Definition at line 1196 of file grcc.h.

3.5.4.10 nNodes

```
int Assign::nNodes
```

Definition at line 1198 of file grcc.h.

3.5.4.11 nEdges

```
int Assign::nEdges
```

Definition at line 1199 of file grcc.h.

3.5.4.12 nExtern

```
int Assign::nExtern
```

Definition at line 1200 of file grcc.h.

3.5.4.13 nETotal

```
int Assign::nETotal
```

Definition at line 1201 of file grcc.h.

3.5.4.14 nAGraphs

```
BigInt Assign::nAGraphs
```

Definition at line 1203 of file grcc.h.

3.5.4.15 wAGraphs

```
Fraction Assign::wAGraphs
```

Definition at line 1204 of file grcc.h.

3.5.4.16 nAOPI

```
BigInt Assign::nAOPI
```

Definition at line 1205 of file grcc.h.

3.5.4.17 wAOPI

```
Fraction Assign::wAOPI
```

Definition at line 1206 of file grcc.h.

3.5.4.18 nodes

```
ANode** Assign::nodes
```

Definition at line 1208 of file grcc.h.

3.5.4.19 edges

```
AEdge** Assign::edges
```

Definition at line 1209 of file grcc.h.

3.5.4.20 checkpoint0

```
CheckPt Assign::checkpoint0
```

Definition at line 1211 of file grcc.h.

3.5.4.21 cplleft

```
int Assign::cplleft[GRCC_MAXNCPLG]
```

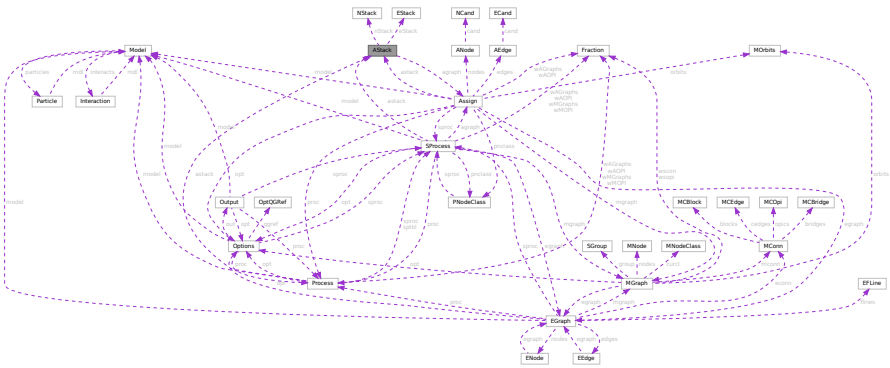
Definition at line 1213 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.6 AStack Class Reference

Collaboration diagram for AStack:



Public Member Functions

- AStack (int nSize, int eSize)
- void setAGraph (Assign ∗ag)
- void checkPoint (CheckPt sav)
- void restore (CheckPt sav)
- void restoreMsg (CheckPt sav, const char ∗msg)
- void prStack (void)
- void pushNode (int n)
- void pushEdge (int e)

Data Fields

- Assign ∗ agraph
- NStack ∗∗ nStack
- int nStackP
- int nSize
- EStack ∗∗ eStack
- int eStackP
- int eSize

3.6.1 Detailed Description

Definition at line 1367 of file grcc.h.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 AStack()

```
AStack::AStack (
            int nSize,
            int eSize )
```

Definition at line 10240 of file grcc.cc.

3.6.2.2 ∼AStack()

```
AStack::~AStack (
            void  )
```

Definition at line 10272 of file grcc.cc.

3.6.3 Member Function Documentation

3.6.3.1 setAGraph()

```
void AStack::setAGraph (
            Assign * ag )
```

Definition at line 10300 of file grcc.cc.

3.6.3.2 checkPoint()

```
void AStack::checkPoint (
            CheckPt sav )
```

Definition at line 10365 of file grcc.cc.

3.6.3.3 restore()

```
void AStack::restore (
            CheckPt sav )
```

Definition at line 10409 of file grcc.cc.

3.6.3.4 restoreMsg()

```
void AStack::restoreMsg (
            CheckPt sav,
            const char * msg )
```

Definition at line 10417 of file grcc.cc.

3.6.3.5 prStack()

```
void AStack::prStack (
            void  )
```

Definition at line 10432 of file grcc.cc.

3.6.3.6 pushNode()

```
void AStack::pushNode (
            int n )
```

Definition at line 10306 of file grcc.cc.

3.6.3.7 pushEdge()

```
void AStack::pushEdge (
            int e )
```

Definition at line 10336 of file grcc.cc.

3.6.4 Field Documentation

3.6.4.1 agraph

```
Assign* AStack::agraph
```

Definition at line 1371 of file grcc.h.

3.6.4.2 nStack

```
NStack** AStack::nStack
```

Definition at line 1373 of file grcc.h.

3.6.4.3 nStackP

```
int AStack::nStackP
```

Definition at line 1374 of file grcc.h.

3.6.4.4 nSize

```
int AStack::nSize
```

Definition at line 1375 of file grcc.h.

3.6.4.5 eStack

```
EStack** AStack::eStack
```

Definition at line 1377 of file grcc.h.

3.6.4.6 eStackP

```
int AStack::eStackP
```

Definition at line 1378 of file grcc.h.

3.6.4.7 eSize

```
int AStack::eSize
```

Definition at line 1379 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.7 bit_field Struct Reference

```
#include <structs.h>
```

Data Fields

- UBYTE bit_0: 1
- UBYTE bit_1: 1
- UBYTE bit_2: 1
- UBYTE bit_3: 1
- UBYTE bit_4: 1
- UBYTE bit_5: 1
- UBYTE bit_6: 1
- UBYTE bit_7: 1

3.7.1 Detailed Description

The struct bit_field is used by set_in, set_set, set_del and set_sub. They in turn are used in pre.c to toggle bits that indicate whether a character can be used as a separator of function arguments. This facility is used in the communication with external channels.

Definition at line 905 of file structs.h.

3.7.2 Field Documentation

3.7.2.1 bit_0

```
UBYTE bit_field::bit_0
```

Definition at line 906 of file structs.h.

3.7.2.2 bit_1

```
UBYTE bit_field::bit_1
```

Definition at line 907 of file structs.h.

3.7.2.3 bit_2

```
UBYTE bit_field::bit_2
```

Definition at line 908 of file structs.h.

3.7.2.4 bit_3

```
UBYTE bit_field::bit_3
```

Definition at line 909 of file structs.h.

3.7.2.5 bit_4

```
UBYTE bit_field::bit_4
```

Definition at line 910 of file structs.h.

3.7.2.6 bit_5

```
UBYTE bit_field::bit_5
```

Definition at line 911 of file structs.h.

3.7.2.7 bit_6

```
UBYTE bit_field::bit_6
```

Definition at line 912 of file structs.h.

3.7.2.8 bit_7

```
UBYTE bit_field::bit_7
```

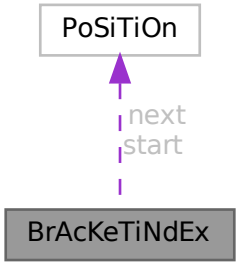
Definition at line 913 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.8 BrAcKeTiNdEx Struct Reference

Collaboration diagram for BrAcKeTiNdEx:



Data Fields

- POSITION start
- POSITION next
- LONG bracket
- LONG termsinbracket

3.8.1 Detailed Description

Definition at line 311 of file structs.h.

3.8.2 Field Documentation

3.8.2.1 start

```
POSITION BrAcKeTiNdEx::start
```

Definition at line 312 of file structs.h.

3.8.2.2 next

```
POSITION BrAcKeTiNdEx::next
```

Definition at line 313 of file structs.h.

3.8.2.3 bracket

```
LONG BrAcKeTiNdEx::bracket
```

Definition at line 314 of file structs.h.

3.8.2.4 termsinbracket

```
LONG BrAcKeTiNdEx::termsinbracket
```

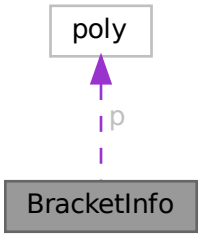
Definition at line 315 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.9 BracketInfo Struct Reference

Collaboration diagram for BracketInfo:



Public Member Functions

- BracketInfo (const std::vector< int > &pattern, int num_terms, const poly *p)
- bool operator< (const BracketInfo &rhs) const

Data Fields

- std::vector< int > pattern
- int num_terms
- int dummy = 0
- const poly * p

3.9.1 Detailed Description

Definition at line 1389 of file polygcd.cc.

3.9.2 Constructor & Destructor Documentation

3.9.2.1 BracketInfo()

```
BracketInfo::BracketInfo (
            const std::vector< int > & pattern,
            int num_terms,
            const poly * p )  [inline]
```

Definition at line 1394 of file polygcd.cc.

3.9.3 Member Function Documentation

3.9.3.1 operator<()

```
bool BracketInfo::operator< (
            const BracketInfo & rhs ) const  [inline]
```

Definition at line 1395 of file polygcd.cc.

3.9.4 Field Documentation

3.9.4.1 pattern

```
std::vector<int> BracketInfo::pattern
```

Definition at line 1390 of file polygcd.cc.

3.9.4.2 num_terms

```
int BracketInfo::num_terms
```

Definition at line 1391 of file polygcd.cc.

3.9.4.3 dummy

```
int BracketInfo::dummy = 0
```

Definition at line 1391 of file polygcd.cc.

3.9.4.4 p

```
const poly* BracketInfo::p
```

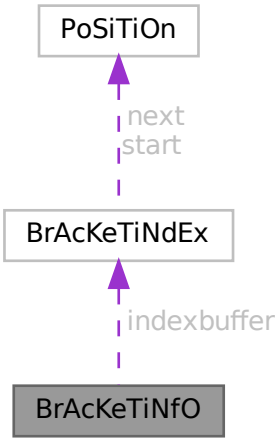
Definition at line 1392 of file polygcd.cc.

The documentation for this struct was generated from the following file:

- polygcd.cc

3.10 BrAcKeTiNfO Struct Reference

Collaboration diagram for BrAcKeTiNfO:



Data Fields

- BRACKETINDEX * indexbuffer
- WORD * bracketbuffer
- LONG bracketbuffersize
- LONG indexbuffersize
- LONG bracketfill
- LONG indexfill
- WORD SortType

3.10.1 Detailed Description

Definition at line 322 of file structs.h.

3.10.2 Field Documentation

3.10.2.1 indexbuffer

```
BRACKETINDEX* BrAcKeTiNfO::indexbuffer
```

[D]

Definition at line 323 of file structs.h.

Referenced by DoRecovery(), and FlushOut().

3.10.2.2 bracketbuffer

```
WORD* BrAcKeTiNfO::bracketbuffer
```

[D]

Definition at line 324 of file structs.h.

Referenced by DoRecovery().

3.10.2.3 bracketbuffersize

```
LONG BrAcKeTiNfO::bracketbuffersize
```

Definition at line 325 of file structs.h.

3.10.2.4 indexbuffersize

```
LONG BrAcKeTiNfO::indexbuffersize
```

Definition at line 326 of file structs.h.

3.10.2.5 bracketfill

```
LONG BrAcKeTiNfO::bracketfill
```

Definition at line 327 of file structs.h.

3.10.2.6 indexfill

```
LONG BrAcKeTiNfO::indexfill
```

Definition at line 328 of file structs.h.

3.10.2.7 SortType

```
WORD BrAcKeTiNfO::SortType
```

The sorting criterium used (like POWERFIRST etc)

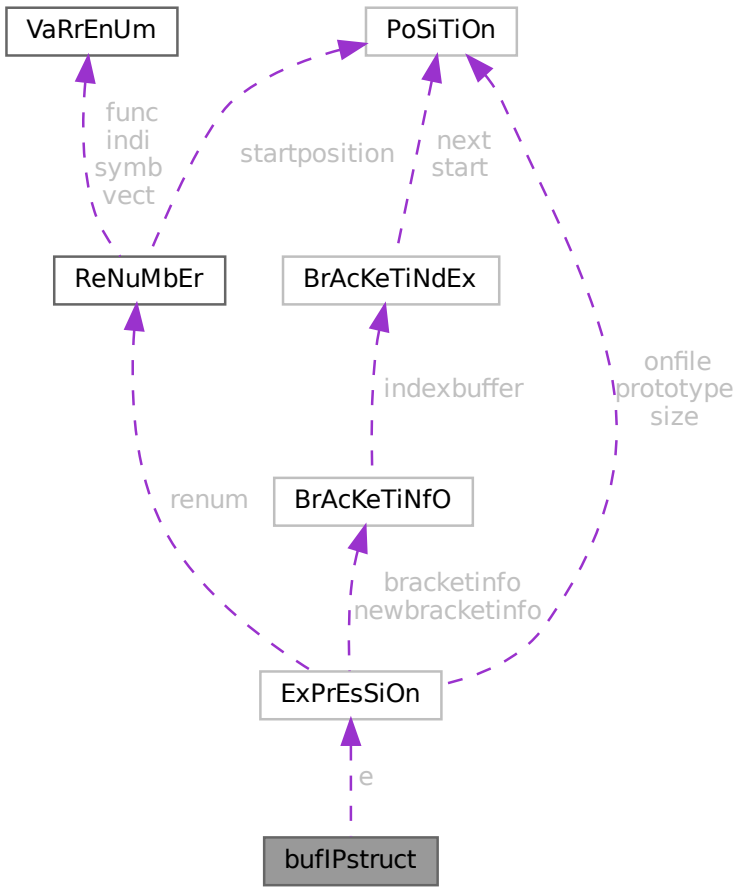
Definition at line 329 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.11 buflPstruct Struct Reference

Collaboration diagram for buflPstruct:



Data Fields

- LONG i
- struct ExPrEsSiOn e

3.11.1 Detailed Description

Definition at line 3928 of file parallel.c.

3.11.2 Field Documentation

3.11.2.1 i

```
LONG bufIPstruct::i
```

Definition at line 3929 of file parallel.c.

3.11.2.2 e

```
struct ExPrEsSiOn bufIPstruct::e
```

Definition at line 3930 of file parallel.c.

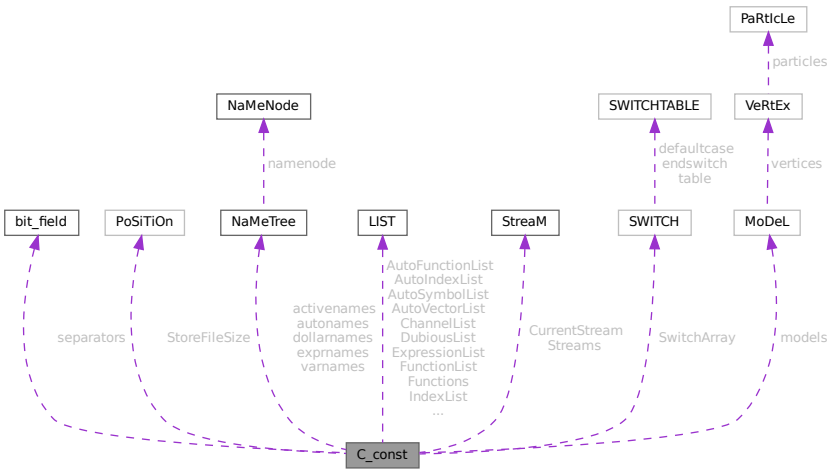
The documentation for this struct was generated from the following file:

- parallel.c

3.12 C_const Struct Reference

```
#include <structs.h>
```

Collaboration diagram for C_const:



Data Fields

- set_of_char separators
- POSITION StoreFileSize
- NAMETREE * dollarnames
- NAMETREE * exprnames
- NAMETREE * varnames
- LIST ChannelList
- LIST DubiousList
- LIST FunctionList
- LIST ExpressionList
- LIST IndexList
- LIST SetElementList
- LIST SetList
- LIST SymbolList
- LIST VectorList
- LIST PotModDolList

- LIST ModOptDolList
- LIST TableBaseList
- LIST cbufList
- LIST AutoSymbolList
- LIST AutoIndexList
- LIST AutoVectorList
- LIST AutoFunctionList
- NAMETREE * autonames
- LIST * Symbols
- LIST * Indices
- LIST * Vectors
- LIST * Functions
- NAMETREE ** activenames
- STREAM * Streams
- STREAM * CurrentStream
- SWITCH * SwitchArray
- MODEL ** models
- WORD * SwitchHeap
- LONG * termstack
- LONG * termsortstack
- UWORD * cmod
- UWORD * powmod
- UWORD * modpowers
- UWORD * halfmod
- WORD * ProtoType
- WORD * WildC
- LONG * IfHeap
- LONG * IfCount
- LONG * IfStack
- UBYTE * iBuffer
- UBYTE * iPointer
- UBYTE * iStop
- UBYTE ** LabelNames
- WORD * FixIndices
- WORD * termsumcheck
- UBYTE * WildcardNames
- int * Labels
- SBYTE * tokens
- SBYTE * toptokens
- SBYTE * endoftokens
- WORD * tokenarglevel
- UWORD * modinverses
- UBYTE * Fortran90Kind
- WORD ** MultiBracketBuf
- UBYTE * extrasym
- WORD * doloopstack
- WORD * doloopnest
- char * CheckpointRunAfter
- char * CheckpointRunBefore
- WORD * IfSumCheck
- WORD * CommuteInSet
- UBYTE * TestValue
- LONG argstack [MAXNEST]
- LONG insidestack [MAXNEST]
- LONG inexprstack [MAXNEST]

- LONG iBufferSize
- LONG TransEname
- LONG ProcessBucketSize
- LONG mProcessBucketSize
- LONG CModule
- LONG ThreadBucketSize
- LONG CheckpointStamp
- LONG CheckpointInterval
- int cbufnum
- int AutoDeclareFlag
- int NoShowInput
- int ShortStats
- int compiletype
- int firstconstindex
- int insidefirst
- int minsidefirst
- int wildflag
- int NumLabels
- int MaxLabels
- int lDefDim
- int lDefDim4
- int NumWildcardNames
- int WildcardBufferSize
- int MaxIf
- int NumStreams
- int MaxNumStreams
- int firstctypemessage
- int tablecheck
- int idoption
- int BottomLevel
- int CompileLevel
- int TokensWriteFlag
- int UnsureDollarMode
- int outsidefun
- int funpowers
- int WarnFlag
- int StatsFlag
- int NamesFlag
- int CodesFlag
- int SetupFlag
- int SortType
- int lSortType
- int ThreadStats
- int FinalStats
- int OldParallelStats
- int ThreadsFlag
- int ThreadBalancing
- int ThreadSortFileSynch
- int ProcessStats
- int BracketNormalize
- int maxtermlevel
- int dumnumflag
- int bracketindexflag
- int parallelflag
- int mparallelflag

- int inparallelflag
- int partodoflag
- int properorderflag
- int vetofilling
- int tablefilling
- int vetotablebasefill
- int exprfillwarning
- int lhdollarflag
- int NoCompress
- int IsFortran90
- int MultiBracketLevels
- int topolynomialflag
- int ffbufnum
- int OldFactArgFlag
- int MemDebugFlag
- int OldGCDflag
- int OldPRFSignFlag
- int WTimeStatsFlag
- int SortReallocateFlag
- int PrintBacktraceFlag
- int FlintPolyFlag
- int HumanStatsFlag
- int GrccVerbose
- int SortVerbose
- int doloopstacksize
- int doolooplevel
- int CheckpointFlag
- int SizeCommuteInSet
- int origin
- int vectorlikeLHS
- int nummodels
- int modelspace
- int ModelLevel
- int ShortStatsMax
- int InnerTest
- WORD argsumcheck [MAXNEST]
- WORD insidesumcheck [MAXNEST]
- WORD inexprsumcheck [MAXNEST]
- WORD RepSumCheck [MAXREPEAT]
- WORD lUniTrace [4]
- WORD RepLevel
- WORD arglevel
- WORD insidelevel
- WORD inexprlevel
- WORD termlevel
- WORD MustTestTable
- WORD DumNum
- WORD ncmod
- WORD npowmod
- WORD modmode
- WORD nhalfmod
- WORD DirtPow
- WORD lUnitTrace
- WORD NwildC
- WORD ComDefer

- WORD CollectFun
- WORD AltCollectFun
- WORD OutputMode
- WORD Cnumpows
- WORD OutputSpaces
- WORD OutNumberType
- WORD DidClean
- WORD IfLevel
- WORD WhileLevel
- WORD SwitchLevel
- WORD SwitchInArray
- WORD MaxSwitch
- WORD LogHandle
- WORD LineLength
- WORD StoreHandle
- WORD HideLevel
- WORD IPolyFun
- WORD IPolyFunInv
- WORD IPolyFunType
- WORD IPolyFunExp
- WORD IPolyFunVar
- WORD IPolyFunPow
- WORD SymChangeFlag
- WORD CollectPercentage
- WORD extrasymbols
- WORD PolyRatFunChanged
- WORD ToBeInFactors
- UBYTE Commercial [COMMERCIALSIZE+2]
- UBYTE debugFlags [MAXFLAGS+2]

3.12.1 Detailed Description

The C_const struct is part of the global data and resides in the ALLGLOBALS struct A under the name #C. We see it used with the macro AC as in AC.exprnames. It contains variables that involve the compiler and objects set during compilation.

Definition at line 1679 of file structs.h.

3.12.2 Field Documentation

3.12.2.1 separators

```
set_of_char C_const::separators
```

Separators in #call and #do

Definition at line 1680 of file structs.h.

3.12.2.2 StoreFileSize

```
POSITION C_const::StoreFileSize
```

Definition at line 1681 of file structs.h.

3.12.2.3 dollarnames

```
NAMETREE* C_const::dollarnames
```

[D] Names of dollar variables

Definition at line 1682 of file structs.h.

3.12.2.4 exprnames

```
NAMETREE* C_const::exprnames
```

[D] Names of expressions

Definition at line 1683 of file structs.h.

3.12.2.5 varnames

```
NAMETREE* C_const::varnames
```

[D] Names of regular variables

Definition at line 1684 of file structs.h.

3.12.2.6 ChannelList

```
LIST C_const::ChannelList
```

Used for the #write statement. Contains CHANNEL

Definition at line 1685 of file structs.h.

3.12.2.7 DubiousList

```
LIST C_const::DubiousList
```

List of dubious variables. Contains DUBIOUSV. If not empty -> no execution

Definition at line 1687 of file structs.h.

3.12.2.8 FunctionList

```
LIST C_const::FunctionList
```

List of functions and properties. Contains FUNCTIONS

Definition at line 1689 of file structs.h.

3.12.2.9 ExpressionList

```
LIST C_const::ExpressionList
```

List of expressions, locations etc.

Definition at line 1690 of file structs.h.

3.12.2.10 IndexList

```
LIST C_const::IndexList
```

List of indices

Definition at line 1691 of file structs.h.

3.12.2.11 SetElementList

```
LIST C_const::SetElementList
```

List of all elements of all sets

Definition at line 1692 of file structs.h.

3.12.2.12 SetList

```
LIST C_const::SetList
```

List of the sets

Definition at line 1693 of file structs.h.

3.12.2.13 SymbolList

```
LIST C_const::SymbolList
```

List of the symbols and their properties

Definition at line 1694 of file structs.h.

3.12.2.14 VectorList

```
LIST C_const::VectorList
```

List of the vectors

Definition at line 1695 of file structs.h.

3.12.2.15 PotModDolList

```
LIST C_const::PotModDolList
```

Potentially changed dollars

Definition at line 1696 of file structs.h.

3.12.2.16 ModOptDolList

```
LIST C_const::ModOptDolList
```

Module Option Dollars list

Definition at line 1697 of file structs.h.

3.12.2.17 TableBaseList

```
LIST C_const::TableBaseList
```

TableBase list

Definition at line 1698 of file structs.h.

3.12.2.18 cbufList

```
LIST C_const::cbufList
```

List of compiler buffers

Definition at line 1702 of file structs.h.

3.12.2.19 AutoSymbolList

```
LIST C_const::AutoSymbolList
```

Definition at line 1706 of file structs.h.

3.12.2.20 AutoIndexList

```
LIST C_const::AutoIndexList
```

Definition at line 1707 of file structs.h.

3.12.2.21 AutoVectorList

```
LIST C_const::AutoVectorList
```

Definition at line 1708 of file structs.h.

3.12.2.22 AutoFunctionList

```
LIST C_const::AutoFunctionList
```

Definition at line 1709 of file structs.h.

3.12.2.23 autonames

```
NAMETREE* C_const::autonames
```

[D] Names in autodeclare

Definition at line 1710 of file structs.h.

3.12.2.24 Symbols

```
LIST* C_const::Symbols
```

Definition at line 1712 of file structs.h.

3.12.2.25 Indices

```
LIST* C_const::Indices
```

Definition at line 1714 of file structs.h.

3.12.2.26 Vectors

```
LIST* C_const::Vectors
```

Definition at line 1715 of file structs.h.

3.12.2.27 Functions

```
LIST* C_const::Functions
```

Definition at line 1716 of file structs.h.

3.12.2.28 activenames

```
NAMETREE** C_const::activenames
```

Definition at line 1717 of file structs.h.

3.12.2.29 Streams

```
STREAM* C_const::Streams
```

(C) Pointer for AutoDeclare statement. Points either to varnames or autonames. [D] The input streams.

Definition at line 1719 of file structs.h.

3.12.2.30 CurrentStream

```
STREAM* C_const::CurrentStream
```

(C) The current input stream. Streams are: do loop, file, prevariable. points into Streams memory.

Definition at line 1720 of file structs.h.

3.12.2.31 SwitchArray

```
SWITCH* C_const::SwitchArray
```

Definition at line 1722 of file structs.h.

3.12.2.32 models

```
MODEL** C_const::models
```

Definition at line 1723 of file structs.h.

3.12.2.33 SwitchHeap

```
WORD* C_const::SwitchHeap
```

Definition at line 1724 of file structs.h.

3.12.2.34 termstack

```
LONG* C_const::termstack
```

[D] Last term statement {offset}

Definition at line 1725 of file structs.h.

3.12.2.35 termsortstack

```
LONG* C_const::termsortstack
```

[D] Last sort statement {offset}

Definition at line 1726 of file structs.h.

3.12.2.36 cmod

```
UWORD* C_const::cmod
```

[D] Local setting of modulus. Pointer to value.

Definition at line 1727 of file structs.h.

3.12.2.37 powmod

```
UWORD* C_const::powmod
```

Local setting printing as powers. Points into cmod memory

Definition at line 1728 of file structs.h.

3.12.2.38 modpowers

```
UWORD* C_const::modpowers
```

[D] The conversion table for mod-> powers.

Definition at line 1729 of file structs.h.

3.12.2.39 halfmod

```
UWORD* C_const::halfmod
```

Definition at line 1730 of file structs.h.

3.12.2.40 ProtoType

```
WORD* C_const::ProtoType
```

Definition at line 1731 of file structs.h.

3.12.2.41 WildC

```
WORD* C_const::WildC
```

Definition at line 1732 of file structs.h.

3.12.2.42 IfHeap

```
LONG* C_const::IfHeap
```

[D] Keeps track of where to go in if

Definition at line 1733 of file structs.h.

3.12.2.43 IfCount

```
LONG* C_const::IfCount
```

[D] Keeps track of where to go in if

Definition at line 1734 of file structs.h.

3.12.2.44 IfStack

```
LONG* C_const::IfStack
```

Keeps track of where to go in if. Points into IfHeap-memory

Definition at line 1735 of file structs.h.

3.12.2.45 iBuffer

```
UBYTE* C_const::iBuffer
```

[D] Compiler input buffer

Definition at line 1736 of file structs.h.

3.12.2.46 iPointer

```
UBYTE* C_const::iPointer
```

Running pointer in the compiler input buffer

Definition at line 1737 of file structs.h.

3.12.2.47 iStop

```
UBYTE* C_const::iStop
```

Top of iBuffer

Definition at line 1738 of file structs.h.

3.12.2.48 LabelNames

```
UBYTE** C_const::LabelNames
```

[D] List of names in label statements

Definition at line 1739 of file structs.h.

3.12.2.49 FixIndices

```
WORD* C_const::FixIndices
```

[D] Buffer of fixed indices

Definition at line 1740 of file structs.h.

3.12.2.50 termsumcheck

```
WORD* C_const::termsumcheck
```

[D] Checking of nesting

Definition at line 1741 of file structs.h.

3.12.2.51 WildcardNames

```
UBYTE* C_const::WildcardNames
```

[D] Names of ?a variables

Definition at line 1742 of file structs.h.

3.12.2.52 Labels

```
int* C_const::Labels
```

Label information for during run. Pointer into LabelNames memory.

Definition at line 1743 of file structs.h.

3.12.2.53 tokens

```
SBYTE* C_const::tokens
```

[D] Array with tokens for tokenizer

Definition at line 1744 of file structs.h.

3.12.2.54 toptokens

```
SBYTE* C_const::toptokens
```

Top of tokens

Definition at line 1745 of file structs.h.

3.12.2.55 endoftokens

```
SBYTE* C_const::endoftokens
```

End of the actual tokens

Definition at line 1746 of file structs.h.

3.12.2.56 tokenarglevel

```
WORD* C_const::tokenarglevel
```

[D] Keeps track of function arguments

Definition at line 1747 of file structs.h.

3.12.2.57 modinverses

```
UWORD* C_const::modinverses
```

Definition at line 1748 of file structs.h.

3.12.2.58 Fortran90Kind

```
UBYTE* C_const::Fortran90Kind
```

Definition at line 1749 of file structs.h.

3.12.2.59 MultiBracketBuf

```
WORD** C_const::MultiBracketBuf
```

Definition at line 1750 of file structs.h.

3.12.2.60 extrasym

```
UBYTE* C_const::extrasym
```

Definition at line 1751 of file structs.h.

3.12.2.61 doloopstack

```
WORD* C_const::doloopstack
```

Definition at line 1752 of file structs.h.

3.12.2.62 doloopnest

```
WORD* C_const::doloopnest
```

Definition at line 1753 of file structs.h.

3.12.2.63 CheckpointRunAfter

```
char* C_const::CheckpointRunAfter
```

[D] Filename of script to be executed *before* creating the snapshot. =0 if no script shall be executed.

Definition at line 1754 of file structs.h.

3.12.2.64 CheckpointRunBefore

```
char* C_const::CheckpointRunBefore
```

[D] Filename of script to be executed *after* having created the snapshot. =0 if no script shall be executed.

Definition at line 1756 of file structs.h.

3.12.2.65 IfSumCheck

```
WORD* C_const::IfSumCheck
```

[D] Keeps track of if-nesting

Definition at line 1758 of file structs.h.

3.12.2.66 CommuteInSet

```
WORD* C_const::CommuteInSet
```

Definition at line 1759 of file structs.h.

3.12.2.67 TestValue

```
UBYTE* C_const::TestValue
```

Definition at line 1760 of file structs.h.

3.12.2.68 argstack

```
LONG C_const::argstack[MAXNEST]
```

Definition at line 1768 of file structs.h.

3.12.2.69 insidestack

```
LONG C_const::insidestack[MAXNEST]
```

Definition at line 1769 of file structs.h.

3.12.2.70 inexprstack

```
LONG C_const::inexprstack[MAXNEST]
```

Definition at line 1770 of file structs.h.

3.12.2.71 iBufferSize

```
LONG C_const::iBufferSize
```

Definition at line 1771 of file structs.h.

3.12.2.72 TransEname

```
LONG C_const::TransEname
```

Definition at line 1772 of file structs.h.

3.12.2.73 ProcessBucketSize

```
LONG C_const::ProcessBucketSize
```

Definition at line 1774 of file structs.h.

3.12.2.74 mProcessBucketSize

```
LONG C_const::mProcessBucketSize
```

Definition at line 1775 of file structs.h.

3.12.2.75 CModule

```
LONG C_const::CModule
```

Definition at line 1776 of file structs.h.

3.12.2.76 ThreadBucketSize

```
LONG C_const::ThreadBucketSize
```

Definition at line 1777 of file structs.h.

3.12.2.77 CheckpointStamp

```
LONG C_const::CheckpointStamp
```

Timestamp of the last created snapshot (set to Timer(0)).

Definition at line 1778 of file structs.h.

3.12.2.78 CheckpointInterval

```
LONG C_const::CheckpointInterval
```

Time interval in milliseconds for snapshots. =0 if snapshots shall be created at the end of *every* module.

Definition at line 1779 of file structs.h.

3.12.2.79 cbufnum

```
int C_const::cbufnum
```

Current compiler buffer

Definition at line 1787 of file structs.h.

3.12.2.80 AutoDeclareFlag

```
int C_const::AutoDeclareFlag
```

Definition at line 1788 of file structs.h.

3.12.2.81 NoShowInput

```
int C_const::NoShowInput
```

(C) Mode of looking for names. Set to NOAUTO (=0) or WITHAUTO (=2), cf. AutoDeclare statement

Definition at line 1790 of file structs.h.

3.12.2.82 ShortStats

```
int C_const::ShortStats
```

Definition at line 1791 of file structs.h.

3.12.2.83 compiletype

```
int C_const::compiletype
```

Definition at line 1792 of file structs.h.

3.12.2.84 firstconstindex

```
int C_const::firstconstindex
```

Definition at line 1793 of file structs.h.

3.12.2.85 insidefirst

```
int C_const::insidefirst
```

Definition at line 1794 of file structs.h.

3.12.2.86 minsidefirst

```
int C_const::minsidefirst
```

Definition at line 1795 of file structs.h.

3.12.2.87 wildflag

```
int C_const::wildflag
```

Definition at line 1796 of file structs.h.

3.12.2.88 NumLabels

```
int C_const::NumLabels
```

Definition at line 1797 of file structs.h.

3.12.2.89 MaxLabels

```
int C_const::MaxLabels
```

Definition at line 1798 of file structs.h.

3.12.2.90 lDefDim

```
int C_const::lDefDim
```

Definition at line 1799 of file structs.h.

3.12.2.91 lDefDim4

```
int C_const::lDefDim4
```

Definition at line 1800 of file structs.h.

3.12.2.92 NumWildcardNames

```
int C_const::NumWildcardNames
```

Definition at line 1801 of file structs.h.

3.12.2.93 WildcardBufferSize

```
int C_const::WildcardBufferSize
```

Definition at line 1802 of file structs.h.

3.12.2.94 MaxIf

```
int C_const::MaxIf
```

Definition at line 1803 of file structs.h.

3.12.2.95 NumStreams

```
int C_const::NumStreams
```

Definition at line 1804 of file structs.h.

3.12.2.96 MaxNumStreams

```
int C_const::MaxNumStreams
```

Definition at line 1805 of file structs.h.

3.12.2.97 firstctypemessage

```
int C_const::firstctypemessage
```

Definition at line 1806 of file structs.h.

3.12.2.98 tablecheck

```
int C_const::tablecheck
```

Definition at line 1807 of file structs.h.

3.12.2.99 idoption

```
int C_const::idoption
```

Definition at line 1808 of file structs.h.

3.12.2.100 BottomLevel

```
int C_const::BottomLevel
```

Definition at line 1809 of file structs.h.

3.12.2.101 CompileLevel

```
int C_const::CompileLevel
```

Definition at line 1810 of file structs.h.

3.12.2.102 TokensWriteFlag

```
int C_const::TokensWriteFlag
```

Definition at line 1811 of file structs.h.

3.12.2.103 UnsureDollarMode

```
int C_const::UnsureDollarMode
```

Definition at line 1812 of file structs.h.

3.12.2.104 outsidefun

```
int C_const::outsidefun
```

Definition at line 1813 of file structs.h.

3.12.2.105 funpowers

```
int C_const::funpowers
```

Definition at line 1814 of file structs.h.

3.12.2.106 WarnFlag

```
int C_const::WarnFlag
```

Definition at line 1815 of file structs.h.

3.12.2.107 StatsFlag

```
int C_const::StatsFlag
```

Definition at line 1816 of file structs.h.

3.12.2.108 NamesFlag

```
int C_const::NamesFlag
```

Definition at line 1817 of file structs.h.

3.12.2.109 CodesFlag

```
int C_const::CodesFlag
```

Definition at line 1818 of file structs.h.

3.12.2.110 SetupFlag

```
int C_const::SetupFlag
```

Definition at line 1819 of file structs.h.

3.12.2.111 SortType

```
int C_const::SortType
```

Definition at line 1820 of file structs.h.

3.12.2.112 lSortType

```
int C_const::lSortType
```

Definition at line 1821 of file structs.h.

3.12.2.113 ThreadStats

```
int C_const::ThreadStats
```

Definition at line 1822 of file structs.h.

3.12.2.114 FinalStats

```
int C_const::FinalStats
```

Definition at line 1823 of file structs.h.

3.12.2.115 OldParallelStats

```
int C_const::OldParallelStats
```

Definition at line 1824 of file structs.h.

3.12.2.116 ThreadsFlag

```
int C_const::ThreadsFlag
```

Definition at line 1825 of file structs.h.

3.12.2.117 ThreadBalancing

```
int C_const::ThreadBalancing
```

Definition at line 1826 of file structs.h.

3.12.2.118 ThreadSortFileSynch

```
int C_const::ThreadSortFileSynch
```

Definition at line 1827 of file structs.h.

3.12.2.119 ProcessStats

```
int C_const::ProcessStats
```

Definition at line 1828 of file structs.h.

3.12.2.120 BracketNormalize

```
int C_const::BracketNormalize
```

Definition at line 1829 of file structs.h.

3.12.2.121 maxtermlevel

```
int C_const::maxtermlevel
```

Definition at line 1830 of file structs.h.

3.12.2.122 dumnumflag

```
int C_const::dumnumflag
```

Definition at line 1831 of file structs.h.

3.12.2.123 bracketindexflag

```
int C_const::bracketindexflag
```

Definition at line 1832 of file structs.h.

3.12.2.124 parallelflag

```
int C_const::parallelflag
```

Definition at line 1833 of file structs.h.

3.12.2.125 mparallelflag

```
int C_const::mparallelflag
```

Definition at line 1834 of file structs.h.

3.12.2.126 inparallelflag

```
int C_const::inparallelflag
```

Definition at line 1835 of file structs.h.

3.12.2.127 partodoflag

```
int C_const::partodoflag
```

Definition at line 1836 of file structs.h.

3.12.2.128 properorderflag

```
int C_const::properorderflag
```

Definition at line 1837 of file structs.h.

3.12.2.129 vetofilling

```
int C_const::vetofilling
```

Definition at line 1838 of file structs.h.

3.12.2.130 tablefilling

```
int C_const::tablefilling
```

Definition at line 1839 of file structs.h.

3.12.2.131 vetotablebasefill

```
int C_const::vetotablebasefill
```

Definition at line 1840 of file structs.h.

3.12.2.132 exprfillwarning

```
int C_const::exprfillwarning
```

Definition at line 1841 of file structs.h.

3.12.2.133 lhdollarflag

```
int C_const::lhdollarflag
```

Definition at line 1842 of file structs.h.

3.12.2.134 NoCompress

```
int C_const::NoCompress
```

Definition at line 1843 of file structs.h.

3.12.2.135 IsFortran90

```
int C_const::IsFortran90
```

Definition at line 1844 of file structs.h.

3.12.2.136 MultiBracketLevels

```
int C_const::MultiBracketLevels
```

Definition at line 1845 of file structs.h.

3.12.2.137 topolynomialflag

```
int C_const::topolynomialflag
```

Definition at line 1846 of file structs.h.

3.12.2.138 ffbufnum

```
int C_const::ffbufnum
```

Definition at line 1847 of file structs.h.

3.12.2.139 OldFactArgFlag

```
int C_const::OldFactArgFlag
```

Definition at line 1848 of file structs.h.

3.12.2.140 MemDebugFlag

```
int C_const::MemDebugFlag
```

Definition at line 1849 of file structs.h.

3.12.2.141 OldGCDflag

```
int C_const::OldGCDflag
```

Definition at line 1850 of file structs.h.

3.12.2.142 OldPRFSignFlag

```
int C_const::OldPRFSignFlag
```

Definition at line 1851 of file structs.h.

3.12.2.143 WTimeStatsFlag

```
int C_const::WTimeStatsFlag
```

Definition at line 1852 of file structs.h.

3.12.2.144 SortReallocateFlag

```
int C_const::SortReallocateFlag
```

Definition at line 1853 of file structs.h.

3.12.2.145 PrintBacktraceFlag

```
int C_const::PrintBacktraceFlag
```

Definition at line 1857 of file structs.h.

3.12.2.146 FlintPolyFlag

```
int C_const::FlintPolyFlag
```

Definition at line 1858 of file structs.h.

3.12.2.147 HumanStatsFlag

```
int C_const::HumanStatsFlag
```

Definition at line 1859 of file structs.h.

3.12.2.148 GrccVerbose

```
int C_const::GrccVerbose
```

Definition at line 1860 of file structs.h.

3.12.2.149 SortVerbose

```
int C_const::SortVerbose
```

Definition at line 1861 of file structs.h.

3.12.2.150 doloopstacksize

```
int C_const::doloopstacksize
```

Definition at line 1862 of file structs.h.

3.12.2.151 dolooplevel

```
int C_const::dolooplevel
```

Definition at line 1863 of file structs.h.

3.12.2.152 CheckpointFlag

```
int C_const::CheckpointFlag
```

Tells preprocessor whether checkpoint code must executed. -1 : do recovery from snapshot, set by command line option; 0 : do nothing; 1 : create snapshots, set by On checkpoint statement

Definition at line 1864 of file structs.h.

3.12.2.153 SizeCommuteInSet

```
int C_const::SizeCommuteInSet
```

Definition at line 1868 of file structs.h.

3.12.2.154 origin

```
int C_const::origin
```

Definition at line 1873 of file structs.h.

3.12.2.155 vectorlikeLHS

```
int C_const::vectorlikeLHS
```

Definition at line 1874 of file structs.h.

3.12.2.156 nummodels

```
int C_const::nummodels
```

Definition at line 1875 of file structs.h.

3.12.2.157 modelspace

```
int C_const::modelspace
```

Definition at line 1876 of file structs.h.

3.12.2.158 ModelLevel

```
int C_const::ModelLevel
```

Definition at line 1877 of file structs.h.

3.12.2.159 ShortStatsMax

```
int C_const::ShortStatsMax
```

Definition at line 1878 of file structs.h.

3.12.2.160 InnerTest

```
int C_const::InnerTest
```

Definition at line 1879 of file structs.h.

3.12.2.161 argsumcheck

```
WORD C_const::argsumcheck[MAXNEST]
```

Definition at line 1880 of file structs.h.

3.12.2.162 insidesumcheck

```
WORD C_const::insidesumcheck[MAXNEST]
```

Definition at line 1881 of file structs.h.

3.12.2.163 inexprsumcheck

```
WORD C_const::inexprsumcheck[MAXNEST]
```

Definition at line 1882 of file structs.h.

3.12.2.164 RepSumCheck

```
WORD C_const::RepSumCheck[MAXREPEAT]
```

Definition at line 1883 of file structs.h.

3.12.2.165 lUniTrace

```
WORD C_const::lUniTrace[4]
```

Definition at line 1884 of file structs.h.

3.12.2.166 RepLevel

```
WORD C_const::RepLevel
```

Definition at line 1885 of file structs.h.

3.12.2.167 arglevel

```
WORD C_const::arglevel
```

Definition at line 1886 of file structs.h.

3.12.2.168 insidelevel

```
WORD C_const::insidelevel
```

Definition at line 1887 of file structs.h.

3.12.2.169 inexprlevel

```
WORD C_const::inexprlevel
```

Definition at line 1888 of file structs.h.

3.12.2.170 termlevel

```
WORD C_const::termlevel
```

Definition at line 1889 of file structs.h.

3.12.2.171 MustTestTable

```
WORD C_const::MustTestTable
```

Definition at line 1890 of file structs.h.

3.12.2.172 DumNum

```
WORD C_const::DumNum
```

Definition at line 1891 of file structs.h.

3.12.2.173 ncmod

```
WORD C_const::ncmod
```

Definition at line 1892 of file structs.h.

3.12.2.174 npowmod

```
WORD C_const::npowmod
```

Definition at line 1893 of file structs.h.

3.12.2.175 modmode

```
WORD C_const::modmode
```

Definition at line 1894 of file structs.h.

3.12.2.176 nhalfmod

```
WORD C_const::nhalfmod
```

Definition at line 1895 of file structs.h.

3.12.2.177 DirtPow

```
WORD C_const::DirtPow
```

Definition at line 1896 of file structs.h.

3.12.2.178 lUnitTrace

```
WORD C_const::lUnitTrace
```

Definition at line 1897 of file structs.h.

3.12.2.179 NwildC

```
WORD C_const::NwildC
```

Definition at line 1898 of file structs.h.

3.12.2.180 ComDefer

```
WORD C_const::ComDefer
```

Definition at line 1899 of file structs.h.

3.12.2.181 CollectFun

```
WORD C_const::CollectFun
```

Definition at line 1900 of file structs.h.

3.12.2.182 AltCollectFun

```
WORD C_const::AltCollectFun
```

Definition at line 1901 of file structs.h.

3.12.2.183 OutputMode

```
WORD C_const::OutputMode
```

Definition at line 1902 of file structs.h.

3.12.2.184 Cnumpows

```
WORD C_const::Cnumpows
```

Definition at line 1903 of file structs.h.

3.12.2.185 OutputSpaces

```
WORD C_const::OutputSpaces
```

Definition at line 1904 of file structs.h.

3.12.2.186 OutNumberType

```
WORD C_const::OutNumberType
```

Definition at line 1905 of file structs.h.

3.12.2.187 DidClean

```
WORD C_const::DidClean
```

Definition at line 1906 of file structs.h.

3.12.2.188 IfLevel

```
WORD C_const::IfLevel
```

Definition at line 1907 of file structs.h.

3.12.2.189 WhileLevel

```
WORD C_const::WhileLevel
```

Definition at line 1908 of file structs.h.

3.12.2.190 SwitchLevel

```
WORD C_const::SwitchLevel
```

Definition at line 1909 of file structs.h.

3.12.2.191 SwitchInArray

```
WORD C_const::SwitchInArray
```

Definition at line 1910 of file structs.h.

3.12.2.192 MaxSwitch

```
WORD C_const::MaxSwitch
```

Definition at line 1911 of file structs.h.

3.12.2.193 LogHandle

```
WORD C_const::LogHandle
```

Definition at line 1912 of file structs.h.

3.12.2.194 LineLength

```
WORD C_const::LineLength
```

Definition at line 1913 of file structs.h.

3.12.2.195 StoreHandle

```
WORD C_const::StoreHandle
```

Definition at line 1914 of file structs.h.

3.12.2.196 HideLevel

```
WORD C_const::HideLevel
```

Definition at line 1915 of file structs.h.

3.12.2.197 lPolyFun

```
WORD C_const::lPolyFun
```

Definition at line 1916 of file structs.h.

3.12.2.198 lPolyFunInv

```
WORD C_const::lPolyFunInv
```

Definition at line 1917 of file structs.h.

3.12.2.199 lPolyFunType

```
WORD C_const::lPolyFunType
```

Definition at line 1918 of file structs.h.

3.12.2.200 lPolyFunExp

```
WORD C_const::lPolyFunExp
```

Definition at line 1919 of file structs.h.

3.12.2.201 lPolyFunVar

```
WORD C_const::lPolyFunVar
```

Definition at line 1920 of file structs.h.

3.12.2.202 lPolyFunPow

```
WORD C_const::lPolyFunPow
```

Definition at line 1921 of file structs.h.

3.12.2.203 SymChangeFlag

```
WORD C_const::SymChangeFlag
```

Definition at line 1922 of file structs.h.

3.12.2.204 CollectPercentage

```
WORD C_const::CollectPercentage
```

Definition at line 1923 of file structs.h.

3.12.2.205 extrasymbols

```
WORD C_const::extrasymbols
```

Definition at line 1924 of file structs.h.

3.12.2.206 PolyRatFunChanged

```
WORD C_const::PolyRatFunChanged
```

Definition at line 1925 of file structs.h.

3.12.2.207 ToBeInFactors

```
WORD C_const::ToBeInFactors
```

Definition at line 1926 of file structs.h.

3.12.2.208 Commercial

```
UBYTE C_const::Commercial[COMMERCIALSIZE+2]
```

Definition at line 1930 of file structs.h.

3.12.2.209 debugFlags

```
UBYTE C_const::debugFlags[MAXFLAGS+2]
```

Definition at line 1931 of file structs.h.

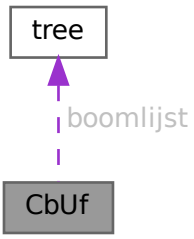
The documentation for this struct was generated from the following file:

- structs.h

3.13 CbUf Struct Reference

`#include <structs.h>`

Collaboration diagram for CbUf:



Data Fields

- WORD * Buffer
- WORD * Top
- WORD * Pointer
- WORD ** lhs
- WORD ** rhs
- LONG * CanCommu
- LONG * NumTerms
- WORD * numdum
- WORD * dimension
- COMPTREE * boomlijst
- LONG BufferSize
- int numlhs
- int numrhs
- int maxlhs
- int maxrhs
- int mnumlhs
- int mnumrhs
- int numtree
- int rootnum
- int MaxTreeSize

3.13.1 Detailed Description

The CBUF struct is used by the compiler. It is a compiler buffer of which since version 3.0 there can be many.

Definition at line 970 of file structs.h.

3.13.2 Field Documentation

3.13.2.1 Buffer

```
WORD* CbUf::Buffer
```

[D] Size in BufferSize

Definition at line 971 of file structs.h.

Referenced by CleanupArgCache(), clearcbuf(), DoubleCbuffer(), finishcbuf(), inicbufs(), PF_BroadcastCBuf(), and Processor().

3.13.2.2 Top

```
WORD* CbUf::Top
```

pointer to the end of the Buffer memory

Definition at line 972 of file structs.h.

Referenced by AddNtoC(), AddNtoL(), DoubleCbuffer(), finishcbuf(), inicbufs(), and PF_BroadcastCBuf().

3.13.2.3 Pointer

```
WORD* CbUf::Pointer
```

pointer into the Buffer memory

Definition at line 973 of file structs.h.

Referenced by AddLHS(), AddNtoC(), AddNtoL(), AddRHS(), CleanupArgCache(), clearcbuf(), DoubleCbuffer(), finishcbuf(), Generator(), inicbufs(), PF_BroadcastCBuf(), and Processor().

3.13.2.4 lhs

```
WORD** CbUf::lhs
```

[D] Size in maxlhs. list of pointers into Buffer.

Definition at line 974 of file structs.h.

Referenced by AddLHS(), DoubleCbuffer(), finishcbuf(), Generator(), inicbufs(), PF_BroadcastCBuf(), Processor(), and TestMatch().

3.13.2.5 rhs

```
WORD** CbUf::rhs
```

[D] Size in maxrhs. list of pointers into Buffer.

Definition at line 975 of file structs.h.

Referenced by AddRHS(), CleanupArgCache(), clearcbuf(), DoubleCbuffer(), finishcbuf(), Generator(), inicbufs(), IniFbuffer(), PF_BroadcastCBuf(), and Processor().

3.13.2.6 CanCommu

```
LONG* CbUf::CanCommu
```

points into rhs memory behind WORD* area.

Definition at line 976 of file structs.h.

Referenced by AddRHS(), finishcbuf(), inicbufs(), IniFbuffer(), and PF_BroadcastCBuf().

3.13.2.7 NumTerms

```
LONG* CbUf::NumTerms
```

points into rhs memory behind CanCommu area

Definition at line 977 of file structs.h.

Referenced by AddRHS(), finishcbuf(), inicbufs(), IniFbuffer(), and PF_BroadcastCBuf().

3.13.2.8 numdum

```
WORD* CbUf::numdum
```

points into rhs memory behind NumTerms

Definition at line 978 of file structs.h.

Referenced by AddRHS(), inicbufs(), IniFbuffer(), and PF_BroadcastCBuf().

3.13.2.9 dimension

```
WORD* CbUf::dimension
```

points into rhs memory behind numdum

Definition at line 979 of file structs.h.

Referenced by AddRHS(), inicbufs(), IniFbuffer(), and PF_BroadcastCBuf().

3.13.2.10 boomlijst

```
COMPTREE* CbUf::boomlijst
```

[D] Number elements in MaxTreeSize

Definition at line 980 of file structs.h.

Referenced by CleanupArgCache(), clearcbuf(), finishcbuf(), inicbufs(), IniFbuffer(), and PF_BroadcastCBuf().

3.13.2.11 BufferSize

```
LONG CbUf::BufferSize
```

Number of allocated WORD's in Buffer

Definition at line 981 of file structs.h.

Referenced by DoubleCbuffer(), finishcbuf(), inicbufs(), and PF_BroadcastCBuf().

3.13.2.12 numlhs

```
int CbUf::numlhs
```

Definition at line 982 of file structs.h.

3.13.2.13 numrhs

```
int CbUf::numrhs
```

Definition at line 983 of file structs.h.

3.13.2.14 maxlhs

```
int CbUf::maxlhs
```

Definition at line 984 of file structs.h.

3.13.2.15 maxrhs

```
int CbUf::maxrhs
```

Definition at line 985 of file structs.h.

3.13.2.16 mnumlhs

```
int CbUf::mnumlhs
```

Definition at line 986 of file structs.h.

3.13.2.17 mnumrhs

```
int CbUf::mnumrhs
```

Definition at line 987 of file structs.h.

3.13.2.18 numtree

```
int CbUf::numtree
```

Definition at line 988 of file structs.h.

3.13.2.19 rootnum

```
int CbUf::rootnum
```

Definition at line 989 of file structs.h.

3.13.2.20 MaxTreeSize

```
int CbUf::MaxTreeSize
```

Definition at line 990 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.14 ChAnNeL Struct Reference

```
#include <structs.h>
```

Data Fields

- char * name
- int handle

3.14.1 Detailed Description

When we read input from text files we have to remember not only their handle but also their name. This is needed for error messages. Hence we call such a file a channel and reserve a struct of type CHANNEL to allow to lay this link.

Definition at line 1000 of file structs.h.

3.14.2 Field Documentation

3.14.2.1 name

```
char* ChAnNeL::name
```

[D] Name of the channel

Definition at line 1001 of file structs.h.

3.14.2.2 handle

```
int ChAnNeL::handle
```

File handle

Definition at line 1002 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.15 COST Struct Reference

Data Fields

- LONG add
- LONG mul
- LONG div
- LONG pow

3.15.1 Detailed Description

Definition at line 1271 of file structs.h.

3.15.2 Field Documentation

3.15.2.1 add

```
LONG COST::add
```

Definition at line 1272 of file structs.h.

3.15.2.2 mul

```
LONG COST::mul
```

Definition at line 1273 of file structs.h.

3.15.2.3 div

```
LONG COST::div
```

Definition at line 1274 of file structs.h.

3.15.2.4 pow

```
LONG COST::pow
```

Definition at line 1275 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.16 CSEEq Struct Reference

Public Member Functions

- bool operator() (const vector< WORD > &lhs, const vector< WORD > &rhs) const

3.16.1 Detailed Description

Definition at line 1059 of file optimize.cc.

3.16.2 Member Function Documentation

3.16.2.1 operator()()

```
bool CSEEq::operator() (
            const vector< WORD > & lhs,
            const vector< WORD > & rhs ) const  [inline]
```

Definition at line 1060 of file optimize.cc.

The documentation for this struct was generated from the following file:

- optimize.cc

3.17 CSEHash Struct Reference

Public Member Functions

- size_t operator() (const vector< WORD > &n) const

3.17.1 Detailed Description

Definition at line 1053 of file optimize.cc.

3.17.2 Member Function Documentation

3.17.2.1 operator()()

```
size_t CSEHash::operator() (
            const vector< WORD > & n ) const  [inline]
```

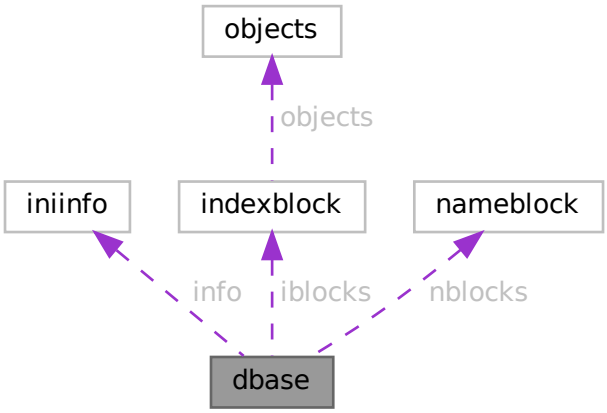
Definition at line 1054 of file optimize.cc.

The documentation for this struct was generated from the following file:

- optimize.cc

3.18 dbase Struct Reference

Collaboration diagram for dbase:



Data Fields

- INIINFO info
- MLONG mode
- MLONG tablenamessize
- MLONG topnumber
- MLONG tablenamefill
- MLONG rwmode
- INDEXBLOCK ** iblocks
- NAMESBLOCK ** nblocks
- FILE * handle
- char * name
- char * fullname
- char * tablenames

3.18.1 Detailed Description

Definition at line 123 of file minos.h.

3.18.2 Field Documentation

3.18.2.1 info

```
INIINFO dbase::info
```

Definition at line 124 of file minos.h.

3.18.2.2 mode

```
MLONG dbase::mode
```

Definition at line 125 of file minos.h.

3.18.2.3 tablenamessize

```
MLONG dbase::tablenamessize
```

Definition at line 126 of file minos.h.

3.18.2.4 topnumber

```
MLONG dbase::topnumber
```

Definition at line 127 of file minos.h.

3.18.2.5 tablenamefill

```
MLONG dbase::tablenamefill
```

Definition at line 128 of file minos.h.

3.18.2.6 rwmode

```
MLONG dbase::rwmode
```

Definition at line 129 of file minos.h.

3.18.2.7 iblocks

```
INDEXBLOCK** dbase::iblocks
```

Definition at line 130 of file minos.h.

3.18.2.8 nblocks

```
NAMESBLOCK** dbase::nblocks
```

Definition at line 131 of file minos.h.

3.18.2.9 handle

```
FILE* dbase::handle
```

Definition at line 132 of file minos.h.

3.18.2.10 name

```
char* dbase::name
```

Definition at line 133 of file minos.h.

3.18.2.11 fullname

```
char* dbase::fullname
```

Definition at line 134 of file minos.h.

3.18.2.12 tablenames

```
char* dbase::tablenames
```

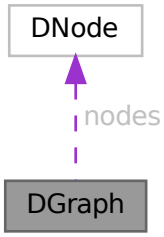
Definition at line 135 of file minos.h.

The documentation for this struct was generated from the following file:

- minos.h

3.19 DGraph Struct Reference

Collaboration diagram for DGraph:



Data Fields

- int nnodes
- int nedges
- DNode * nodes
- DEdge * edges

3.19.1 Detailed Description

Definition at line 255 of file grccparam.h.

3.19.2 Field Documentation

3.19.2.1 nnodes

```
int DGraph::nnodes
```

Definition at line 256 of file grccparam.h.

3.19.2.2 nedges

```
int DGraph::nedges
```

Definition at line 257 of file grccparam.h.

3.19.2.3 nodes

```
DNode* DGraph::nodes
```

Definition at line 258 of file grccparam.h.

3.19.2.4 edges

```
DEdge* DGraph::edges
```

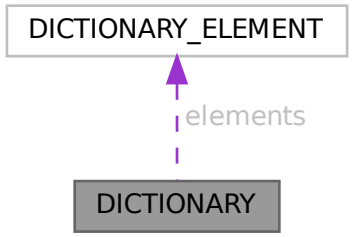
Definition at line 259 of file grccparam.h.

The documentation for this struct was generated from the following file:

- grccparam.h

3.20 DICTIONARY Struct Reference

Collaboration diagram for DICTIONARY:



Data Fields

- DICTIONARY_ELEMENT ** elements
- UBYTE * name
- int sizeelements
- int numelements
- int numbers
- int variables
- int characters
- int funwith
- int gnumelements
- int ranges

3.20.1 Detailed Description

Definition at line 1336 of file structs.h.

3.20.2 Field Documentation

3.20.2.1 elements

```
DICTIONARY_ELEMENT** DICTIONARY::elements
```

Definition at line 1337 of file structs.h.

3.20.2.2 name

```
UBYTE* DICTIONARY::name
```

Definition at line 1338 of file structs.h.

3.20.2.3 sizeelements

```
int DICTIONARY::sizeelements
```

Definition at line 1339 of file structs.h.

3.20.2.4 numelements

```
int DICTIONARY::numelements
```

Definition at line 1340 of file structs.h.

3.20.2.5 numbers

```
int DICTIONARY::numbers
```

Definition at line 1341 of file structs.h.

3.20.2.6 variables

```
int DICTIONARY::variables
```

Definition at line 1342 of file structs.h.

3.20.2.7 characters

```
int DICTIONARY::characters
```

Definition at line 1343 of file structs.h.

3.20.2.8 funwith

```
int DICTIONARY::funwith
```

Definition at line 1344 of file structs.h.

3.20.2.9 gnumelements

```
int DICTIONARY::gnumelements
```

Definition at line 1345 of file structs.h.

3.20.2.10 ranges

```
int DICTIONARY::ranges
```

Definition at line 1346 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.21 DICTIONARY_ELEMENT Struct Reference

Data Fields

- WORD * lhs
- WORD * rhs
- int type
- int size

3.21.1 Detailed Description

Definition at line 1329 of file structs.h.

3.21.2 Field Documentation

3.21.2.1 lhs

```
WORD* DICTIONARY_ELEMENT::lhs
```

Definition at line 1330 of file structs.h.

3.21.2.2 rhs

```
WORD* DICTIONARY_ELEMENT::rhs
```

Definition at line 1331 of file structs.h.

3.21.2.3 type

```
int DICTIONARY_ELEMENT::type
```

Definition at line 1332 of file structs.h.

3.21.2.4 size

```
int DICTIONARY_ELEMENT::size
```

Definition at line 1333 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.22 DiStRiBuTe Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD * obj1
- WORD * obj2
- WORD * out
- WORD sign
- WORD n1
- WORD n2
- WORD n
- WORD cycle [MAXMATCH]

3.22.1 Detailed Description

The struct DISTRIBUTE is used to help the pattern matcher when matching antisymmetric tensors.

Definition at line 1079 of file structs.h.

3.22.2 Field Documentation

3.22.2.1 obj1

```
WORD* DiStRiBuTe::obj1
```

Definition at line 1080 of file structs.h.

3.22.2.2 obj2

```
WORD* DiStRiBuTe::obj2
```

Definition at line 1081 of file structs.h.

3.22.2.3 out

```
WORD* DiStRiBuTe::out
```

Definition at line 1082 of file structs.h.

3.22.2.4 sign

```
WORD DiStRiBuTe::sign
```

Definition at line 1083 of file structs.h.

3.22.2.5 n1

```
WORD DiStRiBuTe::n1
```

Definition at line 1084 of file structs.h.

3.22.2.6 n2

```
WORD DiStRiBuTe::n2
```

Definition at line 1085 of file structs.h.

3.22.2.7 n

```
WORD DiStRiBuTe::n
```

Definition at line 1086 of file structs.h.

3.22.2.8 cycle

```
WORD DiStRiBuTe::cycle[MAXMATCH]
```

Definition at line 1087 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.23 DNode Struct Reference

Data Fields

- int extloop
- int intrct

3.23.1 Detailed Description

Definition at line 248 of file grccparam.h.

3.23.2 Field Documentation

3.23.2.1 extloop

```
int DNode::extloop
```

Definition at line 249 of file grccparam.h.

3.23.2.2 intrct

```
int DNode::intrct
```

Definition at line 250 of file grccparam.h.

The documentation for this struct was generated from the following file:

- grccparam.h

3.24 dollar_buf Struct Reference

3.24.1 Detailed Description

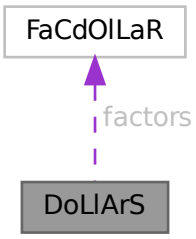
Definition at line 2483 of file parallel.c.

The documentation for this struct was generated from the following file:

- parallel.c

3.25 DoLlArS Struct Reference

Collaboration diagram for DoLlArS:



Data Fields

- WORD * where
- FACDOLLAR * factors
- LONG size
- LONG name
- WORD type
- WORD node
- WORD index
- WORD zero
- WORD numdummies
- WORD nfactors

3.25.1 Detailed Description

Definition at line 541 of file structs.h.

3.25.2 Field Documentation

3.25.2.1 where

```
WORD* DoLlArS::where
```

Definition at line 542 of file structs.h.

3.25.2.2 factors

```
FACDOLLAR* DoLlArS::factors
```

Definition at line 543 of file structs.h.

3.25.2.3 size

```
LONG DoLlArS::size
```

Definition at line 547 of file structs.h.

3.25.2.4 name

```
LONG DoLlArS::name
```

Definition at line 548 of file structs.h.

3.25.2.5 type

```
WORD DoLlArS::type
```

Definition at line 549 of file structs.h.

3.25.2.6 node

```
WORD DoLlArS::node
```

Definition at line 550 of file structs.h.

3.25.2.7 index

```
WORD DoLlArS::index
```

Definition at line 551 of file structs.h.

3.25.2.8 zero

```
WORD DoLlArS::zero
```

Definition at line 552 of file structs.h.

3.25.2.9 numdummies

```
WORD DoLlArS::numdummies
```

Definition at line 553 of file structs.h.

3.25.2.10 nfactors

```
WORD DoLlArS::nfactors
```

Definition at line 554 of file structs.h.

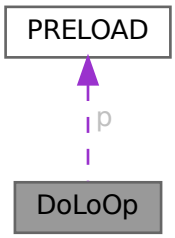
The documentation for this struct was generated from the following file:

- structs.h

3.26 DoLoOp Struct Reference

```
#include <structs.h>
```

Collaboration diagram for DoLoOp:



Data Fields

- PRELOAD p
- UBYTE * name
- UBYTE * vars
- UBYTE * contents
- UBYTE * dollarname
- LONG startlinenumber
- LONG firstnum
- LONG lastnum
- LONG incnum
- int type
- int NoShowInput
- int errorsinloop
- int firstloopcall
- WORD firstdollar
- WORD lastdollar
- WORD incdollar
- WORD NumPreTypes
- WORD PreIfLevel
- WORD PreSwitchLevel

3.26.1 Detailed Description

Each preprocessor do loop has a struct of type DOLOOP to keep track of all relevant parameters like where the beginning of the loop is, what the boundaries, increment and value of the loop parameter are, etc. Also we keep the whole loop inside a buffer of type PRELOAD

Definition at line 876 of file structs.h.

3.26.2 Field Documentation

3.26.2.1 p

```
PRELOAD DoLoOp::p
```

size, name and buffer

Definition at line 877 of file structs.h.

3.26.2.2 name

```
UBYTE* DoLoOp::name
```

pointer into PRELOAD buffer

Definition at line 878 of file structs.h.

3.26.2.3 vars

```
UBYTE* DoLoOp::vars
```

Definition at line 879 of file structs.h.

3.26.2.4 contents

```
UBYTE* DoLoOp::contents
```

Definition at line 880 of file structs.h.

3.26.2.5 dollarname

```
UBYTE* DoLoOp::dollarname
```

For loop over terms in expression. Allocated with Malloc1()

Definition at line 881 of file structs.h.

3.26.2.6 startlinenumber

```
LONG DoLoOp::startlinenumber
```

Definition at line 882 of file structs.h.

3.26.2.7 firstnum

```
LONG DoLoOp::firstnum
```

Definition at line 883 of file structs.h.

3.26.2.8 lastnum

```
LONG DoLoOp::lastnum
```

Definition at line 884 of file structs.h.

3.26.2.9 incnum

```
LONG DoLoOp::incnum
```

Definition at line 885 of file structs.h.

3.26.2.10 type

```
int DoLoOp::type
```

Definition at line 886 of file structs.h.

3.26.2.11 NoShowInput

```
int DoLoOp::NoShowInput
```

Definition at line 887 of file structs.h.

3.26.2.12 errorsinloop

```
int DoLoOp::errorsinloop
```

Definition at line 888 of file structs.h.

3.26.2.13 firstloopcall

```
int DoLoOp::firstloopcall
```

Definition at line 889 of file structs.h.

3.26.2.14 firstdollar

```
WORD DoLoOp::firstdollar
```

Definition at line 890 of file structs.h.

3.26.2.15 lastdollar

```
WORD DoLoOp::lastdollar
```

Definition at line 891 of file structs.h.

3.26.2.16 incdollar

```
WORD DoLoOp::incdollar
```

Definition at line 892 of file structs.h.

3.26.2.17 NumPreTypes

```
WORD DoLoOp::NumPreTypes
```

Definition at line 893 of file structs.h.

3.26.2.18 PrelfLevel

```
WORD DoLoOp::PreIfLevel
```

Definition at line 894 of file structs.h.

3.26.2.19 PreSwitchLevel

```
WORD DoLoOp::PreSwitchLevel
```

Definition at line 895 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.27 DuBiOuS Struct Reference

Data Fields

- LONG name
- WORD node
- WORD dummy

3.27.1 Detailed Description

Definition at line 528 of file structs.h.

3.27.2 Field Documentation

3.27.2.1 name

```
LONG DuBiOuS::name
```

Definition at line 529 of file structs.h.

3.27.2.2 node

```
WORD DuBiOuS::node
```

Definition at line 530 of file structs.h.

3.27.2.3 dummy

```
WORD DuBiOuS::dummy
```

Definition at line 531 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.28 ECand Class Reference

Public Member Functions

- ECand (int dt, int nplist, int ∗plst)
- void prECand (const char ∗msg)

Data Fields

- Bool det
- int nplist
- int plist [GRCC_MAXMPARTICLES2]

3.28.1 Detailed Description

Definition at line 1124 of file grcc.h.

3.28.2 Constructor & Destructor Documentation

3.28.2.1 ECand()

```
ECand::ECand (
            int dt,
            int nplist,
            int * plist )
```

Definition at line 8187 of file grcc.cc.

3.28.2.2 ~ECand()

```
ECand::~ECand (
            void  )
```

Definition at line 8207 of file grcc.cc.

3.28.3 Member Function Documentation

3.28.3.1 prECand()

```
void ECand::prECand (
            const char * msg )
```

Definition at line 8212 of file grcc.cc.

3.28.4 Field Documentation

3.28.4.1 det

```
Bool ECand::det
```

Definition at line 1128 of file grcc.h.

3.28.4.2 nplist

```
int ECand::nplist
```

Definition at line 1129 of file grcc.h.

3.28.4.3 plist

```
int ECand::plist[GRCC_MAXMPARTICLES2]
```

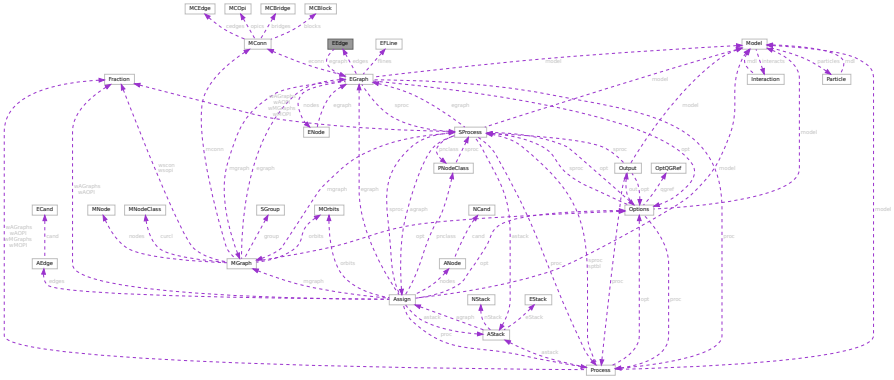
Definition at line 1130 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.29 EEdge Class Reference

Collaboration diagram for EEdge:



Public Member Functions

- EEdge (EGraph ∗egrph, int nedges, int nloops)
- void copy (EEdge ∗ee)
- void setId (EGraph ∗egrph, const int eid)
- void print (void)
- void setType (int typ)
- void setLMom (int k, int dir)
- void setEMom (int nedges, int ∗extn, int dir)

Data Fields

- EGraph ∗ egraph
- int id
- int ext
- int ptcl
- int deleted
- int nodes [2]
- int nlegs [2]
- int ∗ emom
- int ∗ lmom
- int ∗ extMom
- ULong momset
- int momdir
- Bool cut
- int visited
- int conid
- int edtype
- int opicomp
- int dir
- int DUMMYPADDING

3.29.1 Detailed Description

Definition at line 562 of file grcc.h.

3.29.2 Constructor & Destructor Documentation

3.29.2.1 EEdge() [1/2]

```
EEdge::EEdge (
            void  )
```

Definition at line 6144 of file grcc.cc.

3.29.2.2 EEdge() [2/2]

```
EEdge::EEdge (
            EGraph * egrph,
            int nedges,
            int nloops )
```

Definition at line 6166 of file grcc.cc.

3.29.2.3 ~EEdge()

```
EEdge::~EEdge (
            void  )
```

Definition at line 6185 of file grcc.cc.

3.29.3 Member Function Documentation

3.29.3.1 copy()

```
void EEdge::copy (
            EEdge * ee )
```

Definition at line 6196 of file grcc.cc.

3.29.3.2 setId()

```
void EEdge::setId (
            EGraph * egrph,
            const int eid )
```

Definition at line 6208 of file grcc.cc.

3.29.3.3 print()

```
void EEdge::print (
            void  )
```

Definition at line 6244 of file grcc.cc.

3.29.3.4 setType()

```
void EEdge::setType (
            int typ )
```

Definition at line 6215 of file grcc.cc.

3.29.3.5 setLMom()

```
void EEdge::setLMom (
            int k,
            int dir )
```

Definition at line 6238 of file grcc.cc.

3.29.3.6 setEMom()

```
void EEdge::setEMom (
            int nedges,
            int * extn,
            int dir )
```

Definition at line 6226 of file grcc.cc.

3.29.4 Field Documentation

3.29.4.1 egraph

```
EGraph* EEdge::egraph
```

Definition at line 565 of file grcc.h.

3.29.4.2 id

```
int EEdge::id
```

Definition at line 567 of file grcc.h.

3.29.4.3 ext

```
int EEdge::ext
```

Definition at line 568 of file grcc.h.

3.29.4.4 ptcl

```
int EEdge::ptcl
```

Definition at line 569 of file grcc.h.

3.29.4.5 deleted

```
int EEdge::deleted
```

Definition at line 570 of file grcc.h.

3.29.4.6 nodes

```
int EEdge::nodes[2]
```

Definition at line 571 of file grcc.h.

3.29.4.7 nlegs

```
int EEdge::nlegs[2]
```

Definition at line 572 of file grcc.h.

3.29.4.8 emom

```
int* EEdge::emom
```

Definition at line 575 of file grcc.h.

3.29.4.9 lmom

```
int* EEdge::lmom
```

Definition at line 576 of file grcc.h.

3.29.4.10 extMom

```
int* EEdge::extMom
```

Definition at line 577 of file grcc.h.

3.29.4.11 momset

```
ULong EEdge::momset
```

Definition at line 580 of file grcc.h.

3.29.4.12 momdir

```
int EEdge::momdir
```

Definition at line 581 of file grcc.h.

3.29.4.13 cut

```
Bool EEdge::cut
```

Definition at line 583 of file grcc.h.

3.29.4.14 visited

```
int EEdge::visited
```

Definition at line 584 of file grcc.h.

3.29.4.15 conid

```
int EEdge::conid
```

Definition at line 585 of file grcc.h.

3.29.4.16 edtype

```
int EEdge::edtype
```

Definition at line 586 of file grcc.h.

3.29.4.17 opicomp

```
int EEdge::opicomp
```

Definition at line 587 of file grcc.h.

3.29.4.18 dir

```
int EEdge::dir
```

Definition at line 588 of file grcc.h.

3.29.4.19 DUMMYPADDING

```
int EEdge::DUMMYPADDING
```

Definition at line 590 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.30 EFLine Class Reference

Public Member Functions

- void print (const char *msg)

Data Fields

- int elist [GRCC_MAXNODES]
- FLType ftype
- int fkind
- int nlist

3.30.1 Detailed Description

Definition at line 612 of file grcc.h.

3.30.2 Constructor & Destructor Documentation

3.30.2.1 EFLine()

```
EFLine::EFLine (
            void  )
```

Definition at line 6295 of file grcc.cc.

3.30.3 Member Function Documentation

3.30.3.1 print()

```
void EFLine::print (
            const char * msg )
```

Definition at line 6303 of file grcc.cc.

3.30.4 Field Documentation

3.30.4.1 elist

```
int EFLine::elist[GRCC_MAXNODES]
```

Definition at line 614 of file grcc.h.

3.30.4.2 ftype

```
FLType EFLine::ftype
```

Definition at line 615 of file grcc.h.

3.30.4.3 fkind

```
int EFLine::fkind
```

Definition at line 616 of file grcc.h.

3.30.4.4 nlist

```
int EFLine::nlist
```

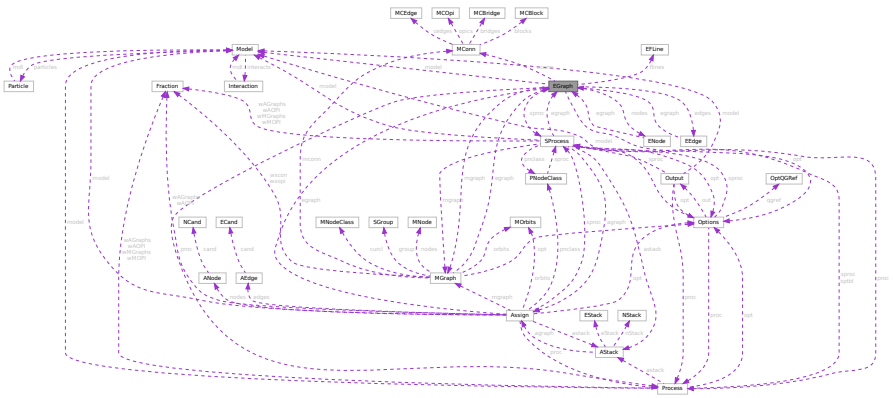
Definition at line 617 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.31 EGraph Class Reference

Collaboration diagram for EGraph:



Public Member Functions

- EGraph (int nnodes, int nedges, int mxdeg)
- void copy (EGraph *eg)
- void print (void)
- void printPy (FILE *fp, long mId)
- void fromDGraph (DGraph *dg)
- void fromMGraph (MGraph *mgraph)
- Bool optQGrafM (Options *opt)
- Bool optQGrafA (Options *opt)
- Bool isOptE (void)
- ENode * setExtern (int n0, int pt, int ndtp)
- Bool isExternal (int nd)
- Bool isFermion (int nd)
- void setExtLoop (int nd, int val)
- void endSetExtLoop (void)
- int connComp (void)
- int connVisit (int nd, int ncc)
- void biconnE (void)
- void biinitE (void)
- void bisearchE (int nd, int *extlst, int *intlst, int *opiext, int *opiloop)
- int findRoot (void)
- int dirEdge (int n, int e)
- void extMomConsv (void)
- int cmpMom (int *lm0, int *em0, int *lm1, int *em1)
- int groupLMom (int *grp, int *ed2gr)
- void chkMomConsv (void)
- void prFLines (void)
- void getFLines (void)
- int fltrace (int fk, int nd0, int *fl)
- void addFLine (const FLType ft, int fk, int nfl, int *fl)
- int legParticle (int ed, int lg)

Data Fields

- Options * opt
- Model * model
- Process * proc
- SProcess * sproc
- MGraph * mgraph
- MConn * econn
- ENode ** nodes
- EEdge ** edges
- BigInt mId
- BigInt aId
- BigInt sId
- BigInt gSubId
- Bool assigned
- int fsign
- BigInt nsym
- BigInt esym
- BigInt nsym1
- BigInt extperm
- BigInt multp

- int pId
- int sNodes
- int sEdges
- int sMaxdeg
- int sLoops
- int nNodes
- int nEdges
- int nExtern
- int maxdeg
- int nLoops
- int totalc
- int nopicomp
- int opi2plp
- int nopi2p
- int nadj2ptv
- int DUMMYPADDING
- int * bidef
- int * bilow
- int * extMom
- int bconn
- int bicount
- int loopm
- int opiCount
- EFLine * flines [GRCC_MAXFLINES]
- int nFlines
- int DUMMYPADDING1

3.31.1 Detailed Description

Definition at line 624 of file grcc.h.

3.31.2 Constructor & Destructor Documentation

3.31.2.1 EGraph()

```
EGraph::EGraph (
            int nnodes,
            int nedges,
            int mxdeg )
```

Definition at line 6328 of file grcc.cc.

3.31.2.2 ~EGraph()

```
EGraph::~EGraph (
            void  )
```

Definition at line 6388 of file grcc.cc.

3.31.3 Member Function Documentation

3.31.3.1 copy()

```
void EGraph::copy (
            EGraph * eg )
```

Definition at line 6419 of file grcc.cc.

3.31.3.2 print()

```
void EGraph::print (
            void  )
```

Definition at line 6818 of file grcc.cc.

3.31.3.3 printPy()

```
void EGraph::printPy (
            FILE * fp,
            long mId )
```

Definition at line 6884 of file grcc.cc.

3.31.3.4 fromDGraph()

```
void EGraph::fromDGraph (
            DGraph * dg )
```

Definition at line 6483 of file grcc.cc.

3.31.3.5 fromMGraph()

```
void EGraph::fromMGraph (
            MGraph * mgraph )
```

Definition at line 6589 of file grcc.cc.

3.31.3.6 optQGrafM()

```
Bool EGraph::optQGrafM (
            Options * opt )
```

Definition at line 7045 of file grcc.cc.

3.31.3.7 optQGrafA()

```
Bool EGraph::optQGrafA (
            Options * opt )
```

Definition at line 7264 of file grcc.cc.

3.31.3.8 isOptE()

```
Bool EGraph::isOptE (
            void  )
```

Definition at line 7289 of file grcc.cc.

3.31.3.9 setExtern()

```
ENode * EGraph::setExtern (
            int n0,
            int pt,
            int ndtp )
```

Definition at line 6771 of file grcc.cc.

3.31.3.10 isExternal()

```
Bool EGraph::isExternal (
            int nd )  [inline]
```

Definition at line 700 of file grcc.h.

3.31.3.11 isFermion()

```
int EGraph::isFermion (
            int nd )
```

Definition at line 7906 of file grcc.cc.

3.31.3.12 setExtLoop()

```
void EGraph::setExtLoop (
            int nd,
            int val )
```

Definition at line 6461 of file grcc.cc.

3.31.3.13 endSetExtLoop()

```
void EGraph::endSetExtLoop (
            void  )
```

Definition at line 6468 of file grcc.cc.

3.31.3.14 connComp()

```
int EGraph::connComp (
            void  )
```

Definition at line 7441 of file grcc.cc.

3.31.3.15 connVisit()

```
int EGraph::connVisit (
            int nd,
            int ncc )
```

Definition at line 7473 of file grcc.cc.

3.31.3.16 biconnE()

```
void EGraph::biconnE (
            void  )
```

Definition at line 7494 of file grcc.cc.

3.31.3.17 biinitE()

```
void EGraph::biinitE (
            void  )
```

Definition at line 7558 of file grcc.cc.

3.31.3.18 bisearchE()

```
void EGraph::bisearchE (
            int nd,
            int * extlst,
            int * intlst,
            int * opiext,
            int * opiloop )
```

Definition at line 7594 of file grcc.cc.

3.31.3.19 findRoot()

```
int EGraph::findRoot (
            void  )
```

Definition at line 7392 of file grcc.cc.

3.31.3.20 dirEdge()

```
int EGraph::dirEdge (
            int n,
            int e )
```

Definition at line 6952 of file grcc.cc.

3.31.3.21 extMomConsv()

```
void EGraph::extMomConsv (
            void  )
```

Definition at line 7799 of file grcc.cc.

3.31.3.22 cmpMom()

```
int EGraph::cmpMom (
            int * lm0,
            int * em0,
            int * lm1,
            int * em1 )
```

Definition at line 6962 of file grcc.cc.

3.31.3.23 groupLMom()

```
int EGraph::groupLMom (
            int * grp,
            int * ed2gr )
```

Definition at line 7012 of file grcc.cc.

3.31.3.24 chkMomConsv()

```
void EGraph::chkMomConsv (
            void  )
```

Definition at line 7827 of file grcc.cc.

3.31.3.25 prFLines()

```
void EGraph::prFLines (
            void  )
```

Definition at line 6939 of file grcc.cc.

3.31.3.26 getFLines()

```
void EGraph::getFLines (
            void  )
```

Definition at line 8023 of file grcc.cc.

3.31.3.27 fltrace()

```
int EGraph::fltrace (
            int fk,
            int nd0,
            int * fl )
```

Definition at line 7920 of file grcc.cc.

3.31.3.28 addFLine()

```
void EGraph::addFLine (
            const FLType ft,
            int fk,
            int nfl,
            int * fl )
```

Definition at line 8117 of file grcc.cc.

3.31.3.29 legParticle()

```
int EGraph::legParticle (
            int ed,
            int lg )
```

Definition at line 7898 of file grcc.cc.

3.31.4 Field Documentation

3.31.4.1 opt

```
Options* EGraph::opt
```

Definition at line 626 of file grcc.h.

3.31.4.2 model

```
Model* EGraph::model
```

Definition at line 627 of file grcc.h.

3.31.4.3 proc

```
Process* EGraph::proc
```

Definition at line 628 of file grcc.h.

3.31.4.4 sproc

```
SProcess* EGraph::sproc
```

Definition at line 629 of file grcc.h.

3.31.4.5 mgraph

```
MGraph* EGraph::mgraph
```

Definition at line 630 of file grcc.h.

3.31.4.6 econn

```
MConn* EGraph::econn
```

Definition at line 631 of file grcc.h.

3.31.4.7 nodes

```
ENode** EGraph::nodes
```

Definition at line 633 of file grcc.h.

3.31.4.8 edges

```
EEdge** EGraph::edges
```

Definition at line 634 of file grcc.h.

3.31.4.9 mId

```
BigInt EGraph::mId
```

Definition at line 636 of file grcc.h.

3.31.4.10 aId

```
BigInt EGraph::aId
```

Definition at line 637 of file grcc.h.

3.31.4.11 sId

```
BigInt EGraph::sId
```

Definition at line 638 of file grcc.h.

3.31.4.12 gSubId

```
BigInt EGraph::gSubId
```

Definition at line 639 of file grcc.h.

3.31.4.13 assigned

```
Bool EGraph::assigned
```

Definition at line 640 of file grcc.h.

3.31.4.14 fsign

```
int EGraph::fsign
```

Definition at line 642 of file grcc.h.

3.31.4.15 nsym

```
BigInt EGraph::nsym
```

Definition at line 643 of file grcc.h.

3.31.4.16 esym

```
BigInt EGraph::esym
```

Definition at line 643 of file grcc.h.

3.31.4.17 nsym1

```
BigInt EGraph::nsym1
```

Definition at line 644 of file grcc.h.

3.31.4.18 extperm

```
BigInt EGraph::extperm
```

Definition at line 645 of file grcc.h.

3.31.4.19 multp

```
BigInt EGraph::multp
```

Definition at line 646 of file grcc.h.

3.31.4.20 pId

```
int EGraph::pId
```

Definition at line 648 of file grcc.h.

3.31.4.21 sNodes

```
int EGraph::sNodes
```

Definition at line 650 of file grcc.h.

3.31.4.22 sEdges

```
int EGraph::sEdges
```

Definition at line 651 of file grcc.h.

3.31.4.23 sMaxdeg

```
int EGraph::sMaxdeg
```

Definition at line 652 of file grcc.h.

3.31.4.24 sLoops

```
int EGraph::sLoops
```

Definition at line 653 of file grcc.h.

3.31.4.25 nNodes

```
int EGraph::nNodes
```

Definition at line 655 of file grcc.h.

3.31.4.26 nEdges

```
int EGraph::nEdges
```

Definition at line 656 of file grcc.h.

3.31.4.27 nExtern

```
int EGraph::nExtern
```

Definition at line 657 of file grcc.h.

3.31.4.28 maxdeg

```
int EGraph::maxdeg
```

Definition at line 658 of file grcc.h.

3.31.4.29 nLoops

```
int EGraph::nLoops
```

Definition at line 659 of file grcc.h.

3.31.4.30 totalc

```
int EGraph::totalc
```

Definition at line 660 of file grcc.h.

3.31.4.31 nopicomp

```
int EGraph::nopicomp
```

Definition at line 663 of file grcc.h.

3.31.4.32 opi2plp

```
int EGraph::opi2plp
```

Definition at line 664 of file grcc.h.

3.31.4.33 nopi2p

```
int EGraph::nopi2p
```

Definition at line 665 of file grcc.h.

3.31.4.34 nadj2ptv

```
int EGraph::nadj2ptv
```

Definition at line 666 of file grcc.h.

3.31.4.35 DUMMYPADDING

```
int EGraph::DUMMYPADDING
```

Definition at line 668 of file grcc.h.

3.31.4.36 bidef

```
int* EGraph::bidef
```

Definition at line 670 of file grcc.h.

3.31.4.37 bilow

```
int* EGraph::bilow
```

Definition at line 671 of file grcc.h.

3.31.4.38 extMom

```
int* EGraph::extMom
```

Definition at line 672 of file grcc.h.

3.31.4.39 bconn

```
int EGraph::bconn
```

Definition at line 673 of file grcc.h.

3.31.4.40 bicount

```
int EGraph::bicount
```

Definition at line 674 of file grcc.h.

3.31.4.41 loopm

```
int EGraph::loopm
```

Definition at line 675 of file grcc.h.

3.31.4.42 opiCount

```
int EGraph::opiCount
```

Definition at line 676 of file grcc.h.

3.31.4.43 flines

```
EFLine* EGraph::flines[GRCC_MAXFLINES]
```

Definition at line 679 of file grcc.h.

3.31.4.44 nFlines

```
int EGraph::nFlines
```

Definition at line 680 of file grcc.h.

3.31.4.45 DUMMYPADDING1

```
int EGraph::DUMMYPADDING1
```

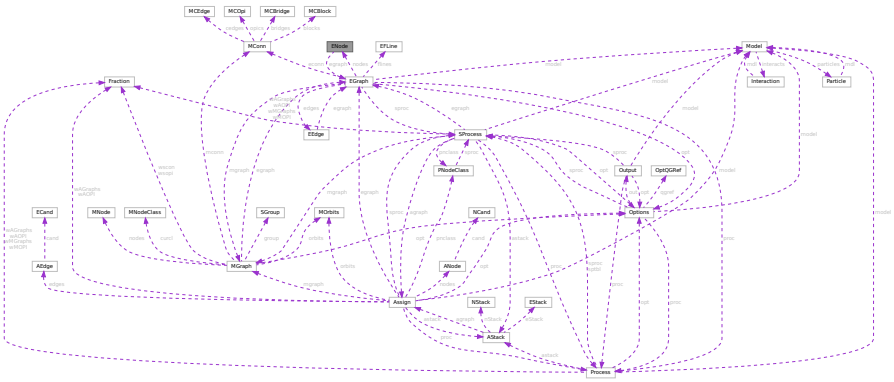
Definition at line 682 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.32 ENode Class Reference

Collaboration diagram for ENode:



Public Member Functions

- ENode (EGraph ∗egrph, int loops, int sdeg)
- void initAss (EGraph ∗egrph, int nid, int sdg)
- void setId (EGraph ∗egrph, const int nid)
- void copy (ENode ∗en)
- void setExtern (int typ, int pt)
- void setType (int typ)
- void print (void)

Data Fields

- EGraph ∗ egraph
- int ∗ edges
- int id
- int maxdeg
- int deg
- int extloop
- int ndtype
- int intrct
- int ∗ klow
- int visited
- int DUMMYPADDING

3.32.1 Detailed Description

Definition at line 528 of file grcc.h.

3.32.2 Constructor & Destructor Documentation

3.32.2.1 ENode() [1/2]

```
ENode::ENode (
            void  )
```

Definition at line 6024 of file grcc.cc.

3.32.2.2 ENode() [2/2]

```
ENode::ENode (
            EGraph * egrph,
            int loops,
            int sdeg )
```

Definition at line 6040 of file grcc.cc.

3.32.2.3 ~ENode()

```
ENode::~ENode (
            void  )
```

Definition at line 6060 of file grcc.cc.

3.32.3 Member Function Documentation

3.32.3.1 initAss()

```
void ENode::initAss (
            EGraph * egrph,
            int nid,
            int sdg )
```

Definition at line 6084 of file grcc.cc.

3.32.3.2 setId()

```
void ENode::setId (
            EGraph * egrph,
            const int nid )
```

Definition at line 6092 of file grcc.cc.

3.32.3.3 copy()

```
void ENode::copy (
            ENode * en )
```

Definition at line 6071 of file grcc.cc.

3.32.3.4 setExtern()

```
void ENode::setExtern (
            int typ,
            int pt )
```

Definition at line 6099 of file grcc.cc.

3.32.3.5 setType()

```
void ENode::setType (
            int typ )
```

Definition at line 6112 of file grcc.cc.

3.32.3.6 print()

```
void ENode::print (
            void  )
```

Definition at line 6125 of file grcc.cc.

3.32.4 Field Documentation

3.32.4.1 egraph

```
EGraph* ENode::egraph
```

Definition at line 530 of file grcc.h.

3.32.4.2 edges

```
int* ENode::edges
```

Definition at line 531 of file grcc.h.

3.32.4.3 id

```
int ENode::id
```

Definition at line 534 of file grcc.h.

3.32.4.4 maxdeg

```
int ENode::maxdeg
```

Definition at line 535 of file grcc.h.

3.32.4.5 deg

```
int ENode::deg
```

Definition at line 536 of file grcc.h.

3.32.4.6 extloop

```
int ENode::extloop
```

Definition at line 537 of file grcc.h.

3.32.4.7 ndtype

```
int ENode::ndtype
```

Definition at line 538 of file grcc.h.

3.32.4.8 intrct

```
int ENode::intrct
```

Definition at line 539 of file grcc.h.

3.32.4.9 klow

```
int* ENode::klow
```

Definition at line 542 of file grcc.h.

3.32.4.10 visited

```
int ENode::visited
```

Definition at line 543 of file grcc.h.

3.32.4.11 DUMMYPADDING

```
int ENode::DUMMYPADDING
```

Definition at line 545 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.33 EStack Class Reference

Public Member Functions

- void print (const char *msg)

Data Fields

- int edgen
- int det
- int nplist
- int plist [GRCC_MAXMPARTICLES2]

3.33.1 Detailed Description

Definition at line 1350 of file grcc.h.

3.33.2 Constructor & Destructor Documentation

3.33.2.1 EStack()

```
EStack::EStack ( )  [inline]
```

Definition at line 1359 of file grcc.h.

3.33.2.2 ~EStack()

```
EStack::~EStack ( )  [inline]
```

Definition at line 1360 of file grcc.h.

3.33.3 Member Function Documentation

3.33.3.1 print()

```
void EStack::print (
            const char * msg )
```

Definition at line 10233 of file grcc.cc.

3.33.4 Field Documentation

3.33.4.1 edgen

```
int EStack::edgen
```

Definition at line 1354 of file grcc.h.

3.33.4.2 det

```
int EStack::det
```

Definition at line 1355 of file grcc.h.

3.33.4.3 nplist

```
int EStack::nplist
```

Definition at line 1356 of file grcc.h.

3.33.4.4 plist

```
int EStack::plist[GRCC_MAXMPARTICLES2]
```

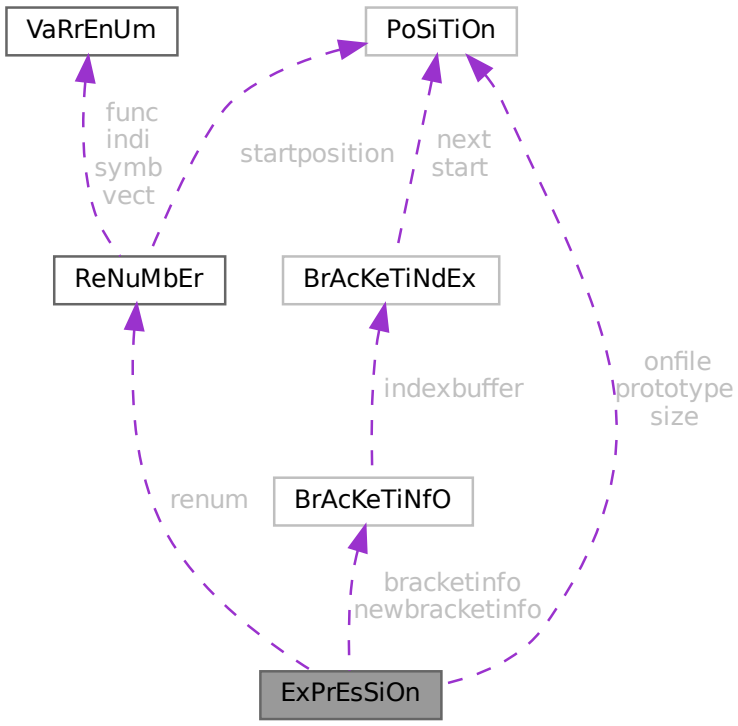
Definition at line 1357 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.34 ExPrEsSiOn Struct Reference

Collaboration diagram for ExPrEsSiOn:



Data Fields

- POSITION onfile
- POSITION prototype
- POSITION size
- RENUMBER renum
- BRACKETINFO * bracketinfo

- BRACKETINFO * newbracketinfo
- WORD * renumlists
- WORD * inmem
- LONG counter
- LONG name
- WORD hidelevel
- WORD vflags
- WORD printflag
- WORD status
- WORD replace
- WORD node
- WORD whichbuffer
- WORD namesize
- WORD compression
- WORD numdummies
- WORD numfactors
- WORD sizeprototype
- WORD uflags

3.34.1 Detailed Description

Definition at line 382 of file structs.h.

3.34.2 Field Documentation

3.34.2.1 onfile

```
POSITION ExPrEsSiOn::onfile
```

Definition at line 383 of file structs.h.

3.34.2.2 prototype

```
POSITION ExPrEsSiOn::prototype
```

Definition at line 384 of file structs.h.

3.34.2.3 size

```
POSITION ExPrEsSiOn::size
```

Definition at line 385 of file structs.h.

3.34.2.4 renum

```
RENUMBER ExPrEsSiOn::renum
```

Definition at line 386 of file structs.h.

3.34.2.5 bracketinfo

```
BRACKETINFO* ExPrEsSiOn::bracketinfo
```

Definition at line 387 of file structs.h.

3.34.2.6 newbracketinfo

```
BRACKETINFO* ExPrEsSiOn::newbracketinfo
```

Definition at line 388 of file structs.h.

3.34.2.7 renumlists

```
WORD* ExPrEsSiOn::renumlists
```

Allocated only for threaded version if variables exist, else points to AN.dummyrenumlist

Definition at line 389 of file structs.h.

Referenced by DoRecovery().

3.34.2.8 inmem

```
WORD* ExPrEsSiOn::inmem
```

Definition at line 391 of file structs.h.

3.34.2.9 counter

```
LONG ExPrEsSiOn::counter
```

Definition at line 392 of file structs.h.

3.34.2.10 name

```
LONG ExPrEsSiOn::name
```

Definition at line 393 of file structs.h.

3.34.2.11 hidelevel

```
WORD ExPrEsSiOn::hidelevel
```

Definition at line 394 of file structs.h.

3.34.2.12 vflags

```
WORD ExPrEsSiOn::vflags
```

Definition at line 395 of file structs.h.

3.34.2.13 printflag

```
WORD ExPrEsSiOn::printflag
```

Definition at line 396 of file structs.h.

3.34.2.14 status

```
WORD ExPrEsSiOn::status
```

Definition at line 397 of file structs.h.

3.34.2.15 replace

```
WORD ExPrEsSiOn::replace
```

Definition at line 398 of file structs.h.

3.34.2.16 node

```
WORD ExPrEsSiOn::node
```

Definition at line 399 of file structs.h.

3.34.2.17 whichbuffer

```
WORD ExPrEsSiOn::whichbuffer
```

Definition at line 400 of file structs.h.

3.34.2.18 namesize

```
WORD ExPrEsSiOn::namesize
```

Definition at line 401 of file structs.h.

3.34.2.19 compression

```
WORD ExPrEsSiOn::compression
```

Definition at line 402 of file structs.h.

3.34.2.20 numdummies

```
WORD ExPrEsSiOn::numdummies
```

Definition at line 403 of file structs.h.

3.34.2.21 numfactors

```
WORD ExPrEsSiOn::numfactors
```

Definition at line 404 of file structs.h.

3.34.2.22 sizeprototype

```
WORD ExPrEsSiOn::sizeprototype
```

Definition at line 405 of file structs.h.

3.34.2.23 uflags

```
WORD ExPrEsSiOn::uflags
```

Definition at line 406 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.35 FaCdOlLaR Struct Reference

Data Fields

- WORD * where
- LONG size
- WORD type
- WORD value

3.35.1 Detailed Description

Definition at line 534 of file structs.h.

3.35.2 Field Documentation

3.35.2.1 where

```
WORD* FaCdOlLaR::where
```

Definition at line 535 of file structs.h.

3.35.2.2 size

```
LONG FaCdOlLaR::size
```

Definition at line 536 of file structs.h.

3.35.2.3 type

```
WORD FaCdOlLaR::type
```

Definition at line 537 of file structs.h.

3.35.2.4 value

```
WORD FaCdOlLaR::value
```

Definition at line 538 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.36 factorized_poly Class Reference

Public Member Functions

- void add_factor (const poly &f, int p=1)
- const std::string tostring () const

Data Fields

- std::vector< poly > factor
- std::vector< int > power

Friends

- std::ostream & **operator**<< (std::ostream &out, const poly &p)

3.36.1 Detailed Description

Definition at line 54 of file polyfact.h.

3.36.2 Member Function Documentation

3.36.2.1 add_factor()

```
void factorized_poly::add_factor (
            const poly & f,
            int p = 1 )
```

Definition at line 125 of file polyfact.cc.

3.36.2.2 tostring()

```
const string factorized_poly::tostring ( ) const
```

Definition at line 59 of file polyfact.cc.

3.36.3 Field Documentation

3.36.3.1 factor

```
std::vector<poly> factorized_poly::factor
```

Definition at line 59 of file polyfact.h.

3.36.3.2 power

```
std::vector<int> factorized_poly::power
```

Definition at line 60 of file polyfact.h.

The documentation for this class was generated from the following files:

- polyfact.h
- polyfact.cc

3.37 FGInput Struct Reference

Data Fields

- long ninitl
- const char * initln [GRCC_MAXNODES]
- int initlc [GRCC_MAXNODES]
- long nfinal
- const char * finaln [GRCC_MAXNODES]
- int finalc [GRCC_MAXNODES]
- int coupl [GRCC_MAXNCPLG]

3.37.1 Detailed Description

Definition at line 221 of file grccparam.h.

3.37.2 Field Documentation

3.37.2.1 ninitl

```
long FGInput::ninitl
```

Definition at line 222 of file grccparam.h.

3.37.2.2 initln

```
const char* FGInput::initln[GRCC_MAXNODES]
```

Definition at line 223 of file grccparam.h.

3.37.2.3 initlc

```
int FGInput::initlc[GRCC_MAXNODES]
```

Definition at line 224 of file grccparam.h.

3.37.2.4 nfinal

```
long FGInput::nfinal
```

Definition at line 225 of file grccparam.h.

3.37.2.5 finaln

```
const char* FGInput::finaln[GRCC_MAXNODES]
```

Definition at line 226 of file grccparam.h.

3.37.2.6 finalc

```
int FGInput::finalc[GRCC_MAXNODES]
```

Definition at line 227 of file grccparam.h.

3.37.2.7 coupl

```
int FGInput::coupl[GRCC_MAXNCPLG]
```

Definition at line 228 of file grccparam.h.

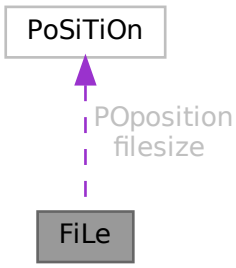
The documentation for this struct was generated from the following file:

- grccparam.h

3.38 FiLe Struct Reference

```
#include <structs.h>
```

Collaboration diagram for FiLe:



Data Fields

- POSITION POposition
- POSITION filesize
- WORD * PObuffer
- WORD * POstop
- WORD * POfill
- WORD * POfull
- char * name
- ULONG numblocks
- ULONG inbuffer
- LONG POsize
- int handle
- int active

3.38.1 Detailed Description

The type FILEHANDLE is the struct that controls all relevant information of a file, whether it is open or not. The file may even not yet exist. There is a system of caches (POBuffer) and as long as the information to be written still fits inside the cache the file may never be created. There are variables that can store information about different types of files, like scratch files or sort files. Depending on what is available in the system we may also have information about gzip compression (currently sort file only) or locks (TFORM).

Definition at line 681 of file structs.h.

3.38.2 Field Documentation

3.38.2.1 POposition

```
POSITION FiLe::POposition
```

Definition at line 682 of file structs.h.

3.38.2.2 filesize

```
POSITION FiLe::filesize
```

Definition at line 683 of file structs.h.

3.38.2.3 POBuffer

```
WORD* FiLe::POBuffer
```

Definition at line 684 of file structs.h.

3.38.2.4 POstop

```
WORD* FiLe::POstop
```

Definition at line 685 of file structs.h.

3.38.2.5 POfill

```
WORD* FiLe::POfill
```

Definition at line 686 of file structs.h.

3.38.2.6 POfull

```
WORD* FiLe::POfull
```

Definition at line 687 of file structs.h.

3.38.2.7 name

```
char* FiLe::name
```

Definition at line 694 of file structs.h.

3.38.2.8 numblocks

```
ULONG FiLe::numblocks
```

Definition at line 699 of file structs.h.

3.38.2.9 inbuffer

```
ULONG FiLe::inbuffer
```

Definition at line 700 of file structs.h.

3.38.2.10 POsize

```
LONG FiLe::POsize
```

Definition at line 701 of file structs.h.

3.38.2.11 handle

```
int FiLe::handle
```

Our own handle. Equal -1 if no file exists.

Definition at line 709 of file structs.h.

Referenced by DoOnePow(), FlushOut(), GetFirstTerm(), MergePatches(), poly_factorize_expression(), poly_unfactorize_expression(), PutIn(), PutOut(), Sflush(), and StageSort().

3.38.2.12 active

```
int FiLe::active
```

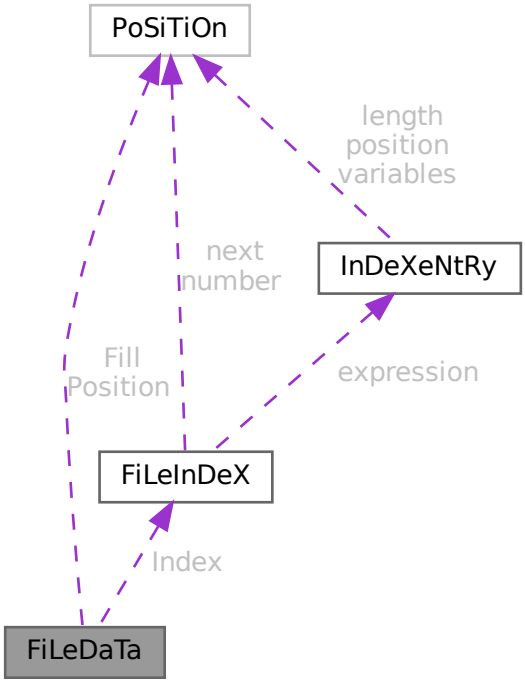
Definition at line 710 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.39 FiLeDaTa Struct Reference

Collaboration diagram for FiLeDaTa:



Data Fields

- FILEINDEX Index
- POSITION Fill
- POSITION Position
- WORD Handle
- WORD dirtyflag

3.39.1 Detailed Description

Definition at line 150 of file structs.h.

3.39.2 Field Documentation

3.39.2.1 Index

```
FILEINDEX FiLeDaTa::Index
```

Definition at line 151 of file structs.h.

3.39.2.2 Fill

```
POSITION FiLeDaTa::Fill
```

Definition at line 152 of file structs.h.

3.39.2.3 Position

```
POSITION FiLeDaTa::Position
```

Definition at line 153 of file structs.h.

3.39.2.4 Handle

```
WORD FiLeDaTa::Handle
```

Definition at line 154 of file structs.h.

3.39.2.5 dirtyflag

```
WORD FiLeDaTa::dirtyflag
```

Definition at line 155 of file structs.h.

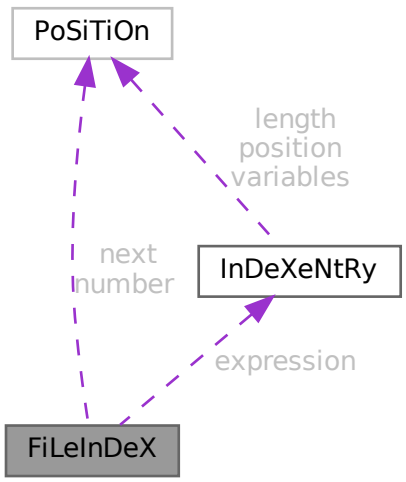
The documentation for this struct was generated from the following file:

- structs.h

3.40 FiLeInDeX Struct Reference

```
#include <structs.h>
```

Collaboration diagram for FiLeInDeX:



Data Fields

- POSITION next
- POSITION number
- INDEXENTRY expression [INFILEINDEX]
- SBYTE empty [EMPTYININDEX]

3.40.1 Detailed Description

Defines the structure of a file index in store-files and save-files.

It contains several entries (see struct InDeXeNtRy) up to a maximum of INFILEINDEX.

The variable number has been made of type POSITION to avoid padding problems with some types of computers/↩ OS and keep system independence of the .sav files.

This struct is always 512 bytes long.

Definition at line 137 of file structs.h.

3.40.2 Field Documentation

3.40.2.1 next

```
POSITION FiLeInDeX::next
```

Position of next FILEINDEX if any

Definition at line 138 of file structs.h.

Referenced by ReadSaveIndex().

3.40.2.2 number

```
POSITION FiLeInDeX::number
```

Number of used entries in this index

Definition at line 139 of file structs.h.

Referenced by ReadSaveIndex().

3.40.2.3 expression

```
INDEXENTRY FiLeInDeX::expression[INFILEINDEX]
```

File index entries

Definition at line 140 of file structs.h.

3.40.2.4 empty

```
SBYTE FiLeInDeX::empty[EMPTYININDEX]
```

Padding to 512 bytes

Definition at line 141 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.41 FixedGlobals Struct Reference

```
#include <structs.h>
```

Data Fields

- WCN Operation [8]
- WCN2 OperaFind [6]
- char * VarType [10]
- char * ExprStat [21]
- char * FunNam [2]
- char * swmes [3]
- char * fname
- char * fname2
- UBYTE * s_one
- WORD fnamebase
- WORD fname2base
- WORD fnamesize
- WORD fname2size
- UINT cTable [256]

3.41.1 Detailed Description

The FIXEDGLOBALS struct is an anachronism. It started as the struct with global variables that needed initialization. It contains the elements Operation and OperaFind which define a very early way of automatically jumping to the proper operation. We find the results of it in parts of the file opera.c Later operations were treated differently in a more transparent way. We never changed the existing code. The most important part is currently the cTable which is used intensively in the compiler.

Definition at line 2551 of file structs.h.

3.41.2 Field Documentation

3.41.2.1 Operation

```
WCN FixedGlobals::Operation[8]
```

Definition at line 2552 of file structs.h.

3.41.2.2 OperaFind

```
WCN2 FixedGlobals::OperaFind[6]
```

Definition at line 2553 of file structs.h.

3.41.2.3 VarType

```
char* FixedGlobals::VarType[10]
```

Definition at line 2554 of file structs.h.

3.41.2.4 ExprStat

```
char* FixedGlobals::ExprStat[21]
```

Definition at line 2555 of file structs.h.

3.41.2.5 FunNam

```
char* FixedGlobals::FunNam[2]
```

Definition at line 2556 of file structs.h.

3.41.2.6 swmes

```
char* FixedGlobals::swmes[3]
```

Definition at line 2557 of file structs.h.

3.41.2.7 fname

```
char* FixedGlobals::fname
```

Definition at line 2558 of file structs.h.

3.41.2.8 fname2

```
char* FixedGlobals::fname2
```

Definition at line 2559 of file structs.h.

3.41.2.9 s_one

```
UBYTE* FixedGlobals::s_one
```

Definition at line 2560 of file structs.h.

3.41.2.10 fnamebase

```
WORD FixedGlobals::fnamebase
```

Definition at line 2561 of file structs.h.

3.41.2.11 fname2base

```
WORD FixedGlobals::fname2base
```

Definition at line 2562 of file structs.h.

3.41.2.12 fnamesize

```
WORD FixedGlobals::fnamesize
```

Definition at line 2563 of file structs.h.

3.41.2.13 fname2size

```
WORD FixedGlobals::fname2size
```

Definition at line 2564 of file structs.h.

3.41.2.14 cTable

```
UINT FixedGlobals::cTable[256]
```

Definition at line 2565 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.42 fixedset Struct Reference

Data Fields

- char * name
- char * description
- int type
- int dimension

3.42.1 Detailed Description

Definition at line 573 of file structs.h.

3.42.2 Field Documentation

3.42.2.1 name

```
char* fixedset::name
```

Definition at line 574 of file structs.h.

3.42.2.2 description

```
char* fixedset::description
```

Definition at line 575 of file structs.h.

3.42.2.3 type

```
int fixedset::type
```

Definition at line 576 of file structs.h.

3.42.2.4 dimension

```
int fixedset::dimension
```

Definition at line 577 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.43 flint::fmpz Class Reference

Public Member Functions

- void print (const string &text)

Data Fields

- fmpz_t d

3.43.1 Detailed Description

Definition at line 81 of file flintinterface.h.

3.43.2 Constructor & Destructor Documentation

3.43.2.1 fmpz()

```
flint::fmpz::fmpz ( )  [inline]
```

Definition at line 84 of file flintinterface.h.

3.43.2.2 ~fmpz()

```
flint::fmpz::~fmpz ( )  [inline]
```

Definition at line 85 of file flintinterface.h.

3.43.3 Member Function Documentation

3.43.3.1 print()

```
void flint::fmpz::print (
            const string & text )  [inline]
```

Definition at line 86 of file flintinterface.h.

3.43.4 Field Documentation

3.43.4.1 d

```
fmpz_t flint::fmpz::d
```

Definition at line 83 of file flintinterface.h.

The documentation for this class was generated from the following file:

- flintinterface.h

3.44 Fraction Class Reference

Public Member Functions

- Fraction (BigInt n, BigInt d)
- void print (const char *msg)
- void setValue (BigInt n, BigInt d)
- void setValue (Fraction &f)
- void add (BigInt n, BigInt d)
- void add (Fraction f)
- void sub (Fraction f)
- BigInt gcd (BigInt n0, BigInt n1)
- void normal (void)
- Bool isEq (Fraction f)

Data Fields

- BigInt num
- BigInt den
- double ratio

3.44.1 Detailed Description

Definition at line 204 of file grcc.h.

3.44.2 Constructor & Destructor Documentation

3.44.2.1 Fraction() [1/2]

```
Fraction::Fraction ( )  [inline]
```

Definition at line 209 of file grcc.h.

3.44.2.2 Fraction() [2/2]

```
Fraction::Fraction (
            BigInt n,
            BigInt d )
```

Definition at line 10450 of file grcc.cc.

3.44.3 Member Function Documentation

3.44.3.1 print()

```
void Fraction::print (
            const char * msg )
```

Definition at line 10461 of file grcc.cc.

3.44.3.2 setValue() [1/2]

```
void Fraction::setValue (
            BigInt n,
            BigInt d )
```

Definition at line 10473 of file grcc.cc.

3.44.3.3 setValue() [2/2]

```
void Fraction::setValue (
            Fraction & f )
```

Definition at line 10481 of file grcc.cc.

3.44.3.4 add() [1/2]

```
void Fraction::add (
            BigInt n,
            BigInt d )
```

Definition at line 10526 of file grcc.cc.

3.44.3.5 add() [2/2]

```
void Fraction::add (
            Fraction f )
```

Definition at line 10549 of file grcc.cc.

3.44.3.6 sub()

```
void Fraction::sub (
            Fraction f )
```

Definition at line 10558 of file grcc.cc.

3.44.3.7 gcd()

```
BigInt Fraction::gcd (
            BigInt n0,
            BigInt n1 )
```

Definition at line 10489 of file grcc.cc.

3.44.3.8 normal()

```
void Fraction::normal (
            void  )
```

Definition at line 10509 of file grcc.cc.

3.44.3.9 isEq()

```
Bool Fraction::isEq (
            Fraction f )
```

Definition at line 10567 of file grcc.cc.

3.44.4 Field Documentation

3.44.4.1 num

```
BigInt Fraction::num
```

Definition at line 206 of file grcc.h.

3.44.4.2 den

```
BigInt Fraction::den
```

Definition at line 206 of file grcc.h.

3.44.4.3 ratio

```
double Fraction::ratio
```

Definition at line 207 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.45 FUN_INFO Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD * location
- int numargs
- int numfunnies
- int numwildcards
- int symmet
- int tensor
- int commute

3.45.1 Detailed Description

The struct FUN_INFO is used for information about functions in the file smart.c which is supposed to intelligently look for patterns in complicated wildcard situations involving symmetric functions.

Definition at line 608 of file structs.h.

3.45.2 Field Documentation

3.45.2.1 location

```
WORD* FUN_INFO::location
```

Definition at line 609 of file structs.h.

3.45.2.2 numargs

```
int FUN_INFO::numargs
```

Definition at line 610 of file structs.h.

3.45.2.3 numfunnies

```
int FUN_INFO::numfunnies
```

Definition at line 611 of file structs.h.

3.45.2.4 numwildcards

```
int FUN_INFO::numwildcards
```

Definition at line 612 of file structs.h.

3.45.2.5 symmet

```
int FUN_INFO::symmet
```

Definition at line 613 of file structs.h.

3.45.2.6 tensor

```
int FUN_INFO::tensor
```

Definition at line 614 of file structs.h.

3.45.2.7 commute

```
int FUN_INFO::commute
```

Definition at line 615 of file structs.h.

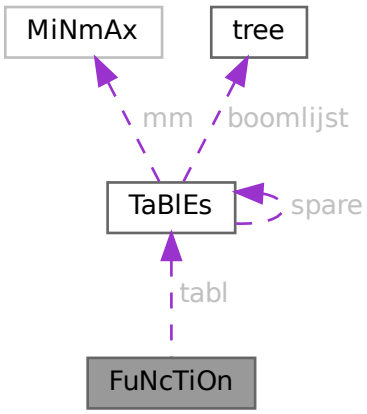
The documentation for this struct was generated from the following file:

- structs.h

3.46 FuNcTiOn Struct Reference

`#include <structs.h>`

Collaboration diagram for FuNcTiOn:



Data Fields

- TABLES tabl
- LONG symminfo
- LONG name
- WORD commute
- WORD complex
- WORD number
- WORD flags
- WORD spec
- WORD symmetric
- WORD node
- WORD namesize
- WORD dimension
- WORD maxnumargs
- WORD minnumargs

3.46.1 Detailed Description

Contains all information about a function. Also used for tables. It is used in the LIST elements of AC.

Definition at line 487 of file structs.h.

3.46.2 Field Documentation

3.46.2.1 tabl

```
TABLES FuNcTiOn::tabl
```

Used if redefined as table. != 0 if function is a table

Definition at line 488 of file structs.h.

3.46.2.2 symminfo

```
LONG FuNcTiOn::symminfo
```

Info regarding symm properties offset in buffer

Definition at line 489 of file structs.h.

3.46.2.3 name

```
LONG FuNcTiOn::name
```

Location in namebuffer of NAMETREE

Definition at line 490 of file structs.h.

Referenced by StartVariables().

3.46.2.4 commute

```
WORD FuNcTiOn::commute
```

Commutation properties

Definition at line 491 of file structs.h.

3.46.2.5 complex

```
WORD FuNcTiOn::complex
```

Properties under complex conjugation

Definition at line 492 of file structs.h.

3.46.2.6 number

`WORD FuNcTiOn::number`

Number when stored in file

Definition at line 493 of file structs.h.

Referenced by ReadSaveIndex().

3.46.2.7 flags

`WORD FuNcTiOn::flags`

Used to indicate usage when storing

Definition at line 494 of file structs.h.

3.46.2.8 spec

`WORD FuNcTiOn::spec`

Regular, Tensor, etc. See FunSpecs.

Definition at line 495 of file structs.h.

Referenced by ReadSaveVariables().

3.46.2.9 symmetric

`WORD FuNcTiOn::symmetric`

0 if symmetric properties

Definition at line 496 of file structs.h.

Referenced by StartVariables().

3.46.2.10 node

`WORD FuNcTiOn::node`

Location in namenode of NAMETREE

Definition at line 497 of file structs.h.

3.46.2.11 namesize

```
WORD FuNcTiOn::namesize
```

Length of the name

Definition at line 498 of file structs.h.

3.46.2.12 dimension

```
WORD FuNcTiOn::dimension
```

Definition at line 499 of file structs.h.

3.46.2.13 maxnumargs

```
WORD FuNcTiOn::maxnumargs
```

Definition at line 500 of file structs.h.

3.46.2.14 minnumargs

```
WORD FuNcTiOn::minnumargs
```

Definition at line 501 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.47 gcd_heuristic_failed Class Reference

3.47.1 Detailed Description

Definition at line 33 of file polygcd.h.

The documentation for this class was generated from the following file:

- polygcd.h

3.48 HANDLERS Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD newlogonly
- WORD newhandle
- WORD oldhandle
- WORD oldlogonly
- WORD oldprinttype
- WORD oldsilent

3.48.1 Detailed Description

The struct HANDLERS is used in the communication with external channels.

Definition at line 942 of file structs.h.

3.48.2 Field Documentation

3.48.2.1 newlogonly

```
WORD HANDLERS::newlogonly
```

Definition at line 943 of file structs.h.

3.48.2.2 newhandle

```
WORD HANDLERS::newhandle
```

Definition at line 944 of file structs.h.

3.48.2.3 oldhandle

```
WORD HANDLERS::oldhandle
```

Definition at line 945 of file structs.h.

3.48.2.4 oldlogonly

```
WORD HANDLERS::oldlogonly
```

Definition at line 946 of file structs.h.

3.48.2.5 oldprinttype

```
WORD HANDLERS::oldprinttype
```

Definition at line 947 of file structs.h.

3.48.2.6 oldsilent

```
WORD HANDLERS::oldsilent
```

Definition at line 948 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.49 IInput Struct Reference

Data Fields

- const char * name
- int icode
- int nplistn
- const char * plistn [GRCC_MAXLEGS]
- int plistc [GRCC_MAXLEGS]
- int cvallist [GRCC_MAXNCPLG]

3.49.1 Detailed Description

Definition at line 193 of file grccparam.h.

3.49.2 Field Documentation

3.49.2.1 name

```
const char* IInput::name
```

Definition at line 194 of file grccparam.h.

3.49.2.2 icode

```
int IInput::icode
```

Definition at line 195 of file grccparam.h.

3.49.2.3 nplistn

```
int IInput::nplistn
```

Definition at line 196 of file grccparam.h.

3.49.2.4 plistn

```
const char* IInput::plistn[GRCC_MAXLEGS]
```

Definition at line 197 of file grccparam.h.

3.49.2.5 plistc

```
int IInput::plistc[GRCC_MAXLEGS]
```

Definition at line 198 of file grccparam.h.

3.49.2.6 cvallist

```
int IInput::cvallist[GRCC_MAXNCPLG]
```

Definition at line 199 of file grccparam.h.

The documentation for this struct was generated from the following file:

- grccparam.h

3.50 InDeX Struct Reference

Data Fields

- LONG name
- WORD type
- WORD dimension
- WORD number
- WORD flags
- WORD nmin4
- WORD node
- WORD namesize

3.50.1 Detailed Description

Definition at line 442 of file structs.h.

3.50.2 Field Documentation

3.50.2.1 name

```
LONG InDeX::name
```

Definition at line 443 of file structs.h.

3.50.2.2 type

```
WORD InDeX::type
```

Definition at line 444 of file structs.h.

3.50.2.3 dimension

```
WORD InDeX::dimension
```

Definition at line 445 of file structs.h.

3.50.2.4 number

```
WORD InDeX::number
```

Definition at line 446 of file structs.h.

3.50.2.5 flags

```
WORD InDeX::flags
```

Definition at line 447 of file structs.h.

3.50.2.6 nmin4

```
WORD InDeX::nmin4
```

Definition at line 448 of file structs.h.

3.50.2.7 node

```
WORD InDeX::node
```

Definition at line 449 of file structs.h.

3.50.2.8 namesize

```
WORD InDeX::namesize
```

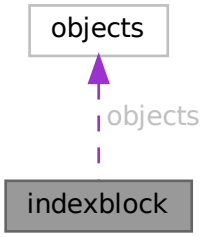
Definition at line 450 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.51 indexblock Struct Reference

Collaboration diagram for indexblock:



Data Fields

- MLONG flags
- MLONG previousblock
- MLONG position
- OBJECTS objects [NUMOBJECTS]

3.51.1 Detailed Description

Definition at line 110 of file minos.h.

3.51.2 Field Documentation

3.51.2.1 flags

```
MLONG indexblock::flags
```

Definition at line 111 of file minos.h.

3.51.2.2 previousblock

```
MLONG indexblock::previousblock
```

Definition at line 112 of file minos.h.

3.51.2.3 position

```
MLONG indexblock::position
```

Definition at line 113 of file minos.h.

3.51.2.4 objects

```
OBJECTS indexblock::objects[NUMOBJECTS]
```

Definition at line 114 of file minos.h.

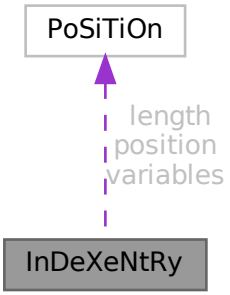
The documentation for this struct was generated from the following file:

- minos.h

3.52 InDeXeNtRy Struct Reference

```
#include <structs.h>
```

Collaboration diagram for InDeXeNtRy:



Data Fields

- POSITION position
- POSITION length
- POSITION variables
- LONG CompressSize
- WORD nsymbols
- WORD nindices
- WORD nvectors
- WORD nfunctions
- WORD size
- SBYTE name [MAXENAME+1]

3.52.1 Detailed Description

Defines the structure of an entry in a file index (see struct FiLeInDeX).

It represents one expression in the file.

Definition at line 100 of file structs.h.

3.52.2 Field Documentation

3.52.2.1 position

`POSITION InDeXeNtRy::position`

Position of the expression itself

Definition at line 101 of file structs.h.

3.52.2.2 length

`POSITION InDeXeNtRy::length`

Length of the expression itself

Definition at line 102 of file structs.h.

3.52.2.3 variables

`POSITION InDeXeNtRy::variables`

Position of the list with variables

Definition at line 103 of file structs.h.

3.52.2.4 CompressSize

`LONG InDeXeNtRy::CompressSize`

Size of buffer before compress

Definition at line 104 of file structs.h.

3.52.2.5 nsymbols

`WORD InDeXeNtRy::nsymbols`

Number of symbols in the list

Definition at line 105 of file structs.h.

Referenced by ReadSaveVariables().

3.52.2.6 nindices

```
WORD InDeXeNtRy::nindices
```

Number of indices in the list

Definition at line 106 of file structs.h.

Referenced by ReadSaveVariables().

3.52.2.7 nvectors

```
WORD InDeXeNtRy::nvectors
```

Number of vectors in the list

Definition at line 107 of file structs.h.

Referenced by ReadSaveVariables().

3.52.2.8 nfunctions

```
WORD InDeXeNtRy::nfunctions
```

Number of functions in the list

Definition at line 108 of file structs.h.

Referenced by ReadSaveVariables().

3.52.2.9 size

```
WORD InDeXeNtRy::size
```

Size of variables field

Definition at line 109 of file structs.h.

3.52.2.10 name

```
SBYTE InDeXeNtRy::name[MAXENAME+1]
```

Name of expression

Definition at line 110 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.53 iniinfo Struct Reference

Data Fields

- MLONG entriesinindex
- MLONG numberofindexblocks
- MLONG firstindexblock
- MLONG lastindexblock
- MLONG numberoftables
- MLONG numberofnamesblocks
- MLONG firstnameblock
- MLONG lastnameblock

3.53.1 Detailed Description

Definition at line 85 of file minos.h.

3.53.2 Field Documentation

3.53.2.1 entriesinindex

```
MLONG iniinfo::entriesinindex
```

Definition at line 87 of file minos.h.

3.53.2.2 numberofindexblocks

```
MLONG iniinfo::numberofindexblocks
```

Definition at line 88 of file minos.h.

3.53.2.3 firstindexblock

```
MLONG iniinfo::firstindexblock
```

Definition at line 89 of file minos.h.

3.53.2.4 lastindexblock

```
MLONG iniinfo::lastindexblock
```

Definition at line 90 of file minos.h.

3.53.2.5 numberoftables

```
MLONG iniinfo::numberoftables
```

Definition at line 91 of file minos.h.

3.53.2.6 numberofnamesblocks

```
MLONG iniinfo::numberofnamesblocks
```

Definition at line 92 of file minos.h.

3.53.2.7 firstnameblock

```
MLONG iniinfo::firstnameblock
```

Definition at line 93 of file minos.h.

3.53.2.8 lastnameblock

```
MLONG iniinfo::lastnameblock
```

Definition at line 94 of file minos.h.

The documentation for this struct was generated from the following file:

- minos.h

3.54 INSIDEINFO Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD * buffer
- int oldcompiletype
- int oldparallelflag
- int oldnumpotmoddollars
- WORD size
- WORD numdollars
- WORD oldcbuf
- WORD oldrbuf
- WORD inscbuf
- WORD oldcnumlhs

3.54.1 Detailed Description

Used for #inside

Definition at line 835 of file structs.h.

3.54.2 Field Documentation

3.54.2.1 buffer

```
WORD* INSIDEINFO::buffer
```

Definition at line 836 of file structs.h.

3.54.2.2 oldcompiletype

```
int INSIDEINFO::oldcompiletype
```

Definition at line 837 of file structs.h.

3.54.2.3 oldparallelflag

```
int INSIDEINFO::oldparallelflag
```

Definition at line 838 of file structs.h.

3.54.2.4 oldnumpotmoddollars

```
int INSIDEINFO::oldnumpotmoddollars
```

Definition at line 839 of file structs.h.

3.54.2.5 size

```
WORD INSIDEINFO::size
```

Definition at line 840 of file structs.h.

3.54.2.6 numdollars

```
WORD INSIDEINFO::numdollars
```

Definition at line 841 of file structs.h.

3.54.2.7 oldcbuf

```
WORD INSIDEINFO::oldcbuf
```

Definition at line 842 of file structs.h.

3.54.2.8 oldrbuf

```
WORD INSIDEINFO::oldrbuf
```

Definition at line 843 of file structs.h.

3.54.2.9 inscbuf

```
WORD INSIDEINFO::inscbuf
```

Definition at line 844 of file structs.h.

3.54.2.10 oldcnumlhs

```
WORD INSIDEINFO::oldcnumlhs
```

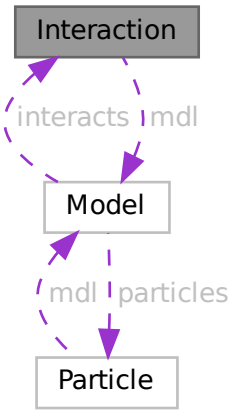
Definition at line 845 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.55 Interaction Class Reference

Collaboration diagram for Interaction:



Public Member Functions

- Interaction (Model ∗modl, int iid, const char ∗nam, int icode, int ∗cpl, int nlgs, int ∗plst, int csm, int lp)
- void prInteraction (void)

Data Fields

- Model ∗ mdl
- char ∗ name
- int ∗ plist
- int ∗ clist
- int ∗ slist
- int id
- int csum
- int nlegs
- int loop
- int icode
- int nplist
- int nclist
- int nslist

3.55.1 Detailed Description

Definition at line 258 of file grcc.h.

3.55.2 Constructor & Destructor Documentation

3.55.2.1 Interaction()

```
Interaction::Interaction (
            Model * modl,
            int iid,
            const char * nam,
            int icode,
            int * cpl,
            int nlgs,
            int * plst,
            int csm,
            int lp )
```

Definition at line 1820 of file grcc.cc.

3.55.2.2 ∼Interaction()

```
Interaction::~Interaction (
            void  )
```

Definition at line 1944 of file grcc.cc.

3.55.3 Member Function Documentation

3.55.3.1 prInteraction()

```
void Interaction::prInteraction (
            void  )
```

Definition at line 1959 of file grcc.cc.

3.55.4 Field Documentation

3.55.4.1 mdl

```
Model* Interaction::mdl
```

Definition at line 260 of file grcc.h.

3.55.4.2 name

```
char* Interaction::name
```

Definition at line 261 of file grcc.h.

3.55.4.3 plist

```
int* Interaction::plist
```

Definition at line 262 of file grcc.h.

3.55.4.4 clist

```
int* Interaction::clist
```

Definition at line 263 of file grcc.h.

3.55.4.5 slist

```
int* Interaction::slist
```

Definition at line 264 of file grcc.h.

3.55.4.6 id

```
int Interaction::id
```

Definition at line 266 of file grcc.h.

3.55.4.7 csum

```
int Interaction::csum
```

Definition at line 267 of file grcc.h.

3.55.4.8 nlegs

```
int Interaction::nlegs
```

Definition at line 268 of file grcc.h.

3.55.4.9 loop

```
int Interaction::loop
```

Definition at line 269 of file grcc.h.

3.55.4.10 icode

```
int Interaction::icode
```

Definition at line 270 of file grcc.h.

3.55.4.11 nplist

```
int Interaction::nplist
```

Definition at line 272 of file grcc.h.

3.55.4.12 nclist

```
int Interaction::nclist
```

Definition at line 273 of file grcc.h.

3.55.4.13 nslist

```
int Interaction::nslist
```

Definition at line 274 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.56 KEYWORD Struct Reference

`#include <structs.h>`

Data Fields

- char * name
- TFUN func
- int type
- int flags

3.56.1 Detailed Description

The KEYWORD struct defines names of commands/statements and the routine to be called when they are encountered by the compiler or preprocessor.

Definition at line 218 of file structs.h.

3.56.2 Field Documentation

3.56.2.1 name

`char* KEYWORD::name`

Definition at line 219 of file structs.h.

3.56.2.2 func

`TFUN KEYWORD::func`

Definition at line 220 of file structs.h.

3.56.2.3 type

`int KEYWORD::type`

Definition at line 221 of file structs.h.

3.56.2.4 flags

`int KEYWORD::flags`

Definition at line 222 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.57 KEYWORDV Struct Reference

`#include <structs.h>`

Data Fields

- char * name
- int * var
- int type
- int flags

3.57.1 Detailed Description

The KEYWORDV struct defines names of commands/statements and the variable to be affected when they are encountered by the compiler or preprocessor.

Definition at line 230 of file structs.h.

3.57.2 Field Documentation

3.57.2.1 name

`char* KEYWORDV::name`

Definition at line 231 of file structs.h.

3.57.2.2 var

`int* KEYWORDV::var`

Definition at line 232 of file structs.h.

3.57.2.3 type

`int KEYWORDV::type`

Definition at line 233 of file structs.h.

3.57.2.4 flags

`int KEYWORDV::flags`

Definition at line 234 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.58 LIST Struct Reference

```
#include <structs.h>
```

Data Fields

- void * lijst
- char * message
- int num
- int maxnum
- int size
- int numglobal
- int numtemp
- int numclear

3.58.1 Detailed Description

Much information is stored in arrays of which we can double the size if the array proves to be too small. Such arrays are controlled by a variable of type LIST. The routines that expand the lists are in the file tools.c

Definition at line 202 of file structs.h.

3.58.2 Field Documentation

3.58.2.1 lijst

```
void* LIST::lijst
```

[D] Holds space for "maxnum" elements of size "size" each

Definition at line 203 of file structs.h.

3.58.2.2 message

```
char* LIST::message
```

Text for Malloc1 when allocating lijst. Set to constant string.

Definition at line 204 of file structs.h.

3.58.2.3 num

```
int LIST::num
```

Number of elements in lijst.

Definition at line 205 of file structs.h.

3.58.2.4 maxnum

```
int LIST::maxnum
```

Maximum number of elements in lijst.

Definition at line 206 of file structs.h.

3.58.2.5 size

```
int LIST::size
```

Size of one element in lijst.

Definition at line 207 of file structs.h.

3.58.2.6 numglobal

```
int LIST::numglobal
```

Marker for position when .global is executed.

Definition at line 208 of file structs.h.

3.58.2.7 numtemp

```
int LIST::numtemp
```

At the moment only needed for sets and setstore.

Definition at line 209 of file structs.h.

3.58.2.8 numclear

```
int LIST::numclear
```

Only for the clear instruction.

Definition at line 210 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.59 longMultiStruct Struct Reference

Collaboration diagram for longMultiStruct:

Data Fields

- UBYTE * buffer
- int bufpos
- int packpos
- int nPacks
- int lastLen
- struct longMultiStruct * next

3.59.1 Detailed Description

Definition at line 1124 of file mpi.c.

3.59.2 Field Documentation

3.59.2.1 buffer

```
UBYTE* longMultiStruct::buffer
```

Definition at line 1125 of file mpi.c.

3.59.2.2 bufpos

```
int longMultiStruct::bufpos
```

Definition at line 1126 of file mpi.c.

3.59.2.3 packpos

```
int longMultiStruct::packpos
```

Definition at line 1127 of file mpi.c.

3.59.2.4 nPacks

```
int longMultiStruct::nPacks
```

Definition at line 1128 of file mpi.c.

3.59.2.5 lastLen

```
int longMultiStruct::lastLen
```

Definition at line 1129 of file mpi.c.

3.59.2.6 next

```
struct longMultiStruct* longMultiStruct::next
```

Definition at line 1132 of file mpi.c.

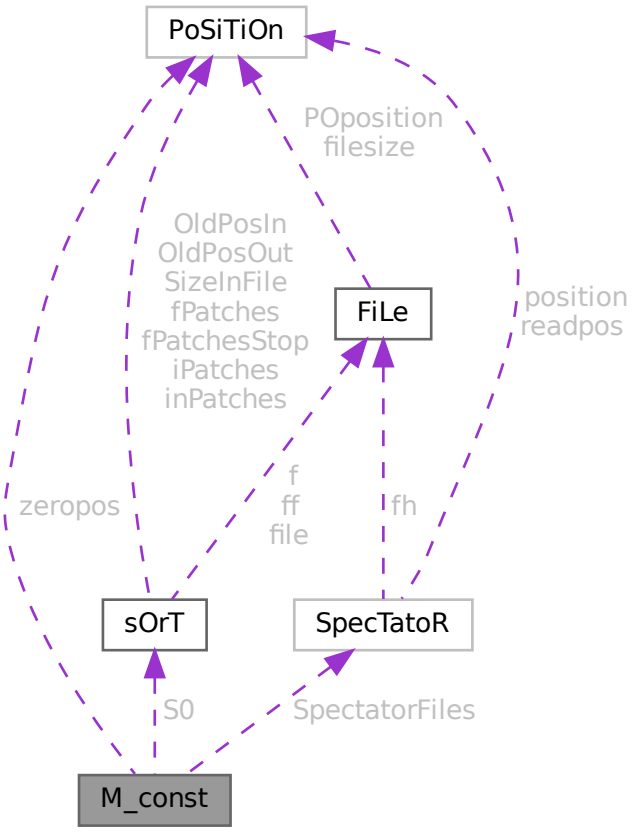
The documentation for this struct was generated from the following file:

- mpi.c

3.60 M_const Struct Reference

```
#include <structs.h>
```

Collaboration diagram for M_const:



Data Fields

- POSITION zeropos
- SORTING * S0
- UWORD * gcmod
- UWORD * gpowmod
- UBYTE * TempDir
- UBYTE * TempSortDir
- UBYTE * IncDir
- UBYTE * InputFileName
- UBYTE * LogFileName
- UBYTE * OutBuffer
- UBYTE * Path
- UBYTE * SetupDir
- UBYTE * SetupFile
- UBYTE * gFortran90Kind
- UBYTE * gextrasym
- UBYTE * ggextrasym
- UBYTE * oldnumextrasymbols
- SPECTATOR * SpectatorFiles

- LONG MaxTer
- LONG CompressSize
- LONG ScratSize
- LONG HideSize
- LONG SizeStoreCache
- LONG MaxStreamSize
- LONG SIOsize
- LONG SLargeSize
- LONG SSmallEsize
- LONG SSmallSize
- LONG STermsInSmall
- LONG MaxBracketBufferSize
- LONG hProcessBucketSize
- LONG gProcessBucketSize
- LONG shmWinSize
- LONG OldChildTime
- LONG OldSecTime
- LONG OldMilliTime
- LONG WorkSize
- LONG gThreadBucketSize
- LONG ggThreadBucketSize
- LONG SumTime
- LONG SpectatorSize
- LONG TimeLimit
- int FileOnlyFlag
- int Interact
- int MaxParLevel
- int OutBufSize
- int SMaxFpatches
- int SMaxPatches
- int StdOut
- int ginsidefirst
- int gDefDim
- int gDefDim4
- int NumFixedSets
- int NumFixedFunctions
- int rbufnum
- int dbufnum
- int sbufnum
- int zbufnum
- int SkipClears
- int gTokensWriteFlag
- int gfunpowers
- int gStatsFlag
- int gNamesFlag
- int gCodesFlag
- int gSortType
- int gproperorderflag
- int hparallelflag
- int gparallelflag
- int totalnumberofthreads
- int gSizeCommuteInSet
- int gThreadStats
- int ggThreadStats
- int gFinalStats

- int ggFinalStats
- int gThreadsFlag
- int ggThreadsFlag
- int gThreadBalancing
- int ggThreadBalancing
- int gThreadSortFileSynch
- int ggThreadSortFileSynch
- int gProcessStats
- int ggProcessStats
- int gOldParallelStats
- int ggOldParallelStats
- int maxFlevels
- int resetTimeOnClear
- int gcNumDollars
- int MultiRun
- int gNoSpacesInNumbers
- int ggNoSpacesInNumbers
- int gIsFortran90
- int PrintTotalSize
- int fbuffersize
- int gOldFactArgFlag
- int ggOldFactArgFlag
- int gnumextrasym
- int ggnumextrasym
- int NumSpectatorFiles
- int SizeForSpectatorFiles
- int gOldGCDflag
- int ggOldGCDflag
- int gWTimeStatsFlag
- int ggWTimeStatsFlag
- int jumpratio
- WORD MaxTal
- WORD IndDum
- WORD DumInd
- WORD WilInd
- WORD gncmod
- WORD gnpowmod
- WORD gmodmode
- WORD gUnitTrace
- WORD gOutputMode
- WORD gOutputSpaces
- WORD gOutNumberType
- WORD gCnumpows
- WORD gUniTrace [4]
- WORD MaxWildcards
- WORD mTraceDum
- WORD OffsetIndex
- WORD OffsetVector
- WORD RepMax
- WORD LogType
- WORD ggStatsFlag
- WORD gLineLength
- WORD qError
- WORD FortranCont
- WORD HoldFlag

- WORD Ordering [15]
- WORD silent
- WORD tracebackflag
- WORD expnum
- WORD denomnum
- WORD facnum
- WORD invfacnum
- WORD sumnum
- WORD sumpnum
- WORD OldOrderFlag
- WORD termfunnum
- WORD matchfunnum
- WORD countfunnum
- WORD gPolyFun
- WORD gPolyFunInv
- WORD gPolyFunType
- WORD gPolyFunExp
- WORD gPolyFunVar
- WORD gPolyFunPow
- WORD dollarzero
- WORD atstartup
- WORD exitflag
- WORD NumStoreCaches
- WORD gIndentSpace
- WORD ggIndentSpace
- WORD gShortStatsMax
- WORD ggShortStatsMax
- WORD gextrasymbols
- WORD ggextrasymbols
- WORD zerorhs
- WORD onerhs
- WORD havesortdir
- WORD vectorzero
- WORD ClearStore
- WORD BracketFactors [8]
- BOOL FromStdin
- BOOL IgnoreDeprecation

3.60.1 Detailed Description

The M_const struct is part of the global data and resides in the ALLGLOBALS struct A under the name #M. We see it used with the macro AM as in AM.S0. It contains global settings at startup or .clear.

Definition at line 1419 of file structs.h.

3.60.2 Field Documentation

3.60.2.1 zeropos

```
POSITION M_const::zeropos
```

Definition at line 1420 of file structs.h.

3.60.2.2 S0

```
SORTING* M_const::S0
```

[D] The main sort buffer

Definition at line 1421 of file structs.h.

3.60.2.3 gcmod

```
UWORD* M_const::gcmod
```

Global setting of modulus. Uses AC.cmod's memory

Definition at line 1422 of file structs.h.

3.60.2.4 gpowmod

```
UWORD* M_const::gpowmod
```

Global setting printing as powers. Uses AC.cmod's memory

Definition at line 1423 of file structs.h.

3.60.2.5 TempDir

```
UBYTE* M_const::TempDir
```

Definition at line 1424 of file structs.h.

3.60.2.6 TempSortDir

```
UBYTE* M_const::TempSortDir
```

Definition at line 1425 of file structs.h.

3.60.2.7 IncDir

```
UBYTE* M_const::IncDir
```

Definition at line 1426 of file structs.h.

3.60.2.8 InputFileName

```
UBYTE* M_const::InputFileName
```

Definition at line 1427 of file structs.h.

3.60.2.9 LogFileName

```
UBYTE* M_const::LogFileName
```

Definition at line 1428 of file structs.h.

3.60.2.10 OutBuffer

```
UBYTE* M_const::OutBuffer
```

Definition at line 1429 of file structs.h.

3.60.2.11 Path

```
UBYTE* M_const::Path
```

Definition at line 1430 of file structs.h.

3.60.2.12 SetupDir

```
UBYTE* M_const::SetupDir
```

Definition at line 1431 of file structs.h.

3.60.2.13 SetupFile

```
UBYTE* M_const::SetupFile
```

Definition at line 1432 of file structs.h.

3.60.2.14 gFortran90Kind

```
UBYTE* M_const::gFortran90Kind
```

Definition at line 1433 of file structs.h.

3.60.2.15 gextrasym

```
UBYTE* M_const::gextrasym
```

Definition at line 1434 of file structs.h.

3.60.2.16 ggextrasym

```
UBYTE* M_const::ggextrasym
```

Definition at line 1435 of file structs.h.

3.60.2.17 oldnumextrasymbols

```
UBYTE* M_const::oldnumextrasymbols
```

Definition at line 1436 of file structs.h.

3.60.2.18 SpectatorFiles

```
SPECTATOR* M_const::SpectatorFiles
```

Definition at line 1437 of file structs.h.

3.60.2.19 MaxTer

```
LONG M_const::MaxTer
```

Definition at line 1445 of file structs.h.

3.60.2.20 CompressSize

```
LONG M_const::CompressSize
```

Definition at line 1446 of file structs.h.

3.60.2.21 ScratSize

```
LONG M_const::ScratSize
```

Definition at line 1447 of file structs.h.

3.60.2.22 HideSize

```
LONG M_const::HideSize
```

Definition at line 1448 of file structs.h.

3.60.2.23 SizeStoreCache

```
LONG M_const::SizeStoreCache
```

Definition at line 1449 of file structs.h.

3.60.2.24 MaxStreamSize

```
LONG M_const::MaxStreamSize
```

Definition at line 1450 of file structs.h.

3.60.2.25 SIOsize

```
LONG M_const::SIOsize
```

Definition at line 1451 of file structs.h.

3.60.2.26 SLargeSize

```
LONG M_const::SLargeSize
```

Definition at line 1452 of file structs.h.

3.60.2.27 SSmallEsize

```
LONG M_const::SSmallEsize
```

Definition at line 1453 of file structs.h.

3.60.2.28 SSmallSize

```
LONG M_const::SSmallSize
```

Definition at line 1454 of file structs.h.

3.60.2.29 STermsInSmall

```
LONG M_const::STermsInSmall
```

Definition at line 1455 of file structs.h.

3.60.2.30 MaxBracketBufferSize

```
LONG M_const::MaxBracketBufferSize
```

Definition at line 1456 of file structs.h.

3.60.2.31 hProcessBucketSize

```
LONG M_const::hProcessBucketSize
```

Definition at line 1457 of file structs.h.

3.60.2.32 gProcessBucketSize

```
LONG M_const::gProcessBucketSize
```

Definition at line 1458 of file structs.h.

3.60.2.33 shmWinSize

```
LONG M_const::shmWinSize
```

Definition at line 1459 of file structs.h.

3.60.2.34 OldChildTime

```
LONG M_const::OldChildTime
```

Definition at line 1460 of file structs.h.

3.60.2.35 OldSecTime

```
LONG M_const::OldSecTime
```

Definition at line 1461 of file structs.h.

3.60.2.36 OldMilliTime

```
LONG M_const::OldMilliTime
```

Definition at line 1462 of file structs.h.

3.60.2.37 WorkSize

```
LONG M_const::WorkSize
```

Definition at line 1463 of file structs.h.

3.60.2.38 gThreadBucketSize

```
LONG M_const::gThreadBucketSize
```

Definition at line 1464 of file structs.h.

3.60.2.39 ggThreadBucketSize

```
LONG M_const::ggThreadBucketSize
```

Definition at line 1465 of file structs.h.

3.60.2.40 SumTime

```
LONG M_const::SumTime
```

Definition at line 1466 of file structs.h.

3.60.2.41 SpectatorSize

```
LONG M_const::SpectatorSize
```

Definition at line 1467 of file structs.h.

3.60.2.42 TimeLimit

```
LONG M_const::TimeLimit
```

Definition at line 1468 of file structs.h.

3.60.2.43 FileOnlyFlag

```
int M_const::FileOnlyFlag
```

Definition at line 1475 of file structs.h.

3.60.2.44 Interact

```
int M_const::Interact
```

Definition at line 1476 of file structs.h.

3.60.2.45 MaxParLevel

```
int M_const::MaxParLevel
```

Definition at line 1477 of file structs.h.

3.60.2.46 OutBufSize

```
int M_const::OutBufSize
```

Definition at line 1478 of file structs.h.

3.60.2.47 SMaxFpatches

```
int M_const::SMaxFpatches
```

Definition at line 1479 of file structs.h.

3.60.2.48 SMaxPatches

```
int M_const::SMaxPatches
```

Definition at line 1480 of file structs.h.

3.60.2.49 StdOut

```
int M_const::StdOut
```

Definition at line 1481 of file structs.h.

3.60.2.50 ginsidefirst

```
int M_const::ginsidefirst
```

Definition at line 1482 of file structs.h.

3.60.2.51 gDefDim

```
int M_const::gDefDim
```

Definition at line 1483 of file structs.h.

3.60.2.52 gDefDim4

```
int M_const::gDefDim4
```

Definition at line 1484 of file structs.h.

3.60.2.53 NumFixedSets

```
int M_const::NumFixedSets
```

Definition at line 1485 of file structs.h.

3.60.2.54 NumFixedFunctions

```
int M_const::NumFixedFunctions
```

Definition at line 1486 of file structs.h.

3.60.2.55 rbufnum

```
int M_const::rbufnum
```

Definition at line 1487 of file structs.h.

3.60.2.56 dbufnum

```
int M_const::dbufnum
```

Definition at line 1488 of file structs.h.

3.60.2.57 sbufnum

```
int M_const::sbufnum
```

Definition at line 1489 of file structs.h.

3.60.2.58 zbufnum

```
int M_const::zbufnum
```

Definition at line 1490 of file structs.h.

3.60.2.59 SkipClears

```
int M_const::SkipClears
```

Definition at line 1491 of file structs.h.

3.60.2.60 gTokensWriteFlag

```
int M_const::gTokensWriteFlag
```

Definition at line 1492 of file structs.h.

3.60.2.61 gfunpowers

```
int M_const::gfunpowers
```

Definition at line 1493 of file structs.h.

3.60.2.62 gStatsFlag

```
int M_const::gStatsFlag
```

Definition at line 1494 of file structs.h.

3.60.2.63 gNamesFlag

```
int M_const::gNamesFlag
```

Definition at line 1495 of file structs.h.

3.60.2.64 gCodesFlag

```
int M_const::gCodesFlag
```

Definition at line 1496 of file structs.h.

3.60.2.65 gSortType

```
int M_const::gSortType
```

Definition at line 1497 of file structs.h.

3.60.2.66 gproperorderflag

```
int M_const::gproperorderflag
```

Definition at line 1498 of file structs.h.

3.60.2.67 hparallelflag

```
int M_const::hparallelflag
```

Definition at line 1499 of file structs.h.

3.60.2.68 gparallelflag

```
int M_const::gparallelflag
```

Definition at line 1500 of file structs.h.

3.60.2.69 totalnumberofthreads

```
int M_const::totalnumberofthreads
```

Definition at line 1501 of file structs.h.

3.60.2.70 gSizeCommuteInSet

```
int M_const::gSizeCommuteInSet
```

Definition at line 1502 of file structs.h.

3.60.2.71 gThreadStats

```
int M_const::gThreadStats
```

Definition at line 1503 of file structs.h.

3.60.2.72 ggThreadStats

```
int M_const::ggThreadStats
```

Definition at line 1504 of file structs.h.

3.60.2.73 gFinalStats

```
int M_const::gFinalStats
```

Definition at line 1505 of file structs.h.

3.60.2.74 ggFinalStats

```
int M_const::ggFinalStats
```

Definition at line 1506 of file structs.h.

3.60.2.75 gThreadsFlag

```
int M_const::gThreadsFlag
```

Definition at line 1507 of file structs.h.

3.60.2.76 ggThreadsFlag

```
int M_const::ggThreadsFlag
```

Definition at line 1508 of file structs.h.

3.60.2.77 gThreadBalancing

```
int M_const::gThreadBalancing
```

Definition at line 1509 of file structs.h.

3.60.2.78 ggThreadBalancing

```
int M_const::ggThreadBalancing
```

Definition at line 1510 of file structs.h.

3.60.2.79 gThreadSortFileSynch

```
int M_const::gThreadSortFileSynch
```

Definition at line 1511 of file structs.h.

3.60.2.80 ggThreadSortFileSynch

```
int M_const::ggThreadSortFileSynch
```

Definition at line 1512 of file structs.h.

3.60.2.81 gProcessStats

```
int M_const::gProcessStats
```

Definition at line 1513 of file structs.h.

3.60.2.82 ggProcessStats

```
int M_const::ggProcessStats
```

Definition at line 1514 of file structs.h.

3.60.2.83 gOldParallelStats

```
int M_const::gOldParallelStats
```

Definition at line 1515 of file structs.h.

3.60.2.84 ggOldParallelStats

```
int M_const::ggOldParallelStats
```

Definition at line 1516 of file structs.h.

3.60.2.85 maxFlevels

```
int M_const::maxFlevels
```

Definition at line 1517 of file structs.h.

3.60.2.86 resetTimeOnClear

```
int M_const::resetTimeOnClear
```

Definition at line 1518 of file structs.h.

3.60.2.87 gcNumDollars

```
int M_const::gcNumDollars
```

Definition at line 1519 of file structs.h.

3.60.2.88 MultiRun

```
int M_const::MultiRun
```

Definition at line 1520 of file structs.h.

3.60.2.89 gNoSpacesInNumbers

```
int M_const::gNoSpacesInNumbers
```

Definition at line 1521 of file structs.h.

3.60.2.90 ggNoSpacesInNumbers

```
int M_const::ggNoSpacesInNumbers
```

Definition at line 1522 of file structs.h.

3.60.2.91 gIsFortran90

```
int M_const::gIsFortran90
```

Definition at line 1523 of file structs.h.

3.60.2.92 PrintTotalSize

```
int M_const::PrintTotalSize
```

Definition at line 1524 of file structs.h.

3.60.2.93 fbuffersize

```
int M_const::fbuffersize
```

Definition at line 1525 of file structs.h.

3.60.2.94 gOldFactArgFlag

```
int M_const::gOldFactArgFlag
```

Definition at line 1526 of file structs.h.

3.60.2.95 ggOldFactArgFlag

```
int M_const::ggOldFactArgFlag
```

Definition at line 1527 of file structs.h.

3.60.2.96 gnumextrasym

```
int M_const::gnumextrasym
```

Definition at line 1528 of file structs.h.

3.60.2.97 ggnumextrasym

```
int M_const::ggnumextrasym
```

Definition at line 1529 of file structs.h.

3.60.2.98 NumSpectatorFiles

```
int M_const::NumSpectatorFiles
```

Definition at line 1530 of file structs.h.

3.60.2.99 SizeForSpectatorFiles

```
int M_const::SizeForSpectatorFiles
```

Definition at line 1531 of file structs.h.

3.60.2.100 gOldGCDflag

```
int M_const::gOldGCDflag
```

Definition at line 1532 of file structs.h.

3.60.2.101 ggOldGCDflag

```
int M_const::ggOldGCDflag
```

Definition at line 1533 of file structs.h.

3.60.2.102 gWTimeStatsFlag

```
int M_const::gWTimeStatsFlag
```

Definition at line 1534 of file structs.h.

3.60.2.103 ggWTimeStatsFlag

```
int M_const::ggWTimeStatsFlag
```

Definition at line 1535 of file structs.h.

3.60.2.104 jumpratio

```
int M_const::jumpratio
```

Definition at line 1536 of file structs.h.

3.60.2.105 MaxTal

```
WORD M_const::MaxTal
```

Definition at line 1537 of file structs.h.

3.60.2.106 IndDum

```
WORD M_const::IndDum
```

Definition at line 1538 of file structs.h.

3.60.2.107 DumInd

```
WORD M_const::DumInd
```

Definition at line 1539 of file structs.h.

3.60.2.108 WilInd

```
WORD M_const::WilInd
```

Definition at line 1540 of file structs.h.

3.60.2.109 gncmod

```
WORD M_const::gncmod
```

Definition at line 1541 of file structs.h.

3.60.2.110 gnpowmod

```
WORD M_const::gnpowmod
```

Definition at line 1542 of file structs.h.

3.60.2.111 gmodmode

```
WORD M_const::gmodmode
```

Definition at line 1543 of file structs.h.

3.60.2.112 gUnitTrace

```
WORD M_const::gUnitTrace
```

Definition at line 1544 of file structs.h.

3.60.2.113 gOutputMode

```
WORD M_const::gOutputMode
```

Definition at line 1545 of file structs.h.

3.60.2.114 gOutputSpaces

```
WORD M_const::gOutputSpaces
```

Definition at line 1546 of file structs.h.

3.60.2.115 gOutNumberType

```
WORD M_const::gOutNumberType
```

Definition at line 1547 of file structs.h.

3.60.2.116 gCnumpows

```
WORD M_const::gCnumpows
```

Definition at line 1548 of file structs.h.

3.60.2.117 gUniTrace

```
WORD M_const::gUniTrace[4]
```

Definition at line 1549 of file structs.h.

3.60.2.118 MaxWildcards

```
WORD M_const::MaxWildcards
```

Definition at line 1550 of file structs.h.

3.60.2.119 mTraceDum

```
WORD M_const::mTraceDum
```

Definition at line 1551 of file structs.h.

3.60.2.120 OffsetIndex

```
WORD M_const::OffsetIndex
```

Definition at line 1552 of file structs.h.

3.60.2.121 OffsetVector

```
WORD M_const::OffsetVector
```

Definition at line 1553 of file structs.h.

3.60.2.122 RepMax

```
WORD M_const::RepMax
```

Definition at line 1554 of file structs.h.

3.60.2.123 LogType

```
WORD M_const::LogType
```

Definition at line 1555 of file structs.h.

3.60.2.124 ggStatsFlag

```
WORD M_const::ggStatsFlag
```

Definition at line 1556 of file structs.h.

3.60.2.125 gLineLength

```
WORD M_const::gLineLength
```

Definition at line 1557 of file structs.h.

3.60.2.126 qError

```
WORD M_const::qError
```

Definition at line 1558 of file structs.h.

3.60.2.127 FortranCont

```
WORD M_const::FortranCont
```

Definition at line 1559 of file structs.h.

3.60.2.128 HoldFlag

```
WORD M_const::HoldFlag
```

Definition at line 1560 of file structs.h.

3.60.2.129 Ordering

```
WORD M_const::Ordering[15]
```

Definition at line 1561 of file structs.h.

3.60.2.130 silent

```
WORD M_const::silent
```

Definition at line 1562 of file structs.h.

3.60.2.131 tracebackflag

```
WORD M_const::tracebackflag
```

Definition at line 1563 of file structs.h.

3.60.2.132 expnum

```
WORD M_const::expnum
```

Definition at line 1564 of file structs.h.

3.60.2.133 denomnum

```
WORD M_const::denomnum
```

Definition at line 1565 of file structs.h.

3.60.2.134 facnum

```
WORD M_const::facnum
```

Definition at line 1566 of file structs.h.

3.60.2.135 invfacnum

```
WORD M_const::invfacnum
```

Definition at line 1567 of file structs.h.

3.60.2.136 sumnum

```
WORD M_const::sumnum
```

Definition at line 1568 of file structs.h.

3.60.2.137 sumpnum

```
WORD M_const::sumpnum
```

Definition at line 1569 of file structs.h.

3.60.2.138 OldOrderFlag

```
WORD M_const::OldOrderFlag
```

Definition at line 1570 of file structs.h.

3.60.2.139 termfunnum

```
WORD M_const::termfunnum
```

Definition at line 1571 of file structs.h.

3.60.2.140 matchfunnum

```
WORD M_const::matchfunnum
```

Definition at line 1572 of file structs.h.

3.60.2.141 countfunnum

```
WORD M_const::countfunnum
```

Definition at line 1573 of file structs.h.

3.60.2.142 gPolyFun

```
WORD M_const::gPolyFun
```

Definition at line 1574 of file structs.h.

3.60.2.143 gPolyFunInv

```
WORD M_const::gPolyFunInv
```

Definition at line 1575 of file structs.h.

3.60.2.144 gPolyFunType

```
WORD M_const::gPolyFunType
```

Definition at line 1576 of file structs.h.

3.60.2.145 gPolyFunExp

```
WORD M_const::gPolyFunExp
```

Definition at line 1577 of file structs.h.

3.60.2.146 gPolyFunVar

```
WORD M_const::gPolyFunVar
```

Definition at line 1578 of file structs.h.

3.60.2.147 gPolyFunPow

```
WORD M_const::gPolyFunPow
```

Definition at line 1579 of file structs.h.

3.60.2.148 dollarzero

```
WORD M_const::dollarzero
```

Definition at line 1580 of file structs.h.

3.60.2.149 atstartup

```
WORD M_const::atstartup
```

Definition at line 1581 of file structs.h.

3.60.2.150 exitflag

```
WORD M_const::exitflag
```

Definition at line 1582 of file structs.h.

3.60.2.151 NumStoreCaches

```
WORD M_const::NumStoreCaches
```

Definition at line 1583 of file structs.h.

3.60.2.152 gIndentSpace

```
WORD M_const::gIndentSpace
```

Definition at line 1584 of file structs.h.

3.60.2.153 ggIndentSpace

```
WORD M_const::ggIndentSpace
```

Definition at line 1585 of file structs.h.

3.60.2.154 gShortStatsMax

```
WORD M_const::gShortStatsMax
```

For On FewerStatistics 10;

Definition at line 1586 of file structs.h.

3.60.2.155 ggShortStatsMax

```
WORD M_const::ggShortStatsMax
```

For On FewerStatistics 10;

Definition at line 1587 of file structs.h.

3.60.2.156 gextrasymbols

```
WORD M_const::gextrasymbols
```

Definition at line 1588 of file structs.h.

3.60.2.157 ggextrasymbols

```
WORD M_const::ggextrasymbols
```

Definition at line 1589 of file structs.h.

3.60.2.158 zerorhs

```
WORD M_const::zerorhs
```

Definition at line 1590 of file structs.h.

3.60.2.159 onerhs

```
WORD M_const::onerhs
```

Definition at line 1591 of file structs.h.

3.60.2.160 havesortdir

```
WORD M_const::havesortdir
```

Definition at line 1592 of file structs.h.

3.60.2.161 vectorzero

```
WORD M_const::vectorzero
```

Definition at line 1593 of file structs.h.

3.60.2.162 ClearStore

```
WORD M_const::ClearStore
```

Definition at line 1594 of file structs.h.

3.60.2.163 BracketFactors

```
WORD M_const::BracketFactors[8]
```

Definition at line 1595 of file structs.h.

3.60.2.164 FromStdin

```
BOOL M_const::FromStdin
```

Definition at line 1596 of file structs.h.

3.60.2.165 IgnoreDeprecation

```
BOOL M_const::IgnoreDeprecation
```

Definition at line 1597 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.61 MCBlock Class Reference

Public Member Functions

- void init (void)

Data Fields

- Edge2n * edges
- int nmedges
- int nartps
- int loop
- int DUMMYPADDING

3.61.1 Detailed Description

Definition at line 985 of file grcc.h.

3.61.2 Constructor & Destructor Documentation

3.61.2.1 MCBlock()

```
MCBlock::MCBlock (
            void  )
```

Definition at line 5512 of file grcc.cc.

3.61.2.2 ~MCBlock()

```
MCBlock::~MCBlock (
            void  )
```

Definition at line 5518 of file grcc.cc.

3.61.3 Member Function Documentation

3.61.3.1 init()

```
void MCBlock::init (
            void  )
```

Definition at line 5523 of file grcc.cc.

3.61.4 Field Documentation

3.61.4.1 edges

```
Edge2n* MCBlock::edges
```

Definition at line 987 of file grcc.h.

3.61.4.2 nmedges

```
int MCBlock::nmedges
```

Definition at line 988 of file grcc.h.

3.61.4.3 nartps

```
int MCBlock::nartps
```

Definition at line 989 of file grcc.h.

3.61.4.4 loop

```
int MCBlock::loop
```

Definition at line 990 of file grcc.h.

3.61.4.5 DUMMYPADDING

```
int MCBlock::DUMMYPADDING
```

Definition at line 992 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.62 MCBridge Class Reference

Data Fields

- Edge2n nodes
- int next

3.62.1 Detailed Description

Definition at line 973 of file grcc.h.

3.62.2 Constructor & Destructor Documentation

3.62.2.1 MCBridge()

```
MCBridge::MCBridge (
            void  )
```

Definition at line 5500 of file grcc.cc.

3.62.2.2 ~MCBridge()

```
MCBridge::~MCBridge (
            void  )
```

Definition at line 5505 of file grcc.cc.

3.62.3 Field Documentation

3.62.3.1 nodes

```
Edge2n MCBridge::nodes
```

Definition at line 975 of file grcc.h.

3.62.3.2 next

```
int MCBridge::next
```

Definition at line 976 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.63 MCEdge Class Reference

Data Fields

- Edge2n nodes
- ULong momset
- int momdir
- int DUMMYPADDING

3.63.1 Detailed Description

Definition at line 936 of file grcc.h.

3.63.2 Constructor & Destructor Documentation

3.63.2.1 MCEdge()

```
MCEdge::MCEdge (
            void  )
```

Definition at line 5459 of file grcc.cc.

3.63.2.2 ~MCEdge()

```
MCEdge::~MCEdge (
            void  )
```

Definition at line 5465 of file grcc.cc.

3.63.3 Field Documentation

3.63.3.1 nodes

```
Edge2n MCEdge::nodes
```

Definition at line 938 of file grcc.h.

3.63.3.2 momset

```
ULong MCEdge::momset
```

Definition at line 939 of file grcc.h.

3.63.3.3 momdir

```
int MCEdge::momdir
```

Definition at line 940 of file grcc.h.

3.63.3.4 DUMMYPADDING

```
int MCEdge::DUMMYPADDING
```

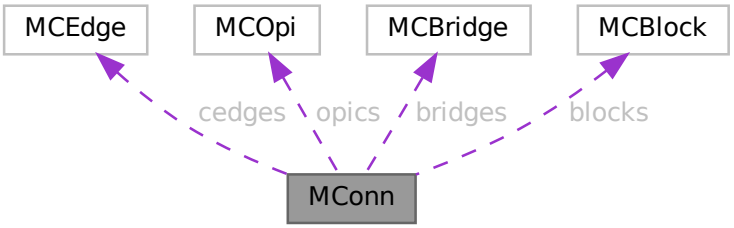
Definition at line 942 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.64 MConn Class Reference

Collaboration diagram for MConn:



Public Member Functions

- MConn (int nnod, int nedg)
- void init (void)
- void initCEdges (MGraph ∗)
- void pushNode (int nd)
- void pushEdge (int n0, int n1)
- void addCEdge (int n0, int n1, ULong momset)
- void addOPIc (MCOpi ∗mopi, int stp)
- void addBridge (int n0, int n1, int nex, int nextot)
- void addArtic (int nd, int mul)
- void addBlock (MCBlock ∗eblk, int stp)
- void addBlockSelf (int nd, int mul)
- void print (void)
- void prEdges (void)

Data Fields

- MCEdge ∗ cedges
- MCOpi ∗ opics
- MCBridge ∗ bridges
- MCBlock ∗ blocks
- int ∗ articuls
- int ∗ opisp
- int ∗ opistk
- Edge2n ∗ blksp
- Edge2n ∗ blkstk
- int snodes
- int sedges
- int nopic
- int nlpopic
- int nctopic
- int nbacked
- int nbridges

- int ne0bridges
- int ne1bridges
- int nselfloops
- int nmultiedges
- int nblocks
- int na1blocks
- int narticuls
- int neblocks
- int nopisp
- int opistkptr
- int nblksp
- int blkstkptr
- int DUMMYPADDING

3.64.1 Detailed Description

Definition at line 1002 of file grcc.h.

3.64.2 Constructor & Destructor Documentation

3.64.2.1 MConn()

```
MConn::MConn (
            int nnod,
            int nedg )
```

Definition at line 5534 of file grcc.cc.

3.64.2.2 ~MConn()

```
MConn::~MConn (
            void  )
```

Definition at line 5557 of file grcc.cc.

3.64.3 Member Function Documentation

3.64.3.1 init()

```
void MConn::init (
            void  )
```

Definition at line 5572 of file grcc.cc.

3.64.3.2 initCEdges()

```
void MConn::initCEdges (
            MGraph * mg )
```

Definition at line 5629 of file grcc.cc.

3.64.3.3 pushNode()

```
void MConn::pushNode (
            int nd )
```

Definition at line 5603 of file grcc.cc.

3.64.3.4 pushEdge()

```
void MConn::pushEdge (
            int n0,
            int n1 )
```

Definition at line 5612 of file grcc.cc.

3.64.3.5 addCEdge()

```
void MConn::addCEdge (
            int n0,
            int n1,
            ULong momset )
```

Definition at line 5655 of file grcc.cc.

3.64.3.6 addOPIc()

```
void MConn::addOPIc (
            MCOpi * mopi,
            int stp )
```

Definition at line 5679 of file grcc.cc.

3.64.3.7 addBridge()

```
void MConn::addBridge (
            int n0,
            int n1,
            int nex,
            int nextot )
```

Definition at line 5706 of file grcc.cc.

3.64.3.8 addArtic()

```
void MConn::addArtic (
            int nd,
            int mul )
```

Definition at line 5723 of file grcc.cc.

3.64.3.9 addBlock()

```
void MConn::addBlock (
            MCBlock * eblk,
            int stp )
```

Definition at line 5734 of file grcc.cc.

3.64.3.10 addBlockSelf()

```
void MConn::addBlockSelf (
            int nd,
            int mul )
```

Definition at line 5758 of file grcc.cc.

3.64.3.11 print()

```
void MConn::print (
            void  )
```

Definition at line 5776 of file grcc.cc.

3.64.3.12 prEdges()

```
void MConn::prEdges (
            void  )
```

Definition at line 5849 of file grcc.cc.

3.64.4 Field Documentation

3.64.4.1 cedges

```
MCEdge* MConn::cedges
```

Definition at line 1005 of file grcc.h.

3.64.4.2 opics

```
MCOpi* MConn::opics
```

Definition at line 1006 of file grcc.h.

3.64.4.3 bridges

```
MCBridge* MConn::bridges
```

Definition at line 1007 of file grcc.h.

3.64.4.4 blocks

```
MCBlock* MConn::blocks
```

Definition at line 1008 of file grcc.h.

3.64.4.5 articuls

```
int* MConn::articuls
```

Definition at line 1009 of file grcc.h.

3.64.4.6 opisp

```
int* MConn::opisp
```

Definition at line 1011 of file grcc.h.

3.64.4.7 opistk

```
int* MConn::opistk
```

Definition at line 1012 of file grcc.h.

3.64.4.8 blksp

```
Edge2n* MConn::blksp
```

Definition at line 1013 of file grcc.h.

3.64.4.9 blkstk

```
Edge2n* MConn::blkstk
```

Definition at line 1014 of file grcc.h.

3.64.4.10 snodes

```
int MConn::snodes
```

Definition at line 1015 of file grcc.h.

3.64.4.11 sedges

```
int MConn::sedges
```

Definition at line 1016 of file grcc.h.

3.64.4.12 nopic

```
int MConn::nopic
```

Definition at line 1019 of file grcc.h.

3.64.4.13 nlpopic

```
int MConn::nlpopic
```

Definition at line 1020 of file grcc.h.

3.64.4.14 nctopic

```
int MConn::nctopic
```

Definition at line 1021 of file grcc.h.

3.64.4.15 nbacked

```
int MConn::nbacked
```

Definition at line 1024 of file grcc.h.

3.64.4.16 nbridges

```
int MConn::nbridges
```

Definition at line 1027 of file grcc.h.

3.64.4.17 ne0bridges

```
int MConn::ne0bridges
```

Definition at line 1028 of file grcc.h.

3.64.4.18 ne1bridges

```
int MConn::ne1bridges
```

Definition at line 1029 of file grcc.h.

3.64.4.19 nselfloops

```
int MConn::nselfloops
```

Definition at line 1030 of file grcc.h.

3.64.4.20 nmultiedges

```
int MConn::nmultiedges
```

Definition at line 1031 of file grcc.h.

3.64.4.21 nblocks

```
int MConn::nblocks
```

Definition at line 1034 of file grcc.h.

3.64.4.22 na1blocks

```
int MConn::na1blocks
```

Definition at line 1035 of file grcc.h.

3.64.4.23 narticuls

```
int MConn::narticuls
```

Definition at line 1036 of file grcc.h.

3.64.4.24 neblocks

```
int MConn::neblocks
```

Definition at line 1037 of file grcc.h.

3.64.4.25 nopisp

```
int MConn::nopisp
```

Definition at line 1040 of file grcc.h.

3.64.4.26 opistkptr

```
int MConn::opistkptr
```

Definition at line 1041 of file grcc.h.

3.64.4.27 nblksp

```
int MConn::nblksp
```

Definition at line 1042 of file grcc.h.

3.64.4.28 blkstkptr

```
int MConn::blkstkptr
```

Definition at line 1043 of file grcc.h.

3.64.4.29 DUMMYPADDING

```
int MConn::DUMMYPADDING
```

Definition at line 1045 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.65 MCOpi Class Reference

Public Member Functions

- void init (void)

Data Fields

- int * nodes
- int nnodes
- int nlegs
- int next
- int nedges
- int loop
- int ctloop
- int mom0lg
- int DUMMYPADDING

3.65.1 Detailed Description

Definition at line 951 of file grcc.h.

3.65.2 Constructor & Destructor Documentation

3.65.2.1 MCOpi()

```
MCOpi::MCOpi (
            void  )
```

Definition at line 5473 of file grcc.cc.

3.65.2.2 ~MCOpi()

```
MCOpi::~MCOpi (
            void  )
```

Definition at line 5479 of file grcc.cc.

3.65.3 Member Function Documentation

3.65.3.1 init()

```
void MCOpi::init (
            void  )
```

Definition at line 5485 of file grcc.cc.

3.65.4 Field Documentation

3.65.4.1 nodes

```
int* MCOpi::nodes
```

Definition at line 953 of file grcc.h.

3.65.4.2 nnodes

```
int MCOpi::nnodes
```

Definition at line 954 of file grcc.h.

3.65.4.3 nlegs

```
int MCOpi::nlegs
```

Definition at line 955 of file grcc.h.

3.65.4.4 next

```
int MCOpi::next
```

Definition at line 956 of file grcc.h.

3.65.4.5 nedges

```
int MCOpi::nedges
```

Definition at line 957 of file grcc.h.

3.65.4.6 loop

```
int MCOpi::loop
```

Definition at line 958 of file grcc.h.

3.65.4.7 ctloop

```
int MCOpi::ctloop
```

Definition at line 959 of file grcc.h.

3.65.4.8 mom0lg

```
int MCOpi::mom0lg
```

Definition at line 960 of file grcc.h.

3.65.4.9 DUMMYPADDING

```
int MCOpi::DUMMYPADDING
```

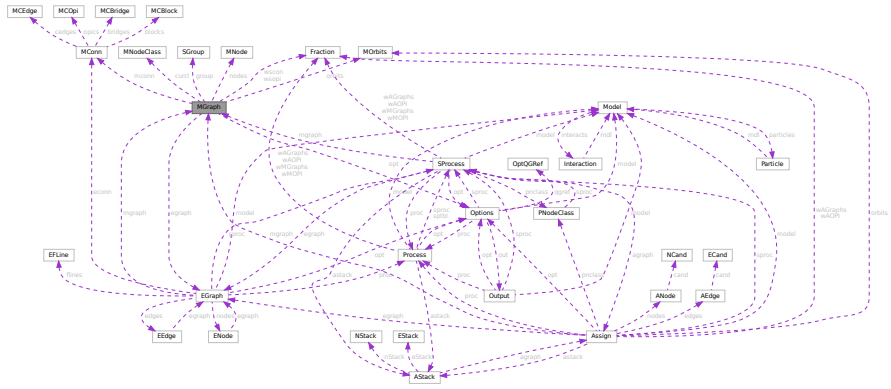
Definition at line 962 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.66 MGraph Class Reference

Collaboration diagram for MGraph:



Public Member Functions

- MGraph (int pid, int ncl, NCInput ∗mgi, Options ∗opt)
- MGraph (int pid, int ncl, int ∗cldeg, int ∗clnum, int ∗clexl, int ∗cmind, int ∗cmaxd, Options ∗opt)
- void init (void)
- BigInt generate (void)
- Bool isExternal (int nd)
- void printAdjMat (MNodeClass ∗cl)
- void print (void)
- void printPy (FILE ∗fp, long mId)
- Bool isConnected (void)
- Bool visit (int nd)
- Bool isIsomorphic (MNodeClass ∗cl)
- void permMat (int size, int ∗perm, int ∗∗mat0, int ∗∗mat1)
- int compMat (int size, int ∗∗mat0, int ∗∗mat1)
- MNodeClass ∗ refineClass (MNodeClass ∗cl)
- void bisearchME (int nd, int pd, int ned, int col, MCOpi ∗mopi, MCBlock ∗mblk, ULong ∗momset, int ∗next, int ∗nart)
- void biconnME (void)
- Bool isOptM (void)
- void connectClass (MNodeClass ∗cl)
- void connectNode (int sc, int ss, MNodeClass ∗cl)
- void connectLeg (int sc, int sn, int tc, int ts, MNodeClass ∗cl)
- void newGraph (MNodeClass ∗cl)

Data Fields

- Options ∗ opt
- SGroup ∗ group
- MOrbits ∗ orbits
- MNode ∗∗ nodes
- BigInt mId
- int ∗ clist
- int pId

- int nNodes
- int nEdges
- int nLoops
- int nExtern
- int nClasses
- int mindeg
- int maxdeg
- int ** adjMat
- MNodeClass * curcl
- EGraph * egraph
- BigInt nsym
- BigInt esym
- BigInt cDiag
- BigInt c1PI
- BigInt cNoTadpole
- BigInt cNoTadBlock
- BigInt c1PINoTadBlock
- Fraction wscon
- Fraction wsopi
- BigInt ngen
- BigInt ngconn
- BigInt nCallRefine
- BigInt discardRefine
- BigInt discardDisc
- BigInt discardIso
- Bool opi
- Bool opiloop
- Bool extself
- Bool selfloop
- Bool multiedge
- Bool tadpole
- Bool tadblock
- Bool block
- Bool bipart
- int DUMMYPADDING1
- MConn * mconn
- int ** modmat
- int * bidef
- int * bilow
- int * bicol
- int bicount
- int DUMMYPADDING2

3.66.1 Detailed Description

Definition at line 793 of file grcc.h.

3.66.2 Constructor & Destructor Documentation

3.66.2.1 MGraph() [1/2]

```
MGraph::MGraph (
            int pid,
            int ncl,
            NCInput * mgi,
            Options * opt )
```

Definition at line 4074 of file grcc.cc.

3.66.2.2 MGraph() [2/2]

```
MGraph::MGraph (
            int pid,
            int ncl,
            int * cldeg,
            int * clnum,
            int * clexl,
            int * cmind,
            int * cmaxd,
            Options * opt )
```

Definition at line 4006 of file grcc.cc.

3.66.2.3 ~MGraph()

```
MGraph::~MGraph (
            void  )
```

Definition at line 4190 of file grcc.cc.

3.66.3 Member Function Documentation

3.66.3.1 init()

```
void MGraph::init (
            void  )
```

Definition at line 4146 of file grcc.cc.

3.66.3.2 generate()

```
BigInt MGraph::generate (
            void  )
```

Definition at line 4961 of file grcc.cc.

3.66.3.3 isExternal()

```
Bool MGraph::isExternal (
            int nd )  [inline]
```

Definition at line 876 of file grcc.h.

3.66.3.4 printAdjMat()

```
void MGraph::printAdjMat (
            MNodeClass * cl )
```

Definition at line 4220 of file grcc.cc.

3.66.3.5 print()

```
void MGraph::print (
            void  )
```

Definition at line 4239 of file grcc.cc.

3.66.3.6 printPy()

```
void MGraph::printPy (
            FILE * fp,
            long mId )
```

Definition at line 4263 of file grcc.cc.

3.66.3.7 isConnected()

```
Bool MGraph::isConnected (
            void  )
```

Definition at line 4294 of file grcc.cc.

3.66.3.8 visit()

```
Bool MGraph::visit (
            int nd )
```

Definition at line 4318 of file grcc.cc.

3.66.3.9 isIsomorphic()

```
Bool MGraph::isIsomorphic (
            MNodeClass * cl )
```

Definition at line 4343 of file grcc.cc.

3.66.3.10 permMat()

```
void MGraph::permMat (
            int size,
            int * perm,
            int ** mat0,
            int ** mat1 )
```

Definition at line 4401 of file grcc.cc.

3.66.3.11 compMat()

```
int MGraph::compMat (
            int size,
            int ** mat0,
            int ** mat1 )
```

Definition at line 4415 of file grcc.cc.

3.66.3.12 refineClass()

```
MNodeClass * MGraph::refineClass (
            MNodeClass * cl )
```

Definition at line 4433 of file grcc.cc.

3.66.3.13 bisearchME()

```
void MGraph::bisearchME (
            int nd,
            int pd,
            int ned,
            int col,
            MCOpi * mopi,
            MCBlock * mblk,
            ULong * momset,
            int * next,
            int * nart )
```

Definition at line 4640 of file grcc.cc.

3.66.3.14 biconnME()

```
void MGraph::biconnME (
            void  )
```

Definition at line 4515 of file grcc.cc.

3.66.3.15 isOptM()

```
Bool MGraph::isOptM (
            void  )
```

Definition at line 5172 of file grcc.cc.

3.66.3.16 connectClass()

```
void MGraph::connectClass (
            MNodeClass * cl )
```

Definition at line 4980 of file grcc.cc.

3.66.3.17 connectNode()

```
void MGraph::connectNode (
            int sc,
            int ss,
            MNodeClass * cl )
```

Definition at line 5008 of file grcc.cc.

3.66.3.18 connectLeg()

```
void MGraph::connectLeg (
            int sc,
            int sn,
            int tc,
            int ts,
            MNodeClass * cl )
```

Definition at line 5025 of file grcc.cc.

3.66.3.19 newGraph()

```
void MGraph::newGraph (
            MNodeClass * cl )
```

Definition at line 5238 of file grcc.cc.

3.66.4 Field Documentation

3.66.4.1 opt

```
Options* MGraph::opt
```

Definition at line 798 of file grcc.h.

3.66.4.2 group

```
SGroup* MGraph::group
```

Definition at line 801 of file grcc.h.

3.66.4.3 orbits

```
MOrbits* MGraph::orbits
```

Definition at line 802 of file grcc.h.

3.66.4.4 nodes

```
MNode** MGraph::nodes
```

Definition at line 804 of file grcc.h.

3.66.4.5 mId

```
BigInt MGraph::mId
```

Definition at line 805 of file grcc.h.

3.66.4.6 clist

```
int* MGraph::clist
```

Definition at line 806 of file grcc.h.

3.66.4.7 pId

```
int MGraph::pId
```

Definition at line 807 of file grcc.h.

3.66.4.8 nNodes

```
int MGraph::nNodes
```

Definition at line 808 of file grcc.h.

3.66.4.9 nEdges

```
int MGraph::nEdges
```

Definition at line 809 of file grcc.h.

3.66.4.10 nLoops

```
int MGraph::nLoops
```

Definition at line 810 of file grcc.h.

3.66.4.11 nExtern

```
int MGraph::nExtern
```

Definition at line 811 of file grcc.h.

3.66.4.12 nClasses

```
int MGraph::nClasses
```

Definition at line 812 of file grcc.h.

3.66.4.13 mindeg

```
int MGraph::mindeg
```

Definition at line 813 of file grcc.h.

3.66.4.14 maxdeg

```
int MGraph::maxdeg
```

Definition at line 814 of file grcc.h.

3.66.4.15 adjMat

```
int** MGraph::adjMat
```

Definition at line 817 of file grcc.h.

3.66.4.16 curcl

```
MNodeClass* MGraph::curcl
```

Definition at line 818 of file grcc.h.

3.66.4.17 egraph

```
EGraph* MGraph::egraph
```

Definition at line 819 of file grcc.h.

3.66.4.18 nsym

```
BigInt MGraph::nsym
```

Definition at line 820 of file grcc.h.

3.66.4.19 esym

```
BigInt MGraph::esym
```

Definition at line 821 of file grcc.h.

3.66.4.20 cDiag

```
BigInt MGraph::cDiag
```

Definition at line 824 of file grcc.h.

3.66.4.21 c1PI

```
BigInt MGraph::c1PI
```

Definition at line 825 of file grcc.h.

3.66.4.22 cNoTadpole

```
BigInt MGraph::cNoTadpole
```

Definition at line 826 of file grcc.h.

3.66.4.23 cNoTadBlock

```
BigInt MGraph::cNoTadBlock
```

Definition at line 827 of file grcc.h.

3.66.4.24 c1PINoTadBlock

```
BigInt MGraph::c1PINoTadBlock
```

Definition at line 828 of file grcc.h.

3.66.4.25 wscon

```
Fraction MGraph::wscon
```

Definition at line 829 of file grcc.h.

3.66.4.26 wsopi

```
Fraction MGraph::wsopi
```

Definition at line 830 of file grcc.h.

3.66.4.27 ngen

```
BigInt MGraph::ngen
```

Definition at line 833 of file grcc.h.

3.66.4.28 ngconn

```
BigInt MGraph::ngconn
```

Definition at line 834 of file grcc.h.

3.66.4.29 nCallRefine

```
BigInt MGraph::nCallRefine
```

Definition at line 836 of file grcc.h.

3.66.4.30 discardRefine

```
BigInt MGraph::discardRefine
```

Definition at line 837 of file grcc.h.

3.66.4.31 discardDisc

```
BigInt MGraph::discardDisc
```

Definition at line 838 of file grcc.h.

3.66.4.32 discardIso

```
BigInt MGraph::discardIso
```

Definition at line 839 of file grcc.h.

3.66.4.33 opi

```
Bool MGraph::opi
```

Definition at line 842 of file grcc.h.

3.66.4.34 opiloop

```
Bool MGraph::opiloop
```

Definition at line 843 of file grcc.h.

3.66.4.35 extself

```
Bool MGraph::extself
```

Definition at line 844 of file grcc.h.

3.66.4.36 selfloop

```
Bool MGraph::selfloop
```

Definition at line 845 of file grcc.h.

3.66.4.37 multiedge

```
Bool MGraph::multiedge
```

Definition at line 846 of file grcc.h.

3.66.4.38 tadpole

```
Bool MGraph::tadpole
```

Definition at line 847 of file grcc.h.

3.66.4.39 tadblock

```
Bool MGraph::tadblock
```

Definition at line 848 of file grcc.h.

3.66.4.40 block

```
Bool MGraph::block
```

Definition at line 849 of file grcc.h.

3.66.4.41 bipart

```
Bool MGraph::bipart
```

Definition at line 850 of file grcc.h.

3.66.4.42 DUMMYPADDING1

```
int MGraph::DUMMYPADDING1
```

Definition at line 852 of file grcc.h.

3.66.4.43 mconn

```
MConn* MGraph::mconn
```

Definition at line 855 of file grcc.h.

3.66.4.44 modmat

```
int** MGraph::modmat
```

Definition at line 858 of file grcc.h.

3.66.4.45 bidef

```
int* MGraph::bidef
```

Definition at line 861 of file grcc.h.

3.66.4.46 bilow

```
int* MGraph::bilow
```

Definition at line 862 of file grcc.h.

3.66.4.47 bicol

```
int* MGraph::bicol
```

Definition at line 863 of file grcc.h.

3.66.4.48 bicount

```
int MGraph::bicount
```

Definition at line 864 of file grcc.h.

3.66.4.49 DUMMYPADDING2

```
int MGraph::DUMMYPADDING2
```

Definition at line 866 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.67 MiNmAx Struct Reference

Data Fields

- WORD mini
- WORD maxi
- WORD size

3.67.1 Detailed Description

Definition at line 301 of file structs.h.

3.67.2 Field Documentation

3.67.2.1 mini

```
WORD MiNmAx::mini
```

Minimum value

Definition at line 302 of file structs.h.

3.67.2.2 maxi

```
WORD MiNmAx::maxi
```

Maximum value

Definition at line 303 of file structs.h.

3.67.2.3 size

```
WORD MiNmAx::size
```

Value of one unit in this position.

Definition at line 304 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.68 MInput Struct Reference

Data Fields

- const char * name
- int defpart
- int ncouple
- const char * cnamlist [GRCC_MAXNCPLG]

3.68.1 Detailed Description

Definition at line 175 of file grccparam.h.

3.68.2 Field Documentation

3.68.2.1 name

```
const char* MInput::name
```

Definition at line 176 of file grccparam.h.

3.68.2.2 defpart

```
int MInput::defpart
```

Definition at line 177 of file grccparam.h.

3.68.2.3 ncouple

```
int MInput::ncouple
```

Definition at line 179 of file grccparam.h.

3.68.2.4 cnamlist

```
const char* MInput::cnamlist[GRCC_MAXNCPLG]
```

Definition at line 180 of file grccparam.h.

The documentation for this struct was generated from the following file:

- grccparam.h

3.69 MNode Class Reference

Public Member Functions

- MNode (int id, int clss, NCInput *mgi)
- MNode (int id, int deg, int extloop, int clss, int cmind, int cmaxd)

Data Fields

- int id
- int deg
- int clss
- int extloop
- int cmindeg
- int cmaxdeg
- int freelg
- int visited

3.69.1 Detailed Description

Definition at line 771 of file grcc.h.

3.69.2 Constructor & Destructor Documentation

3.69.2.1 MNode() [1/2]

```
MNode::MNode (
            int id,
            int clss,
            NCInput * mgi )
```

Definition at line 3968 of file grcc.cc.

3.69.2.2 MNode() [2/2]

```
MNode::MNode (
            int id,
            int deg,
            int extloop,
            int clss,
            int cmind,
            int cmaxd )
```

Definition at line 3986 of file grcc.cc.

3.69.3 Field Documentation

3.69.3.1 id

```
int MNode::id
```

Definition at line 773 of file grcc.h.

3.69.3.2 deg

```
int MNode::deg
```

Definition at line 774 of file grcc.h.

3.69.3.3 clss

```
int MNode::clss
```

Definition at line 775 of file grcc.h.

3.69.3.4 extloop

```
int MNode::extloop
```

Definition at line 776 of file grcc.h.

3.69.3.5 cmindeg

```
int MNode::cmindeg
```

Definition at line 777 of file grcc.h.

3.69.3.6 cmaxdeg

```
int MNode::cmaxdeg
```

Definition at line 778 of file grcc.h.

3.69.3.7 freelg

```
int MNode::freelg
```

Definition at line 780 of file grcc.h.

3.69.3.8 visited

```
int MNode::visited
```

Definition at line 781 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.70 MNodeClass Class Reference

Public Member Functions

- MNodeClass (int nnodes, int nclasses)
- void init (int *cl, int mxdeg, int **adjmat)
- void copy (MNodeClass *mnc)
- int clCmp (int nd0, int nd1, int cn)
- void printMat (void)
- void mkFlist (void)
- void mkNdCl (void)
- void mkClMat (int **adjmat)
- void incMat (int nd, int td, int val)
- int cmpMNCArray (int *a0, int *a1, int ma)
- void reorder (MGraph *mg)

Data Fields

- int clmat [GRCC_MAXNODES][GRCC_MAXNODES]
- int clist [GRCC_MAXNODES]
- int ndcl [GRCC_MAXNODES]
- int flist [GRCC_MAXNODES+1]
- int clord [GRCC_MAXNODES]
- int cmindeg [GRCC_MAXNODES]
- int cmaxdeg [GRCC_MAXNODES]
- int nNodes
- int nClasses
- int maxdeg
- int flg0
- int flg1
- int flg2

3.70.1 Detailed Description

Definition at line 900 of file grcc.h.

3.70.2 Constructor & Destructor Documentation

3.70.2.1 MNodeClass()

```
MNodeClass::MNodeClass (
            int nnodes,
            int nclasses )
```

Definition at line 3721 of file grcc.cc.

3.70.2.2 ~MNodeClass()

```
MNodeClass::~MNodeClass (
            void  )
```

Definition at line 3750 of file grcc.cc.

3.70.3 Member Function Documentation

3.70.3.1 init()

```
void MNodeClass::init (
            int * cl,
            int mxdeg,
            int ** adjmat )
```

Definition at line 3757 of file grcc.cc.

3.70.3.2 copy()

```
void MNodeClass::copy (
            MNodeClass * mnc )
```

Definition at line 3781 of file grcc.cc.

3.70.3.3 clCmp()

```
int MNodeClass::clCmp (
            int nd0,
            int nd1,
            int cn )
```

Definition at line 3837 of file grcc.cc.

3.70.3.4 printMat()

```
void MNodeClass::printMat (
            void  )
```

Definition at line 3888 of file grcc.cc.

3.70.3.5 mkFlist()

```
void MNodeClass::mkFlist (
            void  )
```

Definition at line 3805 of file grcc.cc.

3.70.3.6 mkNdCl()

```
void MNodeClass::mkNdCl (
            void  )
```

Definition at line 3821 of file grcc.cc.

3.70.3.7 mkClMat()

```
void MNodeClass::mkClMat (
            int ** adjmat )
```

Definition at line 3859 of file grcc.cc.

3.70.3.8 incMat()

```
void MNodeClass::incMat (
            int nd,
            int td,
            int val )
```

Definition at line 3879 of file grcc.cc.

3.70.3.9 cmpMNCArray()

```
int MNodeClass::cmpMNCArray (
            int * a0,
            int * a1,
            int ma )
```

Definition at line 3919 of file grcc.cc.

3.70.3.10 reorder()

```
void MNodeClass::reorder (
            MGraph * mg )
```

Definition at line 3935 of file grcc.cc.

3.70.4 Field Documentation

3.70.4.1 clmat

```
int MNodeClass::clmat[GRCC_MAXNODES][GRCC_MAXNODES]
```

Definition at line 902 of file grcc.h.

3.70.4.2 clist

```
int MNodeClass::clist[GRCC_MAXNODES]
```

Definition at line 903 of file grcc.h.

3.70.4.3 ndcl

```
int MNodeClass::ndcl[GRCC_MAXNODES]
```

Definition at line 904 of file grcc.h.

3.70.4.4 flist

```
int MNodeClass::flist[GRCC_MAXNODES+1]
```

Definition at line 905 of file grcc.h.

3.70.4.5 clord

```
int MNodeClass::clord[GRCC_MAXNODES]
```

Definition at line 906 of file grcc.h.

3.70.4.6 cmindeg

```
int MNodeClass::cmindeg[GRCC_MAXNODES]
```

Definition at line 907 of file grcc.h.

3.70.4.7 cmaxdeg

```
int MNodeClass::cmaxdeg[GRCC_MAXNODES]
```

Definition at line 908 of file grcc.h.

3.70.4.8 nNodes

```
int MNodeClass::nNodes
```

Definition at line 909 of file grcc.h.

3.70.4.9 nClasses

```
int MNodeClass::nClasses
```

Definition at line 910 of file grcc.h.

3.70.4.10 maxdeg

```
int MNodeClass::maxdeg
```

Definition at line 911 of file grcc.h.

3.70.4.11 flg0

```
int MNodeClass::flg0
```

Definition at line 913 of file grcc.h.

3.70.4.12 flg1

```
int MNodeClass::flg1
```

Definition at line 914 of file grcc.h.

3.70.4.13 flg2

```
int MNodeClass::flg2
```

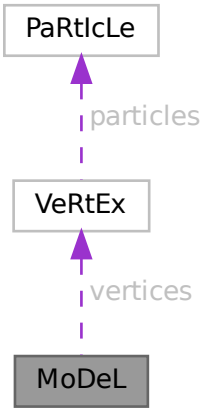
Definition at line 915 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.71 MoDeL Struct Reference

Collaboration diagram for MoDeL:



Data Fields

- VERTEX ** vertices
- WORD * couplings
- UBYTE * name
- void * grccmodel
- WORD legcouple [MAXLEGS+2]
- WORD nparticles
- WORD nvertices
- WORD invertices
- WORD sizevertices
- WORD sizecouplings
- WORD ncouplings
- WORD error
- WORD dummy

3.71.1 Detailed Description

Definition at line 636 of file structs.h.

3.71.2 Field Documentation

3.71.2.1 vertices

```
VERTEX** MoDeL::vertices
```

Definition at line 637 of file structs.h.

3.71.2.2 couplings

```
WORD* MoDeL::couplings
```

Definition at line 638 of file structs.h.

3.71.2.3 name

```
UBYTE* MoDeL::name
```

Definition at line 639 of file structs.h.

3.71.2.4 grccmodel

```
void* MoDeL::grccmodel
```

Definition at line 640 of file structs.h.

3.71.2.5 legcouple

```
WORD MoDeL::legcouple[MAXLEGS+2]
```

Definition at line 641 of file structs.h.

3.71.2.6 nparticles

```
WORD MoDeL::nparticles
```

Definition at line 642 of file structs.h.

3.71.2.7 nvertices

```
WORD MoDeL::nvertices
```

Definition at line 643 of file structs.h.

3.71.2.8 invertices

```
WORD MoDeL::invertices
```

Definition at line 644 of file structs.h.

3.71.2.9 sizevertices

```
WORD MoDeL::sizevertices
```

Definition at line 645 of file structs.h.

3.71.2.10 sizecouplings

```
WORD MoDeL::sizecouplings
```

Definition at line 646 of file structs.h.

3.71.2.11 ncouplings

```
WORD MoDeL::ncouplings
```

Definition at line 647 of file structs.h.

3.71.2.12 error

```
WORD MoDeL::error
```

Definition at line 648 of file structs.h.

3.71.2.13 dummy

```
WORD MoDeL::dummy
```

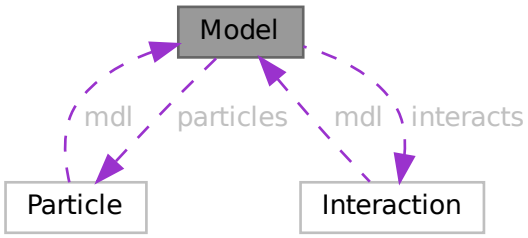
Definition at line 649 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.72 Model Class Reference

Collaboration diagram for Model:



Public Member Functions

- Model (MInput *minp)
- void prModel (void)
- void addParticle (PInput *pinp)
- void addParticleEnd (void)
- void addInteraction (IInput *iinp)
- void addInteractionEnd (void)
- int findParticleName (const char *name)
- int findParticleCode (int pcd)
- int findInteractionName (const char *name)
- int findInteractionCode (int icd)
- char * particleName (int p)
- int particleCode (int p)
- int normalParticle (int pt)
- int antiParticle (int pt)
- int * allParticles (int *len)
- int findMClass (const int cpl, const int dgr)
- void prParticleArray (int n, int *a, const char *msg)

Static Public Member Functions

- static void printMInput (MInput *min)
- static void printPInput (PInput *pin)
- static void printIInput (IInput *iin)

Data Fields

- char * name
- char ** cnlist
- int ncouple
- int nParticles
- Particle ** particles
- int pdef
- int nallPart
- int allPart [GRCC_MAXMPARTICLES2]
- int nintPart
- int intPart [GRCC_MAXMPARTICLES2]
- int nInteracts
- Interaction ** interacts
- int vdef
- int maxnlegs
- int maxcpl
- int maxloop
- int * cplgcp
- int * cplglg
- int * cplgnvl
- int ** cplgvl
- int ncplgcp
- int defpart

3.72.1 Detailed Description

Definition at line 284 of file grcc.h.

3.72.2 Constructor & Destructor Documentation

3.72.2.1 Model()

```
Model::Model (
            MInput * minp )
```

Definition at line 1979 of file grcc.cc.

3.72.2.2 ~Model()

```
Model::~Model (
            void  )
```

Definition at line 2040 of file grcc.cc.

3.72.3 Member Function Documentation

3.72.3.1 prModel()

```
void Model::prModel (
            void  )
```

Definition at line 2082 of file grcc.cc.

3.72.3.2 addParticle()

```
void Model::addParticle (
            PInput * pinp )
```

Definition at line 2131 of file grcc.cc.

3.72.3.3 addParticleEnd()

```
void Model::addParticleEnd (
            void  )
```

Definition at line 2173 of file grcc.cc.

3.72.3.4 addInteraction()

```
void Model::addInteraction (
            IInput * iinp )
```

Definition at line 2211 of file grcc.cc.

3.72.3.5 addInteractionEnd()

```
void Model::addInteractionEnd (
            void  )
```

Definition at line 2340 of file grcc.cc.

3.72.3.6 findParticleName()

```
int Model::findParticleName (
            const char * name )
```

Definition at line 2410 of file grcc.cc.

3.72.3.7 findParticleCode()

```
int Model::findParticleCode (
            int pcd )
```

Definition at line 2428 of file grcc.cc.

3.72.3.8 findInteractionName()

```
int Model::findInteractionName (
            const char * name )
```

Definition at line 2547 of file grcc.cc.

3.72.3.9 findInteractionCode()

```
int Model::findInteractionCode (
            int icd )
```

Definition at line 2560 of file grcc.cc.

3.72.3.10 particleName()

```
char * Model::particleName (
            int p )
```

Definition at line 2443 of file grcc.cc.

3.72.3.11 particleCode()

```
int Model::particleCode (
            int p )
```

Definition at line 2463 of file grcc.cc.

3.72.3.12 normalParticle()

```
int Model::normalParticle (
            int pt )
```

Definition at line 2509 of file grcc.cc.

3.72.3.13 antiParticle()

```
int Model::antiParticle (
            int pt )
```

Definition at line 2526 of file grcc.cc.

3.72.3.14 allParticles()

```
int * Model::allParticles (
            int * len )
```

Definition at line 2540 of file grcc.cc.

3.72.3.15 findMClass()

```
int Model::findMClass (
            const int cpl,
            const int dgr )
```

Definition at line 2496 of file grcc.cc.

3.72.3.16 prParticleArray()

```
void Model::prParticleArray (
            int n,
            int * a,
            const char * msg )
```

Definition at line 2481 of file grcc.cc.

3.72.3.17 printMInput()

```
void Model::printMInput (
            MInput * min )  [static]
```

Definition at line 2573 of file grcc.cc.

3.72.3.18 printPInput()

```
void Model::printPInput (
            PInput * pin )  [static]
```

Definition at line 2593 of file grcc.cc.

3.72.3.19 printIInput()

```
void Model::printIInput (
            IInput * iin )  [static]
```

Definition at line 2601 of file grcc.cc.

3.72.4 Field Documentation

3.72.4.1 name

```
char* Model::name
```

Definition at line 286 of file grcc.h.

3.72.4.2 cnlist

```
char** Model::cnlist
```

Definition at line 287 of file grcc.h.

3.72.4.3 ncouple

```
int Model::ncouple
```

Definition at line 288 of file grcc.h.

3.72.4.4 nParticles

```
int Model::nParticles
```

Definition at line 290 of file grcc.h.

3.72.4.5 particles

```
Particle** Model::particles
```

Definition at line 291 of file grcc.h.

3.72.4.6 pdef

```
int Model::pdef
```

Definition at line 292 of file grcc.h.

3.72.4.7 nallPart

```
int Model::nallPart
```

Definition at line 295 of file grcc.h.

3.72.4.8 allPart

```
int Model::allPart[GRCC_MAXMPARTICLES2]
```

Definition at line 296 of file grcc.h.

3.72.4.9 nintPart

```
int Model::nintPart
```

Definition at line 299 of file grcc.h.

3.72.4.10 intPart

```
int Model::intPart[GRCC_MAXMPARTICLES2]
```

Definition at line 300 of file grcc.h.

3.72.4.11 nInteracts

```
int Model::nInteracts
```

Definition at line 302 of file grcc.h.

3.72.4.12 interacts

```
Interaction** Model::interacts
```

Definition at line 303 of file grcc.h.

3.72.4.13 vdef

```
int Model::vdef
```

Definition at line 304 of file grcc.h.

3.72.4.14 maxnlegs

```
int Model::maxnlegs
```

Definition at line 306 of file grcc.h.

3.72.4.15 maxcpl

```
int Model::maxcpl
```

Definition at line 307 of file grcc.h.

3.72.4.16 maxloop

```
int Model::maxloop
```

Definition at line 308 of file grcc.h.

3.72.4.17 cplgcp

```
int* Model::cplgcp
```

Definition at line 311 of file grcc.h.

3.72.4.18 cplglg

```
int* Model::cplglg
```

Definition at line 312 of file grcc.h.

3.72.4.19 cplgnvl

```
int* Model::cplgnvl
```

Definition at line 313 of file grcc.h.

3.72.4.20 cplgvl

```
int** Model::cplgvl
```

Definition at line 314 of file grcc.h.

3.72.4.21 ncplgcp

```
int Model::ncplgcp
```

Definition at line 315 of file grcc.h.

3.72.4.22 defpart

```
int Model::defpart
```

Definition at line 317 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.73 MODNUM Struct Reference

Data Fields

- UWORD * a
- UWORD * m
- WORD na
- WORD nm

3.73.1 Detailed Description

Definition at line 1278 of file structs.h.

3.73.2 Field Documentation

3.73.2.1 a

```
UWORD* MODNUM::a
```

Definition at line 1279 of file structs.h.

3.73.2.2 m

```
UWORD* MODNUM::m
```

Definition at line 1280 of file structs.h.

3.73.2.3 na

```
WORD MODNUM::na
```

Definition at line 1281 of file structs.h.

3.73.2.4 nm

```
WORD MODNUM::nm
```

Definition at line 1282 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.74 MoDoPtDoLlArS Struct Reference

Data Fields

- WORD number
- WORD type

3.74.1 Detailed Description

Definition at line 561 of file structs.h.

3.74.2 Field Documentation

3.74.2.1 number

```
WORD MoDoPtDoLlArS::number
```

Definition at line 565 of file structs.h.

3.74.2.2 type

```
WORD MoDoPtDoLlArS::type
```

Definition at line 566 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.75 monomial_larger Class Reference

Public Member Functions

- bool operator() (const WORD *a, const WORD *b)

3.75.1 Detailed Description

Definition at line 177 of file poly.h.

3.75.2 Constructor & Destructor Documentation

3.75.2.1 monomial_larger()

```
monomial_larger::monomial_larger ( )  [inline]
```

Definition at line 182 of file poly.h.

3.75.3 Member Function Documentation

3.75.3.1 operator()()

```
bool monomial_larger::operator() (
            const WORD * a,
            const WORD * b )  [inline]
```

Definition at line 188 of file poly.h.

The documentation for this class was generated from the following file:

- poly.h

3.76 MOrbits Class Reference

Public Member Functions

- void print (void)
- void initPerm (int nnodes)
- void fromPerm (int *perm)
- void toOrbits (void)

Data Fields

- int nOrbits
- int nNodes
- int nd2or [GRCC_MAXNODES]
- int or2nd [GRCC_MAXNODES]
- int flist [GRCC_MAXNODES+1]

3.76.1 Detailed Description

Definition at line 1067 of file grcc.h.

3.76.2 Constructor & Destructor Documentation

3.76.2.1 MOrbits()

```
MOrbits::MOrbits (
            void  )
```

Definition at line 5356 of file grcc.cc.

3.76.2.2 ~MOrbits()

```
MOrbits::~MOrbits (
            void  )
```

Definition at line 5363 of file grcc.cc.

3.76.3 Member Function Documentation

3.76.3.1 print()

```
void MOrbits::print (
            void  )
```

Definition at line 5368 of file grcc.cc.

3.76.3.2 initPerm()

```
void MOrbits::initPerm (
            int nnodes )
```

Definition at line 5381 of file grcc.cc.

3.76.3.3 fromPerm()

```
void MOrbits::fromPerm (
            int * perm )
```

Definition at line 5392 of file grcc.cc.

3.76.3.4 toOrbits()

```
void MOrbits::toOrbits (
            void  )
```

Definition at line 5431 of file grcc.cc.

3.76.4 Field Documentation

3.76.4.1 nOrbits

```
int MOrbits::nOrbits
```

Definition at line 1079 of file grcc.h.

3.76.4.2 nNodes

```
int MOrbits::nNodes
```

Definition at line 1080 of file grcc.h.

3.76.4.3 nd2or

```
int MOrbits::nd2or[GRCC_MAXNODES]
```

Definition at line 1081 of file grcc.h.

3.76.4.4 or2nd

```
int MOrbits::or2nd[GRCC_MAXNODES]
```

Definition at line 1082 of file grcc.h.

3.76.4.5 flist

```
int MOrbits::flist[GRCC_MAXNODES+1]
```

Definition at line 1083 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.77 flint::mpoly Class Reference

Public Member Functions

- mpoly (fmpz_mpoly_ctx_struct *ctx_in)
- void print (const string &text)

Data Fields

- fmpz_mpoly_t d

3.77.1 Detailed Description

Definition at line 103 of file flintinterface.h.

3.77.2 Constructor & Destructor Documentation

3.77.2.1 mpoly()

```
flint::mpoly::mpoly (
            fmpz_mpoly_ctx_struct * ctx_in )  [inline], [explicit]
```

Definition at line 108 of file flintinterface.h.

3.77.2.2 ~mpoly()

```
flint::mpoly::~mpoly ( )  [inline]
```

Definition at line 109 of file flintinterface.h.

3.77.3 Member Function Documentation

3.77.3.1 print()

```
void flint::mpoly::print (
            const string & text )  [inline]
```

Definition at line 110 of file flintinterface.h.

3.77.4 Field Documentation

3.77.4.1 d

```
fmpz_mpoly_t flint::mpoly::d
```

Definition at line 107 of file flintinterface.h.

The documentation for this class was generated from the following file:

- flintinterface.h

3.78 flint::mpoly_ctx Class Reference

Public Member Functions

- mpoly_ctx (int64_t nvars)

Data Fields

- fmpz_mpoly_ctx_t d

3.78.1 Detailed Description

Definition at line 126 of file flintinterface.h.

3.78.2 Constructor & Destructor Documentation

3.78.2.1 mpoly_ctx()

```
flint::mpoly_ctx::mpoly_ctx (
            int64_t nvars )  [inline], [explicit]
```

Definition at line 129 of file flintinterface.h.

3.78.2.2 ~mpoly_ctx()

```
flint::mpoly_ctx::~mpoly_ctx ( )  [inline]
```

Definition at line 130 of file flintinterface.h.

3.78.3 Field Documentation

3.78.3.1 d

```
fmpz_mpoly_ctx_t flint::mpoly_ctx::d
```

Definition at line 128 of file flintinterface.h.

The documentation for this class was generated from the following file:

- flintinterface.h

3.79 flint::mpoly_factor Class Reference

Public Member Functions

- mpoly_factor (fmpz_mpoly_ctx_struct *ctx_in)

Data Fields

- fmpz_mpoly_factor_t d

3.79.1 Detailed Description

Definition at line 116 of file flintinterface.h.

3.79.2 Constructor & Destructor Documentation

3.79.2.1 mpoly_factor()

```
flint::mpoly_factor::mpoly_factor (
            fmpz_mpoly_ctx_struct * ctx_in )  [inline], [explicit]
```

Definition at line 121 of file flintinterface.h.

3.79.2.2 ~mpoly_factor()

```
flint::mpoly_factor::~mpoly_factor ( )  [inline]
```

Definition at line 124 of file flintinterface.h.

3.79.3 Field Documentation

3.79.3.1 d

```
fmpz_mpoly_factor_t flint::mpoly_factor::d
```

Definition at line 120 of file flintinterface.h.

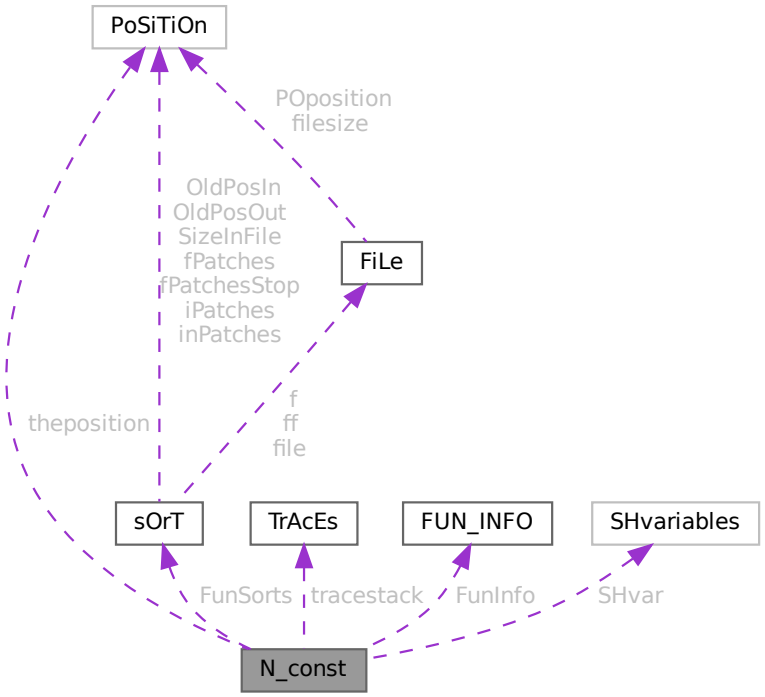
The documentation for this class was generated from the following file:

- flintinterface.h

3.80 N_const Struct Reference

`#include <structs.h>`

Collaboration diagram for N_const:



Data Fields

- POSITION theposition
- WORD * EndNest
- WORD * Frozen
- WORD * FullProto
- WORD * cTerm
- int * RepPoint
- WORD * WildValue
- WORD * WildStop
- WORD * argaddress
- WORD * RepFunList
- WORD * patstop
- WORD * terstop
- WORD * terstart
- WORD * terfirstcomm
- WORD * DumFound
- WORD ** DumPlace
- WORD ** DumFunPlace
- WORD * UsedSymbol

- WORD * UsedVector
- WORD * UsedIndex
- WORD * UsedFunction
- WORD * MaskPointer
- WORD * ForFindOnly
- WORD * findTerm
- WORD * findPattern
- WORD * dummyrenumlist
- int * funargs
- WORD ** funlocs
- int * funinds
- UWORD * NoScrat2
- WORD * ReplaceScrat
- TRACES * tracestack
- WORD * selecttermundo
- WORD * patternbuffer
- WORD * termbuffer
- WORD ** PoinScratch
- WORD ** FunScratch
- WORD * RenumScratch
- FUN_INFO * FunInfo
- WORD ** SplitScratch
- WORD ** SplitScratch1
- SORTING ** FunSorts
- UWORD * SoScratC
- WORD * listinprint
- WORD * currentTerm
- WORD ** arglist
- int * tlistbuf
- WORD * compressSpace
- UWORD * SHcombi
- WORD * poly_vars
- UWORD * cmod
- SHvariables SHvar
- LONG deferskipped
- LONG InScratch
- LONG SplitScratchSize
- LONG InScratch1
- LONG SplitScratchSize1
- LONG ninterms
- LONG SHcombisize
- int NumTotWildArgs
- int UseFindOnly
- int UsedOtherFind
- int ErrorInDollar
- int numfargs
- int numflocs
- int nargs
- int tohunt
- int numoffuns
- int funisize
- int RSsize
- int numtracesctack
- int intracestack
- int numfuninfo

- int NumFunSorts
- int MaxFunSorts
- int arglistsize
- int tlistsize
- int filenum
- int compressSize
- int polysortflag
- int nogroundlevel
- int subsubveto
- WORD MaxRenumScratch
- WORD oldtype
- WORD oldvalue
- WORD NumWild
- WORD RepFunNum
- WORD DisOrderFlag
- WORD WildDirt
- WORD NumFound
- WORD WildReserve
- WORD TeInFun
- WORD TeSuOut
- WORD WildArgs
- WORD WildEat
- WORD PolyNormFlag
- WORD PolyFunTodo
- WORD sizeselecttermundo
- WORD patternbuffersize
- WORD numlistinprint
- WORD ncmod
- WORD ExpectedSign
- WORD SignCheck
- WORD IndDum
- WORD poly_num_vars
- WORD idfunctionflag
- WORD poly_vars_type
- WORD tryterm

3.80.1 Detailed Description

The N_const struct is part of the global data and resides either in the ALLGLOBALS struct A, or the ALLPRIVATES struct B (TFORM) under the name N We see it used with the macro AN as in AN.RepFunNum It has variables that are private to each thread and are used as temporary storage during the expansion of the terms tree.

Definition at line 2206 of file structs.h.

3.80.2 Field Documentation

3.80.2.1 theposition

```
POSITION N_const::theposition
```

Definition at line 2207 of file structs.h.

3.80.2.2 EndNest

```
WORD* N_const::EndNest
```

Definition at line 2208 of file structs.h.

3.80.2.3 Frozen

```
WORD* N_const::Frozen
```

Definition at line 2209 of file structs.h.

3.80.2.4 FullProto

```
WORD* N_const::FullProto
```

Definition at line 2210 of file structs.h.

3.80.2.5 cTerm

```
WORD* N_const::cTerm
```

Definition at line 2211 of file structs.h.

3.80.2.6 RepPoint

```
int* N_const::RepPoint
```

Definition at line 2212 of file structs.h.

3.80.2.7 WildValue

```
WORD* N_const::WildValue
```

Definition at line 2213 of file structs.h.

3.80.2.8 WildStop

```
WORD* N_const::WildStop
```

Definition at line 2214 of file structs.h.

3.80.2.9 argaddress

```
WORD* N_const::argaddress
```

Definition at line 2215 of file structs.h.

3.80.2.10 RepFunList

```
WORD* N_const::RepFunList
```

Definition at line 2216 of file structs.h.

3.80.2.11 patstop

```
WORD* N_const::patstop
```

Definition at line 2217 of file structs.h.

3.80.2.12 terstop

```
WORD* N_const::terstop
```

Definition at line 2218 of file structs.h.

3.80.2.13 terstart

```
WORD* N_const::terstart
```

Definition at line 2219 of file structs.h.

3.80.2.14 terfirstcomm

```
WORD* N_const::terfirstcomm
```

Definition at line 2220 of file structs.h.

3.80.2.15 DumFound

```
WORD* N_const::DumFound
```

Definition at line 2221 of file structs.h.

3.80.2.16 DumPlace

```
WORD** N_const::DumPlace
```

Definition at line 2222 of file structs.h.

3.80.2.17 DumFunPlace

```
WORD** N_const::DumFunPlace
```

Definition at line 2223 of file structs.h.

3.80.2.18 UsedSymbol

```
WORD* N_const::UsedSymbol
```

Definition at line 2224 of file structs.h.

3.80.2.19 UsedVector

```
WORD* N_const::UsedVector
```

Definition at line 2225 of file structs.h.

3.80.2.20 UsedIndex

```
WORD* N_const::UsedIndex
```

Definition at line 2226 of file structs.h.

3.80.2.21 UsedFunction

```
WORD* N_const::UsedFunction
```

Definition at line 2227 of file structs.h.

3.80.2.22 MaskPointer

```
WORD* N_const::MaskPointer
```

Definition at line 2228 of file structs.h.

3.80.2.23 ForFindOnly

```
WORD* N_const::ForFindOnly
```

Definition at line 2229 of file structs.h.

3.80.2.24 findTerm

```
WORD* N_const::findTerm
```

Definition at line 2230 of file structs.h.

3.80.2.25 findPattern

```
WORD* N_const::findPattern
```

Definition at line 2231 of file structs.h.

3.80.2.26 dummyrenumlist

```
WORD* N_const::dummyrenumlist
```

Definition at line 2236 of file structs.h.

3.80.2.27 funargs

```
int* N_const::funargs
```

Definition at line 2237 of file structs.h.

3.80.2.28 funlocs

```
WORD** N_const::funlocs
```

Definition at line 2238 of file structs.h.

3.80.2.29 funinds

```
int* N_const::funinds
```

Definition at line 2239 of file structs.h.

3.80.2.30 NoScrat2

```
UWORD* N_const::NoScrat2
```

Definition at line 2240 of file structs.h.

3.80.2.31 ReplaceScrat

```
WORD* N_const::ReplaceScrat
```

Definition at line 2241 of file structs.h.

3.80.2.32 tracestack

```
TRACES* N_const::tracestack
```

Definition at line 2242 of file structs.h.

3.80.2.33 selecttermundo

```
WORD* N_const::selecttermundo
```

Definition at line 2243 of file structs.h.

3.80.2.34 patternbuffer

```
WORD* N_const::patternbuffer
```

Definition at line 2244 of file structs.h.

3.80.2.35 termbuffer

```
WORD* N_const::termbuffer
```

Definition at line 2245 of file structs.h.

3.80.2.36 PoinScratch

```
WORD** N_const::PoinScratch
```

Definition at line 2246 of file structs.h.

3.80.2.37 FunScratch

```
WORD** N_const::FunScratch
```

Definition at line 2247 of file structs.h.

3.80.2.38 RenumScratch

```
WORD* N_const::RenumScratch
```

Definition at line 2248 of file structs.h.

3.80.2.39 FunInfo

```
FUN_INFO* N_const::FunInfo
```

Definition at line 2249 of file structs.h.

3.80.2.40 SplitScratch

```
WORD** N_const::SplitScratch
```

Definition at line 2250 of file structs.h.

3.80.2.41 SplitScratch1

```
WORD** N_const::SplitScratch1
```

Definition at line 2251 of file structs.h.

3.80.2.42 FunSorts

```
SORTING** N_const::FunSorts
```

Definition at line 2252 of file structs.h.

3.80.2.43 SoScratC

```
UWORD* N_const::SoScratC
```

Definition at line 2253 of file structs.h.

3.80.2.44 listinprint

```
WORD* N_const::listinprint
```

Definition at line 2254 of file structs.h.

3.80.2.45 currentTerm

```
WORD* N_const::currentTerm
```

Definition at line 2255 of file structs.h.

3.80.2.46 arglist

```
WORD** N_const::arglist
```

Definition at line 2256 of file structs.h.

3.80.2.47 tlistbuf

```
int* N_const::tlistbuf
```

Definition at line 2257 of file structs.h.

3.80.2.48 compressSpace

```
WORD* N_const::compressSpace
```

Definition at line 2261 of file structs.h.

3.80.2.49 SHcombi

```
UWORD* N_const::SHcombi
```

Definition at line 2266 of file structs.h.

3.80.2.50 poly_vars

```
WORD* N_const::poly_vars
```

Definition at line 2267 of file structs.h.

3.80.2.51 cmod

```
UWORD* N_const::cmod
```

Definition at line 2268 of file structs.h.

3.80.2.52 SHvar

```
SHvariables N_const::SHvar
```

Definition at line 2269 of file structs.h.

3.80.2.53 deferskipped

```
LONG N_const::deferskipped
```

Definition at line 2270 of file structs.h.

3.80.2.54 InScratch

```
LONG N_const::InScratch
```

Definition at line 2271 of file structs.h.

3.80.2.55 SplitScratchSize

```
LONG N_const::SplitScratchSize
```

Definition at line 2272 of file structs.h.

3.80.2.56 InScratch1

```
LONG N_const::InScratch1
```

Definition at line 2273 of file structs.h.

3.80.2.57 SplitScratchSize1

```
LONG N_const::SplitScratchSize1
```

Definition at line 2274 of file structs.h.

3.80.2.58 ninterms

```
LONG N_const::ninterms
```

Definition at line 2275 of file structs.h.

3.80.2.59 SHcombisize

```
LONG N_const::SHcombisize
```

Definition at line 2284 of file structs.h.

3.80.2.60 NumTotWildArgs

```
int N_const::NumTotWildArgs
```

Definition at line 2285 of file structs.h.

3.80.2.61 UseFindOnly

```
int N_const::UseFindOnly
```

Definition at line 2286 of file structs.h.

3.80.2.62 UsedOtherFind

```
int N_const::UsedOtherFind
```

Definition at line 2287 of file structs.h.

3.80.2.63 ErrorInDollar

```
int N_const::ErrorInDollar
```

Definition at line 2288 of file structs.h.

3.80.2.64 numfargs

```
int N_const::numfargs
```

Definition at line 2289 of file structs.h.

3.80.2.65 numflocs

```
int N_const::numflocs
```

Definition at line 2290 of file structs.h.

3.80.2.66 nargs

```
int N_const::nargs
```

Definition at line 2291 of file structs.h.

3.80.2.67 tohunt

```
int N_const::tohunt
```

Definition at line 2292 of file structs.h.

3.80.2.68 numoffuns

```
int N_const::numoffuns
```

Definition at line 2293 of file structs.h.

3.80.2.69 funisize

```
int N_const::funisize
```

Definition at line 2294 of file structs.h.

3.80.2.70 RSsize

```
int N_const::RSsize
```

Definition at line 2295 of file structs.h.

3.80.2.71 numtracesctack

```
int N_const::numtracesctack
```

Definition at line 2296 of file structs.h.

3.80.2.72 intracestack

```
int N_const::intracestack
```

Definition at line 2297 of file structs.h.

3.80.2.73 numfuninfo

```
int N_const::numfuninfo
```

Definition at line 2298 of file structs.h.

3.80.2.74 NumFunSorts

```
int N_const::NumFunSorts
```

Definition at line 2299 of file structs.h.

3.80.2.75 MaxFunSorts

```
int N_const::MaxFunSorts
```

Definition at line 2300 of file structs.h.

3.80.2.76 arglistsize

```
int N_const::arglistsize
```

Definition at line 2301 of file structs.h.

3.80.2.77 tlistsize

```
int N_const::tlistsize
```

Definition at line 2302 of file structs.h.

3.80.2.78 filenum

```
int N_const::filenum
```

Definition at line 2303 of file structs.h.

3.80.2.79 compressSize

```
int N_const::compressSize
```

Definition at line 2304 of file structs.h.

3.80.2.80 polysortflag

```
int N_const::polysortflag
```

Definition at line 2305 of file structs.h.

3.80.2.81 nogroundlevel

```
int N_const::nogroundlevel
```

Definition at line 2306 of file structs.h.

3.80.2.82 subsubveto

```
int N_const::subsubveto
```

Definition at line 2307 of file structs.h.

3.80.2.83 MaxRenumScratch

```
WORD N_const::MaxRenumScratch
```

Definition at line 2308 of file structs.h.

3.80.2.84 oldtype

```
WORD N_const::oldtype
```

Definition at line 2309 of file structs.h.

3.80.2.85 oldvalue

```
WORD N_const::oldvalue
```

Definition at line 2310 of file structs.h.

3.80.2.86 NumWild

```
WORD N_const::NumWild
```

Definition at line 2311 of file structs.h.

3.80.2.87 RepFunNum

```
WORD N_const::RepFunNum
```

Definition at line 2312 of file structs.h.

3.80.2.88 DisOrderFlag

```
WORD N_const::DisOrderFlag
```

Definition at line 2313 of file structs.h.

3.80.2.89 WildDirt

```
WORD N_const::WildDirt
```

Definition at line 2314 of file structs.h.

3.80.2.90 NumFound

```
WORD N_const::NumFound
```

Definition at line 2315 of file structs.h.

3.80.2.91 WildReserve

```
WORD N_const::WildReserve
```

Definition at line 2316 of file structs.h.

3.80.2.92 TeInFun

```
WORD N_const::TeInFun
```

Definition at line 2317 of file structs.h.

3.80.2.93 TeSuOut

```
WORD N_const::TeSuOut
```

Definition at line 2318 of file structs.h.

3.80.2.94 WildArgs

```
WORD N_const::WildArgs
```

Definition at line 2319 of file structs.h.

3.80.2.95 WildEat

```
WORD N_const::WildEat
```

Definition at line 2320 of file structs.h.

3.80.2.96 PolyNormFlag

```
WORD N_const::PolyNormFlag
```

Definition at line 2321 of file structs.h.

3.80.2.97 PolyFunTodo

```
WORD N_const::PolyFunTodo
```

Definition at line 2322 of file structs.h.

3.80.2.98 sizeselecttermundo

```
WORD N_const::sizeselecttermundo
```

Definition at line 2323 of file structs.h.

3.80.2.99 patternbuffersize

```
WORD N_const::patternbuffersize
```

Definition at line 2324 of file structs.h.

3.80.2.100 numlistinprint

```
WORD N_const::numlistinprint
```

Definition at line 2325 of file structs.h.

3.80.2.101 ncmod

```
WORD N_const::ncmod
```

Definition at line 2326 of file structs.h.

3.80.2.102 ExpectedSign

```
WORD N_const::ExpectedSign
```

Definition at line 2327 of file structs.h.

3.80.2.103 SignCheck

```
WORD N_const::SignCheck
```

Used in pattern matching of antisymmetric functions

Definition at line 2328 of file structs.h.

3.80.2.104 IndDum

```
WORD N_const::IndDum
```

Used in pattern matching of antisymmetric functions

Definition at line 2329 of file structs.h.

3.80.2.105 poly_num_vars

```
WORD N_const::poly_num_vars
```

Definition at line 2330 of file structs.h.

3.80.2.106 idfunctionflag

```
WORD N_const::idfunctionflag
```

Definition at line 2331 of file structs.h.

3.80.2.107 poly_vars_type

```
WORD N_const::poly_vars_type
```

Definition at line 2332 of file structs.h.

3.80.2.108 tryterm

```
WORD N_const::tryterm
```

Definition at line 2333 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.81 nameblock Struct Reference

Data Fields

- MLONG previousblock
- MLONG position
- char names [NAMETABLESIZE]

3.81.1 Detailed Description

Definition at line 117 of file minos.h.

3.81.2 Field Documentation

3.81.2.1 previousblock

```
MLONG nameblock::previousblock
```

Definition at line 118 of file minos.h.

3.81.2.2 position

```
MLONG nameblock::position
```

Definition at line 119 of file minos.h.

3.81.2.3 names

```
char nameblock::names[NAMETABLESIZE]
```

Definition at line 120 of file minos.h.

The documentation for this struct was generated from the following file:

- minos.h

3.82 NaMeNode Struct Reference

```
#include <structs.h>
```

Data Fields

- LONG name
- WORD parent
- WORD left
- WORD right
- WORD balance
- WORD type
- WORD number

3.82.1 Detailed Description

The names of variables are kept in an array. Elements of type NAMENODE define a tree (that is kept balanced) that make it easy and fast to look for variables. See also NAMETREE.

Definition at line 243 of file structs.h.

3.82.2 Field Documentation

3.82.2.1 name

```
LONG NaMeNode::name
```

Offset into NAMETREE::namebuffer.

Definition at line 244 of file structs.h.

3.82.2.2 parent

```
WORD NaMeNode::parent
```

=-1 if no parent.

Definition at line 245 of file structs.h.

3.82.2.3 left

```
WORD NaMeNode::left
```

=-1 if no child.

Definition at line 246 of file structs.h.

3.82.2.4 right

```
WORD NaMeNode::right
```

=-1 if no child.

Definition at line 247 of file structs.h.

3.82.2.5 balance

```
WORD NaMeNode::balance
```

Used for the balancing of the tree.

Definition at line 248 of file structs.h.

3.82.2.6 type

```
WORD NaMeNode::type
```

Type associated with the name. See compiler types.

Definition at line 249 of file structs.h.

3.82.2.7 number

```
WORD NaMeNode::number
```

Number of variable in LIST's like for example C_const::SymbolList.

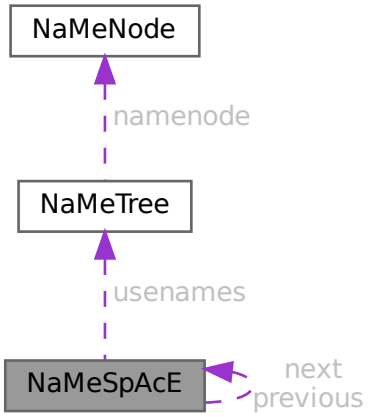
Definition at line 250 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.83 NaMeSpAcE Struct Reference

Collaboration diagram for NaMeSpAcE:



Data Fields

- struct NaMeSpAcE * previous
- struct NaMeSpAcE * next
- NAMETREE * usenames
- UBYTE * name

3.83.1 Detailed Description

Definition at line 657 of file structs.h.

3.83.2 Field Documentation

3.83.2.1 previous

```
struct NaMeSpAcE* NaMeSpAcE::previous
```

Definition at line 658 of file structs.h.

3.83.2.2 next

```
struct NaMeSpAcE* NaMeSpAcE::next
```

Definition at line 659 of file structs.h.

3.83.2.3 usenames

```
NAMETREE* NaMeSpAcE::usenames
```

Definition at line 660 of file structs.h.

3.83.2.4 name

```
UBYTE* NaMeSpAcE::name
```

Definition at line 661 of file structs.h.

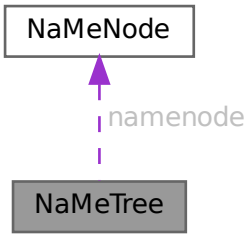
The documentation for this struct was generated from the following file:

- structs.h

3.84 NaMeTree Struct Reference

```
#include <structs.h>
```

Collaboration diagram for NaMeTree:



Data Fields

- NAMENODE * namenode
- UBYTE * namebuffer
- LONG nodesize
- LONG nodefill
- LONG namesize
- LONG namefill
- LONG oldnamefill
- LONG oldnodefill
- LONG globalnamefill
- LONG globalnodefill
- LONG clearnamefill
- LONG clearnodefill
- WORD headnode

3.84.1 Detailed Description

A struct of type NAMETREE controls a complete (balanced) tree of names for the compiler. The compiler maintains several of such trees and the system has been set up in such a way that one could define more of them if we ever want to work with local name spaces.

Definition at line 260 of file structs.h.

3.84.2 Field Documentation

3.84.2.1 namenode

```
NAMENODE* NaMeTree::namenode
```

[D] Vector of NAMENODE's. Number of elements is nodesize. =0 if no memory has been allocated.

Definition at line 261 of file structs.h.

3.84.2.2 namebuffer

```
UBYTE* NaMeTree::namebuffer
```

[D] Buffer that holds all the name strings referred to by the NAMENODE's. Allocation size is namesize. =0 if no memory has been allocated.

Definition at line 263 of file structs.h.

3.84.2.3 nodesize

```
LONG NaMeTree::nodesize
```

Maximum number of elements in namenode.

Definition at line 266 of file structs.h.

3.84.2.4 nodefill

```
LONG NaMeTree::nodefill
```

Number of currently used nodes in namenode.

Definition at line 267 of file structs.h.

3.84.2.5 namesize

```
LONG NaMeTree::namesize
```

Allocation size of namebuffer in bytes.

Definition at line 268 of file structs.h.

3.84.2.6 namefill

```
LONG NaMeTree::namefill
```

Number of bytes occupied.

Definition at line 269 of file structs.h.

3.84.2.7 oldnamefill

```
LONG NaMeTree::oldnamefill
```

UNUSED

Definition at line 270 of file structs.h.

3.84.2.8 oldnodefill

```
LONG NaMeTree::oldnodefill
```

UNUSED

Definition at line 271 of file structs.h.

3.84.2.9 globalnamefill

```
LONG NaMeTree::globalnamefill
```

Set by .global statement to the value of namefill. When a .store command is processed, this value will be used to reset namefill.

Definition at line 272 of file structs.h.

3.84.2.10 globalnodefill

```
LONG NaMeTree::globalnodefill
```

Same usage as globalnamefill, but for nodefill.

Definition at line 274 of file structs.h.

3.84.2.11 clearnamefill

```
LONG NaMeTree::clearnamefill
```

Marks the reset point used by the .clear statement.

Definition at line 275 of file structs.h.

3.84.2.12 clearnodefill

```
LONG NaMeTree::clearnodefill
```

Marks the reset point used by the .clear statement.

Definition at line 276 of file structs.h.

3.84.2.13 headnode

```
WORD NaMeTree::headnode
```

Offset in namenode of head node. =-1 if tree is empty.

Definition at line 277 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.85 NCand Class Reference

Public Member Functions

- NCand (const NCandSt sta, const int dega, const int nilst, int ∗ilst)
- void prNCand (const char ∗msg)

Data Fields

- int deg
- NCandSt st
- int nilist
- int ilist [GRCC_MAXMINTERACT]

3.85.1 Detailed Description

Definition at line 1106 of file grcc.h.

3.85.2 Constructor & Destructor Documentation

3.85.2.1 NCand()

```
NCand::NCand (
            const NCandSt sta,
            const int dega,
            const int nilst,
            int * ilst )
```

Definition at line 8147 of file grcc.cc.

3.85.2.2 ~NCand()

```
NCand::~NCand (
            void  )
```

Definition at line 8174 of file grcc.cc.

3.85.3 Member Function Documentation

3.85.3.1 prNCand()

```
void NCand::prNCand (
            const char * msg )
```

Definition at line 8179 of file grcc.cc.

3.85.4 Field Documentation

3.85.4.1 deg

```
int NCand::deg
```

Definition at line 1110 of file grcc.h.

3.85.4.2 st

```
NCandSt NCand::st
```

Definition at line 1111 of file grcc.h.

3.85.4.3 nilist

```
int NCand::nilist
```

Definition at line 1112 of file grcc.h.

3.85.4.4 ilist

```
int NCand::ilist[GRCC_MAXMINTERACT]
```

Definition at line 1113 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.86 NCInput Struct Reference

Data Fields

- int cldeg
- int clnum
- int cltyp
- int cmind
- int cmaxd
- int ptcl
- int cple

3.86.1 Detailed Description

Definition at line 205 of file grccparam.h.

3.86.2 Field Documentation

3.86.2.1 cldeg

```
int NCInput::cldeg
```

Definition at line 207 of file grccparam.h.

3.86.2.2 clnum

```
int NCInput::clnum
```

Definition at line 208 of file grccparam.h.

3.86.2.3 cltyp

```
int NCInput::cltyp
```

Definition at line 209 of file grccparam.h.

3.86.2.4 cmind

```
int NCInput::cmind
```

Definition at line 210 of file grccparam.h.

3.86.2.5 cmaxd

```
int NCInput::cmaxd
```

Definition at line 211 of file grccparam.h.

3.86.2.6 ptcl

```
int NCInput::ptcl
```

Definition at line 214 of file grccparam.h.

3.86.2.7 cple

```
int NCInput::cple
```

Definition at line 215 of file grccparam.h.

The documentation for this struct was generated from the following file:

- grccparam.h

3.87 NeStInG Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD * termsize
- WORD * funsize
- WORD * argsize

3.87.1 Detailed Description

The NESTING struct is used when we enter the argument of functions and there is the possibility that we have to change something there. Because functions can be nested we have to keep track of all levels of functions in case we have to move the outer layers to make room for a larger function argument.

Definition at line 1032 of file structs.h.

3.87.2 Field Documentation

3.87.2.1 termsize

```
WORD* NeStInG::termsize
```

Definition at line 1033 of file structs.h.

3.87.2.2 funsize

```
WORD* NeStInG::funsize
```

Definition at line 1034 of file structs.h.

3.87.2.3 argsize

```
WORD* NeStInG::argsize
```

Definition at line 1035 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.88 NoDe Struct Reference

Collaboration diagram for NoDe:

Data Fields

- struct NoDe * left
- struct NoDe * rght
- int lloser
- int rloser
- int lsrc
- int rsrc

3.88.1 Detailed Description

A node for the tree of losers in the final sorting on the master.

Definition at line 272 of file parallel.c.

3.88.2 Field Documentation

3.88.2.1 left

```
struct NoDe* NoDe::left
```

Definition at line 273 of file parallel.c.

3.88.2.2 rght

```
struct NoDe* NoDe::rght
```

Definition at line 274 of file parallel.c.

3.88.2.3 lloser

```
int NoDe::lloser
```

Definition at line 275 of file parallel.c.

3.88.2.4 rloser

```
int NoDe::rloser
```

Definition at line 276 of file parallel.c.

3.88.2.5 lsrc

```
int NoDe::lsrc
```

Definition at line 277 of file parallel.c.

3.88.2.6 rsrc

```
int NoDe::rsrc
```

Definition at line 278 of file parallel.c.

The documentation for this struct was generated from the following file:

- parallel.c

3.89 node Struct Reference

Collaboration diagram for node:

Public Member Functions

- node (const WORD *data)
- int cmp (const struct node *rhs) const
- bool operator() (const struct node *lhs, const struct node *rhs) const
- void calcHash ()

Data Fields

- const WORD * data
- struct node * l
- struct node * r
- WORD sign
- UWORD hash

3.89.1 Detailed Description

Definition at line 1495 of file optimize.cc.

3.89.2 Constructor & Destructor Documentation

3.89.2.1 node() [1/2]

```
node::node ( )  [inline]
```

Definition at line 1502 of file optimize.cc.

3.89.2.2 node() [2/2]

```
node::node (
            const WORD * data )  [inline]
```

Definition at line 1503 of file optimize.cc.

3.89.3 Member Function Documentation

3.89.3.1 cmp()

```
int node::cmp (
            const struct node * rhs ) const  [inline]
```

Definition at line 1507 of file optimize.cc.

3.89.3.2 operator()()

```
bool node::operator() (
            const struct node * lhs,
            const struct node * rhs ) const  [inline]
```

Definition at line 1531 of file optimize.cc.

3.89.3.3 calcHash()

```
void node::calcHash ( )  [inline]
```

Definition at line 1536 of file optimize.cc.

3.89.4 Field Documentation

3.89.4.1 data

```
const WORD* node::data
```

Definition at line 1496 of file optimize.cc.

3.89.4.2 l

```
struct node* node::l
```

Definition at line 1497 of file optimize.cc.

3.89.4.3 r

```
struct node* node::r
```

Definition at line 1498 of file optimize.cc.

3.89.4.4 sign

```
WORD node::sign
```

Definition at line 1499 of file optimize.cc.

3.89.4.5 hash

```
UWORD node::hash
```

Definition at line 1500 of file optimize.cc.

The documentation for this struct was generated from the following file:

- optimize.cc

3.90 NodeEq Struct Reference

Public Member Functions

- bool operator() (const NODE *lhs, const NODE *rhs) const

3.90.1 Detailed Description

Definition at line 1557 of file optimize.cc.

3.90.2 Member Function Documentation

3.90.2.1 operator()()

```
bool NodeEq::operator() (
            const NODE * lhs,
            const NODE * rhs ) const  [inline]
```

Definition at line 1558 of file optimize.cc.

The documentation for this struct was generated from the following file:

- optimize.cc

3.91 NodeHash Struct Reference

Public Member Functions

- size_t operator() (const NODE *n) const

3.91.1 Detailed Description

Definition at line 1551 of file optimize.cc.

3.91.2 Member Function Documentation

3.91.2.1 operator()()

```
size_t NodeHash::operator() (
            const NODE * n ) const  [inline]
```

Definition at line 1552 of file optimize.cc.

The documentation for this struct was generated from the following file:

- optimize.cc

3.92 NoRmDaTa Struct Reference

Data Fields

- WORD * psym
- WORD * pvec
- WORD * pdot
- WORD * pdel
- WORD * pind
- WORD ** peps
- WORD ** pden
- WORD ** pcom
- WORD ** pnco
- WORD ** pcon

3.92.1 Detailed Description

Definition at line 1393 of file structs.h.

3.92.2 Field Documentation

3.92.2.1 psym

```
WORD* NoRmDaTa::psym
```

Definition at line 1394 of file structs.h.

3.92.2.2 pvec

```
WORD* NoRmDaTa::pvec
```

Definition at line 1395 of file structs.h.

3.92.2.3 pdot

```
WORD* NoRmDaTa::pdot
```

Definition at line 1396 of file structs.h.

3.92.2.4 pdel

```
WORD* NoRmDaTa::pdel
```

Definition at line 1397 of file structs.h.

3.92.2.5 pind

```
WORD* NoRmDaTa::pind
```

Definition at line 1398 of file structs.h.

3.92.2.6 peps

```
WORD** NoRmDaTa::peps
```

Definition at line 1399 of file structs.h.

3.92.2.7 pden

```
WORD** NoRmDaTa::pden
```

Definition at line 1400 of file structs.h.

3.92.2.8 pcom

```
WORD** NoRmDaTa::pcom
```

Definition at line 1401 of file structs.h.

3.92.2.9 pnco

```
WORD** NoRmDaTa::pnco
```

Definition at line 1402 of file structs.h.

3.92.2.10 pcon

```
WORD** NoRmDaTa::pcon
```

Definition at line 1403 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.93 NStack Class Reference

Public Member Functions

- void print (const char *msg)

Data Fields

- int noden
- int deg
- NCandSt st
- int nilist
- int ilist [GRCC_MAXMINTERACT]

3.93.1 Detailed Description

Definition at line 1332 of file grcc.h.

3.93.2 Constructor & Destructor Documentation

3.93.2.1 NStack()

```
NStack::NStack ( )  [inline]
```

Definition at line 1342 of file grcc.h.

3.93.2.2 ~NStack()

```
NStack::~NStack ( )  [inline]
```

Definition at line 1343 of file grcc.h.

3.93.3 Member Function Documentation

3.93.3.1 print()

```
void NStack::print (
            const char * msg )
```

Definition at line 10226 of file grcc.cc.

3.93.4 Field Documentation

3.93.4.1 noden

```
int NStack::noden
```

Definition at line 1336 of file grcc.h.

3.93.4.2 deg

```
int NStack::deg
```

Definition at line 1337 of file grcc.h.

3.93.4.3 st

```
NCandSt NStack::st
```

Definition at line 1338 of file grcc.h.

3.93.4.4 nilist

```
int NStack::nilist
```

Definition at line 1339 of file grcc.h.

3.93.4.5 ilist

```
int NStack::ilist[GRCC_MAXMINTERACT]
```

Definition at line 1340 of file grcc.h.

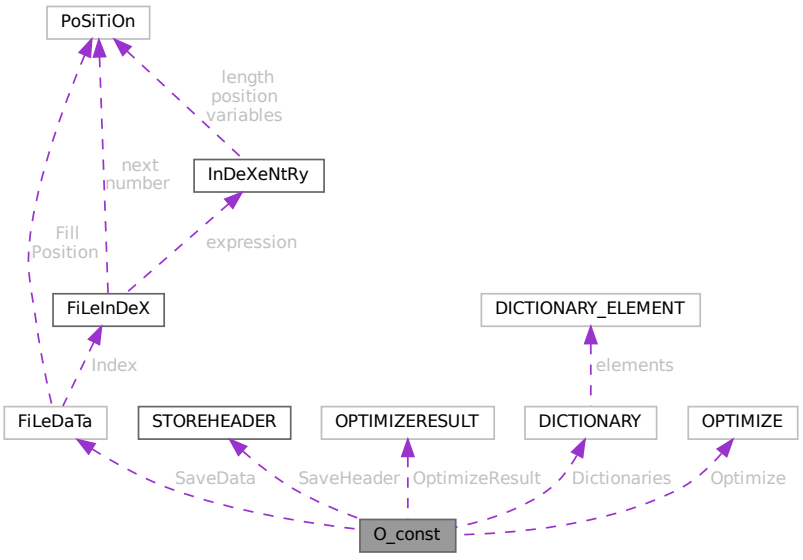
The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.94 O_const Struct Reference

```
#include <structs.h>
```

Collaboration diagram for O_const:



Data Fields

- FILEDATA SaveData
- STOREHEADER SaveHeader
- OPTIMIZERESULT OptimizeResult
- UBYTE * OutputLine
- UBYTE * OutStop
- UBYTE * OutFill
- WORD * bracket
- WORD * termbuf
- WORD * tabstring
- UBYTE * wpos
- UBYTE * wpoin
- UBYTE * DollarOutBuffer
- UBYTE * CurBufWrt
- void(* FlipWORD)(UBYTE *)
- void(* FlipLONG)(UBYTE *)
- void(* FlipPOS)(UBYTE *)
- void(* FlipPOINTER)(UBYTE *)
- void(* ResizeData)(UBYTE *, int, UBYTE *, int)
- void(* ResizeWORD)(UBYTE *, UBYTE *)
- void(* ResizeNCWORD)(UBYTE *, UBYTE *)
- void(* ResizeLONG)(UBYTE *, UBYTE *)
- void(* ResizePOS)(UBYTE *, UBYTE *)
- void(* ResizePOINTER)(UBYTE *, UBYTE *)
- void(* CheckPower)(UBYTE *)
- void(* RenumberVec)(UBYTE *)
- DICTIONARY ** Dictionaries
- UBYTE * tensorList
- WORD * inscheme
- LONG NumInBrack
- LONG wlen
- LONG DollarOutSizeBuffer
- LONG DollarInOutBuffer
- OPTIMIZE Optimize
- int OutInBuffer
- int NoSpacesInNumbers
- int BlockSpaces
- int CurrentDictionary
- int SizeDictionaries
- int NumDictionaries
- int CurDictNumbers
- int CurDictVariables
- int CurDictSpecials
- int CurDictFunWithArgs
- int CurDictNumberWarning
- int CurDictNotInFunctions
- int CurDictInDollars
- int gNumDictionaries
- int IndentSpace
- WORD schemenum
- WORD transFlag
- WORD powerFlag
- WORD mpower
- WORD resizeFlag

- WORD bufferedInd
- WORD OutSkip
- WORD IsBracket
- WORD InFbrack
- WORD PrintType
- WORD FortFirst
- WORD DoubleFlag
- WORD FactorMode
- WORD FactorNum
- WORD ErrorBlock
- WORD OptimizationLevel
- UBYTE FortDotChar

3.94.1 Detailed Description

The O_const struct is part of the global data and resides in the ALLGLOBALS struct A under the name O We see it used with the macro AO as in AO.OutputLine It contains variables that involve the writing of text output and save/store files.

Definition at line 2352 of file structs.h.

3.94.2 Field Documentation

3.94.2.1 SaveData

```
FILEDATA O_const::SaveData
```

Definition at line 2353 of file structs.h.

3.94.2.2 SaveHeader

```
STOREHEADER O_const::SaveHeader
```

Definition at line 2354 of file structs.h.

3.94.2.3 OptimizeResult

```
OPTIMIZERESULT O_const::OptimizeResult
```

Definition at line 2355 of file structs.h.

3.94.2.4 OutputLine

```
UBYTE* O_const::OutputLine
```

Definition at line 2356 of file structs.h.

3.94.2.5 OutStop

```
UBYTE* O_const::OutStop
```

Definition at line 2357 of file structs.h.

3.94.2.6 OutFill

```
UBYTE* O_const::OutFill
```

Definition at line 2358 of file structs.h.

3.94.2.7 bracket

```
WORD* O_const::bracket
```

Definition at line 2359 of file structs.h.

3.94.2.8 termbuf

```
WORD* O_const::termbuf
```

Definition at line 2360 of file structs.h.

3.94.2.9 tabstring

```
WORD* O_const::tabstring
```

Definition at line 2361 of file structs.h.

3.94.2.10 wpos

```
UBYTE* O_const::wpos
```

Definition at line 2362 of file structs.h.

3.94.2.11 wpoin

```
UBYTE* O_const::wpoin
```

Definition at line 2363 of file structs.h.

3.94.2.12 DollarOutBuffer

```
UBYTE* O_const::DollarOutBuffer
```

Definition at line 2364 of file structs.h.

3.94.2.13 CurBufWrt

```
UBYTE* O_const::CurBufWrt
```

Definition at line 2365 of file structs.h.

3.94.2.14 FlipWORD

```
void(* O_const::FlipWORD) (UBYTE *)
```

Definition at line 2366 of file structs.h.

3.94.2.15 FlipLONG

```
void(* O_const::FlipLONG) (UBYTE *)
```

Definition at line 2367 of file structs.h.

3.94.2.16 FlipPOS

```
void(* O_const::FlipPOS) (UBYTE *)
```

Definition at line 2368 of file structs.h.

3.94.2.17 FlipPOINTER

```
void(* O_const::FlipPOINTER) (UBYTE *)
```

Definition at line 2369 of file structs.h.

3.94.2.18 ResizeData

```
void(* O_const::ResizeData) (UBYTE *, int, UBYTE *, int)
```

Definition at line 2370 of file structs.h.

3.94.2.19 ResizeWORD

```
void(* O_const::ResizeWORD) (UBYTE *, UBYTE *)
```

Definition at line 2371 of file structs.h.

3.94.2.20 ResizeNCWORD

```
void(* O_const::ResizeNCWORD) (UBYTE *, UBYTE *)
```

Definition at line 2372 of file structs.h.

3.94.2.21 ResizeLONG

```
void(* O_const::ResizeLONG) (UBYTE *, UBYTE *)
```

Definition at line 2373 of file structs.h.

3.94.2.22 ResizePOS

```
void(* O_const::ResizePOS) (UBYTE *, UBYTE *)
```

Definition at line 2374 of file structs.h.

3.94.2.23 ResizePOINTER

```
void(* O_const::ResizePOINTER) (UBYTE *, UBYTE *)
```

Definition at line 2375 of file structs.h.

3.94.2.24 CheckPower

```
void(* O_const::CheckPower) (UBYTE *)
```

Definition at line 2376 of file structs.h.

3.94.2.25 RenumberVec

```
void(* O_const::RenumberVec) (UBYTE *)
```

Definition at line 2377 of file structs.h.

3.94.2.26 Dictionaries

```
DICTIONARY** O_const::Dictionaries
```

Definition at line 2378 of file structs.h.

3.94.2.27 tensorList

```
UBYTE* O_const::tensorList
```

Definition at line 2379 of file structs.h.

3.94.2.28 inscheme

```
WORD* O_const::inscheme
```

Definition at line 2380 of file structs.h.

3.94.2.29 NumInBrack

```
LONG O_const::NumInBrack
```

Definition at line 2386 of file structs.h.

3.94.2.30 wlen

```
LONG O_const::wlen
```

Definition at line 2387 of file structs.h.

3.94.2.31 DollarOutSizeBuffer

```
LONG O_const::DollarOutSizeBuffer
```

Definition at line 2388 of file structs.h.

3.94.2.32 DollarInOutBuffer

```
LONG O_const::DollarInOutBuffer
```

Definition at line 2389 of file structs.h.

3.94.2.33 Optimize

```
OPTIMIZE O_const::Optimize
```

Definition at line 2394 of file structs.h.

3.94.2.34 OutInBuffer

```
int O_const::OutInBuffer
```

Definition at line 2395 of file structs.h.

3.94.2.35 NoSpacesInNumbers

```
int O_const::NoSpacesInNumbers
```

Definition at line 2396 of file structs.h.

3.94.2.36 BlockSpaces

```
int O_const::BlockSpaces
```

Definition at line 2397 of file structs.h.

3.94.2.37 CurrentDictionary

```
int O_const::CurrentDictionary
```

Definition at line 2398 of file structs.h.

3.94.2.38 SizeDictionaries

```
int O_const::SizeDictionaries
```

Definition at line 2399 of file structs.h.

3.94.2.39 NumDictionaries

```
int O_const::NumDictionaries
```

Definition at line 2400 of file structs.h.

3.94.2.40 CurDictNumbers

```
int O_const::CurDictNumbers
```

Definition at line 2401 of file structs.h.

3.94.2.41 CurDictVariables

```
int O_const::CurDictVariables
```

Definition at line 2402 of file structs.h.

3.94.2.42 CurDictSpecials

```
int O_const::CurDictSpecials
```

Definition at line 2403 of file structs.h.

3.94.2.43 CurDictFunWithArgs

```
int O_const::CurDictFunWithArgs
```

Definition at line 2404 of file structs.h.

3.94.2.44 CurDictNumberWarning

```
int O_const::CurDictNumberWarning
```

Definition at line 2405 of file structs.h.

3.94.2.45 CurDictNotInFunctions

```
int O_const::CurDictNotInFunctions
```

Definition at line 2406 of file structs.h.

3.94.2.46 CurDictInDollars

```
int O_const::CurDictInDollars
```

Definition at line 2407 of file structs.h.

3.94.2.47 gNumDictionaries

```
int O_const::gNumDictionaries
```

Definition at line 2408 of file structs.h.

3.94.2.48 IndentSpace

```
int O_const::IndentSpace
```

Definition at line 2409 of file structs.h.

3.94.2.49 schemenum

```
WORD O_const::schemenum
```

Definition at line 2413 of file structs.h.

3.94.2.50 transFlag

```
WORD O_const::transFlag
```

Definition at line 2414 of file structs.h.

3.94.2.51 powerFlag

```
WORD O_const::powerFlag
```

Definition at line 2415 of file structs.h.

3.94.2.52 mpower

```
WORD O_const::mpower
```

Definition at line 2416 of file structs.h.

3.94.2.53 resizeFlag

```
WORD O_const::resizeFlag
```

Definition at line 2417 of file structs.h.

3.94.2.54 bufferedInd

```
WORD O_const::bufferedInd
```

Definition at line 2418 of file structs.h.

3.94.2.55 OutSkip

```
WORD O_const::OutSkip
```

Definition at line 2419 of file structs.h.

3.94.2.56 IsBracket

```
WORD O_const::IsBracket
```

Definition at line 2420 of file structs.h.

3.94.2.57 InFbrack

```
WORD O_const::InFbrack
```

Definition at line 2421 of file structs.h.

3.94.2.58 PrintType

```
WORD O_const::PrintType
```

Definition at line 2422 of file structs.h.

3.94.2.59 FortFirst

```
WORD O_const::FortFirst
```

Definition at line 2423 of file structs.h.

3.94.2.60 DoubleFlag

```
WORD O_const::DoubleFlag
```

Definition at line 2424 of file structs.h.

3.94.2.61 FactorMode

```
WORD O_const::FactorMode
```

Definition at line 2425 of file structs.h.

3.94.2.62 FactorNum

```
WORD O_const::FactorNum
```

Definition at line 2426 of file structs.h.

3.94.2.63 ErrorBlock

```
WORD O_const::ErrorBlock
```

Definition at line 2427 of file structs.h.

3.94.2.64 OptimizationLevel

```
WORD O_const::OptimizationLevel
```

Definition at line 2428 of file structs.h.

3.94.2.65 FortDotChar

```
UBYTE O_const::FortDotChar
```

Definition at line 2429 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.95 objects Struct Reference

Data Fields

- MLONG position
- MLONG size
- MLONG date
- MLONG tablenumber
- MLONG uncompressed
- MLONG spare1
- MLONG spare2
- MLONG spare3
- char element [ELEMENTSIZE]

3.95.1 Detailed Description

Definition at line 97 of file minos.h.

3.95.2 Field Documentation

3.95.2.1 position

```
MLONG objects::position
```

Definition at line 99 of file minos.h.

3.95.2.2 size

```
MLONG objects::size
```

Definition at line 100 of file minos.h.

3.95.2.3 date

```
MLONG objects::date
```

Definition at line 101 of file minos.h.

3.95.2.4 tablenumber

```
MLONG objects::tablenumber
```

Definition at line 102 of file minos.h.

3.95.2.5 uncompressed

```
MLONG objects::uncompressed
```

Definition at line 103 of file minos.h.

3.95.2.6 spare1

```
MLONG objects::spare1
```

Definition at line 104 of file minos.h.

3.95.2.7 spare2

```
MLONG objects::spare2
```

Definition at line 105 of file minos.h.

3.95.2.8 spare3

```
MLONG objects::spare3
```

Definition at line 106 of file minos.h.

3.95.2.9 element

```
char objects::element[ELEMENTSIZE]
```

Definition at line 107 of file minos.h.

The documentation for this struct was generated from the following file:

- minos.h

3.96 OptDef Struct Reference

Data Fields

- const char * name
- const char * mean
- int defaultv
- int DUMMYPADDING

3.96.1 Detailed Description

Definition at line 124 of file grccparam.h.

3.96.2 Field Documentation

3.96.2.1 name

```
const char* OptDef::name
```

Definition at line 125 of file grccparam.h.

3.96.2.2 mean

```
const char* OptDef::mean
```

Definition at line 126 of file grccparam.h.

3.96.2.3 defaultv

```
int OptDef::defaultv
```

Definition at line 127 of file grccparam.h.

3.96.2.4 DUMMYPADDING

```
int OptDef::DUMMYPADDING
```

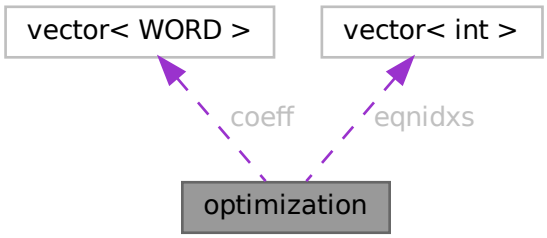
Definition at line 128 of file grccparam.h.

The documentation for this struct was generated from the following file:

- grccparam.h

3.97 optimization Class Reference

Collaboration diagram for optimization:



Public Member Functions

- bool operator< (const optimization &a) const

Data Fields

- int type
- int arg1
- int arg2
- int improve
- vector< WORD > coeff
- vector< int > eqnidxs

3.97.1 Detailed Description

class Optimization

3.97.2 Description

This object represents an optimization. Its type is a number in the range 0 to 5. Depending on this type, the variables arg1, arg2 and coeff indicate:

type==0 : optimization of the form x[arg1] ^ arg2 (coeff=empty) type==1 : optimization of the form x[arg1] * x[arg2] (coeff=empty) type==2 : optimization of the form x[arg1] * coeff (arg2=0) type==3 : optimization of the form x[arg1] + coeff (arg2=0) type==4 : optimization of the form x[arg1] + x[arg2] (coeff=empty) type==5 : optimization of the form x[arg1] - x[arg2] (coeff=empty)

Here, "x[arg]" represents a symbol (if positive) or an extrasymbol (if negative). The represented symbol's id is ABS(x[arg])-1.

"eqns" is a list of equation, where this optimization can be performed.

"improve" is the total improvement of this optimization.

Definition at line 2669 of file optimize.cc.

3.97.3 Member Function Documentation

3.97.3.1 operator<()

```
bool optimization::operator< (
            const optimization & a ) const  [inline]
```

Definition at line 2675 of file optimize.cc.

3.97.4 Field Documentation

3.97.4.1 type

```
int optimization::type
```

Definition at line 2671 of file optimize.cc.

3.97.4.2 arg1

```
int optimization::arg1
```

Definition at line 2671 of file optimize.cc.

3.97.4.3 arg2

```
int optimization::arg2
```

Definition at line 2671 of file optimize.cc.

3.97.4.4 improve

```
int optimization::improve
```

Definition at line 2671 of file optimize.cc.

3.97.4.5 coeff

```
vector<WORD> optimization::coeff
```

Definition at line 2672 of file optimize.cc.

3.97.4.6 eqnidxs

```
vector<int> optimization::eqnidxs
```

Definition at line 2673 of file optimize.cc.

The documentation for this class was generated from the following file:

- optimize.cc

3.98 OPTIMIZE Struct Reference

Data Fields

- union {
  float fval
  int ival [2]
  } **mctsconstant**

- int horner
- int hornerdirection
- int method
- int mctstimelimit
- int mctsnumexpand
- int mctsnumkeep
- int mctsnumrepeat
- int greedytimelimit
- int greedyminnum
- int greedymaxperc
- int printstats
- int debugflags
- int schemeflags
- int mctsdecaymode
- int salter
- union {
  float fval
  int ival [2]
  } **saMaxT**

- union {
  float fval
  int ival [2]
  } **saMinT**

- int spare

3.98.1 Detailed Description

Definition at line 1289 of file structs.h.

3.98.2 Field Documentation

3.98.2.1 fval

```
float OPTIMIZE::fval
```

Definition at line 1291 of file structs.h.

3.98.2.2 ival

```
int OPTIMIZE::ival[2]
```

Definition at line 1292 of file structs.h.

3.98.2.3 horner

```
int OPTIMIZE::horner
```

Definition at line 1294 of file structs.h.

3.98.2.4 hornerdirection

```
int OPTIMIZE::hornerdirection
```

Definition at line 1295 of file structs.h.

3.98.2.5 method

```
int OPTIMIZE::method
```

Definition at line 1296 of file structs.h.

3.98.2.6 mctstimelimit

```
int OPTIMIZE::mctstimelimit
```

Definition at line 1297 of file structs.h.

3.98.2.7 mctsnumexpand

```
int OPTIMIZE::mctsnumexpand
```

Definition at line 1298 of file structs.h.

3.98.2.8 mctsnumkeep

```
int OPTIMIZE::mctsnumkeep
```

Definition at line 1299 of file structs.h.

3.98.2.9 mctsnumrepeat

```
int OPTIMIZE::mctsnumrepeat
```

Definition at line 1300 of file structs.h.

3.98.2.10 greedytimelimit

```
int OPTIMIZE::greedytimelimit
```

Definition at line 1301 of file structs.h.

3.98.2.11 greedyminnum

```
int OPTIMIZE::greedyminnum
```

Definition at line 1302 of file structs.h.

3.98.2.12 greedymaxperc

```
int OPTIMIZE::greedymaxperc
```

Definition at line 1303 of file structs.h.

3.98.2.13 printstats

```
int OPTIMIZE::printstats
```

Definition at line 1304 of file structs.h.

3.98.2.14 debugflags

```
int OPTIMIZE::debugflags
```

Definition at line 1305 of file structs.h.

3.98.2.15 schemeflags

```
int OPTIMIZE::schemeflags
```

Definition at line 1306 of file structs.h.

3.98.2.16 mctsdecaymode

```
int OPTIMIZE::mctsdecaymode
```

Definition at line 1307 of file structs.h.

3.98.2.17 salter

```
int OPTIMIZE::saIter
```

Definition at line 1308 of file structs.h.

3.98.2.18 spare

```
int OPTIMIZE::spare
```

Definition at line 1317 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.99 OPTIMIZERESULT Struct Reference

Data Fields

- WORD * code
- UBYTE * nameofexpr
- LONG codesize
- WORD exprnr
- WORD minvar
- WORD maxvar

3.99.1 Detailed Description

Definition at line 1320 of file structs.h.

3.99.2 Field Documentation

3.99.2.1 code

```
WORD* OPTIMIZERESULT::code
```

Definition at line 1321 of file structs.h.

3.99.2.2 nameofexpr

```
UBYTE* OPTIMIZERESULT::nameofexpr
```

Definition at line 1322 of file structs.h.

3.99.2.3 codesize

```
LONG OPTIMIZERESULT::codesize
```

Definition at line 1323 of file structs.h.

3.99.2.4 exprnr

```
WORD OPTIMIZERESULT::exprnr
```

Definition at line 1324 of file structs.h.

3.99.2.5 minvar

```
WORD OPTIMIZERESULT::minvar
```

Definition at line 1325 of file structs.h.

3.99.2.6 maxvar

```
WORD OPTIMIZERESULT::maxvar
```

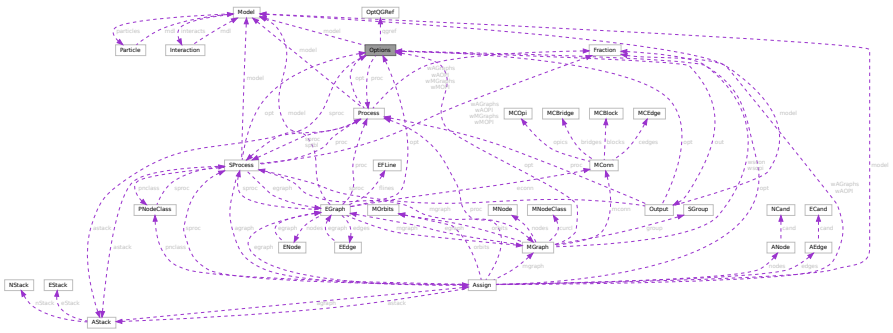
Definition at line 1326 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.100 Options Class Reference

Collaboration diagram for Options:



Public Member Functions

- void setDefaultValues (void)
- void setOldDefaultValues (void)
- void setValue (int ind, int val)
- int getValue (int ind)
- void setQGrafOpt (int *qgopt)
- void print (void)
- void setOutputF (Bool outgrf, const char *fname)
- void setOutputP (Bool outgrp, const char *fname)
- void printLevel (int l)
- void printModel (void)
- void setOutMG (OutEGB *omg, void *pt)
- void setOutAG (OutEG *oag, void *pt)
- void setEndMG (OutEGB *omg, void *pt)
- void setErExit (ErExit *ere, void *pt)
- const OptDef * getDef (void)
- const OptDef * getOldDef (void)
- const OptQGDef * getQGDef (void)
- void begin (Model *mdl)
- void end (void)
- void beginProc (Process *prc)
- void endProc (void)
- void beginSubProc (SProcess *sprc)
- void endSubProc (void)
- void newMGraph (MGraph *mgr)
- void newAGraph (EGraph *egr)
- void outModel (void)

Data Fields

- Model * model
- Process * proc
- SProcess * sproc
- Output * out
- OutEGB * outmg
- OutEGB * endmg
- OutEG * outag
- void * argmg
- void * argemg
- void * argag
- OptQGRef qgref [2 *GRCC_QGRAF_OPT_Size]
- int values [GRCC_OPT_Size]
- int qgopt [GRCC_QGRAF_OPT_Size]
- int nqgopt
- int DUMMYPADDING
- double time0
- double time1

3.100.1 Detailed Description

Definition at line 101 of file grcc.h.

3.100.2 Constructor & Destructor Documentation

3.100.2.1 Options()

```
Options::Options (
            void  )
```

Definition at line 200 of file grcc.cc.

3.100.2.2 ~Options()

```
Options::~Options (
            void  )
```

Definition at line 261 of file grcc.cc.

3.100.3 Member Function Documentation

3.100.3.1 setDefaultValues()

```
void Options::setDefaultValues (
            void  )
```

Definition at line 273 of file grcc.cc.

3.100.3.2 setOldDefaultValues()

```
void Options::setOldDefaultValues (
            void  )
```

Definition at line 289 of file grcc.cc.

3.100.3.3 setValue()

```
void Options::setValue (
            int ind,
            int val )
```

Definition at line 339 of file grcc.cc.

3.100.3.4 getValue()

```
int Options::getValue (
            int ind )
```

Definition at line 354 of file grcc.cc.

3.100.3.5 setQGrafOpt()

```
void Options::setQGrafOpt (
            int * qgopt )
```

Definition at line 368 of file grcc.cc.

3.100.3.6 print()

```
void Options::print (
            void  )
```

Definition at line 443 of file grcc.cc.

3.100.3.7 setOutputF()

```
void Options::setOutputF (
            Bool outgrf,
            const char * fname )
```

Definition at line 481 of file grcc.cc.

3.100.3.8 setOutputP()

```
void Options::setOutputP (
            Bool outgrp,
            const char * fname )
```

Definition at line 488 of file grcc.cc.

3.100.3.9 printLevel()

```
void Options::printLevel (
            int l )
```

Definition at line 333 of file grcc.cc.

3.100.3.10 printModel()

```
void Options::printModel (
            void  )
```

Definition at line 495 of file grcc.cc.

3.100.3.11 setOutMG()

```
void Options::setOutMG (
            OutEGB * omg,
            void * pt )
```

Definition at line 305 of file grcc.cc.

3.100.3.12 setOutAG()

```
void Options::setOutAG (
            OutEG * oag,
            void * pt )
```

Definition at line 319 of file grcc.cc.

3.100.3.13 setEndMG()

```
void Options::setEndMG (
            OutEGB * omg,
            void * pt )
```

Definition at line 312 of file grcc.cc.

3.100.3.14 setErExit()

```
void Options::setErExit (
            ErExit * ere,
            void * pt )
```

Definition at line 326 of file grcc.cc.

3.100.3.15 getDef()

```
const OptDef * Options::getDef (
            void  )
```

Definition at line 463 of file grcc.cc.

3.100.3.16 getOldDef()

```
const OptDef * Options::getOldDef (
            void  )
```

Definition at line 469 of file grcc.cc.

3.100.3.17 getQGDef()

```
const OptQGDef * Options::getQGDef (
            void  )
```

Definition at line 475 of file grcc.cc.

3.100.3.18 begin()

```
void Options::begin (
            Model * mdl )
```

Definition at line 516 of file grcc.cc.

3.100.3.19 end()

```
void Options::end (
            void  )
```

Definition at line 536 of file grcc.cc.

3.100.3.20 beginProc()

```
void Options::beginProc (
            Process * prc )
```

Definition at line 568 of file grcc.cc.

3.100.3.21 endProc()

```
void Options::endProc (
            void  )
```

Definition at line 584 of file grcc.cc.

3.100.3.22 beginSubProc()

```
void Options::beginSubProc (
            SProcess * sprc )
```

Definition at line 673 of file grcc.cc.

3.100.3.23 endSubProc()

```
void Options::endSubProc (
            void  )
```

Definition at line 694 of file grcc.cc.

3.100.3.24 newMGraph()

```
void Options::newMGraph (
            MGraph * mgr )
```

Definition at line 762 of file grcc.cc.

3.100.3.25 newAGraph()

```
void Options::newAGraph (
            EGraph * egr )
```

Definition at line 792 of file grcc.cc.

3.100.3.26 outModel()

```
void Options::outModel (
            void  )
```

Definition at line 507 of file grcc.cc.

3.100.4 Field Documentation

3.100.4.1 model

```
Model* Options::model
```

Definition at line 104 of file grcc.h.

3.100.4.2 proc

```
Process* Options::proc
```

Definition at line 105 of file grcc.h.

3.100.4.3 sproc

```
SProcess* Options::sproc
```

Definition at line 106 of file grcc.h.

3.100.4.4 out

```
Output* Options::out
```

Definition at line 107 of file grcc.h.

3.100.4.5 outmg

```
OutEGB* Options::outmg
```

Definition at line 108 of file grcc.h.

3.100.4.6 endmg

```
OutEGB* Options::endmg
```

Definition at line 109 of file grcc.h.

3.100.4.7 outag

```
OutEG* Options::outag
```

Definition at line 110 of file grcc.h.

3.100.4.8 argmg

```
void* Options::argmg
```

Definition at line 111 of file grcc.h.

3.100.4.9 argemg

```
void* Options::argemg
```

Definition at line 112 of file grcc.h.

3.100.4.10 argag

```
void* Options::argag
```

Definition at line 113 of file grcc.h.

3.100.4.11 qgref

```
OptQGRef Options::qgref[2 *GRCC_QGRAF_OPT_Size]
```

Definition at line 116 of file grcc.h.

3.100.4.12 values

```
int Options::values[GRCC_OPT_Size]
```

Definition at line 117 of file grcc.h.

3.100.4.13 qgopt

```
int Options::qgopt[GRCC_QGRAF_OPT_Size]
```

Definition at line 118 of file grcc.h.

3.100.4.14 nqgopt

```
int Options::nqgopt
```

Definition at line 120 of file grcc.h.

3.100.4.15 DUMMYPADDING

```
int Options::DUMMYPADDING
```

Definition at line 122 of file grcc.h.

3.100.4.16 time0

```
double Options::time0
```

Definition at line 125 of file grcc.h.

3.100.4.17 time1

```
double Options::time1
```

Definition at line 126 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.101 OptQGDef Struct Reference

Data Fields

- const char * name
- const char * cname
- const char * mean

3.101.1 Detailed Description

Definition at line 131 of file grccparam.h.

3.101.2 Field Documentation

3.101.2.1 name

```
const char* OptQGDef::name
```

Definition at line 132 of file grccparam.h.

3.101.2.2 cname

```
const char* OptQGDef::cname
```

Definition at line 133 of file grccparam.h.

3.101.2.3 mean

```
const char* OptQGDef::mean
```

Definition at line 134 of file grccparam.h.

The documentation for this struct was generated from the following file:

- grccparam.h

3.102 OptQGRef Struct Reference

Data Fields

- const char * name
- int index
- int sign

3.102.1 Detailed Description

Definition at line 137 of file grccparam.h.

3.102.2 Field Documentation

3.102.2.1 name

```
const char* OptQGRef::name
```

Definition at line 138 of file grccparam.h.

3.102.2.2 index

```
int OptQGRef::index
```

Definition at line 139 of file grccparam.h.

3.102.2.3 sign

```
int OptQGRef::sign
```

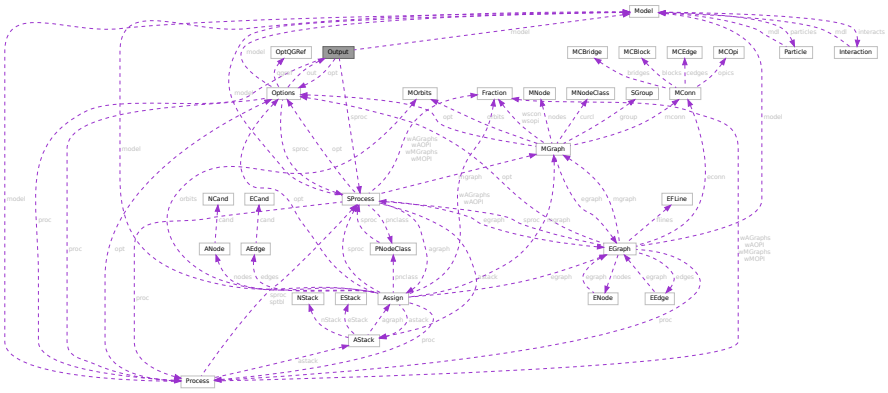
Definition at line 140 of file grccparam.h.

The documentation for this struct was generated from the following file:

- grccparam.h

3.103 Output Class Reference

Collaboration diagram for Output:



Public Member Functions

- Output (Options ∗optn)
- void setOutgrf (const char ∗fname)
- void setOutgrp (const char ∗fname)
- Bool outBeginF (Model ∗mdl, Bool pr)
- Bool outBeginP (Model ∗mdl, Bool pr)
- void outEndF (void)
- void outEndP (void)
- void outProcBeginF (Process ∗prc)
- void outProcBeginP (Process ∗prc)
- void outProcBegin0 (int next, int couple, int loop)
- void outSProcBeginF (SProcess ∗sprc)
- void outSProcBeginP (SProcess ∗sprc)
- void outProcEndF (void)
- void outProcEndP (void)
- void outEGraphF (EGraph ∗egraph)
- void outEGraphP (EGraph ∗egraph)
- void outModelF (void)
- void outModelP (void)

Data Fields

- Options * opt
- Model * model
- Process * proc
- SProcess * sproc
- char * outgrf
- FILE * outgrfp
- char * outgrp
- FILE * outgrpp
- int procId
- Bool outproc

3.103.1 Detailed Description

Definition at line 166 of file grcc.h.

3.103.2 Constructor & Destructor Documentation

3.103.2.1 Output()

```
Output::Output (
            Options * optn )
```

Definition at line 832 of file grcc.cc.

3.103.2.2 ~Output()

```
Output::~Output (
            void  )
```

Definition at line 848 of file grcc.cc.

3.103.3 Member Function Documentation

3.103.3.1 setOutgrf()

```
void Output::setOutgrf (
            const char * fname )
```

Definition at line 879 of file grcc.cc.

3.103.3.2 setOutgrp()

```
void Output::setOutgrp (
            const char * fname )
```

Definition at line 897 of file grcc.cc.

3.103.3.3 outBeginF()

```
Bool Output::outBeginF (
            Model * mdl,
            Bool pr )
```

Definition at line 914 of file grcc.cc.

3.103.3.4 outBeginP()

```
Bool Output::outBeginP (
            Model * mdl,
            Bool pr )
```

Definition at line 960 of file grcc.cc.

3.103.3.5 outEndF()

```
void Output::outEndF (
            void  )
```

Definition at line 999 of file grcc.cc.

3.103.3.6 outEndP()

```
void Output::outEndP (
            void  )
```

Definition at line 1015 of file grcc.cc.

3.103.3.7 outProcBeginF()

```
void Output::outProcBeginF (
            Process * prc )
```

Definition at line 1031 of file grcc.cc.

3.103.3.8 outProcBeginP()

```
void Output::outProcBeginP (
            Process * prc )
```

Definition at line 1076 of file grcc.cc.

3.103.3.9 outProcBegin0()

```
void Output::outProcBegin0 (
            int next,
            int couple,
            int loop )
```

Definition at line 1098 of file grcc.cc.

3.103.3.10 outSProcBeginF()

```
void Output::outSProcBeginF (
            SProcess * sprc )
```

Definition at line 1135 of file grcc.cc.

3.103.3.11 outSProcBeginP()

```
void Output::outSProcBeginP (
            SProcess * sprc )
```

Definition at line 1190 of file grcc.cc.

3.103.3.12 outProcEndF()

```
void Output::outProcEndF (
            void  )
```

Definition at line 1210 of file grcc.cc.

3.103.3.13 outProcEndP()

```
void Output::outProcEndP (
            void  )
```

Definition at line 1229 of file grcc.cc.

3.103.3.14 outEGraphF()

```
void Output::outEGraphF (
            EGraph * egraph )
```

Definition at line 1238 of file grcc.cc.

3.103.3.15 outEGraphP()

```
void Output::outEGraphP (
            EGraph * egraph )
```

Definition at line 1378 of file grcc.cc.

3.103.3.16 outModelF()

```
void Output::outModelF (
            void  )
```

Definition at line 1491 of file grcc.cc.

3.103.3.17 outModelP()

```
void Output::outModelP (
            void  )
```

Definition at line 1572 of file grcc.cc.

3.103.4 Field Documentation

3.103.4.1 opt

```
Options* Output::opt
```

Definition at line 169 of file grcc.h.

3.103.4.2 model

```
Model* Output::model
```

Definition at line 170 of file grcc.h.

3.103.4.3 proc

```
Process* Output::proc
```

Definition at line 171 of file grcc.h.

3.103.4.4 sproc

```
SProcess* Output::sproc
```

Definition at line 172 of file grcc.h.

3.103.4.5 outgrf

```
char* Output::outgrf
```

Definition at line 173 of file grcc.h.

3.103.4.6 outgrfp

```
FILE* Output::outgrfp
```

Definition at line 174 of file grcc.h.

3.103.4.7 outgrp

```
char* Output::outgrp
```

Definition at line 175 of file grcc.h.

3.103.4.8 outgrpp

```
FILE* Output::outgrpp
```

Definition at line 176 of file grcc.h.

3.103.4.9 procId

```
int Output::procId
```

Definition at line 177 of file grcc.h.

3.103.4.10 outproc

```
Bool Output::outproc
```

Definition at line 178 of file grcc.h.

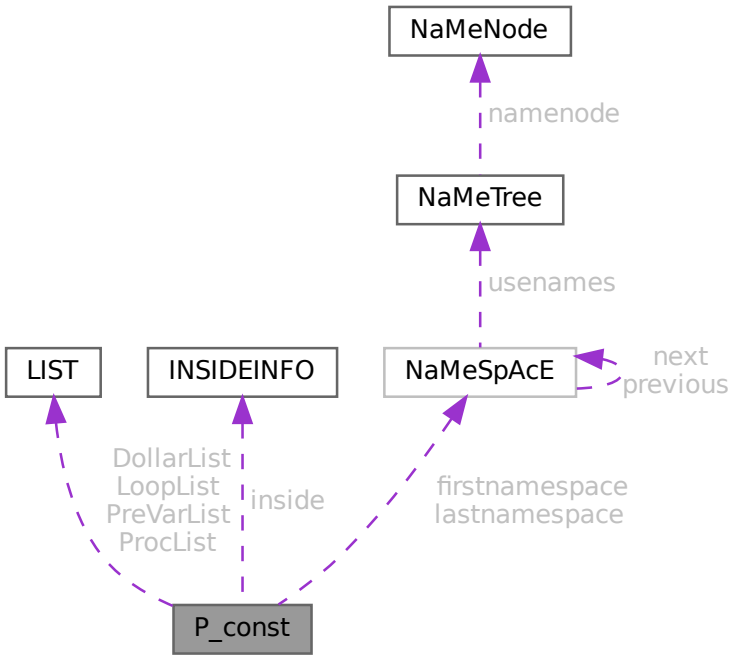
The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.104 P_const Struct Reference

`#include <structs.h>`

Collaboration diagram for P_const:



Data Fields

- LIST DollarList
- LIST PreVarList
- LIST LoopList
- LIST ProcList
- INSIDEINFO inside
- NAMESPACE * firstnamespace
- NAMESPACE * lastnamespace
- UBYTE * fullname
- UBYTE ** PreSwitchStrings
- UBYTE * preStart
- UBYTE * preStop
- UBYTE * preFill
- UBYTE * procedureExtension
- UBYTE * cprocedureExtension
- LONG * PreAssignStack
- int * PreIfStack
- int * PreSwitchModes
- int * PreTypes

- LONG StopWatchZero
- LONG InOutBuf
- LONG pSize
- int PreAssignFlag
- int PreContinuation
- int PreprocFlag
- int iBufError
- int PreOut
- int PreSwitchLevel
- int NumPreSwitchStrings
- int MaxPreTypes
- int NumPreTypes
- int MaxPreIfLevel
- int PreIfLevel
- int PreInsideLevel
- int DelayPrevar
- int AllowDelay
- int lhdollarerror
- int eat
- int gNumPre
- int PreDebug
- int OpenDictionary
- int PreAssignLevel
- int MaxPreAssignLevel
- int fullnamesize
- int FoundFileSetupCount
- WORD DebugFlag
- WORD preError
- UBYTE ComChar
- UBYTE cComChar

3.104.1 Detailed Description

The P_const struct is part of the global data and resides in the ALLGLOBALS struct A under the name P We see it used with the macro AP as in AP.InOutBuf It contains objects that have dealings with the preprocessor.

Definition at line 1610 of file structs.h.

3.104.2 Field Documentation

3.104.2.1 DollarList

```
LIST P_const::DollarList
```

Definition at line 1611 of file structs.h.

3.104.2.2 PreVarList

```
LIST P_const::PreVarList
```

Definition at line 1613 of file structs.h.

3.104.2.3 LoopList

```
LIST P_const::LoopList
```

Definition at line 1615 of file structs.h.

3.104.2.4 ProcList

```
LIST P_const::ProcList
```

Definition at line 1616 of file structs.h.

3.104.2.5 inside

```
INSIDEINFO P_const::inside
```

Definition at line 1617 of file structs.h.

3.104.2.6 firstnamespace

```
NAMESPACE* P_const::firstnamespace
```

Definition at line 1618 of file structs.h.

3.104.2.7 lastnamespace

```
NAMESPACE* P_const::lastnamespace
```

Definition at line 1619 of file structs.h.

3.104.2.8 fullname

```
UBYTE* P_const::fullname
```

Definition at line 1620 of file structs.h.

3.104.2.9 PreSwitchStrings

```
UBYTE** P_const::PreSwitchStrings
```

Definition at line 1621 of file structs.h.

3.104.2.10 preStart

```
UBYTE* P_const::preStart
```

Definition at line 1622 of file structs.h.

3.104.2.11 preStop

```
UBYTE* P_const::preStop
```

Definition at line 1623 of file structs.h.

3.104.2.12 preFill

```
UBYTE* P_const::preFill
```

Definition at line 1624 of file structs.h.

3.104.2.13 procedureExtension

```
UBYTE* P_const::procedureExtension
```

Definition at line 1625 of file structs.h.

3.104.2.14 cprocedureExtension

```
UBYTE* P_const::cprocedureExtension
```

Definition at line 1626 of file structs.h.

3.104.2.15 PreAssignStack

```
LONG* P_const::PreAssignStack
```

Definition at line 1627 of file structs.h.

3.104.2.16 PreIfStack

```
int* P_const::PreIfStack
```

Definition at line 1628 of file structs.h.

3.104.2.17 PreSwitchModes

```
int* P_const::PreSwitchModes
```

Definition at line 1629 of file structs.h.

3.104.2.18 PreTypes

```
int* P_const::PreTypes
```

Definition at line 1630 of file structs.h.

3.104.2.19 StopWatchZero

```
LONG P_const::StopWatchZero
```

Definition at line 1634 of file structs.h.

3.104.2.20 InOutBuf

```
LONG P_const::InOutBuf
```

Definition at line 1635 of file structs.h.

3.104.2.21 pSize

```
LONG P_const::pSize
```

Definition at line 1636 of file structs.h.

3.104.2.22 PreAssignFlag

```
int P_const::PreAssignFlag
```

Definition at line 1637 of file structs.h.

3.104.2.23 PreContinuation

```
int P_const::PreContinuation
```

Definition at line 1638 of file structs.h.

3.104.2.24 PreproFlag

```
int P_const::PreproFlag
```

Definition at line 1639 of file structs.h.

3.104.2.25 iBufError

```
int P_const::iBufError
```

Definition at line 1640 of file structs.h.

3.104.2.26 PreOut

```
int P_const::PreOut
```

Definition at line 1641 of file structs.h.

3.104.2.27 PreSwitchLevel

```
int P_const::PreSwitchLevel
```

Definition at line 1642 of file structs.h.

3.104.2.28 NumPreSwitchStrings

```
int P_const::NumPreSwitchStrings
```

Definition at line 1643 of file structs.h.

3.104.2.29 MaxPreTypes

```
int P_const::MaxPreTypes
```

Definition at line 1644 of file structs.h.

3.104.2.30 NumPreTypes

```
int P_const::NumPreTypes
```

Definition at line 1645 of file structs.h.

3.104.2.31 MaxPreIfLevel

```
int P_const::MaxPreIfLevel
```

Definition at line 1646 of file structs.h.

3.104.2.32 PreIfLevel

```
int P_const::PreIfLevel
```

Definition at line 1647 of file structs.h.

3.104.2.33 PreInsideLevel

```
int P_const::PreInsideLevel
```

Definition at line 1648 of file structs.h.

3.104.2.34 DelayPrevar

```
int P_const::DelayPrevar
```

Definition at line 1649 of file structs.h.

3.104.2.35 AllowDelay

```
int P_const::AllowDelay
```

Definition at line 1650 of file structs.h.

3.104.2.36 lhdollarerror

```
int P_const::lhdollarerror
```

Definition at line 1651 of file structs.h.

3.104.2.37 eat

```
int P_const::eat
```

Definition at line 1652 of file structs.h.

3.104.2.38 gNumPre

```
int P_const::gNumPre
```

Definition at line 1653 of file structs.h.

3.104.2.39 PreDebug

```
int P_const::PreDebug
```

Definition at line 1654 of file structs.h.

3.104.2.40 OpenDictionary

```
int P_const::OpenDictionary
```

Definition at line 1655 of file structs.h.

3.104.2.41 PreAssignLevel

```
int P_const::PreAssignLevel
```

Definition at line 1656 of file structs.h.

3.104.2.42 MaxPreAssignLevel

```
int P_const::MaxPreAssignLevel
```

Definition at line 1657 of file structs.h.

3.104.2.43 fullnamesize

```
int P_const::fullnamesize
```

Definition at line 1658 of file structs.h.

3.104.2.44 FoundFileSetupCount

```
int P_const::FoundFileSetupCount
```

Definition at line 1659 of file structs.h.

3.104.2.45 DebugFlag

```
WORD P_const::DebugFlag
```

Definition at line 1660 of file structs.h.

3.104.2.46 preError

```
WORD P_const::preError
```

Definition at line 1661 of file structs.h.

3.104.2.47 ComChar

```
UBYTE P_const::ComChar
```

Definition at line 1662 of file structs.h.

3.104.2.48 cComChar

```
UBYTE P_const::cComChar
```

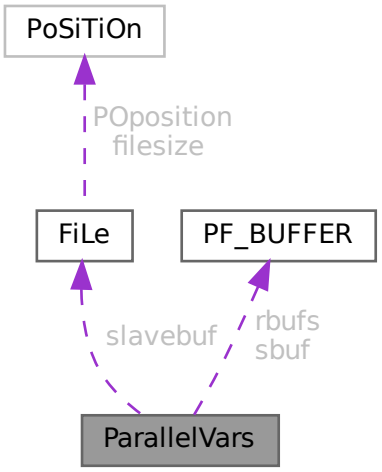
Definition at line 1663 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.105 ParallelVars Struct Reference

Collaboration diagram for ParallelVars:



Data Fields

- FILEHANDLE slavebuf
- PF_BUFFER * sbuf
- PF_BUFFER ** rbufs
- int me
- int numtasks
- int parallel
- int rhsInParallel
- int mkSlaveInfile
- int exprbufsize
- int exprtodo
- int log
- int notMyFault
- WORD numsbufs
- WORD numrbufs

3.105.1 Detailed Description

Definition at line 180 of file parallel.h.

3.105.2 Field Documentation

3.105.2.1 slavebuf

```
FILEHANDLE ParallelVars::slavebuf
```

Definition at line 181 of file parallel.h.

3.105.2.2 sbuf

```
PF_BUFFER* ParallelVars::sbuf
```

Definition at line 183 of file parallel.h.

3.105.2.3 rbufs

```
PF_BUFFER** ParallelVars::rbufs
```

Definition at line 184 of file parallel.h.

3.105.2.4 me

```
int ParallelVars::me
```

Definition at line 185 of file parallel.h.

3.105.2.5 numtasks

```
int ParallelVars::numtasks
```

Definition at line 186 of file parallel.h.

3.105.2.6 parallel

```
int ParallelVars::parallel
```

Definition at line 187 of file parallel.h.

3.105.2.7 rhsInParallel

```
int ParallelVars::rhsInParallel
```

Definition at line 189 of file parallel.h.

3.105.2.8 mkSlaveInfile

```
int ParallelVars::mkSlaveInfile
```

Definition at line 190 of file parallel.h.

3.105.2.9 exprbufsize

```
int ParallelVars::exprbufsize
```

Definition at line 191 of file parallel.h.

3.105.2.10 exprtodo

```
int ParallelVars::exprtodo
```

Definition at line 192 of file parallel.h.

3.105.2.11 log

```
int ParallelVars::log
```

Definition at line 193 of file parallel.h.

3.105.2.12 notMyFault

```
int ParallelVars::notMyFault
```

Definition at line 194 of file parallel.h.

3.105.2.13 numsbufs

```
WORD ParallelVars::numsbufs
```

Definition at line 195 of file parallel.h.

3.105.2.14 numrbufs

```
WORD ParallelVars::numrbufs
```

Definition at line 196 of file parallel.h.

The documentation for this struct was generated from the following file:

- parallel.h

3.106 PaRtl Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD * psize
- WORD * args
- WORD * nargs
- WORD * nfun
- WORD numargs
- WORD numpart
- WORD where

3.106.1 Detailed Description

The struct PARTI is used to help determining whether a partition_ function can be replaced.

Definition at line 1095 of file structs.h.

3.106.2 Field Documentation

3.106.2.1 psize

```
WORD* PaRtI::psize
```

Definition at line 1096 of file structs.h.

3.106.2.2 args

```
WORD* PaRtI::args
```

Definition at line 1097 of file structs.h.

3.106.2.3 nargs

```
WORD* PaRtI::nargs
```

Definition at line 1098 of file structs.h.

3.106.2.4 nfun

```
WORD* PaRtI::nfun
```

Definition at line 1099 of file structs.h.

3.106.2.5 numargs

```
WORD PaRtI::numargs
```

Definition at line 1100 of file structs.h.

3.106.2.6 numpart

```
WORD PaRtI::numpart
```

Definition at line 1101 of file structs.h.

3.106.2.7 where

```
WORD PaRtI::where
```

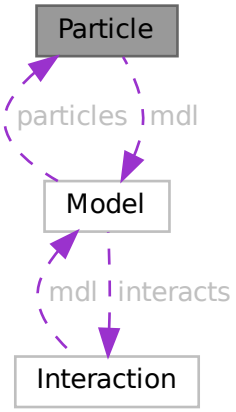
Definition at line 1102 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.107 Particle Class Reference

Collaboration diagram for Particle:



Public Member Functions

- Particle (Model *modl, int pid, PInput *pinp)
- char * particleName (int p)
- int particleCode (int p)
- char * **interactionName** (int p)
- char * aparticle (void)
- void prParticle (void)
- int isNeutral (void)
- const char * typeName (void)
- const char * typeGName (void)

Data Fields

- Model * mdl
- char * name
- char * aname
- int id
- int ptype
- int neutral
- int pcode
- int acode
- int cmindeg
- int cmaxdeg
- int extonly

3.107.1 Detailed Description

Definition at line 227 of file grcc.h.

3.107.2 Constructor & Destructor Documentation

3.107.2.1 Particle()

```
Particle::Particle (
            Model * modl,
            int pid,
            PInput * pinp )
```

Definition at line 1672 of file grcc.cc.

3.107.2.2 ~Particle()

```
Particle::~Particle (
            void  )
```

Definition at line 1749 of file grcc.cc.

3.107.3 Member Function Documentation

3.107.3.1 particleName()

```
char * Particle::particleName (
            int p )
```

Definition at line 1756 of file grcc.cc.

3.107.3.2 particleCode()

```
int Particle::particleCode (
            int p )
```

Definition at line 1766 of file grcc.cc.

3.107.3.3 aparticle()

```
char * Particle::aparticle (
            void  )
```

Definition at line 1794 of file grcc.cc.

3.107.3.4 prParticle()

```
void Particle::prParticle (
            void  )
```

Definition at line 1806 of file grcc.cc.

3.107.3.5 isNeutral()

```
int Particle::isNeutral (
            void  )
```

Definition at line 1776 of file grcc.cc.

3.107.3.6 typeName()

```
const char * Particle::typeName (
            void  )
```

Definition at line 1782 of file grcc.cc.

3.107.3.7 typeGName()

```
const char * Particle::typeGName (
            void  )
```

Definition at line 1788 of file grcc.cc.

3.107.4 Field Documentation

3.107.4.1 mdl

```
Model* Particle::mdl
```

Definition at line 229 of file grcc.h.

3.107.4.2 name

```
char* Particle::name
```

Definition at line 230 of file grcc.h.

3.107.4.3 aname

```
char* Particle::aname
```

Definition at line 231 of file grcc.h.

3.107.4.4 id

```
int Particle::id
```

Definition at line 232 of file grcc.h.

3.107.4.5 ptype

```
int Particle::ptype
```

Definition at line 233 of file grcc.h.

3.107.4.6 neutral

```
int Particle::neutral
```

Definition at line 234 of file grcc.h.

3.107.4.7 pcode

```
int Particle::pcode
```

Definition at line 235 of file grcc.h.

3.107.4.8 acode

```
int Particle::acode
```

Definition at line 236 of file grcc.h.

3.107.4.9 cmindeg

```
int Particle::cmindeg
```

Definition at line 237 of file grcc.h.

3.107.4.10 cmaxdeg

```
int Particle::cmaxdeg
```

Definition at line 238 of file grcc.h.

3.107.4.11 extonly

```
int Particle::extonly
```

Definition at line 240 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.108 PaRtIcLe Struct Reference

Data Fields

- WORD number
- WORD spin
- WORD mass
- WORD type

3.108.1 Detailed Description

Definition at line 618 of file structs.h.

3.108.2 Field Documentation

3.108.2.1 number

```
WORD PaRtIcLe::number
```

Definition at line 619 of file structs.h.

3.108.2.2 spin

```
WORD PaRtIcLe::spin
```

Definition at line 620 of file structs.h.

3.108.2.3 mass

```
WORD PaRtIcLe::mass
```

Definition at line 621 of file structs.h.

3.108.2.4 type

```
WORD PaRtIcLe::type
```

Definition at line 622 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.109 PeRmUtE Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD * objects
- WORD sign
- WORD n
- WORD cycle [MAXMATCH]

3.109.1 Detailed Description

The struct PERM is used to generate all permutations when the pattern matcher has to try to match (anti)symmetric functions.

Definition at line 1056 of file structs.h.

3.109.2 Field Documentation

3.109.2.1 objects

```
WORD* PeRmUtE::objects
```

Definition at line 1057 of file structs.h.

3.109.2.2 sign

```
WORD PeRmUtE::sign
```

Definition at line 1058 of file structs.h.

3.109.2.3 n

```
WORD PeRmUtE::n
```

Definition at line 1059 of file structs.h.

3.109.2.4 cycle

```
WORD PeRmUtE::cycle[MAXMATCH]
```

Definition at line 1060 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.110 PeRmUtEp Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD ** objects
- WORD sign
- WORD n
- WORD cycle [MAXMATCH]

3.110.1 Detailed Description

Like struct PERM but works with pointers.

Definition at line 1067 of file structs.h.

3.110.2 Field Documentation

3.110.2.1 objects

```
WORD** PeRmUtEp::objects
```

Definition at line 1068 of file structs.h.

3.110.2.2 sign

```
WORD PeRmUtEp::sign
```

Definition at line 1069 of file structs.h.

3.110.2.3 n

```
WORD PeRmUtEp::n
```

Definition at line 1070 of file structs.h.

3.110.2.4 cycle

```
WORD PeRmUtEp::cycle[MAXMATCH]
```

Definition at line 1071 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.111 PF_BUFFER Struct Reference

```
#include <parallel.h>
```

Data Fields

- WORD ** buff
- WORD ** fill
- WORD ** full
- WORD ** stop
- MPI_Status * status
- MPI_Status * retstat
- MPI_Request * request
- MPI_Datatype * type
- int * index
- int * tag
- int * from
- int numbufs
- int active

3.111.1 Detailed Description

A struct for nonblocking, unbuffered send of the sorted terms in the PObuffers back to the master using several "rotating" PObuffers.

Definition at line 159 of file parallel.h.

3.111.2 Field Documentation

3.111.2.1 buff

```
WORD** PF_BUFFER::buff
```

Definition at line 160 of file parallel.h.

3.111.2.2 fill

```
WORD** PF_BUFFER::fill
```

Definition at line 161 of file parallel.h.

3.111.2.3 full

```
WORD** PF_BUFFER::full
```

Definition at line 162 of file parallel.h.

3.111.2.4 stop

```
WORD** PF_BUFFER::stop
```

Definition at line 163 of file parallel.h.

3.111.2.5 status

```
MPI_Status* PF_BUFFER::status
```

Definition at line 164 of file parallel.h.

3.111.2.6 retstat

```
MPI_Status* PF_BUFFER::retstat
```

Definition at line 165 of file parallel.h.

3.111.2.7 request

```
MPI_Request* PF_BUFFER::request
```

Definition at line 166 of file parallel.h.

3.111.2.8 type

```
MPI_Datatype* PF_BUFFER::type
```

Definition at line 167 of file parallel.h.

3.111.2.9 index

```
int* PF_BUFFER::index
```

Definition at line 168 of file parallel.h.

3.111.2.10 tag

```
int* PF_BUFFER::tag
```

Definition at line 169 of file parallel.h.

3.111.2.11 from

```
int* PF_BUFFER::from
```

Definition at line 170 of file parallel.h.

3.111.2.12 numbufs

```
int PF_BUFFER::numbufs
```

Definition at line 171 of file parallel.h.

3.111.2.13 active

```
int PF_BUFFER::active
```

Definition at line 172 of file parallel.h.

The documentation for this struct was generated from the following file:

- parallel.h

3.112 PGInput Struct Reference

Data Fields

- int cdeg
- int ctyp
- int cnum
- int cple
- const char * pname
- long pcode

3.112.1 Detailed Description

Definition at line 232 of file grccparam.h.

3.112.2 Field Documentation

3.112.2.1 cdeg

```
int PGInput::cdeg
```

Definition at line 233 of file grccparam.h.

3.112.2.2 ctyp

```
int PGInput::ctyp
```

Definition at line 234 of file grccparam.h.

3.112.2.3 cnum

```
int PGInput::cnum
```

Definition at line 235 of file grccparam.h.

3.112.2.4 cple

```
int PGInput::cple
```

Definition at line 236 of file grccparam.h.

3.112.2.5 pname

```
const char* PGInput::pname
```

Definition at line 238 of file grccparam.h.

3.112.2.6 pcode

```
long PGInput::pcode
```

Definition at line 240 of file grccparam.h.

The documentation for this struct was generated from the following file:

- grccparam.h

3.113 PInput Struct Reference

Data Fields

- const char * name
- const char * aname
- int pcode
- int acode
- const char * ptypen
- int ptypec
- int extonly

3.113.1 Detailed Description

Definition at line 183 of file grccparam.h.

3.113.2 Field Documentation

3.113.2.1 name

```
const char* PInput::name
```

Definition at line 184 of file grccparam.h.

3.113.2.2 aname

```
const char* PInput::aname
```

Definition at line 185 of file grccparam.h.

3.113.2.3 pcode

```
int PInput::pcode
```

Definition at line 186 of file grccparam.h.

3.113.2.4 acode

```
int PInput::acode
```

Definition at line 187 of file grccparam.h.

3.113.2.5 ptypen

```
const char* PInput::ptypen
```

Definition at line 188 of file grccparam.h.

3.113.2.6 ptypec

```
int PInput::ptypec
```

Definition at line 189 of file grccparam.h.

3.113.2.7 extonly

```
int PInput::extonly
```

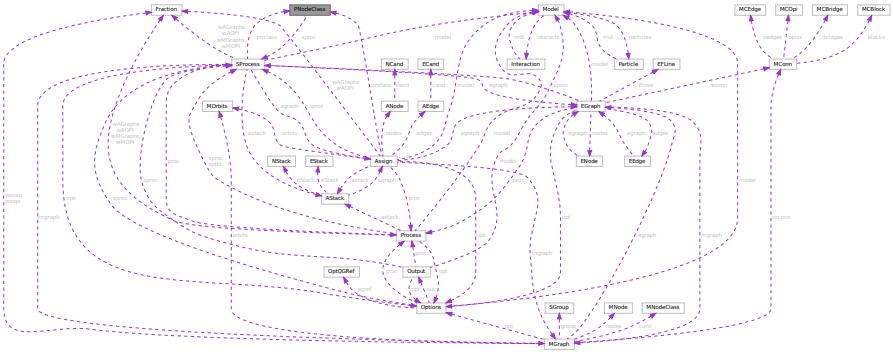
Definition at line 190 of file grccparam.h.

The documentation for this struct was generated from the following file:

- grccparam.h

3.114 PNodeClass Class Reference

Collaboration diagram for PNodeClass:



Public Member Functions

- PNodeClass (SProcess ∗sprc, int nnods, int nclss, NCInput ∗cls)
- PNodeClass (SProcess ∗sprc, int nnods, int nclss, int ∗dgs, int ∗typ, int ∗ptcl, int ∗cpl, int ∗cnt, int ∗cmind, int ∗cmaxd)
- void prPNodeClass (void)
- void prElem (int e)

Data Fields

- SProcess ∗ sproc
- int ∗ deg
- int ∗ type
- int ∗ particle
- int ∗ couple
- int ∗ cmindeg
- int ∗ cmaxdeg
- int ∗ count
- int ∗ cl2nd
- int ∗ nd2cl
- int ∗ cl2mcl
- int nnodes
- int nclass

3.114.1 Detailed Description

Definition at line 348 of file grcc.h.

3.114.2 Constructor & Destructor Documentation

3.114.2.1 PNodeClass() [1/2]

```
PNodeClass::PNodeClass (
            SProcess * sprc,
            int nnods,
            int nclss,
            NCInput * cls )
```

Definition at line 2628 of file grcc.cc.

3.114.2.2 PNodeClass() [2/2]

```
PNodeClass::PNodeClass (
            SProcess * sprc,
            int nnods,
            int nclss,
            int * dgs,
            int * typ,
            int * ptcl,
            int * cpl,
            int * cnt,
            int * cmind,
            int * cmaxd )
```

Definition at line 2702 of file grcc.cc.

3.114.2.3 ~PNodeClass()

```
PNodeClass::~PNodeClass (
            void  )
```

Definition at line 2780 of file grcc.cc.

3.114.3 Member Function Documentation

3.114.3.1 prPNodeClass()

```
void PNodeClass::prPNodeClass (
            void  )
```

Definition at line 2795 of file grcc.cc.

3.114.3.2 prElem()

```
void PNodeClass::prElem (
            int e )
```

Definition at line 2807 of file grcc.cc.

3.114.4 Field Documentation

3.114.4.1 sproc

```
SProcess* PNodeClass::sproc
```

Definition at line 350 of file grcc.h.

3.114.4.2 deg

```
int* PNodeClass::deg
```

Definition at line 351 of file grcc.h.

3.114.4.3 type

```
int* PNodeClass::type
```

Definition at line 352 of file grcc.h.

3.114.4.4 particle

```
int* PNodeClass::particle
```

Definition at line 353 of file grcc.h.

3.114.4.5 couple

```
int* PNodeClass::couple
```

Definition at line 354 of file grcc.h.

3.114.4.6 cmindeg

```
int* PNodeClass::cmindeg
```

Definition at line 355 of file grcc.h.

3.114.4.7 cmaxdeg

```
int* PNodeClass::cmaxdeg
```

Definition at line 356 of file grcc.h.

3.114.4.8 count

```
int* PNodeClass::count
```

Definition at line 357 of file grcc.h.

3.114.4.9 cl2nd

```
int* PNodeClass::cl2nd
```

Definition at line 358 of file grcc.h.

3.114.4.10 nd2cl

```
int* PNodeClass::nd2cl
```

Definition at line 359 of file grcc.h.

3.114.4.11 cl2mcl

```
int* PNodeClass::cl2mcl
```

Definition at line 360 of file grcc.h.

3.114.4.12 nnodes

```
int PNodeClass::nnodes
```

Definition at line 361 of file grcc.h.

3.114.4.13 nclass

```
int PNodeClass::nclass
```

Definition at line 362 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.115 flint::poly Class Reference

Public Member Functions

- void print (const string &text)

Data Fields

- fmpz_poly_t d

3.115.1 Detailed Description

Definition at line 88 of file flintinterface.h.

3.115.2 Constructor & Destructor Documentation

3.115.2.1 poly()

```
flint::poly::poly ( )  [inline]
```

Definition at line 91 of file flintinterface.h.

3.115.2.2 ~poly()

```
flint::poly::~poly ( )  [inline]
```

Definition at line 92 of file flintinterface.h.

3.115.3 Member Function Documentation

3.115.3.1 print()

```
void flint::poly::print (
            const string & text )  [inline]
```

Definition at line 93 of file flintinterface.h.

3.115.4 Field Documentation

3.115.4.1 d

```
fmpz_poly_t flint::poly::d
```

Definition at line 90 of file flintinterface.h.

The documentation for this class was generated from the following file:

- flintinterface.h

3.116 poly Class Reference

Public Member Functions

- poly (PHEAD int, WORD=-1, WORD=1)
- poly (PHEAD const UWORD *, WORD, WORD=-1, WORD=1)
- poly (const poly &, WORD=-1, WORD=1)
- poly & operator+= (const poly &)
- poly & operator-= (const poly &)
- poly & operator*= (const poly &)
- poly & operator/= (const poly &)
- poly & operator%= (const poly &)
- const poly operator+ (const poly &) const
- const poly operator- (const poly &) const
- const poly operator* (const poly &) const
- const poly operator/ (const poly &) const
- const poly operator% (const poly &) const
- bool operator== (const poly &) const
- bool operator!= (const poly &) const
- poly & operator= (const poly &)
- WORD & operator[ ] (int)
- const WORD & operator[ ] (int) const
- void termscopy (const WORD *, int, int)
- void check_memory (int)
- void expand_memory (int)
- bool is_zero () const
- bool is_one () const
- bool is_integer () const
- bool is_monomial () const
- int is_dense_univariate () const
- int sign () const
- int degree (int) const
- int total_degree () const
- int first_variable () const
- int number_of_terms () const
- const std::vector< int > all_variables () const
- const poly integer_lcoeff () const
- const poly lcoeff_univar (int) const
- const poly lcoeff_multivar (int) const
- const poly coefficient (int, int) const
- const poly derivative (int) const
- void setmod (WORD, WORD=1)
- void coefficients_modulo (UWORD *, WORD, bool)
- int size_of_form_notation () const
- int size_of_form_notation_with_den (WORD) const
- const poly & normalize ()
- const std::string to_string () const
- WORD last_monomial_index () const
- WORD * last_monomial () const
- int compare_degree_vector (const poly &) const
- std::vector< int > degree_vector () const
- int **compare_degree_vector** (const std::vector< int > &) const

Static Public Member Functions

- static const poly simple_poly (PHEAD int, int=0, int=1, int=0, int=1)
- static const poly simple_poly (PHEAD int, const poly &, int=1, int=0, int=1)
- static void get_variables (PHEAD const std::vector< WORD * > &, bool, bool)
- static const poly argument_to_poly (PHEAD WORD *, bool, bool, poly *den=NULL)
- static void poly_to_argument (const poly &, WORD *, bool)
- static void poly_to_argument_with_den (const poly &, WORD, const UWORD *, WORD *, bool)
- static const poly from_coefficient_list (PHEAD const std::vector< WORD > &, int, WORD)
- static const std::vector< WORD > to_coefficient_list (const poly &)
- static const std::vector< WORD > coefficient_list_divmod (const std::vector< WORD > &, const std::vector< WORD > &, WORD, int)
- static int monomial_compare (PHEAD const WORD *, const WORD *)
- static void add (const poly &, const poly &, poly &)
- static void sub (const poly &, const poly &, poly &)
- static void mul (const poly &, const poly &, poly &)
- static void div (const poly &, const poly &, poly &)
- static void mod (const poly &, const poly &, poly &)
- static void divmod (const poly &, const poly &, poly &, poly &, bool only_divides)
- static bool divides (const poly &, const poly &)
- static void mul_one_term (const poly &, const poly &, poly &)
- static void mul_univar (const poly &, const poly &, poly &, int)
- static void mul_heap (const poly &, const poly &, poly &)
- static void divmod_one_term (const poly &, const poly &, poly &, poly &, bool)
- static void divmod_univar (const poly &, const poly &, poly &, poly &, int, bool)
- static void divmod_heap (const poly &, const poly &, poly &, poly &, bool, bool, bool &)
- static void push_heap (PHEAD WORD **, int)
- static void pop_heap (PHEAD WORD **, int)

Data Fields

- WORD * terms
- LONG size_of_terms
- WORD modp
- WORD modn

3.116.1 Detailed Description

Definition at line 53 of file poly.h.

3.116.2 Constructor & Destructor Documentation

3.116.2.1 poly() [1/3]

```
poly::poly (
            PHEAD int a,
            WORD modp = -1,
            WORD modn = 1 )
```

Definition at line 54 of file poly.cc.

3.116.2.2 poly() [2/3]

```
poly::poly (
             PHEAD const UWORD * a,
             WORD na,
             WORD modp = -1,
             WORD modn = 1 )
```

Definition at line 83 of file poly.cc.

3.116.2.3 poly() [3/3]

```
poly::poly (
             const poly & a,
             WORD modp = -1,
             WORD modn = 1 )
```

Definition at line 111 of file poly.cc.

3.116.2.4 ~poly()

```
poly::~poly ( )
```

Definition at line 133 of file poly.cc.

3.116.3 Member Function Documentation

3.116.3.1 operator+=()

```
poly & poly::operator+= (
             const poly & a )
```

Definition at line 1966 of file poly.cc.

3.116.3.2 operator-=()

```
poly & poly::operator-= (
             const poly & a )
```

Definition at line 1967 of file poly.cc.

3.116.3.3 operator*=()

```
poly & poly::operator*= (
             const poly & a )
```

Definition at line 1968 of file poly.cc.

3.116.3.4 operator/=()

```
poly & poly::operator/= (
            const poly & a )
```

Definition at line 1969 of file poly.cc.

3.116.3.5 operator%=()

```
poly & poly::operator%= (
            const poly & a )
```

Definition at line 1970 of file poly.cc.

3.116.3.6 operator+()

```
const poly poly::operator+ (
            const poly & a ) const
```

Definition at line 1959 of file poly.cc.

3.116.3.7 operator-()

```
const poly poly::operator- (
            const poly & a ) const
```

Definition at line 1960 of file poly.cc.

3.116.3.8 operator∗()

```
const poly poly::operator* (
            const poly & a ) const
```

Definition at line 1961 of file poly.cc.

3.116.3.9 operator/()

```
const poly poly::operator/ (
            const poly & a ) const
```

Definition at line 1962 of file poly.cc.

3.116.3.10 operator%()

```
const poly poly::operator% (
            const poly & a ) const
```

Definition at line 1963 of file poly.cc.

3.116.3.11 operator==()

```
bool poly::operator== (
            const poly & a ) const
```

Definition at line 1973 of file poly.cc.

3.116.3.12 operator"!=()

```
bool poly::operator!= (
            const poly & a ) const
```

Definition at line 1979 of file poly.cc.

3.116.3.13 operator=()

```
poly & poly::operator= (
            const poly & a )
```

Definition at line 1939 of file poly.cc.

3.116.3.14 operator[]() [1/2]

```
WORD & poly::operator[] (
            int i )  [inline]
```

Definition at line 247 of file poly.h.

3.116.3.15 operator[]() [2/2]

```
const WORD & poly::operator[] (
            int i ) const  [inline]
```

Definition at line 251 of file poly.h.

3.116.3.16 termscopy()

```
void poly::termscopy (
            const WORD * source,
            int dest,
            int num )  [inline]
```

Definition at line 258 of file poly.h.

3.116.3.17 check_memory()

```
void poly::check_memory (
            int i )  [inline]
```

Definition at line 240 of file poly.h.

3.116.3.18 expand_memory()

```
void poly::expand_memory (
            int i )
```

Definition at line 149 of file poly.cc.

3.116.3.19 is_zero()

```
bool poly::is_zero ( ) const
```

Definition at line 2209 of file poly.cc.

3.116.3.20 is_one()

```
bool poly::is_one ( ) const
```

Definition at line 2219 of file poly.cc.

3.116.3.21 is_integer()

```
bool poly::is_integer ( ) const
```

Definition at line 2239 of file poly.cc.

3.116.3.22 is_monomial()

```
bool poly::is_monomial ( ) const
```

Definition at line 2259 of file poly.cc.

3.116.3.23 is_dense_univariate()

```
int poly::is_dense_univariate ( ) const
```

Dense univariate detection

3.116.4 Description

This method returns whether the polynomial is dense and univariate. The possible return values are:

-2 is not dense univariate -1 is no variables n>=0 is univariate in n

3.116.5 Notes

A univariate polynomial is considered dense iff more than half of the coefficients a_0...a_deg are non-zero.

Definition at line 2284 of file poly.cc.

Referenced by divmod(), and mul().

3.116.5.1 sign()

```
int poly::sign ( ) const
```

Definition at line 2039 of file poly.cc.

3.116.5.2 degree()

```
int poly::degree (
            int x ) const
```

Definition at line 2050 of file poly.cc.

3.116.5.3 total_degree()

```
int poly::total_degree ( ) const
```

Definition at line 2063 of file poly.cc.

3.116.5.4 first_variable()

```
int poly::first_variable ( ) const
```

Definition at line 2000 of file poly.cc.

3.116.5.5 number_of_terms()

```
int poly::number_of_terms ( ) const
```

Definition at line 1986 of file poly.cc.

3.116.5.6 all_variables()

```
const vector< int > poly::all_variables ( ) const
```

Definition at line 2016 of file poly.cc.

3.116.5.7 integer_lcoeff()

```
const poly poly::integer_lcoeff ( ) const
```

Definition at line 2083 of file poly.cc.

3.116.5.8 lcoeff_univar()

```
const poly poly::lcoeff_univar (
            int x ) const
```

Definition at line 2133 of file poly.cc.

3.116.5.9 lcoeff_multivar()

```
const poly poly::lcoeff_multivar (
            int x ) const
```

Definition at line 2140 of file poly.cc.

3.116.5.10 coefficient()

```
const poly poly::coefficient (
            int x,
            int n ) const
```

Definition at line 2105 of file poly.cc.

3.116.5.11 derivative()

```
const poly poly::derivative (
            int x ) const
```

Definition at line 2172 of file poly.cc.

3.116.5.12 setmod()

```
void poly::setmod (
            WORD _modp,
            WORD _modn = 1 )
```

Definition at line 179 of file poly.cc.

3.116.5.13 coefficients_modulo()

```
void poly::coefficients_modulo (
            UWORD * a,
            WORD na,
            bool small )
```

Definition at line 215 of file poly.cc.

3.116.5.14 simple_poly() [1/2]

```
const poly poly::simple_poly (
             PHEAD int x,
             int a = 0,
             int b = 1,
             int p = 0,
             int n = 1 )  [static]
```

Definition at line 2317 of file poly.cc.

3.116.5.15 simple_poly() [2/2]

```
const poly poly::simple_poly (
             PHEAD int x,
             const poly & a,
             int b = 1,
             int p = 0,
             int n = 1 )  [static]
```

Definition at line 2356 of file poly.cc.

3.116.5.16 get_variables()

```
void poly::get_variables (
             PHEAD const std::vector< WORD * > & ,
             bool ,
             bool  )  [static]
```

Definition at line 2395 of file poly.cc.

3.116.5.17 argument_to_poly()

```
const poly poly::argument_to_poly (
             PHEAD WORD * e,
             bool with_arghead,
             bool sort_univar,
             poly * den = NULL )  [static]
```

Definition at line 2498 of file poly.cc.

3.116.5.18 poly_to_argument()

```
void poly::poly_to_argument (
             const poly & a,
             WORD * res,
             bool with_arghead )  [static]
```

Definition at line 2611 of file poly.cc.

3.116.5.19 poly_to_argument_with_den()

```
void poly::poly_to_argument_with_den (
            const poly & a,
            WORD nden,
            const UWORD * den,
            WORD * res,
            bool with_arghead )  [static]
```

Definition at line 2683 of file poly.cc.

3.116.5.20 size_of_form_notation()

```
int poly::size_of_form_notation ( ) const
```

Definition at line 2766 of file poly.cc.

3.116.5.21 size_of_form_notation_with_den()

```
int poly::size_of_form_notation_with_den (
            WORD nden ) const
```

Definition at line 2795 of file poly.cc.

3.116.5.22 normalize()

```
const poly & poly::normalize ( )
```

Definition at line 362 of file poly.cc.

3.116.5.23 from_coefficient_list()

```
const poly poly::from_coefficient_list (
            PHEAD const std::vector< WORD > & ,
            int ,
            WORD  )  [static]
```

Definition at line 2885 of file poly.cc.

3.116.5.24 to_coefficient_list()

```
const vector< WORD > poly::to_coefficient_list (
            const poly & a )  [static]
```

Definition at line 2823 of file poly.cc.

3.116.5.25 coefficient_list_divmod()

```
const vector< WORD > poly::coefficient_list_divmod (
            const std::vector< WORD > & ,
            const std::vector< WORD > & ,
            WORD ,
            int  )  [static]
```

Definition at line 2846 of file poly.cc.

3.116.5.26 to_string()

```
const string poly::to_string ( ) const
```

Definition at line 267 of file poly.cc.

3.116.5.27 monomial_compare()

```
int poly::monomial_compare (
            PHEAD const WORD * a,
            const WORD * b )  [static]
```

Definition at line 350 of file poly.cc.

3.116.5.28 last_monomial_index()

```
WORD poly::last_monomial_index ( ) const
```

Definition at line 465 of file poly.cc.

3.116.5.29 last_monomial()

```
WORD * poly::last_monomial ( ) const
```

Definition at line 472 of file poly.cc.

3.116.5.30 compare_degree_vector()

```
int poly::compare_degree_vector (
            const poly & b ) const
```

Definition at line 481 of file poly.cc.

3.116.5.31 degree_vector()

```
std::vector< int > poly::degree_vector ( ) const
```

Definition at line 503 of file poly.cc.

3.116.5.32 add()

```
void poly::add (
            const poly & a,
            const poly & b,
            poly & c )  [static]
```

Definition at line 538 of file poly.cc.

3.116.5.33 sub()

```
void poly::sub (
            const poly & a,
            const poly & b,
            poly & c )  [static]
```

Definition at line 622 of file poly.cc.

3.116.5.34 mul()

```
void poly::mul (
            const poly & a,
            const poly & b,
            poly & c )  [static]
```

Polynomial multiplication

3.116.6 Description

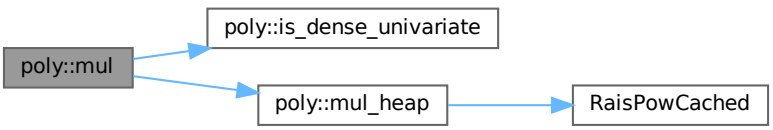
This routine determines which multiplication routine to use for multiplying two polynomials. The logic is as follows:

- If a or b consists of only one term, call mul_one_term;
- Otherwise, if both are univariate and dense, call mul_univar;
- Otherwise, call mul_heap.

Definition at line 1195 of file poly.cc.

References is_dense_univariate(), and mul_heap().

Here is the call graph for this function:



3.116.6.1 div()

```
void poly::div (
            const poly & a,
            const poly & b,
            poly & c )  [static]
```

Definition at line 1915 of file poly.cc.

3.116.6.2 mod()

```
void poly::mod (
            const poly & a,
            const poly & b,
            poly & c )  [static]
```

Definition at line 1927 of file poly.cc.

3.116.6.3 divmod()

```
void poly::divmod (
            const poly & a,
            const poly & b,
            poly & q,
            poly & r,
            bool only_divides )  [static]
```

Polynomial division

3.116.7 Description

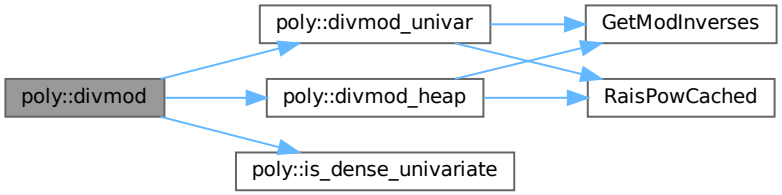
This routine determines which division routine to use for dividing two polynomials. The logic is as follows:

- If b consists of only one term, call divmod_one_term;
- Otherwise, if both are univariate and dense, call divmod_univar;
- Otherwise, call divmod_heap.

Definition at line 1856 of file poly.cc.

References divmod_heap(), divmod_univar(), and is_dense_univariate().

Here is the call graph for this function:



3.116.7.1 divides()

```
bool poly::divides (
            const poly & a,
            const poly & b )  [static]
```

Definition at line 1902 of file poly.cc.

3.116.7.2 mul_one_term()

```
void poly::mul_one_term (
            const poly & a,
            const poly & b,
            poly & c )  [static]
```

Definition at line 755 of file poly.cc.

3.116.7.3 mul_univar()

```
void poly::mul_univar (
            const poly & a,
            const poly & b,
            poly & c,
            int var )  [static]
```

Definition at line 829 of file poly.cc.

3.116.7.4 mul_heap()

```
void poly::mul_heap (
            const poly & a,
            const poly & b,
            poly & c )  [static]
```

Multiplication of polynomials with a heap

3.116.8 Description

Multiplies two multivariate polynomials. The next element of the product is efficiently determined by using a heap. If the product of the maximum power in all variables is small, a hash table is used to add equal terms for extra speed.

A heap element h is formatted as follows:

- h[0] = index in a
- h[1] = index in b
- h[2] = hash code (-1 if no hash is used)
- h[3] = length of coefficient with sign
- h[4...4+AN.poly_num_vars-1] = powers

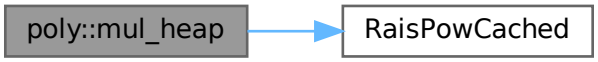
- h[4+AN.poly_num_vars...4+h[3]-1] = coefficient

Definition at line 961 of file poly.cc.

References RaisPowCached().

Referenced by mul().

Here is the call graph for this function:



3.116.8.1 divmod_one_term()

```
void poly::divmod_one_term (
            const poly & a,
            const poly & b,
            poly & q,
            poly & r,
            bool only_divides )  [static]
```

Definition at line 1245 of file poly.cc.

3.116.8.2 divmod_univar()

```
void poly::divmod_univar (
            const poly & a,
            const poly & b,
            poly & q,
            poly & r,
            int var,
            bool only_divides )  [static]
```

Division of dense univariate polynomials.

3.116.9 Description

Divides two dense univariate polynomials. For each power, the method collects all terms that result in that power.

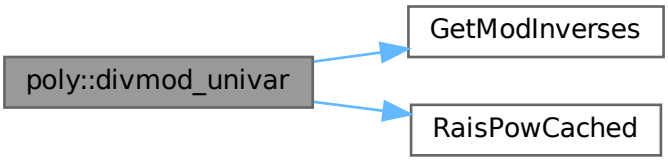
Relevant formula [Q=A/B, P=SUM(p_i*x^i), n=deg(A), m=deg(B)]: q_k = [a_{m+k} - SUM(i=k+1...n-m, b_{m+k-i}*q_i)] / b_m

Definition at line 1376 of file poly.cc.

References GetModInverses(), and RaisPowCached().

Referenced by divmod().

Here is the call graph for this function:



3.116.9.1 divmod_heap()

```
void poly::divmod_heap (
             const poly & a,
             const poly & b,
             poly & q,
             poly & r,
             bool only_divides,
             bool check_div,
             bool & div_fail )  [static]
```

Division of polynomials with a heap

3.116.10 Description

Divides two multivariate polynomials. The next element of the quotient/remainder is efficiently determined by using a heap. If the product of the maximum power in all variables is small, a hash table is used to add equal terms for extra speed.

If the input flag check_div is set then if the result of any coefficient division results in a non-zero remainder (indicating that division has failed over the integers) the output flag div_fail will be set and the division will be terminated early (q, r will be incorrect). If check_div is not set then non-zero remainders from coefficient division will be written into r.

A heap element h is formatted as follows:

- h[0] = index in a
- h[1] = index in b
- h[2] = -1 (no hash is used)
- h[3] = length of coefficient with sign
- h[4...4+AN.poly_num_vars-1] = powers
- h[4+AN.poly_num_vars...4+h[3]-1] = coefficient

Note: the hashing trick as in multiplication cannot be used easily, since there is no tight upperbound on the exponents in the answer.

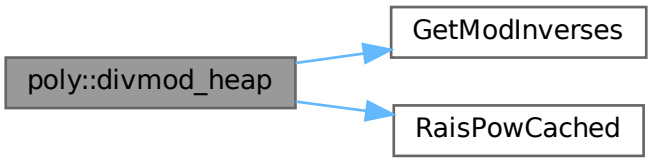
For details, see M. Monagan, "Polynomial Division using Dynamic Array, Heaps, and Packed Exponent Vectors"

Definition at line 1579 of file poly.cc.

References GetModInverses(), and RaisPowCached().

Referenced by divmod().

Here is the call graph for this function:



3.116.10.1 push_heap()

```
void poly::push_heap (
             PHEAD WORD ** heap,
             int n )  [static]
```

Definition at line 739 of file poly.cc.

3.116.10.2 pop_heap()

```
void poly::pop_heap (
             PHEAD WORD ** heap,
             int n )  [static]
```

Definition at line 707 of file poly.cc.

3.116.11 Field Documentation

3.116.11.1 terms

```
WORD* poly::terms
```

Definition at line 62 of file poly.h.

3.116.11.2 size_of_terms

```
LONG poly::size_of_terms
```

Definition at line 63 of file poly.h.

3.116.11.3 modp

```
WORD poly::modp
```

Definition at line 64 of file poly.h.

3.116.11.4 modn

```
WORD poly::modn
```

Definition at line 64 of file poly.h.

The documentation for this class was generated from the following files:

- poly.h
- poly.cc

3.117 flint::poly_factor Class Reference

Data Fields

- fmpz_poly_factor_t d

3.117.1 Detailed Description

Definition at line 97 of file flintinterface.h.

3.117.2 Constructor & Destructor Documentation

3.117.2.1 poly_factor()

```
flint::poly_factor::poly_factor ( )  [inline]
```

Definition at line 100 of file flintinterface.h.

3.117.2.2 ~poly_factor()

```
flint::poly_factor::~poly_factor ( )  [inline]
```

Definition at line 101 of file flintinterface.h.

3.117.3 Field Documentation

3.117.3.1 d

```
fmpz_poly_factor_t flint::poly_factor::d
```

Definition at line 99 of file flintinterface.h.

The documentation for this class was generated from the following file:

- flintinterface.h

3.118 POLYMOD Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD * coefs
- WORD numsym
- WORD arraysize
- WORD polysize
- WORD modnum

3.118.1 Detailed Description

The POLYMOD struct controls one univariate polynomial of which the coefficients have been taken modulus a (prime) number that fits inside a variable of type WORD. The polynomial is stored as an array of coefficients of size WORD.

Definition at line 1246 of file structs.h.

3.118.2 Field Documentation

3.118.2.1 coefs

```
WORD* POLYMOD::coefs
```

Definition at line 1247 of file structs.h.

3.118.2.2 numsym

```
WORD POLYMOD::numsym
```

Definition at line 1248 of file structs.h.

3.118.2.3 arraysize

```
WORD POLYMOD::arraysize
```

Definition at line 1249 of file structs.h.

3.118.2.4 polysize

```
WORD POLYMOD::polysize
```

Definition at line 1250 of file structs.h.

3.118.2.5 modnum

```
WORD POLYMOD::modnum
```

Definition at line 1251 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.119 PoSiTiOn Struct Reference

Data Fields

- off_t p1

3.119.1 Detailed Description

Definition at line 59 of file structs.h.

3.119.2 Field Documentation

3.119.2.1 p1

```
off_t PoSiTiOn::p1
```

Definition at line 60 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.120 PRELOAD Struct Reference

```
#include <structs.h>
```

Data Fields

- UBYTE * buffer
- LONG size

3.120.1 Detailed Description

Used by the preprocessor to load the contents of a doloop or a procedure. The struct PRELOAD is used both in the DOLOOP and PROCEDURE structs.

Definition at line 853 of file structs.h.

3.120.2 Field Documentation

3.120.2.1 buffer

```
UBYTE* PRELOAD::buffer
```

Definition at line 854 of file structs.h.

3.120.2.2 size

```
LONG PRELOAD::size
```

Definition at line 855 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.121 pReVaR Struct Reference

`#include <structs.h>`

Data Fields

- UBYTE * name
- UBYTE * value
- UBYTE * argnames
- int nargs
- int wildarg

3.121.1 Detailed Description

An element of the type PREVAR is needed for each preprocessor variable.

Definition at line 823 of file structs.h.

3.121.2 Field Documentation

3.121.2.1 name

`UBYTE* pReVaR::name`

allocated

Definition at line 824 of file structs.h.

3.121.2.2 value

`UBYTE* pReVaR::value`

points into memory of name

Definition at line 825 of file structs.h.

3.121.2.3 argnames

`UBYTE* pReVaR::argnames`

names of arguments, zero separated. points into memory of name

Definition at line 826 of file structs.h.

3.121.2.4 nargs

```
int pReVaR::nargs
```

0 = regular, >= 1: number of macro arguments. total number

Definition at line 827 of file structs.h.

3.121.2.5 wildarg

```
int pReVaR::wildarg
```

The number of a potential ?var. If none: 0. wildarg<nargs

Definition at line 828 of file structs.h.

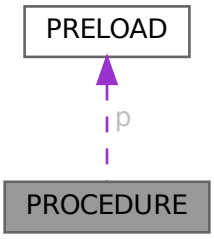
The documentation for this struct was generated from the following file:

- structs.h

3.122 PROCEDURE Struct Reference

```
#include <structs.h>
```

Collaboration diagram for PROCEDURE:



Data Fields

- PRELOAD p
- UBYTE * name
- int loadmode
- int mustfree

3.122.1 Detailed Description

An element of the type PROCEDURE is needed for each procedure in the system.

Definition at line 862 of file structs.h.

3.122.2 Field Documentation

3.122.2.1 p

```
PRELOAD PROCEDURE::p
```

Definition at line 863 of file structs.h.

3.122.2.2 name

```
UBYTE* PROCEDURE::name
```

Definition at line 864 of file structs.h.

3.122.2.3 loadmode

```
int PROCEDURE::loadmode
```

Definition at line 865 of file structs.h.

3.122.2.4 mustfree

```
int PROCEDURE::mustfree
```

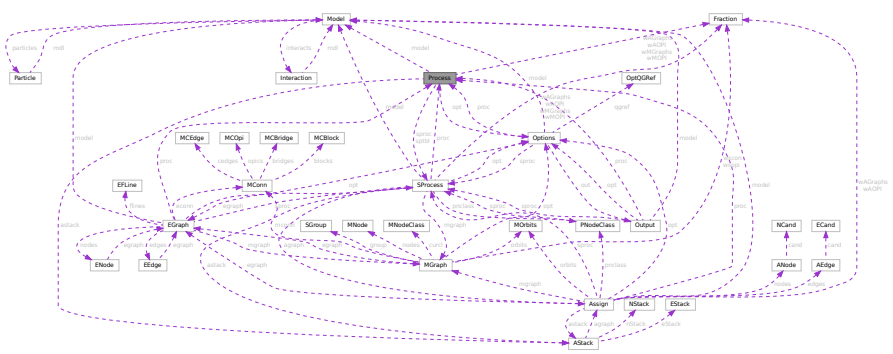
Definition at line 866 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.123 Process Class Reference

Collaboration diagram for Process:



Public Member Functions

- Process (int pid, Model *model, Options *opt, int nin, int *initlPart, int nfin, int *finalPart, int *coupling)
- **Process** (int pid, Model *model, Options *opt, FGInput *fgi)
- void prProcess (void)
- void outProcP (FILE *fp)
- void prProcessP (const char *fname)
- void mkSProcess (void)

Data Fields

- Model * model
- Options * opt
- BigInt mgrcount
- BigInt agrcount
- int * initlPart
- int * finalPart
- int id
- int ninitl
- int nfinal
- int ctotal
- int nExtern
- int loop
- int maxnlegs
- int clist [GRCC_MAXNCPLG]
- int nSubproc
- SProcess * sptbl [GRCC_MAXSUBPROCS]
- SProcess * sproc
- AStack * astack
- BigInt ngraphs
- BigInt nopi
- BigInt wgraphs
- BigInt wopi
- BigInt nMGraphs
- BigInt nMOPI
- Fraction wMGraphs
- Fraction wMOPI
- BigInt nAGraphs
- BigInt nAOPI
- Fraction wAGraphs
- Fraction wAOPI
- double sec

3.123.1 Detailed Description

Definition at line 447 of file grcc.h.

3.123.2 Constructor & Destructor Documentation

3.123.2.1 Process()

```
Process::Process (
            int pid,
            Model * model,
            Options * opt,
            int nin,
            int * initlPart,
            int nfin,
            int * finalPart,
            int * coupling )
```

Definition at line 3340 of file grcc.cc.

3.123.2.2 ~Process()

```
Process::~Process (
            void  )
```

Definition at line 3439 of file grcc.cc.

3.123.3 Member Function Documentation

3.123.3.1 prProcess()

```
void Process::prProcess (
            void  )
```

Definition at line 3450 of file grcc.cc.

3.123.3.2 outProcP()

```
void Process::outProcP (
            FILE * fp )
```

Definition at line 3473 of file grcc.cc.

3.123.3.3 prProcessP()

```
void Process::prProcessP (
            const char * fname )
```

Definition at line 3458 of file grcc.cc.

3.123.3.4 mkSProcess()

```
void Process::mkSProcess (
            void  )
```

Definition at line 3528 of file grcc.cc.

3.123.4 Field Documentation

3.123.4.1 model

```
Model* Process::model
```

Definition at line 449 of file grcc.h.

3.123.4.2 opt

```
Options* Process::opt
```

Definition at line 450 of file grcc.h.

3.123.4.3 mgrcount

```
BigInt Process::mgrcount
```

Definition at line 451 of file grcc.h.

3.123.4.4 agrcount

```
BigInt Process::agrcount
```

Definition at line 452 of file grcc.h.

3.123.4.5 initlPart

```
int* Process::initlPart
```

Definition at line 453 of file grcc.h.

3.123.4.6 finalPart

```
int* Process::finalPart
```

Definition at line 454 of file grcc.h.

3.123.4.7 id

```
int Process::id
```

Definition at line 456 of file grcc.h.

3.123.4.8 ninitl

```
int Process::ninitl
```

Definition at line 457 of file grcc.h.

3.123.4.9 nfinal

```
int Process::nfinal
```

Definition at line 458 of file grcc.h.

3.123.4.10 ctotal

```
int Process::ctotal
```

Definition at line 459 of file grcc.h.

3.123.4.11 nExtern

```
int Process::nExtern
```

Definition at line 460 of file grcc.h.

3.123.4.12 loop

```
int Process::loop
```

Definition at line 461 of file grcc.h.

3.123.4.13 maxnlegs

```
int Process::maxnlegs
```

Definition at line 462 of file grcc.h.

3.123.4.14 clist

```
int Process::clist[GRCC_MAXNCPLG]
```

Definition at line 463 of file grcc.h.

3.123.4.15 nSubproc

```
int Process::nSubproc
```

Definition at line 466 of file grcc.h.

3.123.4.16 sptbl

```
SProcess* Process::sptbl[GRCC_MAXSUBPROCS]
```

Definition at line 467 of file grcc.h.

3.123.4.17 sproc

```
SProcess* Process::sproc
```

Definition at line 468 of file grcc.h.

3.123.4.18 astack

```
AStack* Process::astack
```

Definition at line 471 of file grcc.h.

3.123.4.19 ngraphs

```
BigInt Process::ngraphs
```

Definition at line 474 of file grcc.h.

3.123.4.20 nopi

```
BigInt Process::nopi
```

Definition at line 475 of file grcc.h.

3.123.4.21 wgraphs

```
BigInt Process::wgraphs
```

Definition at line 476 of file grcc.h.

3.123.4.22 wopi

```
BigInt Process::wopi
```

Definition at line 477 of file grcc.h.

3.123.4.23 nMGraphs

```
BigInt Process::nMGraphs
```

Definition at line 480 of file grcc.h.

3.123.4.24 nMOPI

```
BigInt Process::nMOPI
```

Definition at line 481 of file grcc.h.

3.123.4.25 wMGraphs

```
Fraction Process::wMGraphs
```

Definition at line 482 of file grcc.h.

3.123.4.26 wMOPI

```
Fraction Process::wMOPI
```

Definition at line 483 of file grcc.h.

3.123.4.27 nAGraphs

```
BigInt Process::nAGraphs
```

Definition at line 485 of file grcc.h.

3.123.4.28 nAOPI

```
BigInt Process::nAOPI
```

Definition at line 486 of file grcc.h.

3.123.4.29 wAGraphs

```
Fraction Process::wAGraphs
```

Definition at line 487 of file grcc.h.

3.123.4.30 wAOPI

```
Fraction Process::wAOPI
```

Definition at line 488 of file grcc.h.

3.123.4.31 sec

```
double Process::sec
```

Definition at line 490 of file grcc.h.

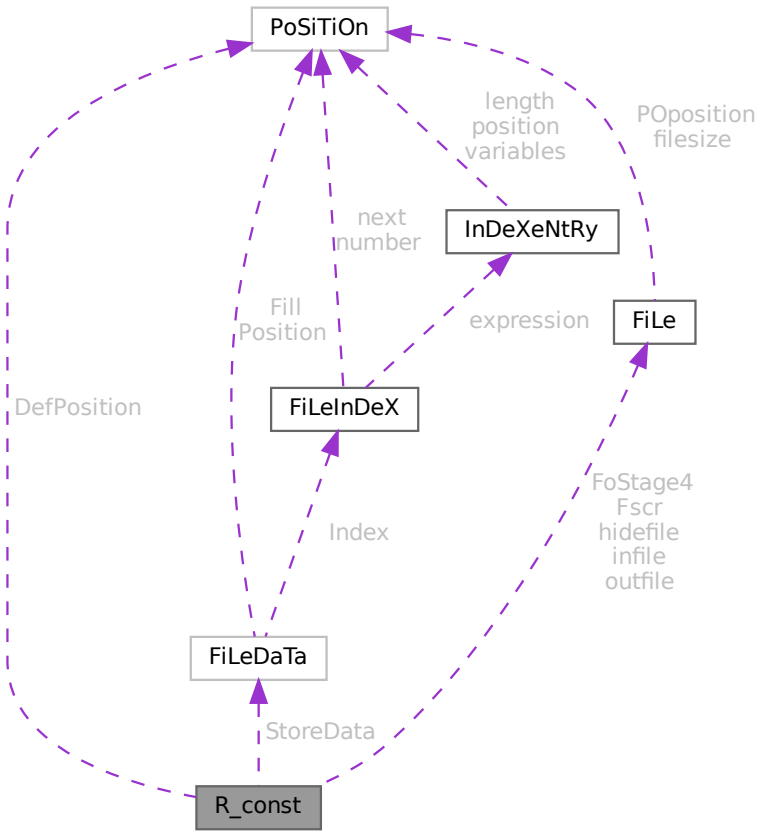
The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.124 R_const Struct Reference

```
#include <structs.h>
```

Collaboration diagram for R_const:



Data Fields

- FILEDATA StoreData
- FILEHANDLE Fscr [3]
- FILEHANDLE FoStage4 [2]
- POSITION DefPosition
- FILEHANDLE * infile
- FILEHANDLE * outfile
- FILEHANDLE * hidefile
- WORD * CompressBuffer
- WORD * ComprTop
- WORD * CompressPointer
- COMPAREDUMMY CompareRoutine
- ULONG * wranfia
- SBYTE * moebiustable
- LONG OldTime
- LONG InInBuf
- LONG InHiBuf
- LONG pWorkSize
- LONG lWorkSize
- LONG posWorkSize
- ULONG wranfseed
- int NoCompress
- int gzipCompress
- int Cnumlhs
- int outtohide
- int wranfcall
- int wranfnpair1
- int wranfnpair2
- WORD GetFile
- WORD KeptInHold
- WORD BracketOn
- WORD MaxBracket
- WORD CurDum
- WORD DeferFlag
- WORD TePos
- WORD sLevel
- WORD Stage4Name
- WORD GetOneFile
- WORD PolyFun
- WORD PolyFunInv
- WORD PolyFunType
- WORD PolyFunExp
- WORD PolyFunVar
- WORD PolyFunPow
- WORD Eside
- WORD MaxDum
- WORD level
- WORD expchanged
- WORD expflags
- WORD CurExpr
- WORD SortType
- WORD ShortSortCount
- WORD modeloptions
- WORD funoffset
- WORD moebiustablesize

3.124.1 Detailed Description

The R_const struct is part of the global data and resides either in the ALLGLOBALS struct A, or the ALLPRIVATES struct B (TFORM) under the name R We see it used with the macro AR as in AR.infile It has the variables that define the running environment and that should be transferred with a term in a multithreaded run.

Definition at line 1990 of file structs.h.

3.124.2 Field Documentation

3.124.2.1 StoreData

```
FILEDATA R_const::StoreData
```

Definition at line 1991 of file structs.h.

3.124.2.2 Fscr

```
FILEHANDLE R_const::Fscr[3]
```

Definition at line 1992 of file structs.h.

3.124.2.3 FoStage4

```
FILEHANDLE R_const::FoStage4[2]
```

Definition at line 1993 of file structs.h.

3.124.2.4 DefPosition

```
POSITION R_const::DefPosition
```

Definition at line 1994 of file structs.h.

3.124.2.5 infile

```
FILEHANDLE* R_const::infile
```

Definition at line 1995 of file structs.h.

3.124.2.6 outfile

```
FILEHANDLE* R_const::outfile
```

Definition at line 1996 of file structs.h.

3.124.2.7 hidefile

```
FILEHANDLE* R_const::hidefile
```

Definition at line 1997 of file structs.h.

3.124.2.8 CompressBuffer

```
WORD* R_const::CompressBuffer
```

Definition at line 1999 of file structs.h.

3.124.2.9 ComprTop

```
WORD* R_const::ComprTop
```

Definition at line 2000 of file structs.h.

3.124.2.10 CompressPointer

```
WORD* R_const::CompressPointer
```

Definition at line 2001 of file structs.h.

3.124.2.11 CompareRoutine

```
COMPAREDUMMY R_const::CompareRoutine
```

Definition at line 2002 of file structs.h.

3.124.2.12 wranfia

```
ULONG* R_const::wranfia
```

Definition at line 2003 of file structs.h.

3.124.2.13 moebiustable

```
SBYTE* R_const::moebiustable
```

Definition at line 2004 of file structs.h.

3.124.2.14 OldTime

```
LONG R_const::OldTime
```

Definition at line 2006 of file structs.h.

3.124.2.15 InInBuf

```
LONG R_const::InInBuf
```

Definition at line 2007 of file structs.h.

3.124.2.16 InHiBuf

```
LONG R_const::InHiBuf
```

Definition at line 2008 of file structs.h.

3.124.2.17 pWorkSize

```
LONG R_const::pWorkSize
```

Definition at line 2009 of file structs.h.

3.124.2.18 lWorkSize

```
LONG R_const::lWorkSize
```

Definition at line 2010 of file structs.h.

3.124.2.19 posWorkSize

```
LONG R_const::posWorkSize
```

Definition at line 2011 of file structs.h.

3.124.2.20 wranfseed

```
ULONG R_const::wranfseed
```

Definition at line 2012 of file structs.h.

3.124.2.21 NoCompress

```
int R_const::NoCompress
```

Definition at line 2013 of file structs.h.

3.124.2.22 gzipCompress

```
int R_const::gzipCompress
```

Definition at line 2014 of file structs.h.

3.124.2.23 Cnumlhs

```
int R_const::Cnumlhs
```

Definition at line 2015 of file structs.h.

3.124.2.24 outtohide

```
int R_const::outtohide
```

Definition at line 2016 of file structs.h.

3.124.2.25 wranfcall

```
int R_const::wranfcall
```

Definition at line 2020 of file structs.h.

3.124.2.26 wranfnpair1

```
int R_const::wranfnpair1
```

Definition at line 2021 of file structs.h.

3.124.2.27 wranfnpair2

```
int R_const::wranfnpair2
```

Definition at line 2022 of file structs.h.

3.124.2.28 GetFile

```
WORD R_const::GetFile
```

Definition at line 2028 of file structs.h.

3.124.2.29 KeptInHold

```
WORD R_const::KeptInHold
```

Definition at line 2029 of file structs.h.

3.124.2.30 BracketOn

```
WORD R_const::BracketOn
```

Definition at line 2030 of file structs.h.

3.124.2.31 MaxBracket

```
WORD R_const::MaxBracket
```

Definition at line 2031 of file structs.h.

3.124.2.32 CurDum

```
WORD R_const::CurDum
```

Definition at line 2032 of file structs.h.

3.124.2.33 DeferFlag

```
WORD R_const::DeferFlag
```

Definition at line 2033 of file structs.h.

3.124.2.34 TePos

```
WORD R_const::TePos
```

Definition at line 2034 of file structs.h.

3.124.2.35 sLevel

```
WORD R_const::sLevel
```

Definition at line 2035 of file structs.h.

3.124.2.36 Stage4Name

```
WORD R_const::Stage4Name
```

Definition at line 2036 of file structs.h.

3.124.2.37 GetOneFile

```
WORD R_const::GetOneFile
```

Definition at line 2037 of file structs.h.

3.124.2.38 PolyFun

```
WORD R_const::PolyFun
```

Definition at line 2038 of file structs.h.

3.124.2.39 PolyFunInv

```
WORD R_const::PolyFunInv
```

Definition at line 2039 of file structs.h.

3.124.2.40 PolyFunType

```
WORD R_const::PolyFunType
```

Definition at line 2040 of file structs.h.

3.124.2.41 PolyFunExp

```
WORD R_const::PolyFunExp
```

Definition at line 2041 of file structs.h.

3.124.2.42 PolyFunVar

```
WORD R_const::PolyFunVar
```

Definition at line 2042 of file structs.h.

3.124.2.43 PolyFunPow

```
WORD R_const::PolyFunPow
```

Definition at line 2043 of file structs.h.

3.124.2.44 Eside

```
WORD R_const::Eside
```

Definition at line 2044 of file structs.h.

3.124.2.45 MaxDum

```
WORD R_const::MaxDum
```

Definition at line 2045 of file structs.h.

3.124.2.46 level

```
WORD R_const::level
```

Definition at line 2046 of file structs.h.

3.124.2.47 expchanged

```
WORD R_const::expchanged
```

Definition at line 2047 of file structs.h.

3.124.2.48 expflags

```
WORD R_const::expflags
```

Definition at line 2048 of file structs.h.

3.124.2.49 CurExpr

```
WORD R_const::CurExpr
```

Definition at line 2049 of file structs.h.

3.124.2.50 SortType

```
WORD R_const::SortType
```

Definition at line 2050 of file structs.h.

3.124.2.51 ShortSortCount

```
WORD R_const::ShortSortCount
```

Definition at line 2051 of file structs.h.

3.124.2.52 modeloptions

```
WORD R_const::modeloptions
```

Definition at line 2052 of file structs.h.

3.124.2.53 funoffset

```
WORD R_const::funoffset
```

Definition at line 2053 of file structs.h.

3.124.2.54 moebiustablesize

```
WORD R_const::moebiustablesize
```

Definition at line 2054 of file structs.h.

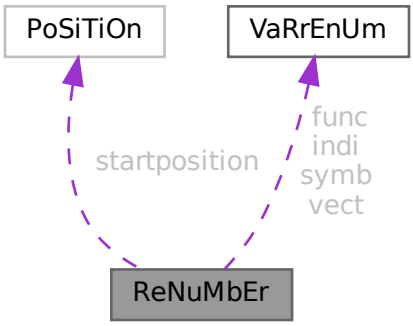
The documentation for this struct was generated from the following file:

- structs.h

3.125 ReNuMbEr Struct Reference

```
#include <structs.h>
```

Collaboration diagram for ReNuMbEr:



Data Fields

- POSITION startposition
- VARRENUM symb
- VARRENUM indi
- VARRENUM vect
- VARRENUM func
- WORD * symnum
- WORD * indnum
- WORD * vecnum
- WORD * funnum

3.125.1 Detailed Description

Only symb.lo gets dynamically allocated. All other pointers points into this memory.

Definition at line 176 of file structs.h.

3.125.2 Field Documentation

3.125.2.1 startposition

```
POSITION ReNuMbEr::startposition
```

Definition at line 177 of file structs.h.

3.125.2.2 symb

```
VARRENUM ReNuMbEr::symb
```

Symbols

Definition at line 179 of file structs.h.

Referenced by DoRecovery(), Generator(), GetFirstTerm(), and TermRenumber().

3.125.2.3 indi

```
VARRENUM ReNuMbEr::indi
```

Indices

Definition at line 180 of file structs.h.

Referenced by DoRecovery(), and TermRenumber().

3.125.2.4 vect

```
VARRENUM ReNuMbEr::vect
```

Vectors

Definition at line 181 of file structs.h.

Referenced by DoRecovery(), and TermRenumber().

3.125.2.5 func

```
VARRENUM ReNuMbEr::func
```

Functions

Definition at line 182 of file structs.h.

Referenced by DoRecovery(), and TermRenumber().

3.125.2.6 symnum

```
WORD* ReNuMbEr::symnum
```

Renumbered symbols

Definition at line 184 of file structs.h.

Referenced by DoRecovery(), and TermRenumber().

3.125.2.7 indnum

```
WORD* ReNuMbEr::indnum
```

Renumbered indices

Definition at line 185 of file structs.h.

Referenced by DoRecovery(), and TermRenumber().

3.125.2.8 vecnum

```
WORD* ReNuMbEr::vecnum
```

Renumbered vectors

Definition at line 186 of file structs.h.

Referenced by DoRecovery(), and TermRenumber().

3.125.2.9 funnum

```
WORD* ReNuMbEr::funnum
```

Renumbered functions

Definition at line 187 of file structs.h.

Referenced by DoRecovery(), and TermRenumber().

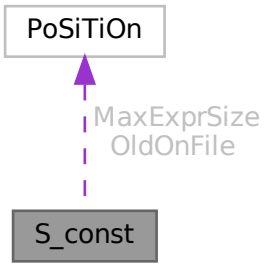
The documentation for this struct was generated from the following file:

- structs.h

3.126 S_const Struct Reference

`#include <structs.h>`

Collaboration diagram for S_const:



Data Fields

- POSITION MaxExprSize
- POSITION * OldOnFile
- WORD * OldNumFactors
- WORD * Oldvflags
- WORD * Olduflags
- int NumOldOnFile
- int NumOldNumFactors
- int MultiThreaded
- int Balancing
- WORD ExecMode
- WORD CollectOverFlag

3.126.1 Detailed Description

The S_const struct is part of the global data and resides in the ALLGLOBALS struct A under the name S We see it used with the macro AS as in AS.ExecMode It has some variables used by the master in multithreaded runs

Definition at line 1945 of file structs.h.

3.126.2 Field Documentation

3.126.2.1 MaxExprSize

```
POSITION S_const::MaxExprSize
```

Definition at line 1946 of file structs.h.

3.126.2.2 OldOnFile

```
POSITION* S_const::OldOnFile
```

Definition at line 1952 of file structs.h.

3.126.2.3 OldNumFactors

```
WORD* S_const::OldNumFactors
```

Definition at line 1953 of file structs.h.

3.126.2.4 Oldvflags

```
WORD* S_const::Oldvflags
```

Definition at line 1954 of file structs.h.

3.126.2.5 Olduflags

```
WORD* S_const::Olduflags
```

Definition at line 1955 of file structs.h.

3.126.2.6 NumOldOnFile

```
int S_const::NumOldOnFile
```

Definition at line 1959 of file structs.h.

3.126.2.7 NumOldNumFactors

```
int S_const::NumOldNumFactors
```

Definition at line 1960 of file structs.h.

3.126.2.8 MultiThreaded

```
int S_const::MultiThreaded
```

Definition at line 1961 of file structs.h.

3.126.2.9 Balancing

```
int S_const::Balancing
```

Definition at line 1968 of file structs.h.

3.126.2.10 ExecMode

```
WORD S_const::ExecMode
```

Definition at line 1969 of file structs.h.

3.126.2.11 CollectOverFlag

```
WORD S_const::CollectOverFlag
```

Definition at line 1971 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.127 SeTs Struct Reference

Data Fields

- LONG name
- WORD type
- WORD first
- WORD last
- WORD node
- WORD namesize
- WORD dimension
- WORD flags

3.127.1 Detailed Description

Definition at line 513 of file structs.h.

3.127.2 Field Documentation

3.127.2.1 name

```
LONG SeTs::name
```

Definition at line 514 of file structs.h.

3.127.2.2 type

```
WORD SeTs::type
```

Definition at line 515 of file structs.h.

3.127.2.3 first

```
WORD SeTs::first
```

Definition at line 516 of file structs.h.

3.127.2.4 last

```
WORD SeTs::last
```

Definition at line 517 of file structs.h.

3.127.2.5 node

```
WORD SeTs::node
```

Definition at line 518 of file structs.h.

3.127.2.6 namesize

```
WORD SeTs::namesize
```

Definition at line 519 of file structs.h.

3.127.2.7 dimension

```
WORD SeTs::dimension
```

Definition at line 520 of file structs.h.

3.127.2.8 flags

```
WORD SeTs::flags
```

Definition at line 521 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.128 SETUPPARAMETERS Struct Reference

```
#include <structs.h>
```

Data Fields

- UBYTE * parameter
- int type
- int flags
- LONG value

3.128.1 Detailed Description

Each setup parameter has one element of the struct SETUPPARAMETERS assigned to it. By binary search in the array of them we can then locate the proper element by name. We have to assume that two ints make a long and either one or two longs make a pointer. The long before the ints and padding gives a problem in the initialization.

Definition at line 1014 of file structs.h.

3.128.2 Field Documentation

3.128.2.1 parameter

```
UBYTE* SETUPPARAMETERS::parameter
```

Definition at line 1015 of file structs.h.

3.128.2.2 type

```
int SETUPPARAMETERS::type
```

Definition at line 1016 of file structs.h.

3.128.2.3 flags

```
int SETUPPARAMETERS::flags
```

Definition at line 1017 of file structs.h.

3.128.2.4 value

```
LONG SETUPPARAMETERS::value
```

Definition at line 1021 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.129 SGroup Class Reference

Public Member Functions

- void print (void)
- void newGroup (int nelm, int nclss, int ∗clss)
- void clearGroup (void)
- void delGroup (void)
- int ∗ genNext (void)
- void addGroup (int ∗p)
- BigInt nElem (void)
- int ∗ nextElem (void)

Data Fields

- BigInt size
- BigInt nelem
- int ∗∗ elem
- int nnodes
- int neclass
- int eclass [GRCC_MAXNODES]
- int cgen
- int csav
- int permg [GRCC_MAXNODES]
- int perms [GRCC_MAXNODES]
- int pgr [GRCC_MAXNODES]
- int pgq [GRCC_MAXNODES]
- int psr [GRCC_MAXNODES]
- int psq [GRCC_MAXNODES]

3.129.1 Detailed Description

Definition at line 732 of file grcc.h.

3.129.2 Constructor & Destructor Documentation

3.129.2.1 SGroup()

```
SGroup::SGroup (
            void  )
```

Definition at line 5874 of file grcc.cc.

3.129.2.2 ∼SGroup()

```
SGroup::∼SGroup (
            void  )
```

Definition at line 5883 of file grcc.cc.

3.129.3 Member Function Documentation

3.129.3.1 print()

```
void SGroup::print (
            void  )
```

Definition at line 5900 of file grcc.cc.

3.129.3.2 newGroup()

```
void SGroup::newGroup (
            int nelm,
            int nclss,
            int * clss )
```

Definition at line 5915 of file grcc.cc.

3.129.3.3 clearGroup()

```
void SGroup::clearGroup (
            void  )
```

Definition at line 5940 of file grcc.cc.

3.129.3.4 delGroup()

```
void SGroup::delGroup (
            void  )
```

Definition at line 5948 of file grcc.cc.

3.129.3.5 genNext()

```
int * SGroup::genNext (
            void  )
```

Definition at line 5964 of file grcc.cc.

3.129.3.6 addGroup()

```
void SGroup::addGroup (
            int * p )
```

Definition at line 5974 of file grcc.cc.

3.129.3.7 nElem()

```
BigInt SGroup::nElem (
            void  )
```

Definition at line 5991 of file grcc.cc.

3.129.3.8 nextElem()

```
int * SGroup::nextElem (
            void  )
```

Definition at line 5997 of file grcc.cc.

3.129.4 Field Documentation

3.129.4.1 size

```
BigInt SGroup::size
```

Definition at line 734 of file grcc.h.

3.129.4.2 nelem

```
BigInt SGroup::nelem
```

Definition at line 735 of file grcc.h.

3.129.4.3 elem

```
int** SGroup::elem
```

Definition at line 736 of file grcc.h.

3.129.4.4 nnodes

```
int SGroup::nnodes
```

Definition at line 737 of file grcc.h.

3.129.4.5 neclass

```
int SGroup::neclass
```

Definition at line 738 of file grcc.h.

3.129.4.6 eclass

```
int SGroup::eclass[GRCC_MAXNODES]
```

Definition at line 739 of file grcc.h.

3.129.4.7 cgen

```
int SGroup::cgen
```

Definition at line 740 of file grcc.h.

3.129.4.8 csav

```
int SGroup::csav
```

Definition at line 741 of file grcc.h.

3.129.4.9 permg

```
int SGroup::permg[GRCC_MAXNODES]
```

Definition at line 742 of file grcc.h.

3.129.4.10 perms

```
int SGroup::perms[GRCC_MAXNODES]
```

Definition at line 743 of file grcc.h.

3.129.4.11 pgr

```
int SGroup::pgr[GRCC_MAXNODES]
```

Definition at line 745 of file grcc.h.

3.129.4.12 pgq

```
int SGroup::pgq[GRCC_MAXNODES]
```

Definition at line 746 of file grcc.h.

3.129.4.13 psr

```
int SGroup::psr[GRCC_MAXNODES]
```

Definition at line 747 of file grcc.h.

3.129.4.14 psq

```
int SGroup::psq[GRCC_MAXNODES]
```

Definition at line 748 of file grcc.h.

The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.130 SHvariables Struct Reference

Data Fields

- WORD * outterm
- WORD * outfun
- WORD * incoef
- WORD * stop1
- WORD * stop2
- WORD * ststop1
- WORD * ststop2
- FINISHUFFLE finishuf
- DO_UFFLE do_uffle
- LONG combilast
- WORD nincoef
- WORD level
- WORD thefunction
- WORD option

3.130.1 Detailed Description

Definition at line 1254 of file structs.h.

3.130.2 Field Documentation

3.130.2.1 outterm

```
WORD* SHvariables::outterm
```

Definition at line 1255 of file structs.h.

3.130.2.2 outfun

```
WORD* SHvariables::outfun
```

Definition at line 1256 of file structs.h.

3.130.2.3 incoef

```
WORD* SHvariables::incoef
```

Definition at line 1257 of file structs.h.

3.130.2.4 stop1

```
WORD* SHvariables::stop1
```

Definition at line 1258 of file structs.h.

3.130.2.5 stop2

```
WORD* SHvariables::stop2
```

Definition at line 1259 of file structs.h.

3.130.2.6 ststop1

```
WORD* SHvariables::ststop1
```

Definition at line 1260 of file structs.h.

3.130.2.7 ststop2

```
WORD* SHvariables::ststop2
```

Definition at line 1261 of file structs.h.

3.130.2.8 finishuf

```
FINISHUFFLE SHvariables::finishuf
```

Definition at line 1262 of file structs.h.

3.130.2.9 do_uffle

```
DO_UFFLE SHvariables::do_uffle
```

Definition at line 1263 of file structs.h.

3.130.2.10 combilast

```
LONG SHvariables::combilast
```

Definition at line 1264 of file structs.h.

3.130.2.11 nincoef

```
WORD SHvariables::nincoef
```

Definition at line 1265 of file structs.h.

3.130.2.12 level

```
WORD SHvariables::level
```

Definition at line 1266 of file structs.h.

3.130.2.13 thefunction

```
WORD SHvariables::thefunction
```

Definition at line 1267 of file structs.h.

3.130.2.14 option

```
WORD SHvariables::option
```

Definition at line 1268 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.131 sOrT Struct Reference

```
#include <structs.h>
```

Collaboration diagram for sOrT:

Data Fields

- FILEHANDLE file
- POSITION SizeInFile [3]
- POSITION OldPosIn
- POSITION OldPosOut
- WORD * lBuffer
- WORD * lTop
- WORD * lFill
- WORD * used
- WORD * sBuffer
- WORD * sTop
- WORD * sTop2
- WORD * sHalf
- WORD * sFill
- WORD ** sPointer
- WORD ** PoinFill
- WORD * cBuffer
- WORD ** Patches
- WORD ** pStop
- WORD ** poina
- WORD ** poin2a
- WORD * ktoi
- WORD * tree
- POSITION * fPatches

- POSITION * inPatches
- POSITION * fPatchesStop
- POSITION * iPatches
- FILEHANDLE * f
- FILEHANDLE ** ff
- LONG sTerms
- LONG LargeSize
- LONG SmallSize
- LONG SmallEsize
- LONG TermsInSmall
- LONG Terms2InSmall
- LONG GenTerms
- LONG TermsLeft
- LONG GenSpace
- LONG SpaceLeft
- LONG putinsize
- LONG ninterms
- LONG verbComparisons
- LONG verbSBsortTerms
- LONG verbSBsortCap
- LONG verbLBsortPatches
- LONG verbLBsortCap
- LONG verbUnsortedSize
- LONG verbMaxTermSize
- int MaxPatches
- int MaxFpatches
- int type
- int lPatch
- int fPatchN1
- int PolyWise
- int PolyFlag
- int cBufferSize
- int maxtermsize
- int newmaxtermsize
- int outputmode
- int stagelevel
- WORD fPatchN
- WORD inNum
- WORD stage4

3.131.1 Detailed Description

The struct SORTING is used to control a sort operation. It includes a small and a large buffer and arrays for keeping track of various stages of the (merge) sorts. Each sort level has its own struct and different levels can have different sizes for its arrays. Also different threads have their own set of SORTING structs.

Definition at line 1114 of file structs.h.

3.131.2 Field Documentation

3.131.2.1 file

```
FILEHANDLE sOrT::file
```

Definition at line 1115 of file structs.h.

3.131.2.2 SizeInFile

```
POSITION sOrT::SizeInFile[3]
```

Definition at line 1116 of file structs.h.

3.131.2.3 OldPosIn

```
POSITION sOrT::OldPosIn
```

Definition at line 1117 of file structs.h.

3.131.2.4 OldPosOut

```
POSITION sOrT::OldPosOut
```

Definition at line 1118 of file structs.h.

3.131.2.5 lBuffer

```
WORD* sOrT::lBuffer
```

Definition at line 1119 of file structs.h.

3.131.2.6 lTop

```
WORD* sOrT::lTop
```

Definition at line 1120 of file structs.h.

3.131.2.7 lFill

```
WORD* sOrT::lFill
```

Definition at line 1121 of file structs.h.

3.131.2.8 used

```
WORD* sOrT::used
```

Definition at line 1122 of file structs.h.

3.131.2.9 sBuffer

```
WORD* sOrT::sBuffer
```

Definition at line 1123 of file structs.h.

3.131.2.10 sTop

```
WORD* sOrT::sTop
```

Definition at line 1124 of file structs.h.

3.131.2.11 sTop2

```
WORD* sOrT::sTop2
```

Definition at line 1125 of file structs.h.

3.131.2.12 sHalf

```
WORD* sOrT::sHalf
```

Definition at line 1126 of file structs.h.

3.131.2.13 sFill

```
WORD* sOrT::sFill
```

Definition at line 1127 of file structs.h.

3.131.2.14 sPointer

```
WORD** sOrT::sPointer
```

Definition at line 1128 of file structs.h.

3.131.2.15 PoinFill

```
WORD** sOrT::PoinFill
```

Definition at line 1129 of file structs.h.

3.131.2.16 cBuffer

```
WORD* sOrT::cBuffer
```

Definition at line 1130 of file structs.h.

3.131.2.17 Patches

```
WORD** sOrT::Patches
```

Definition at line 1131 of file structs.h.

3.131.2.18 pStop

```
WORD** sOrT::pStop
```

Definition at line 1132 of file structs.h.

3.131.2.19 poina

```
WORD** sOrT::poina
```

Definition at line 1133 of file structs.h.

3.131.2.20 poin2a

```
WORD** sOrT::poin2a
```

Definition at line 1134 of file structs.h.

3.131.2.21 ktoi

```
WORD* sOrT::ktoi
```

Definition at line 1135 of file structs.h.

3.131.2.22 tree

```
WORD* sOrT::tree
```

Definition at line 1136 of file structs.h.

3.131.2.23 fPatches

```
POSITION* sOrT::fPatches
```

Definition at line 1142 of file structs.h.

3.131.2.24 inPatches

```
POSITION* sOrT::inPatches
```

Definition at line 1143 of file structs.h.

3.131.2.25 fPatchesStop

```
POSITION* sOrT::fPatchesStop
```

Definition at line 1144 of file structs.h.

3.131.2.26 iPatches

```
POSITION* sOrT::iPatches
```

Definition at line 1145 of file structs.h.

3.131.2.27 f

```
FILEHANDLE* sOrT::f
```

Definition at line 1146 of file structs.h.

3.131.2.28 ff

```
FILEHANDLE** sOrT::ff
```

Definition at line 1147 of file structs.h.

3.131.2.29 sTerms

```
LONG sOrT::sTerms
```

Definition at line 1148 of file structs.h.

3.131.2.30 LargeSize

```
LONG sOrT::LargeSize
```

Definition at line 1149 of file structs.h.

3.131.2.31 SmallSize

```
LONG sOrT::SmallSize
```

Definition at line 1150 of file structs.h.

3.131.2.32 SmallEsize

```
LONG sOrT::SmallEsize
```

Definition at line 1151 of file structs.h.

3.131.2.33 TermsInSmall

```
LONG sOrT::TermsInSmall
```

Definition at line 1152 of file structs.h.

3.131.2.34 Terms2InSmall

```
LONG sOrT::Terms2InSmall
```

Definition at line 1153 of file structs.h.

3.131.2.35 GenTerms

```
LONG sOrT::GenTerms
```

Definition at line 1154 of file structs.h.

3.131.2.36 TermsLeft

```
LONG sOrT::TermsLeft
```

Definition at line 1155 of file structs.h.

3.131.2.37 GenSpace

```
LONG sOrT::GenSpace
```

Definition at line 1156 of file structs.h.

3.131.2.38 SpaceLeft

```
LONG sOrT::SpaceLeft
```

Definition at line 1157 of file structs.h.

3.131.2.39 putinsize

```
LONG sOrT::putinsize
```

Definition at line 1158 of file structs.h.

3.131.2.40 ninterms

```
LONG sOrT::ninterms
```

Definition at line 1159 of file structs.h.

3.131.2.41 verbComparisons

```
LONG sOrT::verbComparisons
```

Definition at line 1160 of file structs.h.

3.131.2.42 verbSBsortTerms

```
LONG sOrT::verbSBsortTerms
```

Definition at line 1161 of file structs.h.

3.131.2.43 verbSBsortCap

```
LONG sOrT::verbSBsortCap
```

Definition at line 1162 of file structs.h.

3.131.2.44 verbLBsortPatches

```
LONG sOrT::verbLBsortPatches
```

Definition at line 1163 of file structs.h.

3.131.2.45 verbLBsortCap

```
LONG sOrT::verbLBsortCap
```

Definition at line 1164 of file structs.h.

3.131.2.46 verbUnsortedSize

```
LONG sOrT::verbUnsortedSize
```

Definition at line 1165 of file structs.h.

3.131.2.47 verbMaxTermSize

```
LONG sOrT::verbMaxTermSize
```

Definition at line 1166 of file structs.h.

3.131.2.48 MaxPatches

```
int sOrT::MaxPatches
```

Definition at line 1167 of file structs.h.

3.131.2.49 MaxFpatches

```
int sOrT::MaxFpatches
```

Definition at line 1168 of file structs.h.

3.131.2.50 type

```
int sOrT::type
```

Definition at line 1169 of file structs.h.

3.131.2.51 lPatch

```
int sOrT::lPatch
```

Definition at line 1170 of file structs.h.

3.131.2.52 fPatchN1

```
int sOrT::fPatchN1
```

Definition at line 1171 of file structs.h.

3.131.2.53 PolyWise

```
int sOrT::PolyWise
```

Definition at line 1172 of file structs.h.

3.131.2.54 PolyFlag

```
int sOrT::PolyFlag
```

Definition at line 1173 of file structs.h.

3.131.2.55 cBufferSize

```
int sOrT::cBufferSize
```

Definition at line 1174 of file structs.h.

3.131.2.56 maxtermsize

```
int sOrT::maxtermsize
```

Definition at line 1175 of file structs.h.

3.131.2.57 newmaxtermsize

```
int sOrT::newmaxtermsize
```

Definition at line 1176 of file structs.h.

3.131.2.58 outputmode

```
int sOrT::outputmode
```

Definition at line 1177 of file structs.h.

3.131.2.59 stagelevel

```
int sOrT::stagelevel
```

Definition at line 1178 of file structs.h.

3.131.2.60 fPatchN

```
WORD sOrT::fPatchN
```

Definition at line 1179 of file structs.h.

3.131.2.61 inNum

```
WORD sOrT::inNum
```

Definition at line 1180 of file structs.h.

3.131.2.62 stage4

```
WORD sOrT::stage4
```

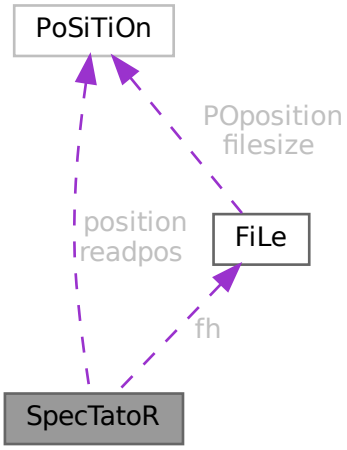
Definition at line 1181 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.132 SpecTatoR Struct Reference

Collaboration diagram for SpecTatoR:



Data Fields

- POSITION position
- POSITION readpos
- FILEHANDLE * fh
- char * name
- WORD exprnumber
- WORD flags

3.132.1 Detailed Description

Definition at line 750 of file structs.h.

3.132.2 Field Documentation

3.132.2.1 position

```
POSITION SpecTatoR::position
```

Definition at line 751 of file structs.h.

3.132.2.2 readpos

```
POSITION SpecTatoR::readpos
```

Definition at line 752 of file structs.h.

3.132.2.3 fh

```
FILEHANDLE* SpecTatoR::fh
```

Definition at line 753 of file structs.h.

3.132.2.4 name

```
char* SpecTatoR::name
```

Definition at line 754 of file structs.h.

3.132.2.5 exprnumber

```
WORD SpecTatoR::exprnumber
```

Definition at line 755 of file structs.h.

3.132.2.6 flags

```
WORD SpecTatoR::flags
```

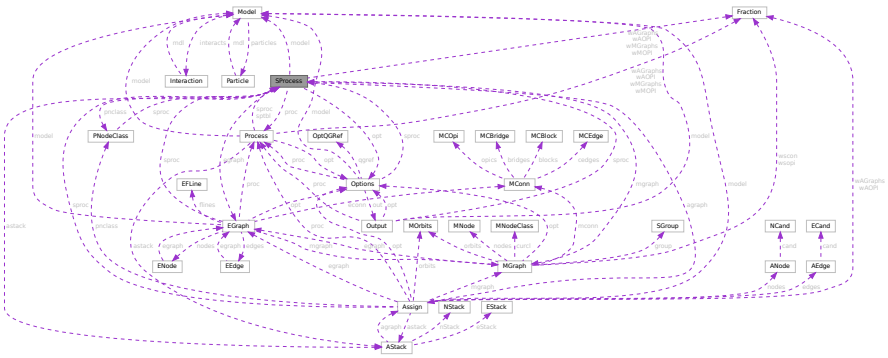
Definition at line 756 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.133 SProcess Class Reference

Collaboration diagram for SProcess:



Public Member Functions

- SProcess (Model ∗mdl, Process ∗prc, Options ∗opts, int sid, int ∗clst, int ncls, NCInput ∗cls)
- SProcess (Model ∗mdl, Process ∗prc, Options ∗opts, int sid, int ∗clst, int ncls, int ∗cdeg, int ∗ctyp, int ∗ptcl, int ∗cpl, int ∗cnum, int ∗cmind, int ∗cmaxd)
- void prSProcess (void)
- BigInt generate (void)
- void assign (MGraph ∗mgr)
- int toMNodeClass (int ∗ctyp, int ∗cldeg, int ∗clnum, int ∗cmind, int ∗cmaxd)
- PNodeClass ∗ match (MGraph ∗mgr)
- void endMGraph (MGraph ∗mgr)
- void endAGraph (EGraph ∗egr)
- void resultMGraph (BigInt nmgraphs, Fraction mwsum, BigInt nmopi, Fraction mwopi)
- void resultAGraph (BigInt nagraphs, Fraction awsum, BigInt naopi, Fraction awopi)

Data Fields

- Model * model
- Process * proc
- Options * opt
- PNodeClass * pnclass
- AStack * astack
- MGraph * mgraph
- EGraph * egraph
- Assign * agraph
- int * cl2nd
- int * nd2cl
- int nclass
- int ninitl
- int nfinal
- int nvert
- int clist [GRCC_MAXNCPLG]
- int id
- int loop
- int nNodes
- int nEdges
- int nExtern
- int ncouple
- int tCouple
- int DUMMYPADDING
- BigInt mgrcount
- BigInt agrcount
- BigInt extperm
- BigInt nMGraphs
- BigInt nMOPI
- Fraction wMGraphs
- Fraction wMOPI
- BigInt nAGraphs
- BigInt nAOPI
- Fraction wAGraphs
- Fraction wAOPI

3.133.1 Detailed Description

Definition at line 373 of file grcc.h.

3.133.2 Constructor & Destructor Documentation

3.133.2.1 SProcess() [1/2]

```
SProcess::SProcess (
            Model * mdl,
            Process * prc,
            Options * opts,
            int sid,
            int * clst,
            int ncls,
            NCInput * cls )
```

Definition at line 2998 of file grcc.cc.

3.133.2.2 SProcess() [2/2]

```
SProcess::SProcess (
            Model * mdl,
            Process * prc,
            Options * opts,
            int sid,
            int * clst,
            int ncls,
            int * cdeg,
            int * ctyp,
            int * ptcl,
            int * cpl,
            int * cnum,
            int * cmind,
            int * cmaxd )
```

Definition at line 2829 of file grcc.cc.

3.133.2.3 ~SProcess()

```
SProcess::~SProcess (
            void  )
```

Definition at line 3160 of file grcc.cc.

3.133.3 Member Function Documentation

3.133.3.1 prSProcess()

```
void SProcess::prSProcess (
            void  )
```

Definition at line 3169 of file grcc.cc.

3.133.3.2 generate()

```
BigInt SProcess::generate (
            void  )
```

Definition at line 3181 of file grcc.cc.

3.133.3.3 assign()

```
void SProcess::assign (
            MGraph * mgr )
```

Definition at line 3225 of file grcc.cc.

3.133.3.4 toMNodeClass()

```
int SProcess::toMNodeClass (
            int * ctyp,
            int * cldeg,
            int * clnum,
            int * cmind,
            int * cmaxd )
```

Definition at line 3206 of file grcc.cc.

3.133.3.5 match()

```
PNodeClass * SProcess::match (
            MGraph * mgr )
```

Definition at line 3242 of file grcc.cc.

3.133.3.6 endMGraph()

```
void SProcess::endMGraph (
            MGraph * mgr )
```

Definition at line 3321 of file grcc.cc.

3.133.3.7 endAGraph()

```
void SProcess::endAGraph (
            EGraph * egr )
```

Definition at line 3331 of file grcc.cc.

3.133.3.8 resultMGraph()

```
void SProcess::resultMGraph (
            BigInt nmgraphs,
            Fraction mwsum,
            BigInt nmopi,
            Fraction mwopi )
```

Definition at line 3303 of file grcc.cc.

3.133.3.9 resultAGraph()

```
void SProcess::resultAGraph (
            BigInt nagraphs,
            Fraction awsum,
            BigInt naopi,
            Fraction awopi )
```

Definition at line 3312 of file grcc.cc.

3.133.4 Field Documentation

3.133.4.1 model

```
Model* SProcess::model
```

Definition at line 380 of file grcc.h.

3.133.4.2 proc

```
Process* SProcess::proc
```

Definition at line 381 of file grcc.h.

3.133.4.3 opt

```
Options* SProcess::opt
```

Definition at line 382 of file grcc.h.

3.133.4.4 pnclass

```
PNodeClass* SProcess::pnclass
```

Definition at line 383 of file grcc.h.

3.133.4.5 astack

```
AStack* SProcess::astack
```

Definition at line 384 of file grcc.h.

3.133.4.6 mgraph

```
MGraph* SProcess::mgraph
```

Definition at line 386 of file grcc.h.

3.133.4.7 egraph

```
EGraph* SProcess::egraph
```

Definition at line 387 of file grcc.h.

3.133.4.8 agraph

```
Assign* SProcess::agraph
```

Definition at line 388 of file grcc.h.

3.133.4.9 cl2nd

```
int* SProcess::cl2nd
```

Definition at line 390 of file grcc.h.

3.133.4.10 nd2cl

```
int* SProcess::nd2cl
```

Definition at line 392 of file grcc.h.

3.133.4.11 nclass

```
int SProcess::nclass
```

Definition at line 393 of file grcc.h.

3.133.4.12 ninitl

```
int SProcess::ninitl
```

Definition at line 395 of file grcc.h.

3.133.4.13 nfinal

```
int SProcess::nfinal
```

Definition at line 396 of file grcc.h.

3.133.4.14 nvert

```
int SProcess::nvert
```

Definition at line 397 of file grcc.h.

3.133.4.15 clist

```
int SProcess::clist[GRCC_MAXNCPLG]
```

Definition at line 399 of file grcc.h.

3.133.4.16 id

```
int SProcess::id
```

Definition at line 400 of file grcc.h.

3.133.4.17 loop

```
int SProcess::loop
```

Definition at line 401 of file grcc.h.

3.133.4.18 nNodes

```
int SProcess::nNodes
```

Definition at line 402 of file grcc.h.

3.133.4.19 nEdges

```
int SProcess::nEdges
```

Definition at line 403 of file grcc.h.

3.133.4.20 nExtern

```
int SProcess::nExtern
```

Definition at line 404 of file grcc.h.

3.133.4.21 ncouple

```
int SProcess::ncouple
```

Definition at line 405 of file grcc.h.

3.133.4.22 tCouple

```
int SProcess::tCouple
```

Definition at line 406 of file grcc.h.

3.133.4.23 DUMMYPADDING

```
int SProcess::DUMMYPADDING
```

Definition at line 408 of file grcc.h.

3.133.4.24 mgrcount

```
BigInt SProcess::mgrcount
```

Definition at line 410 of file grcc.h.

3.133.4.25 agrcount

```
BigInt SProcess::agrcount
```

Definition at line 411 of file grcc.h.

3.133.4.26 extperm

```
BigInt SProcess::extperm
```

Definition at line 412 of file grcc.h.

3.133.4.27 nMGraphs

```
BigInt SProcess::nMGraphs
```

Definition at line 415 of file grcc.h.

3.133.4.28 nMOPI

```
BigInt SProcess::nMOPI
```

Definition at line 416 of file grcc.h.

3.133.4.29 wMGraphs

```
Fraction SProcess::wMGraphs
```

Definition at line 417 of file grcc.h.

3.133.4.30 wMOPI

```
Fraction SProcess::wMOPI
```

Definition at line 418 of file grcc.h.

3.133.4.31 nAGraphs

```
BigInt SProcess::nAGraphs
```

Definition at line 420 of file grcc.h.

3.133.4.32 nAOPI

```
BigInt SProcess::nAOPI
```

Definition at line 421 of file grcc.h.

3.133.4.33 wAGraphs

```
Fraction SProcess::wAGraphs
```

Definition at line 422 of file grcc.h.

3.133.4.34 wAOPI

```
Fraction SProcess::wAOPI
```

Definition at line 423 of file grcc.h.

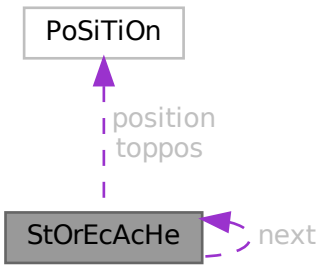
The documentation for this class was generated from the following files:

- grcc.h
- grcc.cc

3.134 StOrEcAcHe Struct Reference

```
#include <structs.h>
```

Collaboration diagram for StOrEcAcHe:



Data Fields

- POSITION position
- POSITION toppos
- struct StOrEcAcHe * next
- WORD buffer [2]

3.134.1 Detailed Description

The struct of type STORECACHE is used by a caching system for reading terms from stored expressions. Each thread should have its own system of caches.

Definition at line 1044 of file structs.h.

3.134.2 Field Documentation

3.134.2.1 position

```
POSITION StOrEcAcHe::position
```

Definition at line 1045 of file structs.h.

3.134.2.2 toppos

```
POSITION StOrEcAcHe::toppos
```

Definition at line 1046 of file structs.h.

3.134.2.3 next

```
struct StOrEcAcHe* StOrEcAcHe::next
```

Definition at line 1047 of file structs.h.

3.134.2.4 buffer

```
WORD StOrEcAcHe::buffer[2]
```

Definition at line 1048 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.135 STOREHEADER Struct Reference

```
#include <structs.h>
```

Data Fields

- UBYTE headermark [8]
- UBYTE lenWORD
- UBYTE lenLONG
- UBYTE lenPOS
- UBYTE lenPOINTER
- UBYTE endianness [16]
- UBYTE sSym
- UBYTE sInd
- UBYTE sVec
- UBYTE sFun
- UBYTE maxpower [16]
- UBYTE wildoffset [16]
- UBYTE revision
- UBYTE reserved [512-8-4-16-4-16-16-1]

3.135.1 Detailed Description

Defines the structure of the file header for store-files and save-files.

The first 8 bytes serve as a unique mark to identity save-files that contain such a header. Older versions of FORM don't have this header and will write the POSITION of the next file index (struct FiLeInDeX) here, which is always different from this pattern.

It is always 512 bytes long.

Definition at line 75 of file structs.h.

3.135.2 Field Documentation

3.135.2.1 headermark

```
UBYTE STOREHEADER::headermark[8]
```

Pattern for header identification. Old versions of FORM have a maximum sizeof(POSITION) of 8

Definition at line 76 of file structs.h.

3.135.2.2 lenWORD

```
UBYTE STOREHEADER::lenWORD
```

Number of bytes for WORD

Definition at line 78 of file structs.h.

Referenced by WriteStoreHeader().

3.135.2.3 lenLONG

```
UBYTE STOREHEADER::lenLONG
```

Number of bytes for LONG

Definition at line 79 of file structs.h.

Referenced by WriteStoreHeader().

3.135.2.4 lenPOS

```
UBYTE STOREHEADER::lenPOS
```

Number of bytes for POSITION

Definition at line 80 of file structs.h.

Referenced by WriteStoreHeader().

3.135.2.5 lenPOINTER

```
UBYTE STOREHEADER::lenPOINTER
```

Number of bytes for void *

Definition at line 81 of file structs.h.

Referenced by WriteStoreHeader().

3.135.2.6 endianness

```
UBYTE STOREHEADER::endianness[16]
```

Used to determine endianness, sizeof(int) should be <= 16

Definition at line 82 of file structs.h.

Referenced by WriteStoreHeader().

3.135.2.7 sSym

```
UBYTE STOREHEADER::sSym
```

sizeof(struct SyMbOl)

Definition at line 83 of file structs.h.

Referenced by WriteStoreHeader().

3.135.2.8 sInd

`UBYTE STOREHEADER::sInd`

sizeof(struct InDeX)

Definition at line 84 of file structs.h.

Referenced by WriteStoreHeader().

3.135.2.9 sVec

`UBYTE STOREHEADER::sVec`

sizeof(struct VeCtOr)

Definition at line 85 of file structs.h.

Referenced by WriteStoreHeader().

3.135.2.10 sFun

`UBYTE STOREHEADER::sFun`

sizeof(struct FuNcTiOn)

Definition at line 86 of file structs.h.

Referenced by WriteStoreHeader().

3.135.2.11 maxpower

`UBYTE STOREHEADER::maxpower[16]`

Maximum power, see #MAXPOWER

Definition at line 87 of file structs.h.

Referenced by WriteStoreHeader().

3.135.2.12 wildoffset

`UBYTE STOREHEADER::wildoffset[16]`

#WILDOFFSET macro

Definition at line 88 of file structs.h.

Referenced by WriteStoreHeader().

3.135.2.13 revision

```
UBYTE STOREHEADER::revision
```

Revision number of save-file system

Definition at line 89 of file structs.h.

3.135.2.14 reserved

```
UBYTE STOREHEADER::reserved[512-8-4-16-4-16-16-1]
```

Padding to 512 bytes

Definition at line 90 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.136 StreaM Struct Reference

```
#include <structs.h>
```

Data Fields

- off_t fileposition
- off_t linenumber
- off_t prevline
- UBYTE * buffer
- UBYTE * pointer
- UBYTE * top
- UBYTE * FoldName
- UBYTE * name
- UBYTE * pname
- LONG buffersize
- LONG bufferposition
- LONG inbuffer
- int previous
- int handle
- int type
- int prevars
- int previousNoShowInput
- int eqnum
- int afterwards
- int olddelay
- int oldnoshowinput
- UBYTE isnextchar
- UBYTE nextchar [2]
- UBYTE reserved

3.136.1 Detailed Description

Input is read from 'streams' which are represented by objects of type STREAM. A stream can be a file, a do-loop, a procedure, the string value of a preprocessor variable ..... When a new stream is opened we have to keep information about where to fall back in the parent stream to allow this to happen even in the middle of reading names etc as would be the case with a`i'b

Definition at line 722 of file structs.h.

3.136.2 Field Documentation

3.136.2.1 fileposition

```
off_t StreaM::fileposition
```

Definition at line 723 of file structs.h.

3.136.2.2 linenumber

```
off_t StreaM::linenumber
```

Definition at line 724 of file structs.h.

3.136.2.3 prevline

```
off_t StreaM::prevline
```

Definition at line 725 of file structs.h.

3.136.2.4 buffer

```
UBYTE* StreaM::buffer
```

[D] Size in buffersize

Definition at line 726 of file structs.h.

3.136.2.5 pointer

```
UBYTE* StreaM::pointer
```

pointer into buffer memory

Definition at line 727 of file structs.h.

3.136.2.6 top

```
UBYTE* StreaM::top
```

pointer into buffer memory

Definition at line 728 of file structs.h.

3.136.2.7 FoldName

```
UBYTE* StreaM::FoldName
```

[D]

Definition at line 729 of file structs.h.

3.136.2.8 name

```
UBYTE* StreaM::name
```

[D]

Definition at line 730 of file structs.h.

3.136.2.9 pname

```
UBYTE* StreaM::pname
```

for DOLLARSTREAM and PREVARSTREAM it points always to name, else it is undefined

Definition at line 731 of file structs.h.

3.136.2.10 buffersize

```
LONG StreaM::buffersize
```

Definition at line 733 of file structs.h.

3.136.2.11 bufferposition

```
LONG StreaM::bufferposition
```

Definition at line 734 of file structs.h.

3.136.2.12 inbuffer

```
LONG StreaM::inbuffer
```

Definition at line 735 of file structs.h.

3.136.2.13 previous

```
int StreaM::previous
```

Definition at line 736 of file structs.h.

3.136.2.14 handle

```
int StreaM::handle
```

Definition at line 737 of file structs.h.

3.136.2.15 type

```
int StreaM::type
```

Definition at line 738 of file structs.h.

3.136.2.16 prevars

```
int StreaM::prevars
```

Definition at line 739 of file structs.h.

3.136.2.17 previousNoShowInput

```
int StreaM::previousNoShowInput
```

Definition at line 740 of file structs.h.

3.136.2.18 eqnum

```
int StreaM::eqnum
```

Definition at line 741 of file structs.h.

3.136.2.19 afterwards

```
int StreaM::afterwards
```

Definition at line 742 of file structs.h.

3.136.2.20 olddelay

```
int StreaM::olddelay
```

Definition at line 743 of file structs.h.

3.136.2.21 oldnoshowinput

```
int StreaM::oldnoshowinput
```

Definition at line 744 of file structs.h.

3.136.2.22 isnextchar

```
UBYTE StreaM::isnextchar
```

Definition at line 745 of file structs.h.

3.136.2.23 nextchar

```
UBYTE StreaM::nextchar[2]
```

Definition at line 746 of file structs.h.

3.136.2.24 reserved

```
UBYTE StreaM::reserved
```

Definition at line 747 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.137 SuBbUf Struct Reference

Data Fields

- WORD subexpnum
- WORD buffernum

3.137.1 Detailed Description

Definition at line 264 of file compiler.c.

3.137.2 Field Documentation

3.137.2.1 subexpnum

```
WORD SuBbUf::subexpnum
```

Definition at line 265 of file compiler.c.

3.137.2.2 buffernum

```
WORD SuBbUf::buffernum
```

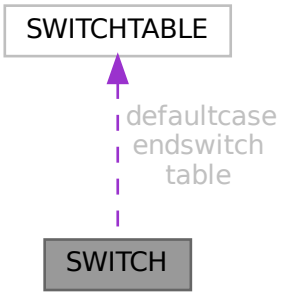
Definition at line 266 of file compiler.c.

The documentation for this struct was generated from the following file:

- compiler.c

3.138 SWITCH Struct Reference

Collaboration diagram for SWITCH:



Data Fields

- SWITCHTABLE * table
- SWITCHTABLE defaultcase
- SWITCHTABLE endswitch
- WORD typetable
- WORD maxcase
- WORD mincase
- WORD numcases
- WORD tablesize
- WORD caseoffset
- WORD iflevel
- WORD whilelevel
- WORD nestingsum
- WORD padding

3.138.1 Detailed Description

Definition at line 1355 of file structs.h.

3.138.2 Field Documentation

3.138.2.1 table

```
SWITCHTABLE* SWITCH::table
```

Definition at line 1356 of file structs.h.

3.138.2.2 defaultcase

```
SWITCHTABLE SWITCH::defaultcase
```

Definition at line 1357 of file structs.h.

3.138.2.3 endswitch

```
SWITCHTABLE SWITCH::endswitch
```

Definition at line 1358 of file structs.h.

3.138.2.4 typetable

```
WORD SWITCH::typetable
```

Definition at line 1359 of file structs.h.

3.138.2.5 maxcase

```
WORD SWITCH::maxcase
```

Definition at line 1360 of file structs.h.

3.138.2.6 mincase

```
WORD SWITCH::mincase
```

Definition at line 1361 of file structs.h.

3.138.2.7 numcases

```
WORD SWITCH::numcases
```

Definition at line 1362 of file structs.h.

3.138.2.8 tablesize

```
WORD SWITCH::tablesize
```

Definition at line 1363 of file structs.h.

3.138.2.9 caseoffset

```
WORD SWITCH::caseoffset
```

Definition at line 1364 of file structs.h.

3.138.2.10 iflevel

```
WORD SWITCH::iflevel
```

Definition at line 1365 of file structs.h.

3.138.2.11 whilelevel

```
WORD SWITCH::whilelevel
```

Definition at line 1366 of file structs.h.

3.138.2.12 nestingsum

```
WORD SWITCH::nestingsum
```

Definition at line 1367 of file structs.h.

3.138.2.13 padding

```
WORD SWITCH::padding
```

Definition at line 1368 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.139 SWITCHTABLE Struct Reference

Data Fields

- WORD ncase
- WORD value
- WORD compbuffer

3.139.1 Detailed Description

Definition at line 1349 of file structs.h.

3.139.2 Field Documentation

3.139.2.1 ncase

```
WORD SWITCHTABLE::ncase
```

Definition at line 1350 of file structs.h.

3.139.2.2 value

```
WORD SWITCHTABLE::value
```

Definition at line 1351 of file structs.h.

3.139.2.3 compbuffer

```
WORD SWITCHTABLE::compbuffer
```

Definition at line 1352 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.140 SyMbOl Struct Reference

Data Fields

- LONG name
- WORD minpower
- WORD maxpower
- WORD complex
- WORD number
- WORD flags
- WORD node
- WORD namesize
- WORD dimension

3.140.1 Detailed Description

Definition at line 416 of file structs.h.

3.140.2 Field Documentation

3.140.2.1 name

```
LONG SyMbOl::name
```

Definition at line 417 of file structs.h.

3.140.2.2 minpower

```
WORD SyMbOl::minpower
```

Definition at line 418 of file structs.h.

3.140.2.3 maxpower

```
WORD SyMbOl::maxpower
```

Definition at line 419 of file structs.h.

3.140.2.4 complex

```
WORD SyMbOl::complex
```

Definition at line 420 of file structs.h.

3.140.2.5 number

```
WORD SyMbOl::number
```

Definition at line 421 of file structs.h.

3.140.2.6 flags

```
WORD SyMbOl::flags
```

Definition at line 422 of file structs.h.

3.140.2.7 node

```
WORD SyMbOl::node
```

Definition at line 423 of file structs.h.

3.140.2.8 namesize

```
WORD SyMbOl::namesize
```

Definition at line 424 of file structs.h.

3.140.2.9 dimension

```
WORD SyMbOl::dimension
```

Definition at line 425 of file structs.h.

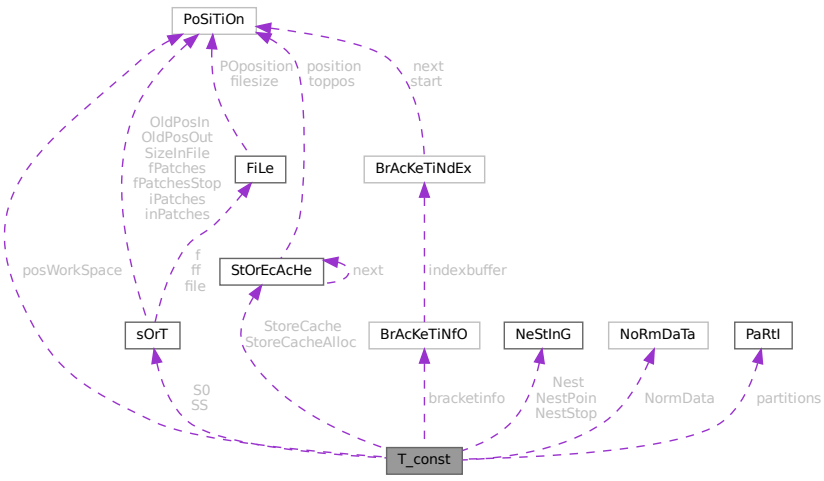
The documentation for this struct was generated from the following file:

- structs.h

3.141 T_const Struct Reference

```
#include <structs.h>
```

Collaboration diagram for T_const:



Data Fields

- SORTING * S0
- SORTING * SS
- NESTING Nest
- NESTING NestStop
- NESTING NestPoin
- WORD * BrackBuf
- STORECACHE StoreCache
- STORECACHE StoreCacheAlloc
- WORD ** pWorkSpace
- LONG * lWorkSpace
- POSITION * posWorkSpace
- WORD * WorkSpace
- WORD * WorkTop
- WORD * WorkPointer
- int * RepCount
- int * RepTop
- WORD * WildArgTaken
- UWORD * factorials
- WORD * small_power_n
- UWORD ** small_power
- UWORD * bernoullis
- WORD * primelist
- LONG * pfac
- LONG * pBer
- WORD * TMaddr
- WORD * WildMask
- WORD * previousEfactor
- WORD ** TermMemHeap
- UWORD ** NumberMemHeap
- UWORD ** CacheNumberMemHeap
- BRACKETINFO * bracketinfo
- WORD ** ListPoly
- WORD * ListSymbols
- UWORD * NumMem
- WORD * TopologiesTerm
- WORD * TopologiesStart
- NORMDATA ** NormData
- LONG NormDataSize
- LONG NormDepth
- PARTI partitions
- LONG sBer
- LONG pWorkPointer
- LONG lWorkPointer
- LONG posWorkPointer
- LONG lnNumMem
- int sfact
- int mfac
- int ebufnum
- int fbufnum
- int allbufnum
- int aebufnum
- int idallflag
- int idallnum

- int idallmaxnum
- int WildcardBufferSize
- int TermMemMax
- int TermMemTop
- int NumberMemMax
- int NumberMemTop
- int CacheNumberMemMax
- int CacheNumberMemTop
- int bracketindexflag
- int optimtimes
- int ListSymbolsSize
- int NumListSymbols
- int numpoly
- int LeaveNegative
- int TrimPower
- WORD small_power_maxx
- WORD small_power_maxn
- WORD dummysubexp [SUBEXPSIZE+4]
- WORD comsym [8]
- WORD comnum [4]
- WORD comfun [FUNHEAD+4]
- WORD comind [7]
- WORD MinVecArg [7+ARGHEAD]
- WORD FunArg [4+ARGHEAD+FUNHEAD]
- WORD locwildvalue [SUBEXPSIZE]
- WORD mulpat [SUBEXPSIZE+5]
- WORD proexp [SUBEXPSIZE+5]
- WORD TMout [40]
- WORD TMbuff
- WORD TMdolfac
- WORD nfac
- WORD nBer
- WORD mBer
- WORD PolyAct
- WORD RecFlag
- WORD inprimelist
- WORD sizeprimelist
- WORD fromindex
- WORD setinterntopo
- WORD setexterntopo
- WORD TopologiesLevel
- WORD TopologiesOptions [2]

3.141.1 Detailed Description

The T_const struct is part of the global data and resides either in the ALLGLOBALS struct A, or the ALLPRIVATES struct B (TFORM) under the name T We see it used with the macro AT as in AT.WorkPointer It has variables that are private to each thread, most of which have to be defined at startup.

Definition at line 2069 of file structs.h.

3.141.2 Field Documentation

3.141.2.1 S0

```
SORTING* T_const::S0
```

Definition at line 2073 of file structs.h.

3.141.2.2 SS

```
SORTING* T_const::SS
```

Definition at line 2074 of file structs.h.

3.141.2.3 Nest

```
NESTING T_const::Nest
```

Definition at line 2075 of file structs.h.

3.141.2.4 NestStop

```
NESTING T_const::NestStop
```

Definition at line 2076 of file structs.h.

3.141.2.5 NestPoin

```
NESTING T_const::NestPoin
```

Definition at line 2077 of file structs.h.

3.141.2.6 BrackBuf

```
WORD* T_const::BrackBuf
```

Definition at line 2078 of file structs.h.

3.141.2.7 StoreCache

```
STORECACHE T_const::StoreCache
```

Definition at line 2079 of file structs.h.

3.141.2.8 StoreCacheAlloc

```
STORECACHE T_const::StoreCacheAlloc
```

Definition at line 2080 of file structs.h.

3.141.2.9 pWorkSpace

```
WORD** T_const::pWorkSpace
```

Definition at line 2081 of file structs.h.

3.141.2.10 lWorkSpace

```
LONG* T_const::lWorkSpace
```

Definition at line 2082 of file structs.h.

3.141.2.11 posWorkSpace

```
POSITION* T_const::posWorkSpace
```

Definition at line 2083 of file structs.h.

3.141.2.12 WorkSpace

```
WORD* T_const::WorkSpace
```

Definition at line 2084 of file structs.h.

3.141.2.13 WorkTop

```
WORD* T_const::WorkTop
```

Definition at line 2085 of file structs.h.

3.141.2.14 WorkPointer

```
WORD* T_const::WorkPointer
```

Definition at line 2086 of file structs.h.

3.141.2.15 RepCount

```
int* T_const::RepCount
```

Definition at line 2087 of file structs.h.

3.141.2.16 RepTop

```
int* T_const::RepTop
```

Definition at line 2088 of file structs.h.

3.141.2.17 WildArgTaken

```
WORD* T_const::WildArgTaken
```

Definition at line 2089 of file structs.h.

3.141.2.18 factorials

```
UWORD* T_const::factorials
```

Definition at line 2090 of file structs.h.

3.141.2.19 small_power_n

```
WORD* T_const::small_power_n
```

Definition at line 2091 of file structs.h.

3.141.2.20 small_power

```
UWORD** T_const::small_power
```

Definition at line 2092 of file structs.h.

3.141.2.21 bernoullis

```
UWORD* T_const::bernoullis
```

Definition at line 2093 of file structs.h.

3.141.2.22 primelist

```
WORD* T_const::primelist
```

Definition at line 2094 of file structs.h.

3.141.2.23 pfac

```
LONG* T_const::pfac
```

Definition at line 2095 of file structs.h.

3.141.2.24 pBer

```
LONG* T_const::pBer
```

Definition at line 2096 of file structs.h.

3.141.2.25 TMaddr

```
WORD* T_const::TMaddr
```

Definition at line 2097 of file structs.h.

3.141.2.26 WildMask

```
WORD* T_const::WildMask
```

Definition at line 2098 of file structs.h.

3.141.2.27 previousEfactor

```
WORD* T_const::previousEfactor
```

Definition at line 2099 of file structs.h.

3.141.2.28 TermMemHeap

```
WORD** T_const::TermMemHeap
```

Definition at line 2100 of file structs.h.

3.141.2.29 NumberMemHeap

```
UWORD** T_const::NumberMemHeap
```

Definition at line 2101 of file structs.h.

3.141.2.30 CacheNumberMemHeap

```
UWORD** T_const::CacheNumberMemHeap
```

Definition at line 2102 of file structs.h.

3.141.2.31 bracketinfo

```
BRACKETINFO* T_const::bracketinfo
```

Definition at line 2103 of file structs.h.

3.141.2.32 ListPoly

```
WORD** T_const::ListPoly
```

Definition at line 2104 of file structs.h.

3.141.2.33 ListSymbols

```
WORD* T_const::ListSymbols
```

Definition at line 2105 of file structs.h.

3.141.2.34 NumMem

```
UWORD* T_const::NumMem
```

Definition at line 2106 of file structs.h.

3.141.2.35 TopologiesTerm

```
WORD* T_const::TopologiesTerm
```

Definition at line 2107 of file structs.h.

3.141.2.36 TopologiesStart

```
WORD* T_const::TopologiesStart
```

Definition at line 2108 of file structs.h.

3.141.2.37 NormData

```
NORMDATA** T_const::NormData
```

Definition at line 2117 of file structs.h.

3.141.2.38 NormDataSize

```
LONG T_const::NormDataSize
```

Definition at line 2118 of file structs.h.

3.141.2.39 NormDepth

```
LONG T_const::NormDepth
```

Definition at line 2119 of file structs.h.

3.141.2.40 partitions

```
PARTI T_const::partitions
```

Definition at line 2120 of file structs.h.

3.141.2.41 sBer

```
LONG T_const::sBer
```

Definition at line 2121 of file structs.h.

3.141.2.42 pWorkPointer

```
LONG T_const::pWorkPointer
```

Definition at line 2122 of file structs.h.

3.141.2.43 lWorkPointer

```
LONG T_const::lWorkPointer
```

Definition at line 2123 of file structs.h.

3.141.2.44 posWorkPointer

```
LONG T_const::posWorkPointer
```

Definition at line 2124 of file structs.h.

3.141.2.45 InNumMem

```
LONG T_const::InNumMem
```

Definition at line 2125 of file structs.h.

3.141.2.46 sfact

```
int T_const::sfact
```

Definition at line 2126 of file structs.h.

3.141.2.47 mfac

```
int T_const::mfac
```

Definition at line 2127 of file structs.h.

3.141.2.48 ebufnum

```
int T_const::ebufnum
```

Definition at line 2128 of file structs.h.

3.141.2.49 fbufnum

```
int T_const::fbufnum
```

Definition at line 2129 of file structs.h.

3.141.2.50 allbufnum

```
int T_const::allbufnum
```

Definition at line 2130 of file structs.h.

3.141.2.51 aebufnum

```
int T_const::aebufnum
```

Definition at line 2131 of file structs.h.

3.141.2.52 idallflag

```
int T_const::idallflag
```

Definition at line 2132 of file structs.h.

3.141.2.53 idallnum

```
int T_const::idallnum
```

Definition at line 2133 of file structs.h.

3.141.2.54 idallmaxnum

```
int T_const::idallmaxnum
```

Definition at line 2134 of file structs.h.

3.141.2.55 WildcardBufferSize

```
int T_const::WildcardBufferSize
```

Definition at line 2135 of file structs.h.

3.141.2.56 TermMemMax

```
int T_const::TermMemMax
```

Definition at line 2144 of file structs.h.

3.141.2.57 TermMemTop

```
int T_const::TermMemTop
```

Definition at line 2145 of file structs.h.

3.141.2.58 NumberMemMax

```
int T_const::NumberMemMax
```

Definition at line 2146 of file structs.h.

3.141.2.59 NumberMemTop

```
int T_const::NumberMemTop
```

Definition at line 2147 of file structs.h.

3.141.2.60 CacheNumberMemMax

```
int T_const::CacheNumberMemMax
```

Definition at line 2148 of file structs.h.

3.141.2.61 CacheNumberMemTop

```
int T_const::CacheNumberMemTop
```

Definition at line 2149 of file structs.h.

3.141.2.62 bracketindexflag

```
int T_const::bracketindexflag
```

Definition at line 2150 of file structs.h.

3.141.2.63 optimtimes

```
int T_const::optimtimes
```

Definition at line 2151 of file structs.h.

3.141.2.64 ListSymbolsSize

```
int T_const::ListSymbolsSize
```

Definition at line 2152 of file structs.h.

3.141.2.65 NumListSymbols

```
int T_const::NumListSymbols
```

Definition at line 2153 of file structs.h.

3.141.2.66 numpoly

```
int T_const::numpoly
```

Definition at line 2154 of file structs.h.

3.141.2.67 LeaveNegative

```
int T_const::LeaveNegative
```

Definition at line 2155 of file structs.h.

3.141.2.68 TrimPower

```
int T_const::TrimPower
```

Definition at line 2156 of file structs.h.

3.141.2.69 small_power_maxx

```
WORD T_const::small_power_maxx
```

Definition at line 2157 of file structs.h.

3.141.2.70 small_power_maxn

```
WORD T_const::small_power_maxn
```

Definition at line 2158 of file structs.h.

3.141.2.71 dummysubexp

```
WORD T_const::dummysubexp[SUBEXPSIZE+4]
```

Definition at line 2159 of file structs.h.

3.141.2.72 comsym

```
WORD T_const::comsym[8]
```

Definition at line 2160 of file structs.h.

3.141.2.73 comnum

```
WORD T_const::comnum[4]
```

Definition at line 2161 of file structs.h.

3.141.2.74 comfun

```
WORD T_const::comfun[FUNHEAD+4]
```

Definition at line 2162 of file structs.h.

3.141.2.75 comind

```
WORD T_const::comind[7]
```

Definition at line 2164 of file structs.h.

3.141.2.76 MinVecArg

```
WORD T_const::MinVecArg[7+ARGHEAD]
```

Definition at line 2165 of file structs.h.

3.141.2.77 FunArg

```
WORD T_const::FunArg[4+ARGHEAD+FUNHEAD]
```

Definition at line 2166 of file structs.h.

3.141.2.78 locwildvalue

```
WORD T_const::locwildvalue[SUBEXPSIZE]
```

Definition at line 2167 of file structs.h.

3.141.2.79 mulpat

```
WORD T_const::mulpat[SUBEXPSIZE+5]
```

Definition at line 2168 of file structs.h.

3.141.2.80 proexp

```
WORD T_const::proexp[SUBEXPSIZE+5]
```

Definition at line 2170 of file structs.h.

3.141.2.81 TMout

```
WORD T_const::TMout[40]
```

Definition at line 2171 of file structs.h.

3.141.2.82 TMbuff

```
WORD T_const::TMbuff
```

Definition at line 2172 of file structs.h.

3.141.2.83 TMdolfac

```
WORD T_const::TMdolfac
```

Definition at line 2173 of file structs.h.

3.141.2.84 nfac

```
WORD T_const::nfac
```

Definition at line 2174 of file structs.h.

3.141.2.85 nBer

```
WORD T_const::nBer
```

Definition at line 2175 of file structs.h.

3.141.2.86 mBer

```
WORD T_const::mBer
```

Definition at line 2176 of file structs.h.

3.141.2.87 PolyAct

```
WORD T_const::PolyAct
```

Definition at line 2177 of file structs.h.

3.141.2.88 RecFlag

```
WORD T_const::RecFlag
```

Definition at line 2178 of file structs.h.

3.141.2.89 inprimelist

```
WORD T_const::inprimelist
```

Definition at line 2179 of file structs.h.

3.141.2.90 sizeprimelist

```
WORD T_const::sizeprimelist
```

Definition at line 2180 of file structs.h.

3.141.2.91 fromindex

```
WORD T_const::fromindex
```

Definition at line 2181 of file structs.h.

3.141.2.92 setinterntopo

```
WORD T_const::setinterntopo
```

Definition at line 2182 of file structs.h.

3.141.2.93 setexterntopo

```
WORD T_const::setexterntopo
```

Definition at line 2183 of file structs.h.

3.141.2.94 TopologiesLevel

```
WORD T_const::TopologiesLevel
```

Definition at line 2184 of file structs.h.

3.141.2.95 TopologiesOptions

```
WORD T_const::TopologiesOptions[2]
```

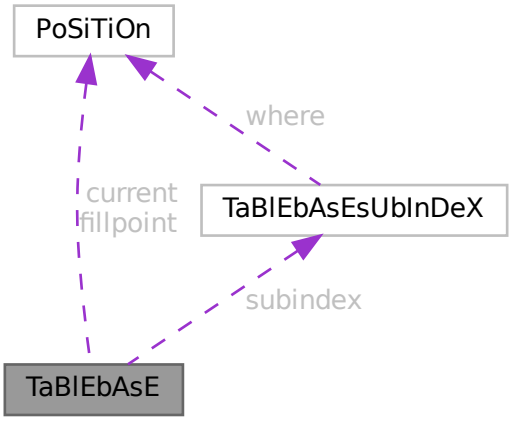
Definition at line 2185 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.142 TaBlEbAsE Struct Reference

Collaboration diagram for TaBlEbAsE:



Data Fields

- POSITION fillpoint
- POSITION current
- UBYTE * name
- int * tablenumbers
- TABLEBASESUBINDEX * subindex
- int numtables

3.142.1 Detailed Description

Definition at line 593 of file structs.h.

3.142.2 Field Documentation

3.142.2.1 fillpoint

```
POSITION TaBlEbAsE::fillpoint
```

Definition at line 594 of file structs.h.

3.142.2.2 current

```
POSITION TaBlEbAsE::current
```

Definition at line 595 of file structs.h.

3.142.2.3 name

```
UBYTE* TaBlEbAsE::name
```

Definition at line 596 of file structs.h.

3.142.2.4 tablenumbers

```
int* TaBlEbAsE::tablenumbers
```

Definition at line 597 of file structs.h.

3.142.2.5 subindex

```
TABLEBASESUBINDEX* TaBlEbAsE::subindex
```

Definition at line 598 of file structs.h.

3.142.2.6 numtables

```
int TaBlEbAsE::numtables
```

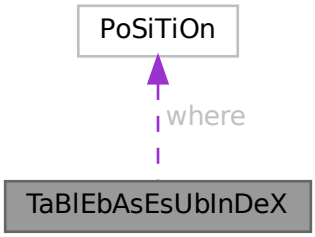
Definition at line 599 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.143 TaBlEbAsEsUbInDeX Struct Reference

Collaboration diagram for TaBlEbAsEsUbInDeX:



Data Fields

- POSITION where
- LONG size

3.143.1 Detailed Description

Definition at line 584 of file structs.h.

3.143.2 Field Documentation

3.143.2.1 where

```
POSITION TaBlEbAsEsUbInDeX::where
```

Definition at line 585 of file structs.h.

3.143.2.2 size

```
LONG TaBlEbAsEsUbInDeX::size
```

Definition at line 586 of file structs.h.

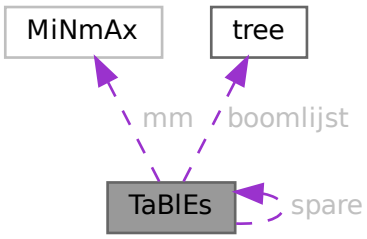
The documentation for this struct was generated from the following file:

- structs.h

3.144 TaBIEs Struct Reference

`#include <structs.h>`

Collaboration diagram for TaBIEs:



Data Fields

- WORD * tablepointers
- WORD * prototype
- WORD * pattern
- MINMAX * mm
- WORD * flags
- COMPTREE * boomlijst
- UBYTE * argtail
- struct TaBIEs * spare
- WORD * buffers
- LONG totind
- LONG reserved
- LONG defined
- LONG mdefined
- int prototypeSize
- int numind
- int bounds
- int strict
- int sparse
- int numtree
- int rootnum
- int MaxTreeSize
- WORD bufnum
- WORD bufferssize
- WORD buffersfill
- WORD tablenum
- WORD mode
- WORD numdummies

3.144.1 Detailed Description

buffers, mm, flags, and prototype are always dynamically allocated, tablepointers only if needed (=0 if unallocated), boomlijst and argtail only for sparse tables.

Allocation is done for both the normal and the stub instance (spare), except for prototype and argtail which share memory.

Definition at line 342 of file structs.h.

3.144.2 Field Documentation

3.144.2.1 tablepointers

```
WORD* TaBlEs::tablepointers
```

[D] Start in tablepointers table.

Definition at line 343 of file structs.h.

Referenced by DoRecovery().

3.144.2.2 prototype

```
WORD* TaBlEs::prototype
```

[D] The wildcard prototyping for arguments

Definition at line 348 of file structs.h.

Referenced by DoRecovery().

3.144.2.3 pattern

```
WORD* TaBlEs::pattern
```

The pattern with which to match the arguments

Definition at line 349 of file structs.h.

Referenced by DoRecovery().

3.144.2.4 mm

```
MINMAX* TaBlEs::mm
```

[D] Array bounds, dimension by dimension. # elements = numind.

Definition at line 351 of file structs.h.

Referenced by DoRecovery().

3.144.2.5 flags

```
WORD* TaBlEs::flags
```

[D] Is element in use ? etc. # elements = numind.

Definition at line 352 of file structs.h.

Referenced by DoRecovery().

3.144.2.6 boomlijst

```
COMPTREE* TaBlEs::boomlijst
```

[D] Tree for searching in sparse tables

Definition at line 353 of file structs.h.

Referenced by DoRecovery().

3.144.2.7 argtail

```
UBYTE* TaBlEs::argtail
```

[D] The arguments in characters. Starts for tablebase with parenthesis to indicate tail

Definition at line 354 of file structs.h.

Referenced by DoRecovery().

3.144.2.8 spare

```
struct TaBlEs* TaBlEs::spare
```

[D] For tablebase. Alternatingly stubs and real

Definition at line 356 of file structs.h.

Referenced by DoRecovery().

3.144.2.9 buffers

```
WORD* TaBlEs::buffers
```

[D] When we use more than one compiler buffer.

Definition at line 357 of file structs.h.

Referenced by AddRHS(), and DoRecovery().

3.144.2.10 totind

`LONG TaBlEs::totind`

Total number requested

Definition at line 358 of file structs.h.

Referenced by DoRecovery().

3.144.2.11 reserved

`LONG TaBlEs::reserved`

Total reservation in tablepointers for sparse

Definition at line 359 of file structs.h.

Referenced by DoRecovery().

3.144.2.12 defined

`LONG TaBlEs::defined`

Number of table elements that are defined

Definition at line 360 of file structs.h.

3.144.2.13 mdefined

`LONG TaBlEs::mdefined`

Same as defined but after .global

Definition at line 361 of file structs.h.

3.144.2.14 prototypeSize

`int TaBlEs::prototypeSize`

Size of allocated memory for prototype in bytes.

Definition at line 362 of file structs.h.

Referenced by DoRecovery().

3.144.2.15 numind

```
int TaBlEs::numind
```

Number of array indices

Definition at line 363 of file structs.h.

Referenced by DoRecovery().

3.144.2.16 bounds

```
int TaBlEs::bounds
```

Array bounds check on/off.

Definition at line 364 of file structs.h.

3.144.2.17 strict

```
int TaBlEs::strict
```

>0: all must be defined. <0: undefined not substitute

Definition at line 365 of file structs.h.

3.144.2.18 sparse

```
int TaBlEs::sparse
```

0 --> sparse table

Definition at line 366 of file structs.h.

Referenced by DoRecovery().

3.144.2.19 numtree

```
int TaBlEs::numtree
```

For the tree for sparse tables

Definition at line 367 of file structs.h.

3.144.2.20 rootnum

```
int TaBlEs::rootnum
```

For the tree for sparse tables

Definition at line 368 of file structs.h.

3.144.2.21 MaxTreeSize

```
int TaBlEs::MaxTreeSize
```

For the tree for sparse tables

Definition at line 369 of file structs.h.

Referenced by DoRecovery().

3.144.2.22 bufnum

```
WORD TaBlEs::bufnum
```

Each table potentially its own buffer

Definition at line 370 of file structs.h.

Referenced by AddRHS().

3.144.2.23 bufferssize

```
WORD TaBlEs::bufferssize
```

When we use more than one compiler buffer

Definition at line 371 of file structs.h.

Referenced by AddRHS(), and DoRecovery().

3.144.2.24 buffersfill

```
WORD TaBlEs::buffersfill
```

When we use more than one compiler buffer

Definition at line 372 of file structs.h.

Referenced by AddRHS().

3.144.2.25 tablenum

```
WORD TaBlEs::tablenum
```

For testing of tableuse

Definition at line 373 of file structs.h.

3.144.2.26 mode

```
WORD TaBlEs::mode
```

0: normal, 1: stub

Definition at line 374 of file structs.h.

3.144.2.27 numdummies

```
WORD TaBlEs::numdummies
```

Definition at line 375 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.145 TERMINFO Struct Reference

Data Fields

- WORD * term
- void * currentModel
- void * currentMODEL
- WORD * legcouple [MAXLEGS+1]
- LONG numtopo
- LONG numdia
- WORD diaoffset
- WORD level
- WORD externalset
- WORD internalset
- WORD numextern
- WORD flags

3.145.1 Detailed Description

Definition at line 1371 of file structs.h.

3.145.2 Field Documentation

3.145.2.1 term

```
WORD* TERMINFO::term
```

Definition at line 1372 of file structs.h.

3.145.2.2 currentModel

```
void* TERMINFO::currentModel
```

Definition at line 1373 of file structs.h.

3.145.2.3 currentMODEL

```
void* TERMINFO::currentMODEL
```

Definition at line 1374 of file structs.h.

3.145.2.4 legcouple

```
WORD* TERMINFO::legcouple[MAXLEGS+1]
```

Definition at line 1375 of file structs.h.

3.145.2.5 numtopo

```
LONG TERMINFO::numtopo
```

Definition at line 1376 of file structs.h.

3.145.2.6 numdia

```
LONG TERMINFO::numdia
```

Definition at line 1377 of file structs.h.

3.145.2.7 diaoffset

```
WORD TERMINFO::diaoffset
```

Definition at line 1378 of file structs.h.

3.145.2.8 level

```
WORD TERMINFO::level
```

Definition at line 1379 of file structs.h.

3.145.2.9 externalset

```
WORD TERMINFO::externalset
```

Definition at line 1380 of file structs.h.

3.145.2.10 internalset

```
WORD TERMINFO::internalset
```

Definition at line 1381 of file structs.h.

3.145.2.11 numextern

```
WORD TERMINFO::numextern
```

Definition at line 1382 of file structs.h.

3.145.2.12 flags

```
WORD TERMINFO::flags
```

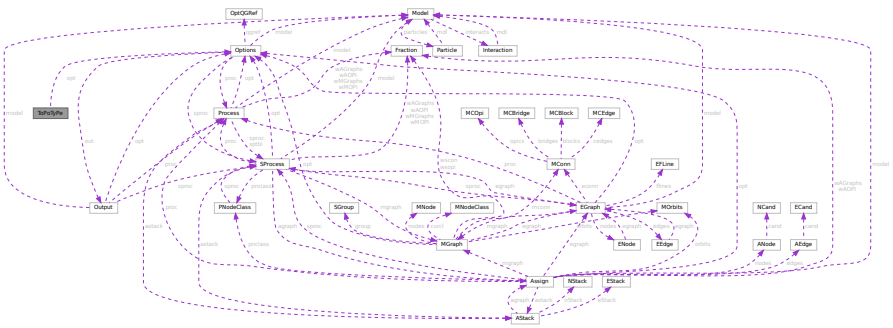
Definition at line 1383 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.146 ToPoTyPe Struct Reference

Collaboration diagram for ToPoTyPe:



Data Fields

- WORD * vert
- WORD * vertmax
- Options * opt
- int cldeg [MAXPOINTS]
- int clnum [MAXPOINTS]
- int clext [MAXPOINTS]
- int cmind [MAXLEGS+1]
- int cmaxd [MAXLEGS+1]
- int ncl
- int nvert

3.146.1 Detailed Description

Definition at line 43 of file diawrap.cc.

3.146.2 Field Documentation

3.146.2.1 vert

```
WORD* ToPoTyPe::vert
```

Definition at line 44 of file diawrap.cc.

3.146.2.2 vertmax

```
WORD* ToPoTyPe::vertmax
```

Definition at line 45 of file diawrap.cc.

3.146.2.3 opt

```
Options* ToPoTyPe::opt
```

Definition at line 46 of file diawrap.cc.

3.146.2.4 cldeg

```
int ToPoTyPe::cldeg[MAXPOINTS]
```

Definition at line 47 of file diawrap.cc.

3.146.2.5 clnum

```
int ToPoTyPe::clnum[MAXPOINTS]
```

Definition at line 47 of file diawrap.cc.

3.146.2.6 clext

```
int ToPoTyPe::clext[MAXPOINTS]
```

Definition at line 47 of file diawrap.cc.

3.146.2.7 cmind

```
int ToPoTyPe::cmind[MAXLEGS+1]
```

Definition at line 48 of file diawrap.cc.

3.146.2.8 cmaxd

```
int ToPoTyPe::cmaxd[MAXLEGS+1]
```

Definition at line 48 of file diawrap.cc.

3.146.2.9 ncl

```
int ToPoTyPe::ncl
```

Definition at line 49 of file diawrap.cc.

3.146.2.10 nvert

```
int ToPoTyPe::nvert
```

Definition at line 49 of file diawrap.cc.

The documentation for this struct was generated from the following file:

- diawrap.cc

3.147 TrAcEn Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD * accu
- WORD * accup
- WORD * termp
- WORD * perm
- WORD * inlist
- WORD sgn
- WORD num
- WORD level
- WORD factor
- WORD allsign

3.147.1 Detailed Description

The struct TRACEN keeps track of the progress during the expansion of a 4-dimensional trace. Each time a term gets generated the expansion tree continues in the next statement. When it returns it has to know where to continue.

Definition at line 805 of file structs.h.

3.147.2 Field Documentation

3.147.2.1 accu

```
WORD* TrAcEn::accu
```

Definition at line 806 of file structs.h.

3.147.2.2 accup

```
WORD* TrAcEn::accup
```

Definition at line 807 of file structs.h.

3.147.2.3 termp

```
WORD* TrAcEn::termp
```

Definition at line 808 of file structs.h.

3.147.2.4 perm

```
WORD* TrAcEn::perm
```

Definition at line 809 of file structs.h.

3.147.2.5 inlist

```
WORD* TrAcEn::inlist
```

Definition at line 810 of file structs.h.

3.147.2.6 sgn

```
WORD TrAcEn::sgn
```

Definition at line 811 of file structs.h.

3.147.2.7 num

```
WORD TrAcEn::num
```

Definition at line 811 of file structs.h.

3.147.2.8 level

```
WORD TrAcEn::level
```

Definition at line 811 of file structs.h.

3.147.2.9 factor

```
WORD TrAcEn::factor
```

Definition at line 811 of file structs.h.

3.147.2.10 allsign

```
WORD TrAcEn::allsign
```

Definition at line 811 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.148 TrAcEs Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD * accu
- WORD * accup
- WORD * termp
- WORD * perm
- WORD * inlist
- WORD * nt3
- WORD * nt4
- WORD * j3
- WORD * j4
- WORD * e3
- WORD * e4
- WORD * eers
- WORD * mepf
- WORD * mdel
- WORD * pepf
- WORD * pdel
- WORD sgn
- WORD stap
- WORD step1
- WORD kstep
- WORD mdum
- WORD gamm
- WORD ad
- WORD a3
- WORD a4
- WORD lc3
- WORD lc4
- WORD sign1
- WORD sign2
- WORD gamma5
- WORD num
- WORD level
- WORD factor
- WORD allsign
- WORD finalstep

3.148.1 Detailed Description

The struct TRACES keeps track of the progress during the expansion of a 4-dimensional trace. Each time a term gets generated the expansion tree continues in the next statement. When it returns it has to know where to continue. The 4-dimensional traces are more complicated than the n-dimensional traces (see TRACEN) because of the extra tricks that can be used. They are responsible for the shorter final expressions.

Definition at line 773 of file structs.h.

3.148.2 Field Documentation

3.148.2.1 accu

```
WORD* TrAcEs::accu
```

Definition at line 774 of file structs.h.

3.148.2.2 accup

```
WORD* TrAcEs::accup
```

Definition at line 775 of file structs.h.

3.148.2.3 termp

```
WORD* TrAcEs::termp
```

Definition at line 776 of file structs.h.

3.148.2.4 perm

```
WORD* TrAcEs::perm
```

Definition at line 777 of file structs.h.

3.148.2.5 inlist

```
WORD* TrAcEs::inlist
```

Definition at line 778 of file structs.h.

3.148.2.6 nt3

```
WORD* TrAcEs::nt3
```

Definition at line 779 of file structs.h.

3.148.2.7 nt4

```
WORD* TrAcEs::nt4
```

Definition at line 780 of file structs.h.

3.148.2.8 j3

```
WORD* TrAcEs::j3
```

Definition at line 781 of file structs.h.

3.148.2.9 j4

```
WORD* TrAcEs::j4
```

Definition at line 782 of file structs.h.

3.148.2.10 e3

```
WORD* TrAcEs::e3
```

Definition at line 783 of file structs.h.

3.148.2.11 e4

```
WORD* TrAcEs::e4
```

Definition at line 784 of file structs.h.

3.148.2.12 eers

```
WORD* TrAcEs::eers
```

Definition at line 785 of file structs.h.

3.148.2.13 mepf

```
WORD* TrAcEs::mepf
```

Definition at line 786 of file structs.h.

3.148.2.14 mdel

```
WORD* TrAcEs::mdel
```

Definition at line 787 of file structs.h.

3.148.2.15 pepf

```
WORD* TrAcEs::pepf
```

Definition at line 788 of file structs.h.

3.148.2.16 pdel

```
WORD* TrAcEs::pdel
```

Definition at line 789 of file structs.h.

3.148.2.17 sgn

```
WORD TrAcEs::sgn
```

Definition at line 790 of file structs.h.

3.148.2.18 stap

```
WORD TrAcEs::stap
```

Definition at line 791 of file structs.h.

3.148.2.19 step1

```
WORD TrAcEs::step1
```

Definition at line 792 of file structs.h.

3.148.2.20 kstep

```
WORD TrAcEs::kstep
```

Definition at line 792 of file structs.h.

3.148.2.21 mdum

```
WORD TrAcEs::mdum
```

Definition at line 792 of file structs.h.

3.148.2.22 gamm

```
WORD TrAcEs::gamm
```

Definition at line 793 of file structs.h.

3.148.2.23 ad

```
WORD TrAcEs::ad
```

Definition at line 793 of file structs.h.

3.148.2.24 a3

```
WORD TrAcEs::a3
```

Definition at line 793 of file structs.h.

3.148.2.25 a4

```
WORD TrAcEs::a4
```

Definition at line 793 of file structs.h.

3.148.2.26 lc3

```
WORD TrAcEs::lc3
```

Definition at line 793 of file structs.h.

3.148.2.27 lc4

```
WORD TrAcEs::lc4
```

Definition at line 793 of file structs.h.

3.148.2.28 sign1

```
WORD TrAcEs::sign1
```

Definition at line 794 of file structs.h.

3.148.2.29 sign2

```
WORD TrAcEs::sign2
```

Definition at line 794 of file structs.h.

3.148.2.30 gamma5

```
WORD TrAcEs::gamma5
```

Definition at line 794 of file structs.h.

3.148.2.31 num

```
WORD TrAcEs::num
```

Definition at line 794 of file structs.h.

3.148.2.32 level

```
WORD TrAcEs::level
```

Definition at line 794 of file structs.h.

3.148.2.33 factor

```
WORD TrAcEs::factor
```

Definition at line 794 of file structs.h.

3.148.2.34 allsign

```
WORD TrAcEs::allsign
```

Definition at line 794 of file structs.h.

3.148.2.35 finalstep

```
WORD TrAcEs::finalstep
```

Definition at line 795 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.149 tree Struct Reference

```
#include <structs.h>
```

Data Fields

- int parent
- int left
- int right
- int value
- int blnce
- int usage

3.149.1 Detailed Description

The subexpressions in the compiler are kept track of in a (balanced) tree to reduce the need for subexpressions and hence save much space in large rhs expressions (like when we have xxxxxxx occurrences of objects like f(x+1,x+1) in which each x+1 becomes a subexpression. The struct that controls this tree is COMPTREE.

Definition at line 288 of file structs.h.

3.149.2 Field Documentation

3.149.2.1 parent

```
int tree::parent
```

Index of parent

Definition at line 289 of file structs.h.

Referenced by IniFbuffer().

3.149.2.2 left

`int tree::left`

Left child (if not -1)

Definition at line 290 of file structs.h.

Referenced by IniFbuffer().

3.149.2.3 right

`int tree::right`

Right child (if not -1)

Definition at line 291 of file structs.h.

Referenced by IniFbuffer().

3.149.2.4 value

`int tree::value`

The object to be sorted and searched

Definition at line 292 of file structs.h.

Referenced by IniFbuffer().

3.149.2.5 blnce

`int tree::blnce`

Balance factor

Definition at line 293 of file structs.h.

Referenced by IniFbuffer().

3.149.2.6 usage

`int tree::usage`

Number of uses in some types of trees

Definition at line 294 of file structs.h.

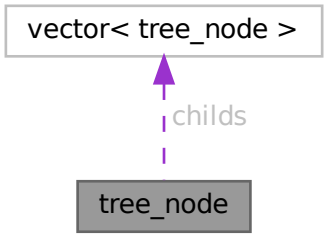
Referenced by CleanupArgCache(), and IniFbuffer().

The documentation for this struct was generated from the following file:

- structs.h

3.150 tree_node Class Reference

Collaboration diagram for tree_node:



Public Member Functions

- tree_node (int _var=0)
- tree_node (const tree_node &other)
- tree_node & operator= (const tree_node &other)

Data Fields

- vector< tree_node > childs
- double sum_results
- int num_visits
- WORD var
- bool finished

3.150.1 Detailed Description

Definition at line 125 of file optimize.cc.

3.150.2 Constructor & Destructor Documentation

3.150.2.1 tree_node() [1/2]

```
tree_node::tree_node (
            int _var = 0 )  [inline]
```

Definition at line 133 of file optimize.cc.

3.150.2.2 tree_node() [2/2]

```
tree_node::tree_node (
            const tree_node & other )  [inline]
```

Definition at line 136 of file optimize.cc.

3.150.3 Member Function Documentation

3.150.3.1 operator=()

```
tree_node & tree_node::operator= (
            const tree_node & other )  [inline]
```

Definition at line 143 of file optimize.cc.

3.150.4 Field Documentation

3.150.4.1 childs

```
vector<tree_node> tree_node::childs
```

Definition at line 127 of file optimize.cc.

3.150.4.2 sum_results

```
double tree_node::sum_results
```

Definition at line 128 of file optimize.cc.

3.150.4.3 num_visits

```
int tree_node::num_visits
```

Definition at line 129 of file optimize.cc.

3.150.4.4 var

```
WORD tree_node::var
```

Definition at line 130 of file optimize.cc.

3.150.4.5 finished

```
bool tree_node::finished
```

Definition at line 131 of file optimize.cc.

The documentation for this class was generated from the following file:

- optimize.cc

3.151 VaRrEnUm Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD * start
- WORD * lo
- WORD * hi

3.151.1 Detailed Description

Contains the pointers to an array in which a binary search will be performed.

Definition at line 164 of file structs.h.

3.151.2 Field Documentation

3.151.2.1 start

```
WORD* VaRrEnUm::start
```

Start point for search. Points inbetween lo and hi

Definition at line 165 of file structs.h.

Referenced by DoRecovery().

3.151.2.2 lo

```
WORD* VaRrEnUm::lo
```

Start of memory area

Definition at line 166 of file structs.h.

Referenced by DoRecovery(), Generator(), and GetFirstTerm().

3.151.2.3 hi

```
WORD* VaRrEnUm::hi
```

End of memory area

Definition at line 167 of file structs.h.

Referenced by DoRecovery().

The documentation for this struct was generated from the following file:

- structs.h

3.152 VeCtOr Struct Reference

Data Fields

- LONG name
- WORD complex
- WORD number
- WORD flags
- WORD node
- WORD namesize
- WORD dimension

3.152.1 Detailed Description

Definition at line 462 of file structs.h.

3.152.2 Field Documentation

3.152.2.1 name

```
LONG VeCtOr::name
```

Definition at line 463 of file structs.h.

3.152.2.2 complex

```
WORD VeCtOr::complex
```

Definition at line 464 of file structs.h.

3.152.2.3 number

```
WORD VeCtOr::number
```

Definition at line 465 of file structs.h.

3.152.2.4 flags

```
WORD VeCtOr::flags
```

Definition at line 466 of file structs.h.

3.152.2.5 node

```
WORD VeCtOr::node
```

Definition at line 467 of file structs.h.

3.152.2.6 namesize

```
WORD VeCtOr::namesize
```

Definition at line 468 of file structs.h.

3.152.2.7 dimension

```
WORD VeCtOr::dimension
```

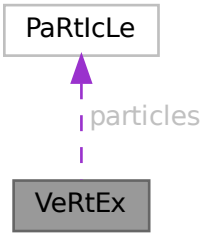
Definition at line 469 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.153 VeRtEx Struct Reference

Collaboration diagram for VeRtEx:



Data Fields

- PARTICLE particles [MAXPARTICLES]
- WORD couplings [2 ∗MAXCOUPLINGS]
- WORD nparticles
- WORD ncouplings
- WORD type
- WORD error
- WORD externonly
- WORD spare

3.153.1 Detailed Description

Definition at line 625 of file structs.h.

3.153.2 Field Documentation

3.153.2.1 particles

```
PARTICLE VeRtEx::particles[MAXPARTICLES]
```

Definition at line 626 of file structs.h.

3.153.2.2 couplings

```
WORD VeRtEx::couplings[2 *MAXCOUPLINGS]
```

Definition at line 627 of file structs.h.

3.153.2.3 nparticles

```
WORD VeRtEx::nparticles
```

Definition at line 628 of file structs.h.

3.153.2.4 ncouplings

```
WORD VeRtEx::ncouplings
```

Definition at line 629 of file structs.h.

3.153.2.5 type

```
WORD VeRtEx::type
```

Definition at line 630 of file structs.h.

3.153.2.6 error

```
WORD VeRtEx::error
```

Definition at line 631 of file structs.h.

3.153.2.7 externonly

```
WORD VeRtEx::externonly
```

Definition at line 632 of file structs.h.

3.153.2.8 spare

```
WORD VeRtEx::spare
```

Definition at line 633 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.154 X_const Struct Reference

```
#include <structs.h>
```

Data Fields

- UBYTE * currentPrompt
- UBYTE * shellname
- UBYTE * stderrname
- int timeout
- int killSignal
- int killWholeGroup
- int daemonize
- int currentExternalChannel

3.154.1 Detailed Description

The X_const struct is part of the global data and resides in the ALLGLOBALS struct A under the name X We see it used with the macro AX as in AX.timeout It contains variables that involve communication with external programs

Definition at line 2445 of file structs.h.

3.154.2 Field Documentation

3.154.2.1 currentPrompt

```
UBYTE* X_const::currentPrompt
```

Definition at line 2446 of file structs.h.

3.154.2.2 shellname

```
UBYTE* X_const::shellname
```

Definition at line 2447 of file structs.h.

3.154.2.3 stderrname

```
UBYTE* X_const::stderrname
```

Definition at line 2449 of file structs.h.

3.154.2.4 timeout

```
int X_const::timeout
```

Definition at line 2451 of file structs.h.

3.154.2.5 killSignal

```
int X_const::killSignal
```

Definition at line 2454 of file structs.h.

3.154.2.6 killWholeGroup

```
int X_const::killWholeGroup
```

Definition at line 2455 of file structs.h.

3.154.2.7 daemonize

```
int X_const::daemonize
```

Definition at line 2457 of file structs.h.

3.154.2.8 currentExternalChannel

```
int X_const::currentExternalChannel
```

Definition at line 2458 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

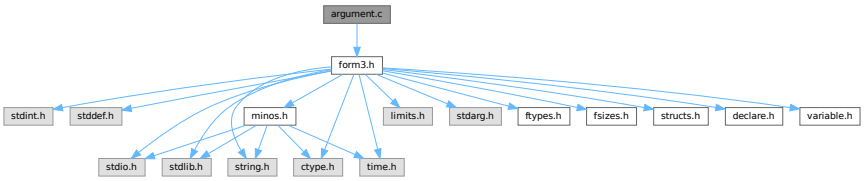
Chapter 4

File Documentation

4.1 argument.c File Reference

```
#include "form3.h"
```

Include dependency graph for argument.c:



Macros

- #define NEWORDER

Functions

- int execarg (PHEAD WORD *term, WORD level)
- int execterm (PHEAD WORD *term, WORD level)
- int ArgumentImplode (PHEAD WORD *term, WORD *thelist)
- int ArgumentExplode (PHEAD WORD *term, WORD *thelist)
- int ArgFactorize (PHEAD WORD *argin, WORD *argout)
- WORD FindArg (PHEAD WORD *a)
- int InsertArg (PHEAD WORD *argin, WORD *argout, int par)
- int CleanupArgCache (PHEAD WORD bufnum)
- int ArgSymbolMerge (WORD *t1, WORD *t2)
- int ArgDotproductMerge (WORD *t1, WORD *t2)
- WORD * TakeArgContent (PHEAD WORD *argin, WORD *argout)
- WORD * MakeInteger (PHEAD WORD *argin, WORD *argout, WORD *argfree)
- WORD * MakeMod (PHEAD WORD *argin, WORD *argout, WORD *argfree)
- void SortWeights (LONG *weights, LONG *extraspace, WORD number)

4.1.1 Detailed Description

Contains the routines that deal with the execution phase of the argument and related statements (like term)

Definition in file argument.c.

4.1.2 Macro Definition Documentation

4.1.2.1 NEWORDER

```
#define NEWORDER
```

Factorizes an argument in general notation (meaning that the first word of the argument is a positive size indicator) Input (argin): pointer to the complete argument Output (argout): Pointer to where the output should be written. This is in the WorkSpace Return value should be negative if anything goes wrong.

The notation of the output should be a string of arguments terminated by the number zero.

Originally we sorted in a way that the constants came last. This gave conflicts with the dollar and expression factorizations (in the expressions we wanted the zero first and then followed by the constants).

Definition at line 2068 of file argument.c.

4.1.3 Function Documentation

4.1.3.1 execarg()

```
int execarg (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 56 of file argument.c.

4.1.3.2 execterm()

```
int execterm (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 1777 of file argument.c.

4.1.3.3 ArgumentImplode()

```
int ArgumentImplode (
            PHEAD WORD * term,
            WORD * thelist )
```

Definition at line 1853 of file argument.c.

4.1.3.4 ArgumentExplode()

```
int ArgumentExplode (
            PHEAD WORD * term,
            WORD * thelist )
```

Definition at line 1957 of file argument.c.

4.1.3.5 ArgFactorize()

```
int ArgFactorize (
            PHEAD WORD * argin,
            WORD * argout )
```

Definition at line 2070 of file argument.c.

4.1.3.6 FindArg()

```
WORD FindArg (
            PHEAD WORD * a )
```

Looks the argument up in the (workers) table. If it is found the number in the table is returned (plus one to make it positive). If it is not found we look in the compiler provided table. If it is found - the number in the table is returned (minus one to make it negative). If in neither table we return zero.

Definition at line 2525 of file argument.c.

4.1.3.7 InsertArg()

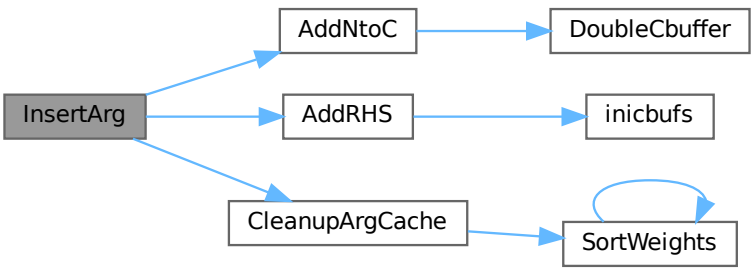
```
int InsertArg (
            PHEAD WORD * argin,
            WORD * argout,
            int par )
```

Inserts the argument into the (workers) table. If the table is too full we eliminate half of it. The eliminated elements are the ones that have not been used most recently, weighted by their total use and age(?). If par == 0 it inserts in the regular factorization cache If par == 1 it inserts in the cache defined with the FactorCache statement

Definition at line 2549 of file argument.c.

References AddNtoC(), AddRHS(), and CleanupArgCache().

Here is the call graph for this function:



4.1.3.8 CleanupArgCache()

```
int CleanupArgCache (
            PHEAD WORD bufnum )
```

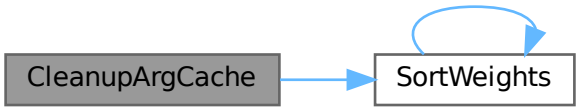
Cleans up the argument factorization cache. We throw half the elements. For a weight of what we want to keep we use the product of usage and the number in the buffer.

Definition at line 2584 of file argument.c.

References CbUf::boomlijst, CbUf::Buffer, CbUf::Pointer, CbUf::rhs, SortWeights(), and tree::usage.

Referenced by InsertArg().

Here is the call graph for this function:



4.1.3.9 ArgSymbolMerge()

```
int ArgSymbolMerge (
            WORD * t1,
            WORD * t2 )
```

Definition at line 2655 of file argument.c.

4.1.3.10 ArgDotproductMerge()

```
int ArgDotproductMerge (
            WORD * t1,
            WORD * t2 )
```

Definition at line 2710 of file argument.c.

4.1.3.11 TakeArgContent()

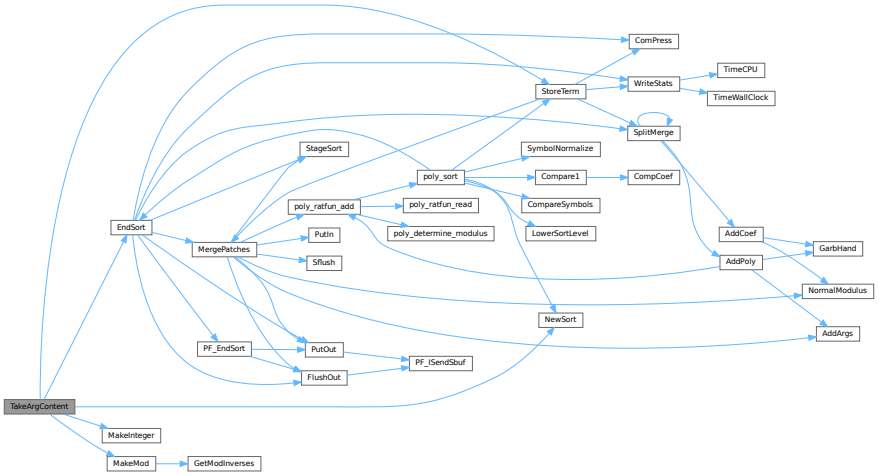
```
WORD * TakeArgContent (
            PHEAD WORD * argin,
            WORD * argout )
```

Implements part of the old ExecArg in which we take common factors from arguments with more than one term. The common pieces are put in argout as a sequence of arguments. The part with the multiple terms that are now relative prime is put in argfree which is allocated via TermMalloc and is given as the return value. The difference with the old code is that negative powers are always removed. Hence it is as in MakeInteger in which only numerators will be left: now only zero or positive powers will be remaining.

Definition at line 2778 of file argument.c.

References EndSort(), MakeInteger(), MakeMod(), NewSort(), and StoreTerm().

Here is the call graph for this function:



4.1.3.12 MakeInteger()

```
WORD * MakeInteger (
            PHEAD WORD * argin,
            WORD * argout,
            WORD * argfree )
```

For normalizing everything to integers we have to determine for all elements of this argument the LCM of the denominators and the GCD of the numerators. The input argument is in argin. The number that comes out should go to argout. The new pointer in the argout buffer is the return value. The normalized argument is in argfree.

Definition at line 3328 of file argument.c.

Referenced by TakeArgContent().

4.1.3.13 MakeMod()

```
WORD * MakeMod (
            PHEAD WORD * argin,
            WORD * argout,
            WORD * argfree )
```

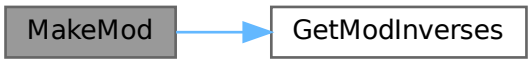
Similar to MakeInteger but now with modulus arithmetic using only a one WORD 'prime'. We make the coefficient of the first term in the argument equal to one. Already the coefficients are taken modulus AN.cmod and AN.ncmod == 1

Definition at line 3499 of file argument.c.

References GetModInverses().

Referenced by TakeArgContent().

Here is the call graph for this function:



4.1.3.14 SortWeights()

```
void SortWeights (
            LONG * weights,
            LONG * extraspace,
            WORD number )
```

Sorts an array of LONGS in the same way SplitMerge (in sort.c) works We use gradual division in two.

Definition at line 3544 of file argument.c.

References SortWeights().

Referenced by CleanupArgCache(), and SortWeights().

Here is the call graph for this function:

4.2 argument.c

Go to the documentation of this file.

```
00001
00007 /* #[ License : */
00008 /*
00009  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00010  *   When using this file you are requested to refer to the publication
00011  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00012  *   This is considered a matter of courtesy as the development was paid
00013  *   for by FOM the Dutch physics granting agency and we would like to
00014  *   be able to track its scientific use to convince FOM of its value
00015  *   for the community.
00016  *
00017  *   This file is part of FORM.
00018  *
00019  *   FORM is free software: you can redistribute it and/or modify it under the
00020  *   terms of the GNU General Public License as published by the Free Software
00021  *   Foundation, either version 3 of the License, or (at your option) any later
00022  *   version.
00023  *
00024  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00025  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00026  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00027  *   details.
00028  *
00029  *   You should have received a copy of the GNU General Public License along
00030  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00031  */
00032 /* #] License : */
00033
00034 /*
00035     #[ include : argument.c
00036 */
00037
00038 #include "form3.h"
00039
00040 /*
00041     #] include :
00042     #[ execarg :
00043
00044     Executes the subset of statements in an argument environment.
00045     The calling routine should be of the type
00046     if ( C->lhs[level][0] == TYPEARG ) {
00047         if ( execarg(term,level) ) goto GenCall;
00048         level = C->lhs[level][2];
00049         goto SkipCount;
00050     }
00051     Note that there will be cases in which extra space is needed.
00052     In addition the compare with C->numlhs isn't very fine, because we
00053     need to insert a different value (C->lhs[level][2]).
00054 */
00055
00056 int execarg(PHEAD WORD *term, WORD level)
00057 {
00058     GETBIDENTITY
00059     WORD *t, *r, *m, *v;
00060     WORD *start, *stop, *rstop, *r1, *r2 = 0, *r3 = 0, *r4, *r5, *r6, *r7, *r8, *r9;
00061     WORD *mm, *mstop, *rnext, *rr, *factor, type, ngcd, nq;
00062     CBUF *C = cbuf+AM.rbufnum, *CC = cbuf+AT.ebufnum;
00063     WORD i, j, k, oldnumlhs = AR.Cnumlhs, count, olddefer = AR.DeferFlag;
00064     int action = 0;
00065     WORD oldnumrhs = CC->numrhs, size, pow, jj;
00066     LONG oldcpointer = CC->Pointer - CC->Buffer, oldppointer = AT.pWorkPointer, lp;
00067     WORD *oldwork = AT.WorkPointer, *oldwork2, scale, renorm;
00068     WORD kLCM = 0, kGCD = 0, kGCD2, kkLCM = 0, jLCM = 0, jGCD, sign = 1;
00069     int ii, didpolyratfun;
00070     UWORD *EAscrat, *GCDbuffer = 0, *GCDbuffer2 = 0, *LCMbuffer = 0, *LCMb = 0, *LCMc = 0;
00071     AT.WorkPointer += *term;
00072     start = C->lhs[level];
00073     AR.Cnumlhs = start[2];
00074     stop = start + start[1];
00075     type = *start;
00076     scale = start[4];
00077     renorm = start[5];
00078     start += TYPEARGHEADSIZE;
00079 /*
00080     #[ Dollars :
00081 */
00082     if ( renorm && start[1] != 0 ) {/* We have to evaluate $ symbols inside () */
00083         t = start+1; factor = oldwork2 = v = AT.WorkPointer;
00084         i = *t; t++;
00085         *v++ = i+3; i--; NCOPY(v,t,i);
00086         *v++ = 1; *v++ = 1; *v++ = 3;
00087         AT.WorkPointer = v;
```

```
00088         start = t; AR.Eside = LHSIDEX;
00089         NewSort(BHEAD0);
00090         if ( Generator(BHEAD factor,AR.Cnumlhs) ) {
00091             LowerSortLevel();
00092             AT.WorkPointer = oldwork;
00093             return(-1);
00094         }
00095         AT.WorkPointer = v;
00096         if ( EndSort(BHEAD factor,0) < 0 ) {}
00097         if ( *factor && *(factor+*factor) != 0 ) {
00098             MLOCK(ErrorMessageLock);
00099             MesPrint("&$ in () does not evaluate into a single term");
00100             MUNLOCK(ErrorMessageLock);
00101             return(-1);
00102         }
00103         AR.Eside = RHSIDE;
00104         if ( *factor > 0 ) {
00105             v = factor+*factor;
00106             v -= ABS(v[-1]);
00107             *factor = v-factor;
00108         }
00109         AT.WorkPointer = v;
00110     }
00111     else {
00112         if ( *start < 0 ) {
00113             factor = start + 1;
00114             start += -*start;
00115         }
00116         else factor = 0;
00117     }
00118 /*
00119     #] Dollars :
00120 */
00121     t = term;
00122     r = t + *t;
00123     rstop = r - ABS(r[-1]);
00124     t++;
00125 /*
00126     #[ Argument detection : + argument statement
00127 */
00128 /*
00129     Allocate Numbers for MakeInteger here, for re-use in case multiple
00130     functions are treated in the same term.
00131 */
00132     if ( type == TYPENORM4 ) {
00133         GCDbuffer = NumberMalloc("execarg");
00134         GCDbuffer2 = NumberMalloc("execarg");
00135         LCMbuffer = NumberMalloc("execarg");
00136         LCMb = NumberMalloc("execarg"); LCMc = NumberMalloc("execarg");
00137     }
00138     didpolyratfun = 0;
00139     while ( t < rstop ) {
00140         if ( *t >= FUNCTION && functions[*t-FUNCTION].spec <= 0 ) {
00141 /*
00142             We have a function. First count the number of arguments.
00143             Tensors are excluded.
00144 */
00145             count = 0;
00146             v = t;
00147             m = t + FUNHEAD;
00148             r = t + t[1];
00149             while ( m < r ) {
00150                 count++;
00151                 NEXTARG(m)
00152             }
00153             if ( count <= 0 ) { t += t[1]; continue; }
00154 /*
00155             Now we take the arguments one by one and test for a match
00156 */
00157             for ( i = 1; i <= count; i++ ) {
00158                 m = start;
00159                 while ( m < stop ) {
00160                     r = m + m[1];
00161                     j = *r++;
00162                     if ( j > 1 ) {
00163                         while ( --j > 0 ) {
00164                             if ( *r == i ) goto RightNum;
00165                             r++;
00166                         }
00167                         m = r;
00168                         continue;
00169                     }
00170 RightNum:
00171                     if ( m[1] == 2 ) {
00172 #ifdef WITHFLOAT
00173                         if ( *t != FLOATFUN || TestFloat(t) == 0 )
00174 #endif
```

```
00175                           {
00176                               m += 2;
00177                               m += *m;
00178                               goto HaveTodo;
00179                           }
00180 #ifdef WITHFLOAT
00181                           else {
00182                               m += 2;
00183                           }
00184 #endif
00185                       }
00186                       else {
00187                           r = m + m[1];
00188                           m += 2;
00189                           while ( m < r ) {
00190                               if ( *m == CSET ) {
00191                                   r1 = SetElements + Sets[m[1]].first;
00192                                   r2 = SetElements + Sets[m[1]].last;
00193                                   while ( r1 < r2 ) {
00194                                       if ( *r1++ == *t ) goto HaveTodo;
00195                                   }
00196                               }
00197                               else if ( m[1] == *t ) goto HaveTodo;
00198                               m += 2;
00199                           }
00200                       }
00201                       m += *m;
00202                   }
00203                   continue;
00204 HaveTodo:
00205 /*
00206                   If we come here we have to do the argument i (first is 1).
00207 */
00208                   sign = 1;
00209                   action = 1;
00210                   if ( *t == AR.PolyFun ) didpolyratfun = 1;
00211                   v[2] |= DIRTYFLAG;
00212                   r = t + FUNHEAD;
00213                   j = i;
00214                   while ( --j > 0 ) { NEXTARG(r) }
00215                   if ( ( type == TYPESPLITARG ) || ( type == TYPESPLITFIRSTARG )
00216                    || ( type == TYPESPLITLASTARG ) ) {
00217                       if ( *t > FUNCTION && *r > 0 ) {
00218                           WantAddPointers(2);
00219                           AT.pWorkSpace[AT.pWorkPointer++] = t;
00220                           AT.pWorkSpace[AT.pWorkPointer++] = r;
00221                       }
00222                       continue;
00223                   }
00224                   else if ( type == TYPESPLITARG2 ) {
00225                       if ( *t > FUNCTION && *r > 0 ) {
00226                           WantAddPointers(2);
00227                           AT.pWorkSpace[AT.pWorkPointer++] = t;
00228                           AT.pWorkSpace[AT.pWorkPointer++] = r;
00229                       }
00230                       continue;
00231                   }
00232                   else if ( type == TYPEFACTARG || type == TYPEFACTARG2 ) {
00233                       if ( *t > FUNCTION || *t == DENOMINATOR ) {
00234                           if ( *r > 0 ) {
00235                           mm = r + ARGHEAD; mstop = r + *r;
00236                           if ( mm + *mm < mstop ) {
00237                               WantAddPointers(2);
00238                               AT.pWorkSpace[AT.pWorkPointer++] = t;
00239                               AT.pWorkSpace[AT.pWorkPointer++] = r;
00240                               continue;
00241                           }
00242                           if ( *mm == 1+ABS(mstop[-1]) ) continue;
00243                           if ( mstop[-3] != 1 || mstop[-2] != 1
00244                               || mstop[-1] != 3 ) {
00245                               WantAddPointers(2);
00246                               AT.pWorkSpace[AT.pWorkPointer++] = t;
00247                               AT.pWorkSpace[AT.pWorkPointer++] = r;
00248                               continue;
00249                           }
00250                           GETSTOP(mm,mstop); mm++;
00251                           if ( mm + mm[1] < mstop ) {
00252                               WantAddPointers(2);
00253                               AT.pWorkSpace[AT.pWorkPointer++] = t;
00254                               AT.pWorkSpace[AT.pWorkPointer++] = r;
00255                               continue;
00256                           }
00257                           if ( *mm == SYMBOL && ( mm[1] > 4 ||
00258                               ( mm[3] != 1 && mm[3] != -1 ) ) ) {
00259                               WantAddPointers(2);
00260                               AT.pWorkSpace[AT.pWorkPointer++] = t;
00261                               AT.pWorkSpace[AT.pWorkPointer++] = r;
```

```
00262                           continue;
00263                       }
00264                       else if ( *mm == DOTPRODUCT && ( mm[1] > 5 ||
00265                           ( mm[4] != 1 && mm[4] != -1 ) ) ) {
00266                           WantAddPointers(2);
00267                           AT.pWorkSpace[AT.pWorkPointer++] = t;
00268                           AT.pWorkSpace[AT.pWorkPointer++] = r;
00269                           continue;
00270                       }
00271                       else if ( ( *mm == DELTA || *mm == VECTOR )
00272                            && mm[1] > 4 ) {
00273                           WantAddPointers(2);
00274                           AT.pWorkSpace[AT.pWorkPointer++] = t;
00275                           AT.pWorkSpace[AT.pWorkPointer++] = r;
00276                           continue;
00277                       }
00278                       }
00279                       else if ( factor && *factor == 4 && factor[2] == 1 ) {
00280                           WantAddPointers(2);
00281                           AT.pWorkSpace[AT.pWorkPointer++] = t;
00282                           AT.pWorkSpace[AT.pWorkPointer++] = r;
00283                           continue;
00284                       }
00285                       else if ( factor && *factor == 0
00286                       && ( *r == -SNUMBER && r[1] != 1 ) ) {
00287                           WantAddPointers(2);
00288                           AT.pWorkSpace[AT.pWorkPointer++] = t;
00289                           AT.pWorkSpace[AT.pWorkPointer++] = r;
00290                           continue;
00291                       }
00292                       else if ( *r == -MINVECTOR ) {
00293                           WantAddPointers(2);
00294                           AT.pWorkSpace[AT.pWorkPointer++] = t;
00295                           AT.pWorkSpace[AT.pWorkPointer++] = r;
00296                           continue;
00297                       }
00298                   }
00299                   continue;
00300               }
00301               else if ( type == TYPENORM || type == TYPENORM2 || type == TYPENORM3 || type ==
      TYPENORM4 ) {
00302                   if ( *r < 0 ) {
00303                       WORD rone;
00304                       if ( *r == -MINVECTOR ) { rone = -1; *r = -INDEX; }
00305                       else if ( *r != -SNUMBER || r[1] == 1 || r[1] == 0 ) continue;
00306                       else { rone = r[1]; r[1] = 1; }
00307 /*
00308                       Now we must multiply the general coefficient by r[1]
00309 */
00310                       if ( scale && ( factor == 0 || *factor ) ) {
00311                           action = 1;
00312                           v[2] |= DIRTYFLAG;
00313                           if ( rone < 0 ) {
00314                               if ( type == TYPENORM3 ) k = 1;
00315                               else k = -1;
00316                               rone = -rone;
00317                           }
00318                           else k = 1;
00319                           r1 = term + *term;
00320                           size = r1[-1];
00321                           size = REDLENG(size);
00322                           if ( scale > 0 ) {
00323                               for ( jj = 0; jj < scale; jj++ ) {
00324                                   if ( Mully(BHEAD (UWORD *)rstop,&size,(UWORD *)(&rone),k) )
00325                                       goto execargerr;
00326                               }
00327                           }
00328                           else {
00329                               for ( jj = 0; jj > scale; jj-- ) {
00330                                   if ( Divvy(BHEAD (UWORD *)rstop,&size,(UWORD *)(&rone),k) )
00331                                       goto execargerr;
00332                               }
00333                           }
00334                           size = INCLENG(size);
00335                           k = size < 0 ? -size: size;
00336                           rstop[k-1] = size;
00337                           *term = (WORD)(rstop - term) + k;
00338                       }
00339                       continue;
00340                   }
00341 /*
00342                   Now we have to find a reference term.
00343                   If factor is defined and *factor != 0 we have to
00344                   look for the first term that matches the pattern exactly
00345                   Otherwise the first term plays this role
00346                   If its coefficient is not one,
00347                   we must set up a division of the whole argument by
```

```
00348                     this coefficient, and a multiplication of the term
00349                     when the type is not equal to TYPENORM2.
00350                     We first multiply the coefficient of the term.
00351                     Then we set up the division.
00352 
00353                     First find the magic term
00354 */
00355                     if ( type == TYPENORM4 ) {
00356 /*
00357                         For normalizing everything to integers we have to
00358                         determine for all elements of this argument the LCM of
00359                         the denominators and the GCD of the numerators.
00360                         The buffers have been allocated already.
00361 */
00362                         r4 = r + *r;
00363                         r1 = r + ARGHEAD;
00364 /*
00365                         First take the first term to load up the LCM and the GCD
00366 */
00367                         r2 = r1 + *r1;
00368                         j = r2[-1];
00369                         if ( j < 0 ) sign = -1;
00370                         r3 = r2 - ABS(j);
00371                         k = REDLENG(j);
00372                         if ( k < 0 ) k = -k;
00373                         while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00374                         for ( kGCD = 0; kGCD < k; kGCD++ ) GCDbuffer[kGCD] = r3[kGCD];
00375                         k = REDLENG(j);
00376                         if ( k < 0 ) k = -k;
00377                         r3 += k;
00378                         while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00379                         for ( kLCM = 0; kLCM < k; kLCM++ ) LCMbuffer[kLCM] = r3[kLCM];
00380                         r1 = r2;
00381 /*
00382                         Now go through the rest of the terms in this argument.
00383 */
00384                         while ( r1 < r4 ) {
00385                             r2 = r1 + *r1;
00386                             j = r2[-1];
00387                             r3 = r2 - ABS(j);
00388                             k = REDLENG(j);
00389                             if ( k < 0 ) k = -k;
00390                             while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00391                             if ( ( ( GCDbuffer[0] == 1 ) && ( kGCD == 1 ) ) ) {
00392 /*
00393                                 GCD is already 1
00394 */
00395                             }
00396                             else if ( ( ( k != 1 ) || ( r3[0] != 1 ) ) ) {
00397                                 if ( GcdLong(BHEAD GCDbuffer,kGCD,(UWORD *)r3,k,GCDbuffer2,&kGCD2) ) {
00398                                     NumberFree(GCDbuffer,"execarg");
00399                                     NumberFree(GCDbuffer2,"execarg");
00400                                     NumberFree(LCMbuffer,"execarg");
00401                                     NumberFree(LCMb,"execarg"); NumberFree(LCMc,"execarg");
00402                                     goto execargerr;
00403                                 }
00404                                 kGCD = kGCD2;
00405                                 for ( ii = 0; ii < kGCD; ii++ ) GCDbuffer[ii] = GCDbuffer2[ii];
00406                             }
00407                             else {
00408                                 kGCD = 1; GCDbuffer[0] = 1;
00409                             }
00410                             k = REDLENG(j);
00411                             if ( k < 0 ) k = -k;
00412                             r3 += k;
00413                             while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00414                             if ( ( ( LCMbuffer[0] == 1 ) && ( kLCM == 1 ) ) ) {
00415                                 for ( kLCM = 0; kLCM < k; kLCM++ )
00416                                     LCMbuffer[kLCM] = r3[kLCM];
00417                             }
00418                             else if ( ( k != 1 ) || ( r3[0] != 1 ) ) {
00419                                 if ( GcdLong(BHEAD LCMbuffer,kLCM,(UWORD *)r3,k,LCMb,&kkLCM) ) {
00420                                     NumberFree(GCDbuffer,"execarg"); NumberFree(GCDbuffer2,"execarg");
00421                                     NumberFree(LCMbuffer,"execarg"); NumberFree(LCMb,"execarg");
      NumberFree(LCMc,"execarg");
00422                                     goto execargerr;
00423                                 }
00424                                 DivLong((UWORD *)r3,k,LCMb,kkLCM,LCMb,&kkLCM,LCMc,&jLCM);
00425                                 MulLong(LCMbuffer,kLCM,LCMb,kkLCM,LCMc,&jLCM);
00426                                 for ( kLCM = 0; kLCM < jLCM; kLCM++ )
00427                                     LCMbuffer[kLCM] = LCMc[kLCM];
00428                             }
00429                             else {} /* LCM doesn't change */
00430                             r1 = r2;
00431                         }
00432 /*
00433                         Now put the factor together: GCD/LCM
```

```
00434 */
00435                         r3 = (WORD *)(GCDbuffer);
00436                         if ( kGCD == kLCM ) {
00437                             for ( jGCD = 0; jGCD < kGCD; jGCD++ )
00438                                 r3[jGCD+kGCD] = LCMbuffer[jGCD];
00439                             k = kGCD;
00440                         }
00441                         else if ( kGCD > kLCM ) {
00442                             for ( jGCD = 0; jGCD < kLCM; jGCD++ )
00443                                 r3[jGCD+kGCD] = LCMbuffer[jGCD];
00444                             for ( jGCD = kLCM; jGCD < kGCD; jGCD++ )
00445                                 r3[jGCD+kGCD] = 0;
00446                             k = kGCD;
00447                         }
00448                         else {
00449                             for ( jGCD = kGCD; jGCD < kLCM; jGCD++ )
00450                                 r3[jGCD] = 0;
00451                             for ( jGCD = 0; jGCD < kLCM; jGCD++ )
00452                                 r3[jGCD+kLCM] = LCMbuffer[jGCD];
00453                             k = kLCM;
00454                         }
00455
00456                         j = 2*k+1;
00457 /*
00458                         Now we have to correct the overall factor
00459 */
00460                         if ( scale && ( factor == 0 || *factor > 0 ) )
00461                             goto ScaledVariety;
00462 /*
00463                         The if was added 28-nov-2012 to give MakeInteger also
00464                         the (0) option.
00465 */
00466                         if ( scale && ( factor == 0 || *factor ) ) {
00467                             size = term[*term-1];
00468                             size = REDLENG(size);
00469                             if ( MulRat(BHEAD (UWORD *)rstop,size,(UWORD *)r3,k,
00470                                     (UWORD *)rstop,&size) ) goto execargerr;
00471                             size = INCLENG(size);
00472                             k = size < 0 ? -size: size;
00473                             rstop[k-1] = size*sign;
00474                             *term = (WORD)(rstop - term) + k;
00475                         }
00476                     }
00477                     else {
00478                         if ( factor && *factor >= 1 ) {
00479                             r4 = r + *r;
00480                             r1 = r + ARGHEAD;
00481                             while ( r1 < r4 ) {
00482                                 r2 = r1 + *r1;
00483                                 r3 = r2 - ABS(r2[-1]);
00484                                 j = r3 - r1;
00485                                 r5 = factor;
00486                                 if ( j != *r5 ) { r1 = r2; continue; }
00487                                 r5++; r6 = r1+1;
00488                                 while ( --j > 0 ) {
00489                                     if ( *r5 != *r6 ) break;
00490                                     r5++; r6++;
00491                                 }
00492                                 if ( j > 0 ) { r1 = r2; continue; }
00493                                 break;
00494                             }
00495                             if ( r1 >= r4 ) continue;
00496                         }
00497                         else {
00498                             r1 = r + ARGHEAD;
00499                             r2 = r1 + *r1;
00500                             r3 = r2 - ABS(r2[-1]);
00501                         }
00502                         if ( *r3 == 1 && r3[1] == 1 ) {
00503                             if ( r2[-1] == 3 ) continue;
00504                             if ( r2[-1] == -3 && type == TYPENORM3 ) continue;
00505                         }
00506                         action = 1;
00507                         v[2] |= DIRTYFLAG;
00508                         j = r2[-1];
00509                         k = REDLENG(j);
00510                         if ( j < 0 ) j = -j;
00511                         if ( type == TYPENORM && scale && ( factor == 0 || *factor ) ) {
00512 /*
00513                             Now we correct the overall factor
00514 */
00515 ScaledVariety:;
00516                             size = term[*term-1];
00517                             size = REDLENG(size);
00518                             if ( scale > 0 ) {
00519                                 for ( jj = 0; jj < scale; jj++ ) {
00520                                     if ( MulRat(BHEAD (UWORD *)rstop,size,(UWORD *)r3,k,
```

```
00521                                       (UWORD *)rstop,&size) ) goto execargerr;
00522                                 }
00523                             }
00524                             else {
00525                                 for ( jj = 0; jj > scale; jj-- ) {
00526                                     if ( DivRat(BHEAD (UWORD *)rstop,size,(UWORD *)r3,k,
00527                                         (UWORD *)rstop,&size) ) goto execargerr;
00528                                 }
00529                             }
00530                             size = INCLENG(size);
00531                             k = size < 0 ? -size: size;
00532                             rstop[k-1] = size*sign;
00533                             *term = (WORD)(rstop - term) + k;
00534                         }
00535                     }
00536 /*
00537                     We generate a statement for adapting all terms in the
00538                     argument successively
00539 */
00540                     r4 = AddRHS(AT.ebufnum,1);
00541                     while ( (r4+j+12) > CC->Top ) r4 = DoubleCbuffer(AT.ebufnum,r4,3);
00542                     *r4++ = j+1;
00543                     i = (j-1)/2;  /* was (j-1)*2  ????? 17-oct-2017 */
00544                     for ( k = 0; k < i; k++ ) *r4++ = r3[i+k];
00545                     for ( k = 0; k < i; k++ ) *r4++ = r3[k];
00546                     if ( ( type == TYPENORM3 ) || ( type == TYPENORM4 ) ) *r4++ = j*sign;
00547                     else *r4++ = r3[j-1];
00548                     *r4++ = 0;
00549                     CC->rhs[CC->numrhs+1] = r4;
00550                     CC->Pointer = r4;
00551                     AT.mulpat[5] = CC->numrhs;
00552                     AT.mulpat[7] = AT.ebufnum;
00553                 }
00554                 else if ( type == TYPEARGTOEXTRASYMBOL ) {
00555                     WORD n;
00556                     if ( r[0] < 0 ) {
00557                         /* The argument is in the fast notation. */
00558                         WORD tmp[MaX(9,FUNHEAD+5)];
00559                         switch ( r[0] ) {
00560                             case -SNUMBER:
00561                                 if ( r[1] == 0 ) {
00562                                     tmp[0] = 0;
00563                                 }
00564                                 else {
00565                                     tmp[0] = 4;
00566                                     tmp[1] = ABS(r[1]);
00567                                     tmp[2] = 1;
00568                                     tmp[3] = r[1] > 0 ? 3 : -3;
00569                                     tmp[4] = 0;
00570                                 }
00571                                 break;
00572                             case -SYMBOL:
00573                                 tmp[0] = 8;
00574                                 tmp[1] = SYMBOL;
00575                                 tmp[2] = 4;
00576                                 tmp[3] = r[1];
00577                                 tmp[4] = 1;
00578                                 tmp[5] = 1;
00579                                 tmp[6] = 1;
00580                                 tmp[7] = 3;
00581                                 tmp[8] = 0;
00582                                 break;
00583                             case -INDEX:
00584                             case -VECTOR:
00585                             case -MINVECTOR:
00586                                 tmp[0] = 7;
00587                                 tmp[1] = INDEX;
00588                                 tmp[2] = 3;
00589                                 tmp[3] = r[1];
00590                                 tmp[4] = 1;
00591                                 tmp[5] = 1;
00592                                 tmp[6] = r[0] != -MINVECTOR ? 3 : -3;
00593                                 tmp[7] = 0;
00594                                 break;
00595                             default:
00596                                 if ( r[0] <= -FUNCTION ) {
00597                                     tmp[0] = FUNHEAD+4;
00598                                     tmp[1] = -r[0];
00599                                     tmp[2] = FUNHEAD;
00600                                     ZeroFillRange(tmp,3,1+FUNHEAD);
00601                                     tmp[FUNHEAD+1] = 1;
00602                                     tmp[FUNHEAD+2] = 1;
00603                                     tmp[FUNHEAD+3] = 3;
00604                                     tmp[FUNHEAD+4] = 0;
00605                                     break;
00606                                 }
00607                                 else {
```

```
00608 /* INTERNAL_ERROR_EXCL_START */
00609                                     MLOCK(ErrorMessageLock);
00610                                     MesPrint("!>Unknown fast notation found (TYPEARGTOEXTRASYMBOL)");
00611                                     MUNLOCK(ErrorMessageLock);
00612                                     return(-1);
00613 /* INTERNAL_ERROR_EXCL_STOP */
00614                                 }
00615                             }
00616                             n = FindSubexpression(tmp);
00617                         }
00618                         else {
00619                             /*
00620                              * NOTE: writing to r[r[0]] is legal. As long as we work
00621                              * in a part of the term, at least the coefficient of
00622                              * the term must follow.
00623                              */
00624                             WORD old_rr0 = r[r[0]];
00625                             r[r[0]] = 0;  /* zero-terminated */
00626                             n = FindSubexpression(r+ARGHEAD);
00627                             r[r[0]] = old_rr0;
00628                         }
00629                         /* Put the new argument in the work space. */
00630                         if ( AT.WorkPointer+2 > AT.WorkTop ) {
00631                             MLOCK(ErrorMessageLock);
00632                             MesWork();
00633                             MUNLOCK(ErrorMessageLock);
00634                             return(-1);
00635                         }
00636                         r1 = AT.WorkPointer;
00637                         if ( scale ) {  /* means "tonumber" */
00638                             r1[0] = -SNUMBER;
00639                             r1[1] = n;
00640                         }
00641                         else {
00642                             r1[0] = -SYMBOL;
00643                             r1[1] = MAXVARIABLES-n;
00644                         }
00645                         /* We need r2, r3, m and k to shift the data. */
00646                         r2 = r + (r[0] > 0 ? r[0] : r[0] <= -FUNCTION ? 1 : 2);
00647                         r3 = r;
00648                         m = r1+ARGHEAD+2;
00649                         k = 2;
00650                         goto do_shift;
00651                     }
00652                     r3 = r;
00653                     AR.DeferFlag = 0;
00654                     if ( *r > 0 ) {
00655                         NewSort(BHEAD0);
00656                         action = 1;
00657                         r2 = r + *r;
00658                         r += ARGHEAD;
00659                         while ( r < r2 ) {  /* Sum over the terms */
00660                             m = AT.WorkPointer;
00661                             j = *r;
00662                             while ( --j >= 0 ) *m++ = *r++;
00663                             r1 = AT.WorkPointer;
00664                             AT.WorkPointer = m;
00665 /*
00666                             What to do with dummy indices?
00667 */
00668                             if ( type == TYPENORM || type == TYPENORM2 || type == TYPENORM3 || type ==
      TYPENORM4 ) {
00669                                 if ( MultDo(BHEAD r1,AT.mulpat) ) goto execargerr;
00670                                 AT.WorkPointer = r1 + *r1;
00671                             }
00672                             if ( Generator(BHEAD r1,level) ) goto execargerr;
00673                             AT.WorkPointer = r1;
00674                         }
00675                     }
00676                     else {
00677                         r2 = r + (( *r <= -FUNCTION ) ? 1:2);
00678                         r1 = AT.WorkPointer;
00679                         ToGeneral(r,r1,0);
00680                         m = r1 + ARGHEAD;
00681                         AT.WorkPointer = r1 + *r1;
00682                         NewSort(BHEAD0);
00683                         action = 1;
00684 /*
00685                         What to do with dummy indices?
00686 */
00687                         if ( type == TYPENORM || type == TYPENORM2 || type == TYPENORM3 || type ==
      TYPENORM4 ) {
00688                             if ( MultDo(BHEAD m,AT.mulpat) ) goto execargerr;
00689                             AT.WorkPointer = m + *m;
00690                         }
00691                         if ( (*m != 0 ) && Generator(BHEAD m,level) ) goto execargerr;
00692                         AT.WorkPointer = r1;
```

```
00693                 }
00694                 if ( EndSort(BHEAD AT.WorkPointer+ARGHEAD,1) < 0 ) goto execargerr;
00695                 AR.DeferFlag = olddefer;
00696 /*
00697                 Now shift the sorted entity over the old argument.
00698 */
00699                 m = AT.WorkPointer+ARGHEAD;
00700                 while ( *m ) m += *m;
00701                 k = WORDDIF(m,AT.WorkPointer);
00702                 *AT.WorkPointer = k;
00703                 AT.WorkPointer[1] = 0;
00704                 if ( ToFast(AT.WorkPointer,AT.WorkPointer) ) {
00705                     if ( *AT.WorkPointer <= -FUNCTION ) k = 1;
00706                     else k = 2;
00707                 }
00708 do_shift:
00709                 if ( *r3 > 0 ) j = k - *r3;
00710                 else if ( *r3 <= -FUNCTION ) j = k - 1;
00711                 else j = k - 2;
00712
00713                 t[1] += j;
00714                 action = 1;
00715                 v[2] |= DIRTYFLAG;
00716                 if ( j > 0 ) {
00717                     r = m + j;
00718                     while ( m > AT.WorkPointer ) *--r = *--m;
00719                     AT.WorkPointer = r;
00720                     m = term + *term;
00721                     r = m + j;
00722                     while ( m > r2 ) *--r = *--m;
00723                 }
00724                 else if ( j < 0 ) {
00725                     r = r2 + j;
00726                     r1 = term + *term;
00727                     while ( r2 < r1 ) *r++ = *r2++;
00728                 }
00729                 r = r3;
00730                 m = AT.WorkPointer;
00731                 NCOPY(r,m,k);
00732                 *term += j;
00733                 rstop += j;
00734                 CC->numrhs = oldnumrhs;
00735                 CC->Pointer = CC->Buffer + oldcpointer;
00736             }
00737         }
00738         t += t[1];
00739     }
00740 /*
00741         If TYPENORM4, we allocated Number buffers before the above while loop. Free them.
00742 */
00743     if ( type == TYPENORM4 ) {
00744         NumberFree(GCDbuffer,"execarg");
00745         NumberFree(GCDbuffer2,"execarg");
00746         NumberFree(LCMbuffer,"execarg");
00747         NumberFree(LCMb,"execarg"); NumberFree(LCMc,"execarg");
00748     }
00749     if ( didpolyratfun ) {
00750         PolyFunDirty(BHEAD term);
00751         didpolyratfun = 0;
00752     }
00753 /*
00754     #] Argument detection :
00755     #[ SplitArg : + varieties
00756 */
00757     if ( ( type == TYPESPLITARG || type == TYPESPLITARG2
00758      || type == TYPESPLITFIRSTARG || type == TYPESPLITLASTARG ) &&
00759     AT.pWorkPointer > oldppointer ) {
00760         t = term+1;
00761         r1 = AT.WorkPointer + 1;
00762         lp = oldppointer;
00763         while ( t < rstop ) {
00764             if ( lp < AT.pWorkPointer && t == AT.pWorkSpace[lp] ) {
00765                 v = t;
00766                 m = t + FUNHEAD;
00767                 r = t + t[1];
00768                 r2 = r1; while ( t < m ) *r1++ = *t++;
00769                 while ( m < r ) {
00770                     t = m;
00771                     NEXTARG(m)
00772                     if ( lp >= AT.pWorkPointer || t != AT.pWorkSpace[lp+1] ) {
00773                         if ( *t > 0 ) t[1] = 0;
00774                         while ( t < m ) *r1++ = *t++;
00775                         continue;
00776                     }
00777 /*
00778                     Now we have a nontrivial argument that should be done.
00779 */
```

```
00780                   lp += 2;
00781                   action = 1;
00782                   v[2] |= DIRTYFLAG;
00783                   r3 = t + *t;
00784                   t += ARGHEAD;
00785                   if ( type == TYPESPLITFIRSTARG ) {
00786                       r4 = r1; r5 = t; r7 = oldwork;
00787                       *r1++ = *t + ARGHEAD;
00788                       for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
00789                       j = 0;
00790                       while ( t < r3 ) {
00791                           i = *t;
00792                           if ( j == 0 ) {
00793                               NCOPY(r7,t,i)
00794                               j++;
00795                           }
00796                           else {
00797                               NCOPY(r1,t,i)
00798                           }
00799                       }
00800                       *r4 = r1 - r4;
00801                       if ( j ) {
00802                           if ( ToFast(r4,r4) ) {
00803                               r1 = r4;
00804                               if ( *r1 > -FUNCTION ) r1++;
00805                               r1++;
00806                           }
00807                           r7 = oldwork;
00808                           while ( --j >= 0 ) {
00809                               r4 = r1; i = *r7;
00810                               *r1++ = i+ARGHEAD; *r1++ = 0;
00811                               FILLARG(r1);
00812                               NCOPY(r1,r7,i)
00813                               if ( ToFast(r4,r4) ) {
00814                                   r1 = r4;
00815                                   if ( *r1 > -FUNCTION ) r1++;
00816                                   r1++;
00817                               }
00818                           }
00819                       }
00820                       t = r3;
00821                   }
00822                   else if ( type == TYPESPLITLASTARG ) {
00823                       r4 = r1; r5 = t; r7 = oldwork;
00824                       *r1++ = *t + ARGHEAD;
00825                       for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
00826                       j = 0;
00827                       while ( t < r3 ) {
00828                           i = *t;
00829                           if ( t+i >= r3 ) {
00830                               NCOPY(r7,t,i)
00831                               j++;
00832                           }
00833                           else {
00834                               NCOPY(r1,t,i)
00835                           }
00836                       }
00837                       *r4 = r1 - r4;
00838                       if ( j ) {
00839                           if ( ToFast(r4,r4) ) {
00840                               r1 = r4;
00841                               if ( *r1 > -FUNCTION ) r1++;
00842                               r1++;
00843                           }
00844                           r7 = oldwork;
00845                           while ( --j >= 0 ) {
00846                               r4 = r1; i = *r7;
00847                               *r1++ = i+ARGHEAD; *r1++ = 0;
00848                               FILLARG(r1);
00849                               NCOPY(r1,r7,i)
00850                               if ( ToFast(r4,r4) ) {
00851                                   r1 = r4;
00852                                   if ( *r1 > -FUNCTION ) r1++;
00853                                   r1++;
00854                               }
00855                           }
00856                       }
00857                       t = r3;
00858                   }
00859                   else if ( factor == 0 || ( type == TYPESPLITARG2 && *factor == 0 ) ) {
00860                       while ( t < r3 ) {
00861                           r4 = r1;
00862                           *r1++ = *t + ARGHEAD;
00863                           for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
00864                           i = *t;
00865                           while ( --i >= 0 ) *r1++ = *t++;
00866                           if ( ToFast(r4,r4) ) {
```

```
00867                               r1 = r4;
00868                               if ( *r1 > -FUNCTION ) r1++;
00869                               r1++;
00870                           }
00871                       }
00872                   }
00873                   else if ( type == TYPESPLITARG2 ) {
00874 /*
00875                       Here we better put the pattern matcher at work?
00876                       Remember: there are no wildcards.
00877 */
00878                       WORD *oRepFunList = AN.RepFunList;
00879                       WORD *oWildMask = AT.WildMask, *oWildValue = AN.WildValue;
00880                       AN.WildValue = AT.locwildvalue; AT.WildMask = AT.locwildvalue+2;
00881                       AN.NumWild = 0;
00882                       r4 = r1; r5 = t; r7 = oldwork;
00883                       *r1++ = *t + ARGHEAD;
00884                       for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
00885                       j = 0;
00886                       while ( t < r3 ) {
00887                           AN.UseFindOnly = 0; oldwork2 = AT.WorkPointer;
00888                           AN.RepFunList = r1;
00889                           AT.WorkPointer = r1+AN.RepFunNum+2;
00890                           i = *t;
00891                           if ( FindRest(BHEAD t,factor) &&
00892                            ( AN.UsedOtherFind || FindOnce(BHEAD t,factor) ) ) {
00893                               NCOPY(r7,t,i)
00894                               j++;
00895                           }
00896                           else if ( factor[0] == FUNHEAD+1 && factor[1] >= FUNCTION ) {
00897                               WORD *rr1 = t+1, *rr2 = t+i;
00898                               rr2 -= ABS(rr2[-1]);
00899                               while ( rr1 < rr2 ) {
00900                                   if ( *rr1 == factor[1] ) break;
00901                                   rr1 += rr1[1];
00902                               }
00903                               if ( rr1 < rr2 ) {
00904                                   NCOPY(r7,t,i)
00905                                   j++;
00906                               }
00907                               else {
00908                                   NCOPY(r1,t,i)
00909                               }
00910                           }
00911                           else {
00912                               NCOPY(r1,t,i)
00913                           }
00914                           AT.WorkPointer = oldwork2;
00915                       }
00916                       AN.RepFunList = oRepFunList;
00917                       *r4 = r1 - r4;
00918                       if ( j ) {
00919                           if ( ToFast(r4,r4) ) {
00920                               r1 = r4;
00921                               if ( *r1 > -FUNCTION ) r1++;
00922                               r1++;
00923                           }
00924                           r7 = oldwork;
00925                           while ( --j >= 0 ) {
00926                               r4 = r1; i = *r7;
00927                               *r1++ = i+ARGHEAD; *r1++ = 0;
00928                               FILLARG(r1);
00929                               NCOPY(r1,r7,i)
00930                               if ( ToFast(r4,r4) ) {
00931                                   r1 = r4;
00932                                   if ( *r1 > -FUNCTION ) r1++;
00933                                   r1++;
00934                               }
00935                           }
00936                       }
00937                       t = r3;
00938                       AT.WildMask = oWildMask; AN.WildValue = oWildValue;
00939                   }
00940                   else {
00941 /*
00942                       This code deals with splitting off a single term
00943 */
00944                       r4 = r1; r5 = t;
00945                       *r1++ = *t + ARGHEAD;
00946                       for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
00947                       j = 0;
00948                       while ( t < r3 ) {
00949                           r6 = t + *t; r6 -= ABS(r6[-1]);
00950                           if ( (r6 - t) == *factor ) {
00951                               k = *factor - 1;
00952                               for ( ; k > 0; k-- ) {
00953                                   if ( t[k] != factor[k] ) break;
```

```
00954                                   }
00955                                   if ( k <= 0 ) {
00956                                       j = r3 - t; t += *t; continue;
00957                                   }
00958                               }
00959                               else if ( (r6 - t) == 1 && *factor == 0 ) {
00960                                   j = r3 - t; t += *t; continue;
00961                               }
00962                               i = *t;
00963                               NCOPY(r1,t,i)
00964                           }
00965                           *r4 = r1 - r4;
00966                           if ( j ) {
00967                               if ( ToFast(r4,r4) ) {
00968                                   r1 = r4;
00969                                   if ( *r1 > -FUNCTION ) r1++;
00970                                   r1++;
00971                               }
00972                               t = r3 - j;
00973                               r4 = r1;
00974                               *r1++ = *t + ARGHEAD;
00975                               for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
00976                               i = *t;
00977                               while ( --i >= 0 ) *r1++ = *t++;
00978                               if ( ToFast(r4,r4) ) {
00979                                   r1 = r4;
00980                                   if ( *r1 > -FUNCTION ) r1++;
00981                                   r1++;
00982                               }
00983                           }
00984                           t = r3;
00985                       }
00986                   }
00987                   r2[1] = r1 - r2;
00988               }
00989               else {
00990                   r = t + t[1];
00991                   while ( t < r ) *r1++ = *t++;
00992               }
00993           }
00994           r = term + *term;
00995           while ( t < r ) *r1++ = *t++;
00996           m = AT.WorkPointer;
00997           i = m[0] = r1 - m;
00998           t = term;
00999           while ( --i >= 0 ) *t++ = *m++;
01000           if ( AT.WorkPointer < m ) AT.WorkPointer = m;
01001       }
01002   /*
01003       #] SplitArg :
01004       #[ FACTARG :
01005   */
01006       if ( ( type == TYPEFACTARG || type == TYPEFACTARG2 ) &&
01007       AT.pWorkPointer > oldppointer ) {
01008           t = term+1;
01009           r1 = AT.WorkPointer + 1;
01010           lp = oldppointer;
01011           while ( t < rstop ) {
01012               if ( lp < AT.pWorkPointer && AT.pWorkSpace[lp] == t ) {
01013                   v = t;
01014                   m = t + FUNHEAD;
01015                   r = t + t[1];
01016                   r2 = r1; while ( t < m ) *r1++ = *t++;
01017                   while ( m < r ) {
01018                       rr = t = m;
01019                       NEXTARG(m)
01020                       if ( lp >= AT.pWorkPointer || AT.pWorkSpace[lp+1] != t ) {
01021                           if ( *t > 0 ) t[1] = 0;
01022                           while ( t < m ) *r1++ = *t++;
01023                           continue;
01024                       }
01025   /*
01026                       Now we have a nontrivial argument that should be studied.
01027                       Try to find common factors.
01028   */
01029                       lp += 2;
01030                       if ( *t < 0 ) {
01031                           if ( factor && ( *factor == 0 && *t == -SNUMBER ) ) {
01032                               *r1++ = *t++;
01033                               if ( *t == 0 ) *r1++ = *t++;
01034                               else { *r1++ = 1; t++; }
01035                               continue;
01036                           }
01037                           else if ( factor && *factor == 4 && factor[2] == 1 ) {
01038                               if ( *t == -SNUMBER ) {
01039                                   if ( factor[3] < 0 || t[1] >= 0 ) {
01040                                       while ( t < m ) *r1++ = *t++;
```

```
01041                                   }
01042                                   else {
01043                                       *r1++ = -SNUMBER; *r1++ = -1;
01044                                       *r1++ = *t++; *r1++ = -*t++;
01045                                   }
01046                               }
01047                               else {
01048                                   while ( t < m ) *r1++ = *t++;
01049                                   *r1++ = -SNUMBER; *r1++ = 1;
01050                               }
01051                               continue;
01052                           }
01053                           else if ( *t == -MINVECTOR ) {
01054                               if ( AC.OldFactArgFlag == NEWFACTARG ) {
01055                                   *r1++ = -SNUMBER; *r1++ = -1;
01056                                   *r1++ = -VECTOR; t++; *r1++ = *t++;
01057                               }
01058                               else {
01059                                   *r1++ = -VECTOR; t++; *r1++ = *t++;
01060                                   *r1++ = -SNUMBER; *r1++ = -1;
01061                                   *r1++ = -SNUMBER; *r1++ = 1;
01062                               }
01063                               continue;
01064                           }
01065                       }
01066 /*
01067                       Now we have a nontrivial argument
01068 */
01069                       r3 = t + *t;
01070                       t += ARGHEAD;  r5 = t; /* Store starting point */
01071                       /* We have terms from r5 to r3 */
01072                       if ( r5+*r5 == r3 && factor ) { /* One term only */
01073                           if ( *factor == 0 ) {
01074                               GETSTOP(t,r6);
01075                               r9 = r1; *r1++ = 0; *r1++ = 1;
01076                               FILLARG(r1);
01077                               *r1++ = (r6-t)+3; t++;
01078                               while ( t < r6 ) *r1++ = *t++;
01079                               *r1++ = 1; *r1++ = 1; *r1++ = 3;
01080                               *r9 = r1-r9;
01081                               if ( ToFast(r9,r9) ) {
01082                                   if ( *r9 <= -FUNCTION ) r1 = r9+1;
01083                                   else r1 = r9+2;
01084                               }
01085                               t = r3; continue;
01086                           }
01087                           if ( factor[0] == 4 && factor[2] == 1 ) {
01088                               GETSTOP(t,r6);
01089                               r7 = r1; *r1++ = (r6-t)+3+ARGHEAD; *r1++ = 0;
01090                               FILLARG(r1);
01091                               *r1++ = (r6-t)+3; t++;
01092                               while ( t < r6 ) *r1++ = *t++;
01093                               *r1++ = 1; *r1++ = 1; *r1++ = 3;
01094                               if ( ToFast(r7,r7) ) {
01095                                   if ( *r7 <= -FUNCTION ) r1 = r7+1;
01096                                   else r1 = r7+2;
01097                               }
01098                               if ( r3[-1] < 0 && factor[3] > 0 ) {
01099                                   *r1++ = -SNUMBER; *r1++ = -1;
01100                                   if ( r3[-1] == -3 && r3[-2] == 1
01101                                   && ( r3[-3] & MAXPOSITIVE ) == r3[-3] ) {
01102                                       *r1++ = -SNUMBER; *r1++ = r3[-3];
01103                                   }
01104                                   else {
01105                                       *r1++ = (r3-r6)+1+ARGHEAD;
01106                                       *r1++ = 0;
01107                                       FILLARG(r1);
01108                                       *r1++ = (r3-r6+1);
01109                                       while ( t < r3 ) *r1++ = *t++;
01110                                       r1[-1] = -r1[-1];
01111                                   }
01112                               }
01113                               else {
01114                                   if ( ( r3[-1] == -3 || r3[-1] == 3 )
01115                                   && r3[-2] == 1
01116                                   && ( r3[-3] & MAXPOSITIVE ) == r3[-3] ) {
01117                                       *r1++ = -SNUMBER; *r1++ = r3[-3];
01118                                       if ( r3[-1] < 0 ) r1[-1] = - r1[-1];
01119                                   }
01120                                   else {
01121                                       *r1++ = (r3-r6)+1+ARGHEAD;
01122                                       *r1++ = 0;
01123                                       FILLARG(r1);
01124                                       *r1++ = (r3-r6+1);
01125                                       while ( t < r3 ) *r1++ = *t++;
01126                                   }
01127                               }
```

```
01128                           t = r3; continue;
01129                       }
01130                   }
01131 /*
01132                   Now we take the first term and look for its pieces
01133                   inside the other terms.
01134
01135                   It is at this point that a more general factorization
01136                   routine could take over (allowing for writing the output
01137                   properly of course).
01138 */
01139                   if ( AC.OldFactArgFlag == NEWFACTARG ) {
01140                   if ( factor == 0 ) {
01141                       WORD *oldworkpointer2 = AT.WorkPointer;
01142                       AT.WorkPointer = r1 + AM.MaxTer+FUNHEAD;
01143                       if ( ArgFactorize(BHEAD t-ARGHEAD,r1) < 0 ) {
01144                           MesCall("ExecArg");
01145                           return(-1);
01146                       }
01147                       AT.WorkPointer = oldworkpointer2;
01148                       t = r3;
01149                       while ( *r1 ) { NEXTARG(r1) }
01150                   }
01151                   else {
01152                       rnext = t + *t;
01153                       GETSTOP(t,r6);
01154                       t++;
01155                       t = r5; pow = 1;
01156                       while ( t < r3 ) {
01157                           t += *t; if ( t[-1] > 0 ) { pow = 0; break; }
01158                       }
01159 /*
01160                       We have to add here the code for computing the GCD
01161                       and to divide it out.
01162
01163               #[ Numerical factor :
01164 */
01165                       t = r5;
01166                       EAscrat = (UWORD *)(TermMalloc("execarg"));
01167                       if ( t + *t == r3 ) {
01168                           if ( factor == 0 || *factor > 2 ) {
01169                             if ( pow > 0 ) {
01170                               *r1++ = -SNUMBER; *r1++ = -1;
01171                               t = r5;
01172                               while ( t < r3 ) {
01173                                   t += *t; t[-1] = -t[-1];
01174                               }
01175                             }
01176                             t = rr; *r1++ = *t++; *r1++ = 1; t++;
01177                             COPYARG(r1,t);
01178                             while ( t < m ) *r1++ = *t++;
01179                           }
01180                       }
01181                       else {
01182                       GETSTOP(t,r6);
01183                       ngcd = t[t[0]-1];
01184                       i = abs(ngcd)-1;
01185                       while ( --i >= 0 ) EAscrat[i] = r6[i];
01186                       t += *t;
01187                       while ( t < r3 ) {
01188                           GETSTOP(t,r6);
01189                           i = t[t[0]-1];
01190                           if ( AccumGCD(BHEAD EAscrat,&ngcd,(UWORD *)r6,i) ) goto execargerr;
01191                           if ( ngcd == 3 && EAscrat[0] == 1 && EAscrat[1] == 1 ) break;
01192                           t += *t;
01193                       }
01194 /*
01195                       if ( ngcd != 3 || EAscrat[0] != 1 || EAscrat[1] != 1 )
01196 */
01197                       {
01198                           if ( pow ) ngcd = -ngcd;
01199                           t = r5; r9 = r1; *r1++ = t[-ARGHEAD]; *r1++ = 1;
01200                           FILLARG(r1); ngcd = REDLENG(ngcd);
01201                           while ( t < r3 ) {
01202                               GETSTOP(t,r6);
01203                               r7 = t; r8 = r1;
01204                               while ( r7 < r6) *r1++ = *r7++;
01205                               t += *t;
01206                               i = REDLENG(t[-1]);
01207                               if ( DivRat(BHEAD (UWORD *)r6,i,EAscrat,ngcd,(UWORD *)r1,&nq) ) goto
      execargerr;
01208                               nq = INCLENG(nq);
01209                               i = ABS(nq)-1;
01210                               r1 += i; *r1++ = nq; *r8 = r1-r8;
01211                           }
01212                           *r9 = r1-r9;
01213                           ngcd = INCLENG(ngcd);
```

```
01214                           i = ABS(ngcd)-1;
01215                           if ( factor && *factor == 0 ) {}
01216                           else if ( ( factor && factor[0] == 4 && factor[2] == 1
01217                           && factor[3] == -3 ) || pow == 0 ) {
01218                               r9 = r1; *r1++ = ARGHEAD+2+i; *r1++ = 0;
01219                               FILLARG(r1); *r1++ = i+2;
01220                               for ( j = 0; j < i; j++ ) *r1++ = EAscrat[j];
01221                               *r1++ = ngcd;
01222                               if ( ToFast(r9,r9) ) r1 = r9+2;
01223                           }
01224                           else if ( factor && factor[0] == 4 && factor[2] == 1
01225                           && factor[3] > 0 && pow ) {
01226                               if ( ngcd < 0 ) ngcd = -ngcd;
01227                               *r1++ = -SNUMBER; *r1++ = -1;
01228                               r9 = r1; *r1++ = ARGHEAD+2+i; *r1++ = 0;
01229                               FILLARG(r1); *r1++ = i+2;
01230                               for ( j = 0; j < i; j++ ) *r1++ = EAscrat[j];
01231                               *r1++ = ngcd;
01232                               if ( ToFast(r9,r9) ) r1 = r9+2;
01233                           }
01234                           else {
01235                               if ( ngcd < 0 ) ngcd = -ngcd;
01236                               if ( pow ) { *r1++ = -SNUMBER; *r1++ = -1; }
01237                               if ( ngcd != 3 || EAscrat[0] != 1 || EAscrat[1] != 1 ) {
01238                                   r9 = r1; *r1++ = ARGHEAD+2+i; *r1++ = 0;
01239                                   FILLARG(r1); *r1++ = i+2;
01240                                   for ( j = 0; j < i; j++ ) *r1++ = EAscrat[j];
01241                                   *r1++ = ngcd;
01242                                   if ( ToFast(r9,r9) ) r1 = r9+2;
01243                               }
01244                           }
01245                       }
01246 /*
01247             #] Numerical factor :
01248                       else {
01249 onetermnew:;
01250
01251                           if ( factor == 0 || *factor > 2 ) {
01252                             if ( pow > 0 ) {
01253                               *r1++ = -SNUMBER; *r1++ = -1;
01254                               t = r5;
01255                               while ( t < r3 ) {
01256                                   t += *t; t[-1] = -t[-1];
01257                               }
01258                             }
01259                             t = rr; *r1++ = *t++; *r1++ = 1; t++;
01260                             COPYARG(r1,t);
01261                             while ( t < m ) *r1++ = *t++;
01262                           }
01263                       }
01264 onetermnew:;
01265 */
01266                       }
01267                       TermFree(EAscrat,"execarg");
01268                   }
01269                   }
01270                   else {  /* AC.OldFactArgFlag is ON */
01271                   {
01272                   WORD *mnext, ncom;
01273                   rnext = t + *t;
01274                   GETSTOP(t,r6);
01275                   t++;
01276                   if ( factor == 0 ) {
01277                     while ( t < r6 ) {
01278 /*
01279             #[ SYMBOL :
01280 */
01281                       if ( *t == SYMBOL ) {
01282                           r7 = t; r8 = t + t[1]; t += 2;
01283                           while ( t < r8 ) {
01284                               pow = t[1];
01285                               mm = rnext;
01286                               while ( mm < r3 ) {
01287                                   mnext = mm + *mm;
01288                                   GETSTOP(mm,mstop); mm++;
01289                                   while ( mm < mstop ) {
01290                                       if ( *mm != SYMBOL ) mm += mm[1];
01291                                       else break;
01292                                   }
01293                                   if ( *mm == SYMBOL ) {
01294                                       mstop = mm + mm[1]; mm += 2;
01295                                       while ( *mm != *t && mm < mstop ) mm += 2;
01296                                       if ( mm >= mstop ) pow = 0;
01297                                       else if ( pow > 0 && mm[1] > 0 ) {
01298                                           if ( mm[1] < pow ) pow = mm[1];
01299                                       }
01300                                       else if ( pow < 0 && mm[1] < 0 ) {
```

```
01301                                       if ( mm[1] > pow ) pow = mm[1];
01302                                   }
01303                                   else pow = 0;
01304                               }
01305                               else pow = 0;
01306                               if ( pow == 0 ) break;
01307                               mm = mnext;
01308                           }
01309                           if ( pow == 0 ) { t += 2; continue; }
01310 /*
01311                           We have a factor
01312 */
01313                           action = 1; i = pow;
01314                           if ( i > 0 ) {
01315                               while ( --i >= 0 ) {
01316                                   *r1++ = -SYMBOL;
01317                                   *r1++ = *t;
01318                               }
01319                           }
01320                           else {
01321                               while ( i++ < 0 ) {
01322                                   *r1++ = 8 + ARGHEAD;
01323                                   for ( j = 1; j < ARGHEAD; j++ ) *r1++ = 0;
01324                                   *r1++ = 8; *r1++ = SYMBOL;
01325                                   *r1++ = 4; *r1++ = *t; *r1++ = -1;
01326                                   *r1++ = 1; *r1++ = 1; *r1++ = 3;
01327                               }
01328                           }
01329 /*
01330                           Now we have to remove the symbols
01331 */
01332                           t[1] -= pow;
01333                           mm = rnext;
01334                           while ( mm < r3 ) {
01335                               mnext = mm + *mm;
01336                               GETSTOP(mm,mstop); mm++;
01337                               while ( mm < mstop ) {
01338                                   if ( *mm != SYMBOL ) mm += mm[1];
01339                                   else break;
01340                               }
01341                               mstop = mm + mm[1]; mm += 2;
01342                               while ( mm < mstop && *mm != *t ) mm += 2;
01343                               mm[1] -= pow;
01344                               mm = mnext;
01345                           }
01346                           t += 2;
01347                       }
01348                   }
01349 /*
01350               #] SYMBOL :
01351               #[ DOTPRODUCT :
01352 */
01353                   else if ( *t == DOTPRODUCT ) {
01354                       r7 = t; r8 = t + t[1]; t += 2;
01355                       while ( t < r8 ) {
01356                           pow = t[2];
01357                           mm = rnext;
01358                           while ( mm < r3 ) {
01359                               mnext = mm + *mm;
01360                               GETSTOP(mm,mstop); mm++;
01361                               while ( mm < mstop ) {
01362                                   if ( *mm != DOTPRODUCT ) mm += mm[1];
01363                                   else break;
01364                               }
01365                               if ( *mm == DOTPRODUCT ) {
01366                                   mstop = mm + mm[1]; mm += 2;
01367                                   while ( ( *mm != *t || mm[1] != t[1] )
01368                                        && mm < mstop ) mm += 3;
01369                                   if ( mm >= mstop ) pow = 0;
01370                                   else if ( pow > 0 && mm[2] > 0 ) {
01371                                       if ( mm[2] < pow ) pow = mm[2];
01372                                   }
01373                                   else if ( pow < 0 && mm[2] < 0 ) {
01374                                       if ( mm[2] > pow ) pow = mm[2];
01375                                   }
01376                                   else pow = 0;
01377                               }
01378                               else pow = 0;
01379                               if ( pow == 0 ) break;
01380                               mm = mnext;
01381                           }
01382                           if ( pow == 0 ) { t += 3; continue; }
01383 /*
01384                           We have a factor
01385 */
01386                           action = 1; i = pow;
01387                           if ( i > 0 ) {
```

```
01388                                   while ( --i >= 0 ) {
01389                                       *r1++ = 9 + ARGHEAD;
01390                                       for ( j = 1; j < ARGHEAD; j++ ) *r1++ = 0;
01391                                       *r1++ = 9; *r1++ = DOTPRODUCT;
01392                                       *r1++ = 5; *r1++ = *t; *r1++ = t[1]; *r1++ = 1;
01393                                       *r1++ = 1; *r1++ = 1; *r1++ = 3;
01394                                   }
01395                               }
01396                               else {
01397                                   while ( i++ < 0 ) {
01398                                       *r1++ = 9 + ARGHEAD;
01399                                       for ( j = 1; j < ARGHEAD; j++ ) *r1++ = 0;
01400                                       *r1++ = 9; *r1++ = DOTPRODUCT;
01401                                       *r1++ = 5; *r1++ = *t; *r1++ = t[1]; *r1++ = -1;
01402                                       *r1++ = 1; *r1++ = 1; *r1++ = 3;
01403                                   }
01404                               }
01405 /*
01406                               Now we have to remove the dotproducts
01407 */
01408                               t[2] -= pow;
01409                               mm = rnext;
01410                               while ( mm < r3 ) {
01411                                   mnext = mm + *mm;
01412                                   GETSTOP(mm,mstop); mm++;
01413                                   while ( mm < mstop ) {
01414                                       if ( *mm != DOTPRODUCT ) mm += mm[1];
01415                                       else break;
01416                                   }
01417                                   mstop = mm + mm[1]; mm += 2;
01418                                   while ( mm < mstop && ( *mm != *t
01419                                       || mm[1] != t[1] ) ) mm += 3;
01420                                   mm[2] -= pow;
01421                                   mm = mnext;
01422                               }
01423                               t += 3;
01424                           }
01425                       }
01426 /*
01427           #] DOTPRODUCT :
01428           #[ DELTA/VECTOR :
01429 */
01430                       else if ( *t == DELTA || *t == VECTOR ) {
01431                           r7 = t; r8 = t + t[1]; t += 2;
01432                           while ( t < r8 ) {
01433                               mm = rnext;
01434                               pow = 1;
01435                               while ( mm < r3 ) {
01436                                   mnext = mm + *mm;
01437                                   GETSTOP(mm,mstop); mm++;
01438                                   while ( mm < mstop ) {
01439                                       if ( *mm != *r7 ) mm += mm[1];
01440                                       else break;
01441                                   }
01442                                   if ( *mm == *r7 ) {
01443                                       mstop = mm + mm[1]; mm += 2;
01444                                       while ( ( *mm != *t || mm[1] != t[1] )
01445                                           && mm < mstop ) mm += 2;
01446                                       if ( mm >= mstop ) pow = 0;
01447                                   }
01448                                   else pow = 0;
01449                                   if ( pow == 0 ) break;
01450                                   mm = mnext;
01451                               }
01452                               if ( pow == 0 ) { t += 2; continue; }
01453 /*
01454                               We have a factor
01455 */
01456                               action = 1;
01457                               *r1++ = 8 + ARGHEAD;
01458                               for ( j = 1; j < ARGHEAD; j++ ) *r1++ = 0;
01459                               *r1++ = 8; *r1++ = *r7;
01460                               *r1++ = 4; *r1++ = *t; *r1++ = t[1];
01461                               *r1++ = 1; *r1++ = 1; *r1++ = 3;
01462 /*
01463                               Now we have to remove the delta's/vectors
01464 */
01465                               mm = rnext;
01466                               while ( mm < r3 ) {
01467                                   mnext = mm + *mm;
01468                                   GETSTOP(mm,mstop); mm++;
01469                                   while ( mm < mstop ) {
01470                                       if ( *mm != *r7 ) mm += mm[1];
01471                                       else break;
01472                                   }
01473                                   mstop = mm + mm[1]; mm += 2;
01474                                   while ( mm < mstop && (
```

```
01475                                   *mm != *t || mm[1] != t[1] ) ) mm += 2;
01476                                *mm = mm[1] = NOINDEX;
01477                                mm = mnext;
01478                            }
01479                            *t = t[1] = NOINDEX;
01480                            t += 2;
01481                        }
01482                    }
01483 /*
01484            #] DELTA/VECTOR :
01485            #[ INDEX :
01486 */
01487                    else if ( *t == INDEX ) {
01488                        r7 = t; r8 = t + t[1]; t += 2;
01489                        while ( t < r8 ) {
01490                            mm = rnext;
01491                            pow = 1;
01492                            while ( mm < r3 ) {
01493                                mnext = mm + *mm;
01494                                GETSTOP(mm,mstop); mm++;
01495                                while ( mm < mstop ) {
01496                                    if ( *mm != *r7 ) mm += mm[1];
01497                                    else break;
01498                                }
01499                                if ( *mm == *r7 ) {
01500                                    mstop = mm + mm[1]; mm += 2;
01501                                    while ( *mm != *t
01502                                        && mm < mstop ) mm++;
01503                                    if ( mm >= mstop ) pow = 0;
01504                                }
01505                                else pow = 0;
01506                                if ( pow == 0 ) break;
01507                                mm = mnext;
01508                            }
01509                            if ( pow == 0 ) { t++; continue; }
01510 /*
01511                            We have a factor
01512 */
01513                            action = 1;
01514 /*
01515                            The next looks like an error.
01516                            We should have here a VECTOR or INDEX like object
01517
01518                            *r1++ = 7 + ARGHEAD;
01519                            for ( j = 1; j < ARGHEAD; j++ ) *r1++ = 0;
01520                            *r1++ = 7; *r1++ = *r7;
01521                            *r1++ = 3; *r1++ = *t;
01522                            *r1++ = 1; *r1++ = 1; *r1++ = 3;
01523
01524                            Replace this by:  (11-apr-2007)
01525 */
01526                            if ( *t < 0 ) { *r1++ = -VECTOR; }
01527                            else { *r1++ = -INDEX; }
01528                            *r1++ = *t;
01529 /*
01530                            Now we have to remove the index
01531 */
01532                            *t = NOINDEX;
01533                            mm = rnext;
01534                            while ( mm < r3 ) {
01535                                mnext = mm + *mm;
01536                                GETSTOP(mm,mstop); mm++;
01537                                while ( mm < mstop ) {
01538                                    if ( *mm != *r7 ) mm += mm[1];
01539                                    else break;
01540                                }
01541                                mstop = mm + mm[1]; mm += 2;
01542                                while ( mm < mstop &&
01543                                  *mm != *t ) mm += 1;
01544                                *mm = NOINDEX;
01545                                mm = mnext;
01546                            }
01547                            t += 1;
01548                        }
01549                    }
01550 /*
01551            #] INDEX :
01552            #[ FUNCTION :
01553 */
01554                    else if ( *t >= FUNCTION ) {
01555 /*
01556                        In the next code we should actually look inside
01557                        the DENOMINATOR or EXPONENT for noncommuting objects
01558 */
01559                        if ( *t >= FUNCTION &&
01560                            functions[*t-FUNCTION].commute == 0 ) ncom = 0;
01561                        else ncom = 1;
```

```
01562                               if ( ncom ) {
01563                                   mm = r5 + 1;
01564                                   while ( mm < t && ( *mm == DUMMYFUN
01565                                   || *mm == DUMMYTEN ) ) mm += mm[1];
01566                                   if ( mm < t ) { t += t[1]; continue; }
01567                               }
01568                               mm = rnext; pow = 1;
01569                               while ( mm < r3 ) {
01570                                   mnext = mm + *mm;
01571                                   GETSTOP(mm,mstop); mm++;
01572                                   while ( mm < mstop ) {
01573                                       if ( *mm == *t && mm[1] == t[1] ) {
01574                                           for ( i = 2; i < t[1]; i++ ) {
01575                                               if ( mm[i] != t[i] ) break;
01576                                           }
01577                                           if ( i >= t[1] )
01578                                               { mm += mm[1]; goto nextmterm; }
01579                                       }
01580                                       if ( ncom && *mm != DUMMYFUN && *mm != DUMMYTEN )
01581                                           { pow = 0; break; }
01582                                       mm += mm[1];
01583                                   }
01584                                   if ( mm >= mstop ) pow = 0;
01585                                   if ( pow == 0 ) break;
01586 nextmterm:                        mm = mnext;
01587                               }
01588                               if ( pow == 0 ) { t += t[1]; continue; }
01589 /*
01590                               Copy the function
01591 */
01592                               action = 1;
01593                               *r1++ = t[1] + 4 + ARGHEAD;
01594                               for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
01595                               *r1++ = t[1] + 4;
01596                               for ( i = 0; i < t[1]; i++ ) *r1++ = t[i];
01597                               *r1++ = 1; *r1++ = 1; *r1++ = 3;
01598 /*
01599                               Now we have to take out the functions
01600 */
01601                               mm = rnext;
01602                               while ( mm < r3 ) {
01603                                   mnext = mm + *mm;
01604                                   GETSTOP(mm,mstop); mm++;
01605                                   while ( mm < mstop ) {
01606                                       if ( *mm == *t && mm[1] == t[1] ) {
01607                                           for ( i = 2; i < t[1]; i++ ) {
01608                                               if ( mm[i] != t[i] ) break;
01609                                           }
01610                                           if ( i >= t[1] ) {
01611                                               if ( functions[*t-FUNCTION].spec > 0 )
01612                                                   *mm = DUMMYTEN;
01613                                               else
01614                                                   *mm = DUMMYFUN;
01615                                               mm += mm[1];
01616                                               goto nextterm;
01617                                           }
01618                                       }
01619                                       mm += mm[1];
01620                                   }
01621 nextterm:                        mm = mnext;
01622                               }
01623                               if ( functions[*t-FUNCTION].spec > 0 )
01624                                       *t = DUMMYTEN;
01625                               else
01626                                       *t = DUMMYFUN;
01627                               action = 1;
01628                               v[2] = DIRTYFLAG;
01629                               t += t[1];
01630                           }
01631 /*
01632             #] FUNCTION :
01633 */
01634                           else {
01635                               t += t[1];
01636                           }
01637                       }
01638                   }
01639                   t = r5; pow = 1;
01640                   while ( t < r3 ) {
01641                       t += *t; if ( t[-1] > 0 ) { pow = 0; break; }
01642                   }
01643 /*
01644                   We have to add here the code for computing the GCD
01645                   and to divide it out.
01646 */
01647 /*
01648             #[ Numerical factor :
```

```
01649 */
01650                     t = r5;
01651                     EAscrat = (UWORD *)(TermMalloc("execarg"));
01652                     if ( t + *t == r3 ) goto oneterm;
01653                     GETSTOP(t,r6);
01654                     ngcd = t[t[0]-1];
01655                     i = abs(ngcd)-1;
01656                     while ( --i >= 0 ) EAscrat[i] = r6[i];
01657                     t += *t;
01658                     while ( t < r3 ) {
01659                         GETSTOP(t,r6);
01660                         i = t[t[0]-1];
01661                         if ( AccumGCD(BHEAD EAscrat,&ngcd,(UWORD *)r6,i) ) goto execargerr;
01662                         if ( ngcd == 3 && EAscrat[0] == 1 && EAscrat[1] == 1 ) break;
01663                         t += *t;
01664                     }
01665                     if ( ngcd != 3 || EAscrat[0] != 1 || EAscrat[1] != 1 ) {
01666                         if ( pow ) ngcd = -ngcd;
01667                         t = r5; r9 = r1; *r1++ = t[-ARGHEAD]; *r1++ = 1;
01668                         FILLARG(r1); ngcd = REDLENG(ngcd);
01669                         while ( t < r3 ) {
01670                             GETSTOP(t,r6);
01671                             r7 = t; r8 = r1;
01672                             while ( r7 < r6) *r1++ = *r7++;
01673                             t += *t;
01674                             i = REDLENG(t[-1]);
01675                             if ( DivRat(BHEAD (UWORD *)r6,i,EAscrat,ngcd,(UWORD *)r1,&nq) ) goto
    execargerr;
01676                             nq = INCLENG(nq);
01677                             i = ABS(nq)-1;
01678                             r1 += i; *r1++ = nq; *r8 = r1-r8;
01679                         }
01680                         *r9 = r1-r9;
01681                         ngcd = INCLENG(ngcd);
01682                         i = ABS(ngcd)-1;
01683                         if ( factor && *factor == 0 ) {}
01684                         else if ( ( factor && factor[0] == 4 && factor[2] == 1
01685                         && factor[3] == -3 ) || pow == 0 ) {
01686                             r9 = r1; *r1++ = ARGHEAD+2+i; *r1++ = 0;
01687                             FILLARG(r1); *r1++ = i+2;
01688                             for ( j = 0; j < i; j++ ) *r1++ = EAscrat[j];
01689                             *r1++ = ngcd;
01690                             if ( ToFast(r9,r9) ) r1 = r9+2;
01691                         }
01692                         else if ( factor && factor[0] == 4 && factor[2] == 1
01693                         && factor[3] > 0 && pow ) {
01694                             if ( ngcd < 0 ) ngcd = -ngcd;
01695                             *r1++ = -SNUMBER; *r1++ = -1;
01696                             r9 = r1; *r1++ = ARGHEAD+2+i; *r1++ = 0;
01697                             FILLARG(r1); *r1++ = i+2;
01698                             for ( j = 0; j < i; j++ ) *r1++ = EAscrat[j];
01699                             *r1++ = ngcd;
01700                             if ( ToFast(r9,r9) ) r1 = r9+2;
01701                         }
01702                         else {
01703                             if ( ngcd < 0 ) ngcd = -ngcd;
01704                             if ( pow ) { *r1++ = -SNUMBER; *r1++ = -1; }
01705                             if ( ngcd != 3 || EAscrat[0] != 1 || EAscrat[1] != 1 ) {
01706                                 r9 = r1; *r1++ = ARGHEAD+2+i; *r1++ = 0;
01707                                 FILLARG(r1); *r1++ = i+2;
01708                                 for ( j = 0; j < i; j++ ) *r1++ = EAscrat[j];
01709                                 *r1++ = ngcd;
01710                                 if ( ToFast(r9,r9) ) r1 = r9+2;
01711                             }
01712                         }
01713                     }
01714 /*
01715             #] Numerical factor :
01716 */
01717                     else {
01718 oneterm:;
01719                         if ( factor == 0 || *factor > 2 ) {
01720                         if ( pow > 0 ) {
01721                             *r1++ = -SNUMBER; *r1++ = -1;
01722                             t = r5;
01723                             while ( t < r3 ) {
01724                                 t += *t; t[-1] = -t[-1];
01725                             }
01726                         }
01727                         t = rr; *r1++ = *t++; *r1++ = 1; t++;
01728                         COPYARG(r1,t);
01729                         while ( t < m ) *r1++ = *t++;
01730                         }
01731                     }
01732                     TermFree(EAscrat,"execarg");
01733                     }
01734                     } /* AC.OldFactArgFlag */
```

```
01735                 }
01736 /* r1 is fout in ons voorbeeld. */
01737                 r2[1] = r1 - r2;
01738                 action = 1;
01739                 v[2] = DIRTYFLAG;
01740             }
01741             else {
01742                 r = t + t[1];
01743                 while ( t < r ) *r1++ = *t++;
01744             }
01745         }
01746         r = term + *term;
01747         while ( t < r ) *r1++ = *t++;
01748         m = AT.WorkPointer;
01749         i = m[0] = r1 - m;
01750         t = term;
01751         while ( --i >= 0 ) *t++ = *m++;
01752         if ( AT.WorkPointer < t ) AT.WorkPointer = t;
01753     }
01754 /*
01755     #] FACTARG :
01756 */
01757     AR.Cnumlhs = oldnumlhs;
01758     if ( action && Normalize(BHEAD term) ) goto execargerr;
01759     AT.WorkPointer = oldwork;
01760     if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
01761     AT.pWorkPointer = oldppointer;
01762     return(action);
01763 execargerr:
01764     AT.WorkPointer = oldwork;
01765     AT.pWorkPointer = oldppointer;
01766     MLOCK(ErrorMessageLock);
01767     MesCall("execarg");
01768     MUNLOCK(ErrorMessageLock);
01769     return(-1);
01770 }
01771
01772 /*
01773     #] execarg :
01774     #[ execterm :
01775 */
01776
01777 int execterm(PHEAD WORD *term, WORD level)
01778 {
01779     GETBIDENTITY
01780     CBUF *C = cbuf+AM.rbufnum;
01781     WORD oldnumlhs = AR.Cnumlhs;
01782     WORD maxisat = C->lhs[level][2];
01783     WORD *buffer1 = 0;
01784     WORD *oldworkpointer = AT.WorkPointer;
01785     WORD *t1, i;
01786     WORD olddeferflag = AR.DeferFlag, tryterm = 0;
01787     AR.DeferFlag = 0;
01788     do {
01789         AR.Cnumlhs = C->lhs[level][3];
01790         NewSort(BHEAD0);
01791 /*
01792         Normally for function arguments we do not use PolyFun/PolyRatFun.
01793         Hence NewSort sets the corresponding variables to zero.
01794         Here we overwrite that.
01795 */
01796         AN.FunSorts[AR.sLevel]->PolyFlag = ( AR.PolyFun != 0 ) ? AR.PolyFunType: 0;
01797         if ( AR.PolyFun == 0 ) { AN.FunSorts[AR.sLevel]->PolyFlag = 0; }
01798         else if ( AR.PolyFunType == 1 ) { AN.FunSorts[AR.sLevel]->PolyFlag = 1; }
01799         else if ( AR.PolyFunType == 2 ) {
01800             if ( AR.PolyFunExp == 2 ) AN.FunSorts[AR.sLevel]->PolyFlag = 1;
01801             else                      AN.FunSorts[AR.sLevel]->PolyFlag = 2;
01802         }
01803         if ( buffer1 ) {
01804             term = buffer1;
01805             while ( *term ) {
01806                 t1 = oldworkpointer;
01807                 i = *term; while ( --i >= 0 ) *t1++ = *term++;
01808                 AT.WorkPointer = t1;
01809                 if ( Generator(BHEAD oldworkpointer,level) ) goto exectermerr;
01810             }
01811         }
01812         else {
01813             if ( Generator(BHEAD term,level) ) goto exectermerr;
01814         }
01815         if ( buffer1 ) {
01816             if ( tryterm ) { TermFree(buffer1,"buffer in sort statement"); tryterm = 0; }
01817             else { M_free((void *)buffer1,"buffer in sort statement"); }
01818             buffer1 = 0;
01819         }
01820         AN.tryterm = 1;
01821         if ( EndSort(BHEAD (WORD *)((void *)(&buffer1)),2) < 0 ) goto exectermerr;
```

```
01822         tryterm = AN.tryterm; AN.tryterm = 0;
01823         level = AR.Cnumlhs;
01824     } while ( AR.Cnumlhs < maxisat );
01825     AR.Cnumlhs = oldnumlhs;
01826     AR.DeferFlag = olddeferflag;
01827     term = buffer1;
01828     while ( *term ) {
01829         t1 = oldworkpointer;
01830         i = *term; while ( --i >= 0 ) *t1++ = *term++;
01831         AT.WorkPointer = t1;
01832         if ( Generator(BHEAD oldworkpointer,level) ) goto exectermerr;
01833     }
01834     if ( tryterm ) { TermFree(buffer1,"buffer in term statement"); tryterm = 0; }
01835     else { M_free(buffer1,"buffer in term statement"); }
01836     buffer1 = 0;
01837     AT.WorkPointer = oldworkpointer;
01838     return(0);
01839 exectermerr:
01840     AT.WorkPointer = oldworkpointer;
01841     AR.DeferFlag = olddeferflag;
01842     MLOCK(ErrorMessageLock);
01843     MesCall("execterm");
01844     MUNLOCK(ErrorMessageLock);
01845     return(-1);
01846 }
01847
01848 /*
01849     #] execterm :
01850     #[ ArgumentImplode :
01851 */
01852
01853 int ArgumentImplode(PHEAD WORD *term, WORD *thelist)
01854 {
01855     GETBIDENTITY
01856     WORD *liststart, *liststop, *inlist;
01857     WORD *w, *t, *tend, *tstop, *tt, *ttstop, *ttt, ncount, i;
01858     int action = 0;
01859     liststop = thelist + thelist[1];
01860     liststart = thelist + 2;
01861     t = term;
01862     tend = t + *t;
01863     tstop = tend - ABS(tend[-1]);
01864     t++;
01865     while ( t < tstop ) {
01866         if ( *t >= FUNCTION ) {
01867             inlist = liststart;
01868             while ( inlist < liststop && *inlist != *t ) inlist += inlist[1];
01869             if ( inlist < liststop ) {
01870                 tt = t; ttstop = t + t[1]; w = AT.WorkPointer;
01871                 for ( i = 0; i < FUNHEAD; i++ ) *w++ = *tt++;
01872                 while ( tt < ttstop ) {
01873                     ncount = 0;
01874                     if ( *tt == -SNUMBER && tt[1] == 0 ) {
01875                         ncount = 1; ttt = tt; tt += 2;
01876                         while ( tt < ttstop && *tt == -SNUMBER && tt[1] == 0 ) {
01877                             ncount++; tt += 2;
01878                         }
01879                     }
01880                     if ( ncount > 0 ) {
01881                         if ( tt < ttstop && *tt == -SNUMBER && ( tt[1] == 1 || tt[1] == -1 ) ) {
01882                             *w++ = -SNUMBER;
01883                             *w++ = (ncount+1) * tt[1];
01884                             tt += 2;
01885                             action = 1;
01886                         }
01887                         else if ( ( tt[0] == tt[ARGHEAD] + ARGHEAD )
01888                         && ( ABS(tt[tt[0]-1]) == 3 )
01889                         && ( tt[tt[0]-2] == 1 )
01890                         && ( tt[tt[0]-3] == 1 ) ) { /* Single term with coef +/- 1 */
01891                             i = *tt; NCOPY(w,tt,i)
01892                             w[-3] = ncount+1;
01893                             action = 1;
01894                         }
01895                         else if ( *tt == -SYMBOL ) {
01896                             *w++ = ARGHEAD+8;
01897                             *w++ = 0;
01898                             FILLARG(w)
01899                             *w++ = 8;
01900                             *w++ = SYMBOL;
01901                             *w++ = tt[1];
01902                             *w++ = 1;
01903                             *w++ = ncount+1; *w++ = 1; *w++ = 3;
01904                             tt += 2;
01905                             action = 1;
01906                         }
01907                         else if ( *tt <= -FUNCTION ) {
01908                             *w++ = ARGHEAD+FUNHEAD+4;
```

```
01909                               *w++ = 0;
01910                               FILLARG(w)
01911                               *w++ = -*tt++;
01912                               *w++ = FUNHEAD+4;
01913                               FILLFUN(w)
01914                               *w++ = ncount+1; *w++ = 1; *w++ = 3;
01915                               action = 1;
01916                           }
01917                           else {
01918                               while ( ttt < tt ) *w++ = *ttt++;
01919                               if ( tt < ttstop && *tt == -SNUMBER ) {
01920                                   *w++ = *tt++; *w++ = *tt++;
01921                               }
01922                           }
01923                       }
01924                       else if ( *tt <= -FUNCTION ) {
01925                           *w++ = *tt++;
01926                       }
01927                       else if ( *tt < 0 ) {
01928                           *w++ = *tt++;
01929                           *w++ = *tt++;
01930                       }
01931                       else {
01932                           i = *tt; NCOPY(w,tt,i)
01933                       }
01934                   }
01935                   AT.WorkPointer[1] = w - AT.WorkPointer;
01936                   while ( tt < tend ) *w++ = *tt++;
01937                   ttt = AT.WorkPointer; tt = t;
01938                   while ( ttt < w ) *tt++ = *ttt++;
01939                   term[0] = tt - term;
01940                   AT.WorkPointer = tt;
01941                   tend = tt; tstop = tt - ABS(tt[-1]);
01942               }
01943           }
01944           t += t[1];
01945       }
01946       if ( action ) {
01947           if ( Normalize(BHEAD term) ) return(-1);
01948       }
01949       return(0);
01950   }
01951
01952   /*
01953       #] ArgumentImplode :
01954       #[ ArgumentExplode :
01955   */
01956
01957   int ArgumentExplode(PHEAD WORD *term, WORD *thelist)
01958   {
01959       GETBIDENTITY
01960       WORD *liststart, *liststop, *inlist, *old;
01961       WORD *w, *t, *tend, *tstop, *tt, *ttstop, *ttt, ncount, i;
01962       int action = 0;
01963       LONG x;
01964       liststop = thelist + thelist[1];
01965       liststart = thelist + 2;
01966       t = term;
01967       tend = t + *t;
01968       tstop = tend - ABS(tend[-1]);
01969       t++;
01970       while ( t < tstop ) {
01971           if ( *t >= FUNCTION ) {
01972               inlist = liststart;
01973               while ( inlist < liststop && *inlist != *t ) inlist += inlist[1];
01974               if ( inlist < liststop ) {
01975                   tt = t; ttstop = t + t[1]; w = AT.WorkPointer;
01976                   for ( i = 0; i < FUNHEAD; i++ ) *w++ = *tt++;
01977                   while ( tt < ttstop ) {
01978                       if ( *tt == -SNUMBER && tt[1] != 0 ) {
01979                           if ( tt[1] < AM.MaxTer/((WORD)sizeof(WORD)*4)
01980                               && tt[1] > -(AM.MaxTer/((WORD)sizeof(WORD)*4))
01981                               && ( tt[1] > 1 || tt[1] < -1 ) ) {
01982                               ncount = ABS(tt[1]);
01983                               while ( ncount > 1 ) {
01984                                   *w++ = -SNUMBER; *w++ = 0; ncount--;
01985                               }
01986                               *w++ = -SNUMBER;
01987                               if ( tt[1] < 0 ) *w++ = -1;
01988                               else             *w++ =  1;
01989                               tt += 2;
01990                               action = 1;
01991                           }
01992                           else {
01993                               *w++ = *tt++; *w++ = *tt++;
01994                           }
01995                       }
```

```
01996                     else if ( *tt <= -FUNCTION ) {
01997                         *w++ = *tt++;
01998                     }
01999                     else if ( *tt < 0 ) {
02000                         *w++ = *tt++;
02001                         *w++ = *tt++;
02002                     }
02003                     else if ( tt[0] == tt[ARGHEAD]+ARGHEAD ) {
02004                         ttt = tt + tt[0] - 1;
02005                         i = (ABS(ttt[0])-1)/2;
02006                         if ( i > 1 ) {
02007 TooMany:                     old = AN.currentTerm;
02008                             AN.currentTerm = term;
02009                             MLOCK(ErrorMessageLock);
02010                             MesPrint("Too many arguments in output of ArgExplode");
02011                             MesPrint("Term = %t");
02012                             MUNLOCK(ErrorMessageLock);
02013                             AN.currentTerm = old;
02014                             return(-1);
02015                         }
02016                         if ( ttt[-1] != 1 ) goto NoExplode;
02017                         x = ttt[-2];
02018                         if ( 2*x > (AT.WorkTop-w)-*term ) goto TooMany;
02019                         ncount = x - 1;
02020                         while ( ncount > 0 ) {
02021                             *w++ = -SNUMBER; *w++ = 0; ncount--;
02022                         }
02023                         ttt[-2] = 1;
02024                         i = *tt; NCOPY(w,tt,i)
02025                         action = 1;
02026                     }
02027                     else {
02028 NoExplode:
02029                         i = *tt; NCOPY(w,tt,i)
02030                     }
02031                 }
02032                 AT.WorkPointer[1] = w - AT.WorkPointer;
02033                 while ( tt < tend ) *w++ = *tt++;
02034                 ttt = AT.WorkPointer; tt = t;
02035                 while ( ttt < w ) *tt++ = *ttt++;
02036                 term[0] = tt - term;
02037                 AT.WorkPointer = tt;
02038                 tend = tt; tstop = tt - ABS(tt[-1]);
02039             }
02040         }
02041         t += t[1];
02042     }
02043     if ( action ) {
02044         if ( Normalize(BHEAD term) ) return(-1);
02045     }
02046     return(0);
02047 }
02048
02049 /*
02050     #] ArgumentExplode :
02051     #[ ArgFactorize :
02052 */
02068 #define NEWORDER
02069
02070 int ArgFactorize(PHEAD WORD *argin, WORD *argout)
02071 {
02072 /*
02073     #[ step 0 : Declarations and initializations
02074 */
02075     WORD *argfree, *argextra, *argcopy, *t, *tstop, *a, *a1, *a2;
02076 #ifdef NEWORDER
02077     WORD *tt;
02078 #endif
02079     WORD startebuf = cbuf[AT.ebufnum].numrhs,oldword;
02080     WORD oldsorttype = AR.SortType, numargs;
02081     int error = 0, action = 0, i, ii, number, sign = 1;
02082
02083     *argout = 0;
02084 /*
02085     #] step 0 :
02086     #[ step 1 : Take care of ordering
02087 */
02088     AR.SortType = SORTHIGHFIRST;
02089     if ( oldsorttype != AR.SortType ) {
02090         NewSort(BHEAD0);
02091         oldword = argin[*argin]; argin[*argin] = 0;
02092         t = argin+ARGHEAD;
02093         while ( *t ) {
02094             tstop = t + *t;
02095             if ( AN.ncmod != 0 ) {
02096                 if ( AN.ncmod != 1 || ( (WORD)AN.cmod[0] < 0 ) ) {
02097                     MLOCK(ErrorMessageLock);
```

```
02098                   MesPrint("Factorization modulus a number, greater than a WORD not implemented.");
02099                   MUNLOCK(ErrorMessageLock);
02100                   Terminate(-1);
02101               }
02102               if ( Modulus(t) ) {
02103                   MLOCK(ErrorMessageLock);
02104                   MesCall("ArgFactorize");
02105                   MUNLOCK(ErrorMessageLock);
02106                   Terminate(-1);
02107               }
02108               if ( !*t) { t = tstop; continue; }
02109           }
02110           StoreTerm(BHEAD t);
02111           t = tstop;
02112       }
02113       /* par = 1, in case the arg has more than SubTermsInSmall terms */
02114       EndSort(BHEAD argin+ARGHEAD,1);
02115       argin[*argin] = oldword;
02116   }
02117 /*
02118   #] step 1 :
02119   #[ step 2 : take out the 'content'.
02120 */
02121   argfree = TakeArgContent(BHEAD argin,argout);
02122   {
02123       a1 = argout;
02124       while ( *a1 ) {
02125           if ( a1[0] == -SNUMBER && ( a1[1] == 1 || a1[1] == -1 ) ) {
02126               if ( a1[1] == -1 ) { sign = -sign; a1[1] = 1; }
02127               if ( a1[2] ) {
02128                   a = t = a1+2; while ( *t ) NEXTARG(t);
02129                   i = t - a1-2;
02130                   t = a1; NCOPY(t,a,i);
02131                   *t = 0;
02132                   continue;
02133               }
02134               else {
02135                   a1[0] = 0;
02136               }
02137               break;
02138           }
02139           else if ( a1[0] == FUNHEAD+ARGHEAD+4 && a1[ARGHEAD] == FUNHEAD+4
02140           && a1[*a1-1] == 3 && a1[*a1-2] == 1 && a1[*a1-3] == 1
02141           && a1[ARGHEAD+1] >= FUNCTION ) {
02142               a = t = a1+*a1; while ( *t ) NEXTARG(t);
02143               i = t - a;
02144               *a1 = -a1[ARGHEAD+1]; t = a1+1; NCOPY(t,a,i);
02145               *t = 0;
02146           }
02147           NEXTARG(a1);
02148       }
02149   }
02150   if ( argfree == 0 ) {
02151       argfree = argin;
02152   }
02153   else if ( argfree[0] == ( argfree[ARGHEAD]+ARGHEAD ) ) {
02154       Normalize(BHEAD argfree+ARGHEAD);
02155       argfree[0] = argfree[ARGHEAD]+ARGHEAD;
02156       argfree[1] = 0;
02157       if ( ( argfree[0] == ARGHEAD+4 ) && ( argfree[ARGHEAD+3] == 3 )
02158           && ( argfree[ARGHEAD+1] == 1 ) && ( argfree[ARGHEAD+2] == 1 ) ) {
02159           goto return0;
02160       }
02161   }
02162   else {
02163 /*
02164       The way we took out objects is rather brutish. We have to
02165       normalize
02166 */
02167       NewSort(BHEAD0);
02168       t = argfree+ARGHEAD;
02169       while ( *t ) {
02170           tstop = t + *t;
02171           Normalize(BHEAD t);
02172           StoreTerm(BHEAD t);
02173           t = tstop;
02174       }
02175       /* par = 1, in case the arg has more than SubTermsInSmall terms */
02176       EndSort(BHEAD argfree+ARGHEAD,1);
02177       t = argfree+ARGHEAD;
02178       while ( *t ) t += *t;
02179       *argfree = t - argfree;
02180   }
02181 /*
02182   #] step 2 :
02183   #[ step 3 : look whether we have done this one already.
02184 */
```

```
02185     if ( ( number = FindArg(BHEAD argfree) ) != 0 ) {
02186         if ( number > 0 ) t = cbuf[AT.fbufnum].rhs[number-1];
02187         else              t = cbuf[AC.ffbufnum].rhs[-number-1];
02188 /*
02189         Now position on the result. Remember we have in the cache:
02190                     inputarg,0,outputargs,0
02191         t is currently at inputarg. *inputarg is always positive.
02192         in principle this holds also for the arguments in the output
02193         but we take no risks here (in case of future developments).
02194 */
02195         t += *t; t++;
02196         tstop = t;
02197         ii = 0;
02198         while ( *tstop ) {
02199             if ( *tstop == -SNUMBER && tstop[1] == -1 ) {
02200                 sign = -sign; ii += 2;
02201             }
02202             NEXTARG(tstop);
02203         }
02204         a = argout; while ( *a ) NEXTARG(a);
02205 #ifndef NEWORDER
02206         if ( sign == -1 ) { *a++ = -SNUMBER; *a++ = -1; *a = 0; sign = 1; }
02207 #endif
02208         i = tstop - t - ii;
02209         ii = a - argout;
02210         a2 = a; a1 = a + i;
02211         *a1 = 0;
02212         while ( ii > 0 ) { *--a1 = *--a2; ii--; }
02213         a = argout;
02214         while ( *t ) {
02215             if ( *t == -SNUMBER && t[1] == -1 ) { t += 2; }
02216             else { COPY1ARG(a,t) }
02217         }
02218         goto return0;
02219     }
02220 /*
02221     #] step 3 :
02222     #[ step 4 : invoke ConvertToPoly
02223
02224                 We make a copy first in case there are no factors
02225 */
02226     argcopy = TermMalloc("argcopy");
02227     for ( i = 0; i <= *argfree; i++ ) argcopy[i] = argfree[i];
02228
02229     tstop = argfree + *argfree;
02230     {
02231         WORD sumcommu = 0;
02232         t = argfree + ARGHEAD;
02233         while ( t < tstop ) {
02234             sumcommu += DoesCommu(t);
02235             t += *t;
02236         }
02237         if ( sumcommu > 1 ) {
02238             MLOCK(ErrorMessageLock);
02239             MesPrint("ERROR: Cannot factorize an argument with more than one noncommuting object");
02240             MUNLOCK(ErrorMessageLock);
02241             Terminate(-1);
02242         }
02243     }
02244     t = argfree + ARGHEAD;
02245
02246     while ( t < tstop ) {
02247         if ( ( t[1] != SYMBOL ) && ( *t != (ABS(t[*t-1])+1) ) ) {
02248             action = 1; break;
02249         }
02250         t += *t;
02251     }
02252     if ( action ) {
02253         t = argfree + ARGHEAD;
02254         argextra = AT.WorkPointer;
02255         NewSort(BHEAD0);
02256         while ( t < tstop ) {
02257             if ( LocalConvertToPoly(BHEAD t,argextra,startebuf,0) < 0 ) {
02258                 error = -1;
02259 getout:
02260                 AR.SortType = oldsorttype;
02261                 TermFree(argcopy,"argcopy");
02262                 if ( argfree != argin ) TermFree(argfree,"argfree");
02263                 MesCall("ArgFactorize");
02264                 Terminate(-1);
02265                 return(-1);
02266             }
02267             StoreTerm(BHEAD argextra);
02268             t += *t; argextra += *argextra;
02269         }
02270         /* par = 1, in case the arg has more than SubTermsInSmall terms */
02271         if ( EndSort(BHEAD argfree+ARGHEAD,1) < 0 ) { error = -2; goto getout; }
```

```
02272         t = argfree + ARGHEAD;
02273         while ( *t > 0 ) t += *t;
02274         *argfree = t - argfree;
02275     }
02276 /*
02277     #] step 4 :
02278     #[ step 5 : If not in the tables, we have to do this by hard work.
02279 */
02280
02281     a = argout;
02282     while ( *a ) NEXTARG(a);
02283     if ( poly_factorize_argument(BHEAD argfree,a) < 0 ) {
02284         MesCall("ArgFactorize");
02285         error = -1;
02286     }
02287 /*
02288     #] step 5 :
02289     #[ step 6 : use now ConvertFromPoly
02290
02291                 Be careful: there should be more than one argument now.
02292 */
02293     if ( error == 0 && action ) {
02294         a1 = a; NEXTARG(a1);
02295         if ( *a1 != 0 ) {
02296             CBUF *C = cbuf+AC.cbufnum;
02297             CBUF *CC = cbuf+AT.ebufnum;
02298             WORD *oldworkpointer = AT.WorkPointer;
02299             WORD *argcopy2 = TermMalloc("argcopy2"), *a1, *a2;
02300             a1 = a; a2 = argcopy2;
02301             while ( *a1 ) {
02302                 if ( *a1 < 0 ) {
02303                     if ( *a1 > -FUNCTION ) *a2++ = *a1++;
02304                     *a2++ = *a1++; *a2 = 0;
02305                     continue;
02306                 }
02307                 t = a1 + ARGHEAD;
02308                 tstop = a1 + *a1;
02309                 argextra = AT.WorkPointer;
02310                 NewSort(BHEAD0);
02311                 while ( t < tstop ) {
02312                     if ( ConvertFromPoly(BHEAD t,argextra,numxsymbol,CC->numrhs-startebuf+numxsymbol
02313                     ,startebuf-numxsymbol,1) <= 0 ) {
02314                         TermFree(argcopy2,"argcopy2");
02315                         LowerSortLevel();
02316                         error = -3;
02317                         goto getout;
02318                     }
02319                     t += *t;
02320                     AT.WorkPointer = argextra + *argextra;
02321 /*
02322                     ConvertFromPoly leaves terms with subexpressions. Hence:
02323 */
02324                     if ( Generator(BHEAD argextra,C->numlhs) ) {
02325                         TermFree(argcopy2,"argcopy2");
02326                         LowerSortLevel();
02327                         error = -4;
02328                         goto getout;
02329                     }
02330                 }
02331                 AT.WorkPointer = oldworkpointer;
02332                 /* par = 1, in case the factor has more than SubTermsInSmall terms */
02333                 if ( EndSort(BHEAD a2+ARGHEAD,1) < 0 ) { error = -5; goto getout; }
02334                 t = a2+ARGHEAD; while ( *t ) t += *t;
02335                 *a2 = t - a2; a2[1] = 0; ZEROARG(a2);
02336                 ToFast(a2,a2); NEXTARG(a2);
02337                 a1 = tstop;
02338             }
02339             i = a2 - argcopy2;
02340             a2 = argcopy2; a1 = a;
02341             NCOPY(a1,a2,i);
02342             *a1 = 0;
02343             TermFree(argcopy2,"argcopy2");
02344 /*
02345             Erase the entries we made temporarily in cbuf[AT.ebufnum]
02346 */
02347             CC->numrhs = startebuf;
02348         }
02349         else {  /* no factorization. recover the argument from before step 3. */
02350             for ( i = 0; i <= *argcopy; i++ ) a[i] = argcopy[i];
02351         }
02352     }
02353 /*
02354     #] step 6 :
02355     #[ step 7 : Add this one to the tables.
02356
02357                 Possibly drop some elements in the tables
02358                 when they become too full.
```

```
02359 */
02360     if ( error == 0 && AN.ncmod == 0 ) {
02361         if ( InsertArg(BHEAD argcopy,a,0) < 0 ) { error = -1; }
02362     }
02363 /*
02364     #] step 7 :
02365     #[ step 8 : Clean up and return.
02366
02367         Change the order of the arguments in argout and a.
02368         Use argcopy as spare space.
02369 */
02370     ii = a - argout;
02371     for ( i = 0; i < ii; i++ ) argcopy[i] = argout[i];
02372     a1 = a;
02373     while ( *a1 ) {
02374         if ( *a1 == -SNUMBER && a1[1] < 0 ) {
02375             sign = -sign; a1[1] = -a1[1];
02376             if ( a1[1] == 1 ) {
02377                 a2 = a1+2; while ( *a2 ) NEXTARG(a2);
02378                 i = a2-a1-2; a2 = a1+2;
02379                 NCOPY(a1,a2,i);
02380                 *a1 = 0;
02381             }
02382             while ( *a1 ) NEXTARG(a1);
02383             break;
02384         }
02385         else {
02386             if ( *a1 > 0 && *a1 == a1[ARGHEAD]+ARGHEAD && a1[*a1-1] < 0 ) {
02387                 a1[*a1-1] = -a1[*a1-1]; sign = -sign;
02388             }
02389             if ( *a1 == ARGHEAD+4 && a1[ARGHEAD+1] == 1 && a1[ARGHEAD+2] == 1 ) {
02390                 a2 = a1+ARGHEAD+4; while ( *a2 ) NEXTARG(a2);
02391                 i = a2-a1-ARGHEAD-4; a2 = a1+ARGHEAD+4;
02392                 NCOPY(a1,a2,i);
02393                 *a1 = 0;
02394                 break;
02395             }
02396             while ( *a1 ) NEXTARG(a1);
02397             break;
02398         }
02399         NEXTARG(a1);
02400     }
02401     i = a1 - a;
02402     a2 = argout;
02403     NCOPY(a2,a,i);
02404     for ( i = 0; i < ii; i++ ) *a2++ = argcopy[i];
02405 #ifndef NEWORDER
02406     if ( sign == -1 ) { *a2++ = -SNUMBER; *a2++ = -1; sign = 1; }
02407 #endif
02408     *a2 = 0;
02409     TermFree(argcopy,"argcopy");
02410 return0:
02411     if ( argfree != argin ) TermFree(argfree,"argfree");
02412     if ( oldsorttype != AR.SortType ) {
02413         AR.SortType = oldsorttype;
02414         a = argout;
02415         while ( *a ) {
02416             if ( *a > 0 ) {
02417                 NewSort(BHEAD0);
02418                 oldword = a[*a]; a[*a] = 0;
02419                 t = a+ARGHEAD;
02420                 while ( *t ) {
02421                     tstop = t + *t;
02422                     StoreTerm(BHEAD t);
02423                     t = tstop;
02424                 }
02425                 /* par = 1, in case the factor has more than SubTermsInSmall terms */
02426                 EndSort(BHEAD a+ARGHEAD,1);
02427                 a[*a] = oldword;
02428                 a += *a;
02429             }
02430             else { NEXTARG(a); }
02431         }
02432     }
02433 #ifdef NEWORDER
02434     t = argout; numargs = 0;
02435     while ( *t ) {
02436         tt = t;
02437         NEXTARG(t);
02438         if ( *tt == ABS(t[-1])+1+ARGHEAD && sign == -1 ) { t[-1] = -t[-1]; sign = 1; }
02439         else if ( *tt == -SNUMBER && sign == -1 ) { tt[1] = -tt[1]; sign = 1; }
02440         numargs++;
02441     }
02442     if ( sign == -1 ) {
02443         *t++ = -SNUMBER; *t++ = -1; *t = 0; sign = 1; numargs++;
02444     }
02445 #else
```

```
02446 /*
02447     Now we have to sort the arguments
02448     First have the number of 'nontrivial/nonnumerical' arguments
02449     Then make a piece of code like in FullSymmetrize with that number
02450     of arguments to be symmetrized.
02451     Put a function in front
02452     Call the Symmetrize routine
02453 */
02454     t = argout; numargs = 0;
02455     while ( *t && *t != -SNUMBER && ( *t < 0 || ( ABS(t[*t-1]) != *t-1 ) ) ) {
02456         NEXTARG(t);
02457         numargs++;
02458     }
02459 #endif
02460     if ( numargs > 1 ) {
02461         WORD *Lijst;
02462         WORD x[3];
02463         x[0] = argout[-FUNHEAD];
02464         x[1] = argout[-FUNHEAD+1];
02465         x[2] = argout[-FUNHEAD+2];
02466         while ( *t ) { NEXTARG(t); }
02467         argout[-FUNHEAD] = SQRTFUNCTION;
02468         argout[-FUNHEAD+1] = t-argout+FUNHEAD;
02469         argout[-FUNHEAD+2] = 0;
02470         AT.WorkPointer = t+1;
02471         Lijst = AT.WorkPointer;
02472         for ( i = 0; i < numargs; i++ ) Lijst[i] = i;
02473         AT.WorkPointer += numargs;
02474         error = Symmetrize(BHEAD argout-FUNHEAD,Lijst,numargs,1,SYMMETRIC);
02475         AT.WorkPointer = Lijst;
02476         argout[-FUNHEAD] = x[0];
02477         argout[-FUNHEAD+1] = x[1];
02478         argout[-FUNHEAD+2] = x[2];
02479 #ifdef NEWORDER
02480 /*
02481         Now we have to get a potential numerical argument to the first position
02482 */
02483         tstop = argout; while ( *tstop ) { NEXTARG(tstop); }
02484         t = argout; number = 0;
02485         while ( *t ) {
02486             tt = t; NEXTARG(t);
02487             if ( *tt == -SNUMBER ) {
02488                 if ( number == 0 ) break;
02489                 x[0] = tt[1];
02490                 while ( tt > argout ) { *--t = *--tt; }
02491                 argout[0] = -SNUMBER; argout[1] = x[0];
02492                 break;
02493             }
02494             else if ( *tt == ABS(t[-1])+1+ARGHEAD ) {
02495                 if ( number == 0 ) break;
02496                 ii = t - tt;
02497                 for ( i = 0; i < ii; i++ ) tstop[i] = tt[i];
02498                 while ( tt > argout ) { *--t = *--tt; }
02499                 for ( i = 0; i < ii; i++ ) argout[i] = tstop[i];
02500                 *tstop = 0;
02501                 break;
02502             }
02503             number++;
02504         }
02505 #endif
02506     }
02507 /*
02508     #] step 8 :
02509 */
02510     return(error);
02511 }
02512
02513 /*
02514     #] ArgFactorize :
02515     #[ FindArg :
02516 */
02525 WORD FindArg(PHEAD WORD *a)
02526 {
02527     int number;
02528     if ( AN.ncmod != 0 ) return(0); /* no room for mod stuff */
02529     number = FindTree(AT.fbufnum,a);
02530     if ( number >= 0 ) return(number+1);
02531     number = FindTree(AC.ffbufnum,a);
02532     if ( number >= 0 ) return(-number-1);
02533     return(0);
02534 }
02535
02536 /*
02537     #] FindArg :
02538     #[ InsertArg :
02539 */
02549 int InsertArg(PHEAD WORD *argin, WORD *argout,int par)
```

```
02550 {
02551     CBUF *C;
02552     WORD *a, i, bufnum;
02553     if ( par == 0 ) {
02554         bufnum = AT.fbufnum;
02555         C = cbuf+bufnum;
02556         if ( C->numrhs >= (C->maxrhs-2) ) CleanupArgCache(BHEAD AT.fbufnum);
02557     }
02558     else if ( par == 1 ) {
02559         bufnum = AC.ffbufnum;
02560         C = cbuf+bufnum;
02561     }
02562     else { return(-1); }
02563     AddRHS(bufnum,1);
02564     AddNtoC(bufnum,*argin,argin,1);
02565     AddToCB(C,0)
02566     a = argout; while ( *a ) NEXTARG(a);
02567     i = a - argout;
02568     AddNtoC(bufnum,i,argout,2);
02569     AddToCB(C,0)
02570     return(InsTree(bufnum,C->numrhs));
02571 }
02572
02573 /*
02574     #] InsertArg :
02575     #[ CleanupArgCache :
02576 */
02584 int CleanupArgCache(PHEAD WORD bufnum)
02585 {
02586     CBUF *C = cbuf + bufnum;
02587     COMPTREE *boomlijst = C->boomlijst;
02588     LONG *weights = (LONG *)Malloc1(2*(C->numrhs+1)*sizeof(LONG),"CleanupArgCache");
02589     LONG w, whalf, *extraweights;
02590     WORD *a, *to, *from;
02591     int i,j,k;
02592     for ( i = 1; i <= C->numrhs; i++ ) {
02593         weights[i] = ((LONG)i) * (boomlijst[i].usage);
02594     }
02595 /*
02596         Now sort the weights and determine the halfway weight
02597 */
02598     extraweights = weights+C->numrhs+1;
02599     SortWeights(weights+1,extraweights,C->numrhs);
02600     whalf = weights[C->numrhs/2+1];
02601 /*
02602         We should drop everybody with a weight < whalf.
02603 */
02604     to = C->Buffer;
02605     k = 1;
02606     for ( i = 1; i <= C->numrhs; i++ ) {
02607         from = C->rhs[i]; w = ((LONG)i) * (boomlijst[i].usage);
02608         if ( w >= whalf ) {
02609             if ( i < C->numrhs-1 ) {
02610                 if ( to == from ) {
02611                     to = C->rhs[i+1];
02612                 }
02613                 else {
02614                     j = C->rhs[i+1] - from;
02615                     C->rhs[k] = to;
02616                     NCOPY(to,from,j)
02617                 }
02618             }
02619             else if ( to == from ) {
02620                 to += *to + 1; while ( *to ) NEXTARG(to); to++;
02621             }
02622             else {
02623                 a = from; a += *a+1; while ( *a ) NEXTARG(a); a++;
02624                 j = a - from;
02625                 C->rhs[k] = to;
02626                 NCOPY(to,from,j)
02627             }
02628             weights[k++] = boomlijst[i].usage;
02629         }
02630     }
02631     C->numrhs = --k;
02632     C->Pointer = to;
02633 /*
02634         Next we need to rebuild the tree.
02635         Note that this can probably be done much faster by using the
02636         remains of the old tree !!!!!!!!!!!!!!!
02637 */
02638     ClearTree(AT.fbufnum);
02639     for ( i = 1; i <= k; i++ ) {
02640         InsTree(AT.fbufnum,i);
02641         boomlijst[i].usage = weights[i];
02642     }
02643 /*
```

```
02644         And cleanup
02645 */
02646     M_free(weights,"CleanupArgCache");
02647     return(0);
02648 }
02649
02650 /*
02651     #] CleanupArgCache :
02652     #[ ArgSymbolMerge :
02653 */
02654
02655 int ArgSymbolMerge(WORD *t1, WORD *t2)
02656 {
02657     WORD *t1e = t1+t1[1];
02658     WORD *t2e = t2+t2[1];
02659     WORD *t1a = t1+2;
02660     WORD *t2a = t2+2;
02661     WORD *t3;
02662     while ( t1a < t1e && t2a < t2e ) {
02663         if ( *t1a < *t2a ) {
02664             if ( t1a[1] >= 0 ) {
02665                 t3 = t1a+2;
02666                 while ( t3 < t1e ) { t3[-2] = *t3; t3[-1] = t3[1]; t3 += 2; }
02667                 t1e -= 2;
02668             }
02669             else t1a += 2;
02670         }
02671         else if ( *t1a > *t2a ) {
02672             if ( t2a[1] >= 0 ) t2a += 2;
02673             else {
02674                 t3 = t1e;
02675                 while ( t3 > t1a ) { *t3 = t3[-2]; t3[1] = t3[-1]; t3 -= 2; }
02676                 *t1a++ = *t2a++;
02677                 *t1a++ = *t2a++;
02678                 t1e += 2;
02679             }
02680         }
02681         else {
02682             if ( t2a[1] < t1a[1] ) t1a[1] = t2a[1];
02683             t1a += 2; t2a += 2;
02684         }
02685     }
02686     while ( t2a < t2e ) {
02687         if ( t2a[1] < 0 ) {
02688             *t1a++ = *t2a++;
02689             *t1a++ = *t2a++;
02690         }
02691         else t2a += 2;
02692     }
02693     while ( t1a < t1e ) {
02694         if ( t1a[1] >= 0 ) {
02695             t3 = t1a+2;
02696             while ( t3 < t1e ) { t3[-2] = *t3; t3[-1] = t3[1]; t3 += 2; }
02697             t1e -= 2;
02698         }
02699         else t1a += 2;
02700     }
02701     t1[1] = t1a - t1;
02702     return(0);
02703 }
02704
02705 /*
02706     #] ArgSymbolMerge :
02707     #[ ArgDotproductMerge :
02708 */
02709
02710 int ArgDotproductMerge(WORD *t1, WORD *t2)
02711 {
02712     WORD *t1e = t1+t1[1];
02713     WORD *t2e = t2+t2[1];
02714     WORD *t1a = t1+2;
02715     WORD *t2a = t2+2;
02716     WORD *t3;
02717     while ( t1a < t1e && t2a < t2e ) {
02718         if ( *t1a < *t2a || ( *t1a == *t2a && t1a[1] < t2a[1] ) ) {
02719             if ( t1a[2] >= 0 ) {
02720                 t3 = t1a+3;
02721                 while ( t3 < t1e ) { t3[-3] = *t3; t3[-2] = t3[1]; t3[-1] = t3[2]; t3 += 3; }
02722                 t1e -= 3;
02723             }
02724             else t1a += 3;
02725         }
02726         else if ( *t1a > *t2a || ( *t1a == *t2a && t1a[1] > t2a[1] ) ) {
02727             if ( t2a[2] >= 0 ) t2a += 3;
02728             else {
02729                 t3 = t1e;
02730                 while ( t3 > t1a ) { *t3 = t3[-3]; t3[1] = t3[-2]; t3[2] = t3[-1]; t3 -= 3; }
```

```
02731                 *t1a++ = *t2a++;
02732                 *t1a++ = *t2a++;
02733                 *t1a++ = *t2a++;
02734                 t1e += 3;
02735             }
02736         }
02737         else {
02738             if ( t2a[2] < t1a[2] ) t1a[2] = t2a[2];
02739             t1a += 3; t2a += 3;
02740         }
02741     }
02742     while ( t2a < t2e ) {
02743         if ( t2a[2] < 0 ) {
02744             *t1a++ = *t2a++;
02745             *t1a++ = *t2a++;
02746             *t1a++ = *t2a++;
02747         }
02748         else t2a += 3;
02749     }
02750     while ( t1a < t1e ) {
02751         if ( t1a[2] >= 0 ) {
02752             t3 = t1a+3;
02753             while ( t3 < t1e ) { t3[-3] = *t3; t3[-2] = t3[1]; t3[-1] = t3[2]; t3 += 3; }
02754             t1e -= 3;
02755         }
02756         else t1a += 2;
02757     }
02758     t1[1] = t1a - t1;
02759     return(0);
02760 }
02761
02762 /*
02763     #] ArgDotproductMerge :
02764     #[ TakeArgContent :
02765 */
02778 WORD *TakeArgContent(PHEAD WORD *argin, WORD *argout)
02779 {
02780     GETBIDENTITY
02781     WORD *t, *rnext, *r1, *r2, *r3, *r5, *r6, *r7, *r8, *r9;
02782     WORD pow, *mm, *mnext, *mstop, *argin2 = argin, *argin3 = argin, *argfree;
02783     WORD ncom;
02784     int j, i, act;
02785     r5 = t = argin + ARGHEAD;
02786     r3 = argin + *argin;
02787     rnext = t + *t;
02788     GETSTOP(t,r6);
02789     r1 = argout;
02790     t++;
02791 /*
02792         First pass: arrange everything but the symbols and dotproducts.
02793         They need separate treatment because we have to take out negative
02794         powers.
02795 */
02796     while ( t < r6 ) {
02797 /*
02798             #[ DELTA/VECTOR :
02799 */
02800         if ( *t == DELTA || *t == VECTOR ) {
02801             r7 = t; r8 = t + t[1]; t += 2;
02802             while ( t < r8 ) {
02803                 mm = rnext;
02804                 pow = 1;
02805                 while ( mm < r3 ) {
02806                     mnext = mm + *mm;
02807                     GETSTOP(mm,mstop); mm++;
02808                     while ( mm < mstop ) {
02809                         if ( *mm != *r7 ) mm += mm[1];
02810                         else break;
02811                     }
02812                     if ( *mm == *r7 ) {
02813                         mstop = mm + mm[1]; mm += 2;
02814                         while ( ( *mm != *t || mm[1] != t[1] )
02815                             && mm < mstop ) mm += 2;
02816                         if ( mm >= mstop ) pow = 0;
02817                     }
02818                     else pow = 0;
02819                     if ( pow == 0 ) break;
02820                     mm = mnext;
02821                 }
02822                 if ( pow == 0 ) { t += 2; continue; }
02823 /*
02824                 We have a factor
02825 */
02826                 *r1++ = 8 + ARGHEAD;
02827                 for ( j = 1; j < ARGHEAD; j++ ) *r1++ = 0;
02828                 *r1++ = 8; *r1++ = *r7;
02829                 *r1++ = 4; *r1++ = *t; *r1++ = t[1];
```

```
02830                 *r1++ = 1; *r1++ = 1; *r1++ = 3;
02831                 argout = r1;
02832 /*
02833                 Now we have to remove the delta's/vectors
02834 */
02835                 mm = rnext;
02836                 while ( mm < r3 ) {
02837                     mnext = mm + *mm;
02838                     GETSTOP(mm,mstop); mm++;
02839                     while ( mm < mstop ) {
02840                         if ( *mm != *r7 ) mm += mm[1];
02841                         else break;
02842                     }
02843                     mstop = mm + mm[1]; mm += 2;
02844                     while ( mm < mstop && (
02845                      *mm != *t || mm[1] != t[1] ) ) mm += 2;
02846                     *mm = mm[1] = NOINDEX;
02847                     mm = mnext;
02848                 }
02849                 *t = t[1] = NOINDEX;
02850                 t += 2;
02851             }
02852         }
02853 /*
02854             #] DELTA/VECTOR :
02855             #[ INDEX :
02856 */
02857         else if ( *t == INDEX ) {
02858             r7 = t; r8 = t + t[1]; t += 2;
02859             while ( t < r8 ) {
02860                 mm = rnext;
02861                 pow = 1;
02862                 while ( mm < r3 ) {
02863                     mnext = mm + *mm;
02864                     GETSTOP(mm,mstop); mm++;
02865                     while ( mm < mstop ) {
02866                         if ( *mm != *r7 ) mm += mm[1];
02867                         else break;
02868                     }
02869                     if ( *mm == *r7 ) {
02870                         mstop = mm + mm[1]; mm += 2;
02871                         while ( *mm != *t
02872                             && mm < mstop ) mm++;
02873                         if ( mm >= mstop ) pow = 0;
02874                     }
02875                     else pow = 0;
02876                     if ( pow == 0 ) break;
02877                     mm = mnext;
02878                 }
02879                 if ( pow == 0 ) { t++; continue; }
02880 /*
02881                 We have a factor
02882 */
02883                 if ( *t < 0 ) { *r1++ = -VECTOR; }
02884                 else          { *r1++ = -INDEX; }
02885                 *r1++ = *t;
02886                 argout = r1;
02887 /*
02888                 Now we have to remove the index
02889 */
02890                 *t = NOINDEX;
02891                 mm = rnext;
02892                 while ( mm < r3 ) {
02893                     mnext = mm + *mm;
02894                     GETSTOP(mm,mstop); mm++;
02895                     while ( mm < mstop ) {
02896                         if ( *mm != *r7 ) mm += mm[1];
02897                         else break;
02898                     }
02899                     mstop = mm + mm[1]; mm += 2;
02900                     while ( mm < mstop &&
02901                      *mm != *t ) mm += 1;
02902                     *mm = NOINDEX;
02903                     mm = mnext;
02904                 }
02905                 t += 1;
02906             }
02907         }
02908 /*
02909             #] INDEX :
02910             #[ FUNCTION :
02911 */
02912         else if ( *t >= FUNCTION ) {
02913 /*
02914             In the next code we should actually look inside
02915             the DENOMINATOR or EXPONENT for noncommuting objects
02916 */
```

```
02917             if ( *t >= FUNCTION &&
02918                 functions[*t-FUNCTION].commute == 0 ) ncom = 0;
02919             else ncom = 1;
02920             if ( ncom ) {
02921                 mm = r5 + 1;
02922                 while ( mm < t && ( *mm == DUMMYFUN
02923                 || *mm == DUMMYTEN ) ) mm += mm[1];
02924                 if ( mm < t ) { t += t[1]; continue; }
02925             }
02926             mm = rnext; pow = 1;
02927             while ( mm < r3 ) {
02928                 mnext = mm + *mm;
02929                 GETSTOP(mm,mstop); mm++;
02930                 while ( mm < mstop ) {
02931                     if ( *mm == *t && mm[1] == t[1] ) {
02932                         for ( i = 2; i < t[1]; i++ ) {
02933                             if ( mm[i] != t[i] ) break;
02934                         }
02935                         if ( i >= t[1] )
02936                             { mm += mm[1]; goto nextmterm; }
02937                     }
02938                     if ( ncom && *mm != DUMMYFUN && *mm != DUMMYTEN )
02939                         { pow = 0; break; }
02940                     mm += mm[1];
02941                 }
02942                 if ( mm >= mstop ) pow = 0;
02943                 if ( pow == 0 ) break;
02944 nextmterm:      mm = mnext;
02945             }
02946             if ( pow == 0 ) { t += t[1]; continue; }
02947 /*
02948             Copy the function
02949 */
02950             *r1++ = t[1] + 4 + ARGHEAD;
02951             for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
02952             *r1++ = t[1] + 4;
02953             for ( i = 0; i < t[1]; i++ ) *r1++ = t[i];
02954             *r1++ = 1; *r1++ = 1; *r1++ = 3;
02955             argout = r1;
02956 /*
02957             Now we have to take out the functions
02958 */
02959             mm = rnext;
02960             while ( mm < r3 ) {
02961                 mnext = mm + *mm;
02962                 GETSTOP(mm,mstop); mm++;
02963                 while ( mm < mstop ) {
02964                     if ( *mm == *t && mm[1] == t[1] ) {
02965                         for ( i = 2; i < t[1]; i++ ) {
02966                             if ( mm[i] != t[i] ) break;
02967                         }
02968                         if ( i >= t[1] ) {
02969                             if ( functions[*t-FUNCTION].spec > 0 )
02970                                 *mm = DUMMYTEN;
02971                             else
02972                                 *mm = DUMMYFUN;
02973                             mm += mm[1];
02974                             goto nextterm;
02975                         }
02976                     }
02977                     mm += mm[1];
02978                 }
02979 nextterm:                       mm = mnext;
02980             }
02981             if ( functions[*t-FUNCTION].spec > 0 )
02982                     *t = DUMMYTEN;
02983             else
02984                     *t = DUMMYFUN;
02985             t += t[1];
02986         }
02987 /*
02988             #] FUNCTION :
02989 */
02990         else {
02991             t += t[1];
02992         }
02993     }
02994 /*
02995             #[ SYMBOL :
02996
02997         Now collect all symbols. We can use the space after r1 as storage
02998 */
02999     t = argin+ARGHEAD;
03000     rnext = t + *t;
03001     r2 = r1;
03002     while ( t < r3 ) {
03003         GETSTOP(t,r6);
```

```
03004         t++;
03005         act = 0;
03006         while ( t < r6 ) {
03007             if ( *t == SYMBOL ) {
03008                 act = 1;
03009                 i = t[1];
03010                 NCOPY(r2,t,i)
03011             }
03012             else { t += t[1]; }
03013         }
03014         if ( act == 0 ) {
03015             *r2++ = SYMBOL; *r2++ = 2;
03016         }
03017         t = rnext; rnext = rnext + *rnext;
03018     }
03019     *r2 = 0;
03020     argin2 = argin;
03021 /*
03022         Now we have a list of all symbols as a sequence of SYMBOL subterms.
03023         Any symbol that is absent in a subterm has power zero.
03024         We now need a list of all minimum powers.
03025         This can be done by subsequent merges.
03026 */
03027     r7 = r1;          /* The first object into which we merge. */
03028     r8 = r7 + r7[1];  /* The object that gets merged into r7.  */
03029     while ( *r8 ) {
03030         r2 = r8 + r8[1]; /* Next object */
03031         ArgSymbolMerge(r7,r8);
03032         r8 = r2;
03033     }
03034 /*
03035         Now we have to divide by the object in r7 and take it apart as factors.
03036         The division can be simple if there are no negative powers.
03037 */
03038     if ( r7[1] > 2 ) {
03039         r8 = r7+2;
03040         r2 = r7 + r7[1];
03041         act = 0;
03042         pow = 0;
03043         while ( r8 < r2 ) {
03044             if ( r8[1] < 0 ) { act = 1; pow += -r8[1]*(ARGHEAD+8); }
03045             else { pow += 2*r8[1]; }
03046             r8 += 2;
03047         }
03048 /*
03049         The amount of space we need to move r7 is given in pow
03050 */
03051         if ( act == 0 ) {   /* this can be done 'in situ' */
03052             t = argin + ARGHEAD;
03053             while ( t < r3 ) {
03054                 rnext = t + *t;
03055                 GETSTOP(t,r6);
03056                 t++;
03057                 while ( t < r6 ) {
03058                     if ( *t != SYMBOL ) { t += t[1]; continue; }
03059                     r8 = r7+2; r9 = t + t[1]; t += 2;
03060                     while ( ( t < r9 ) && ( r8 < r2 ) ) {
03061                         if ( *t == *r8 ) {
03062                             t[1] -= r8[1]; t += 2; r8 += 2;
03063                         }
03064                         else { /* *t must be < than *r8 !!! */
03065                             t += 2;
03066                         }
03067                     }
03068                     t = r9;
03069                 }
03070                 t = rnext;
03071             }
03072 /*
03073             And now the factors that go to argout.
03074             First we have to move r7 out of the way.
03075 */
03076             r8 = r7+pow; i = r7[1];
03077             while ( --i >= 0 ) r8[i] = r7[i];
03078             r2 += pow;
03079             r8 += 2;
03080             while ( r8 < r2 ) {
03081                 for ( i = 0; i < r8[1]; i++ ) { *r1++ = -SYMBOL; *r1++ = *r8; }
03082                 r8 += 2;
03083             }
03084         }
03085         else {  /* this needs a new location */
03086             argin2 = TermMalloc("TakeArgContent2");
03087 /*
03088             We have to multiply the inverse of r7 into argin
03089             The answer should go to argin2.
03090 */
```

```
03091             r5 = argin2; *r5++ = 0; *r5++ = 0; FILLARG(r5);
03092             t = argin+ARGHEAD;
03093             while ( t < r3 ) {
03094                 rnext = t + *t;
03095                 GETSTOP(t,r6);
03096                 r9 = r5;
03097                 *r5++ = *t++ + r7[1];
03098                 while ( t < r6 ) *r5++ = *t++;
03099                 i = r7[1] - 2; r8 = r7+2;
03100                 *r5++ = r7[0]; *r5++ = r7[1];
03101                 while ( i > 0 ) { *r5++ = *r8++; *r5++ = -*r8++; i -= 2; }
03102                 while ( t < rnext ) *r5++ = *t++;
03103                 Normalize(BHEAD r9);
03104                 r5 = r9 + *r9;
03105             }
03106             *r5 = 0;
03107             *argin2 = r5-argin2;
03108 /*
03109             We may have to sort the terms in argin2.
03110 */
03111             NewSort(BHEAD0);
03112             t = argin2+ARGHEAD;
03113             while ( *t ) {
03114                 StoreTerm(BHEAD t);
03115                 t += *t;
03116             }
03117             t = argin2+ARGHEAD;
03118             if ( EndSort(BHEAD t,0) < 0 ) goto Irreg;
03119             while ( *t ) t += *t;
03120             *argin2 = t - argin2;
03121             r3 = t;
03122 /*
03123             And now the factors that go to argout.
03124             First we have to move r7 out of the way.
03125 */
03126             r8 = r7+pow; i = r7[1];
03127             while ( --i >= 0 ) r8[i] = r7[i];
03128             r2 += pow;
03129             r8 += 2;
03130             while ( r8 < r2 ) {
03131                 if ( r8[1] >= 0 ) {
03132                     for ( i = 0; i < r8[1]; i++ ) { *r1++ = -SYMBOL; *r1++ = *r8; }
03133                 }
03134                 else {
03135                     for ( i = 0; i < -r8[1]; i++ ) {
03136                         *r1++ = ARGHEAD+8; *r1++ = 0;
03137                         FILLARG(r1);
03138                         *r1++ = 8; *r1++ = SYMBOL; *r1++ = 4; *r1++ = *r8;
03139                         *r1++ = -1; *r1++ = 1; *r1++ = 1; *r1++ = 3;
03140                     }
03141                 }
03142                 r8 += 2;
03143             }
03144             argout = r1;
03145         }
03146     }
03147 /*
03148             #] SYMBOL :
03149             #[ DOTPRODUCT :
03150
03151         Now collect all dotproducts. We can use the space after r1 as storage
03152 */
03153       t = argin2+ARGHEAD;
03154       rnext = t + *t;
03155       r2 = r1;
03156       while ( t < r3 ) {
03157         GETSTOP(t,r6);
03158         t++;
03159         act = 0;
03160         while ( t < r6 ) {
03161             if ( *t == DOTPRODUCT ) {
03162                 act = 1;
03163                 i = t[1];
03164                 NCOPY(r2,t,i)
03165             }
03166             else { t += t[1]; }
03167         }
03168         if ( act == 0 ) {
03169             *r2++ = DOTPRODUCT; *r2++ = 2;
03170         }
03171         t = rnext; rnext = rnext + *rnext;
03172       }
03173       *r2 = 0;
03174       argin3 = argin2;
03175 /*
03176         Now we have a list of all dotproducts as a sequence of DOTPRODUCT
03177         subterms. Any dotproduct that is absent in a subterm has power zero.
```

```
03178         We now need a list of all minimum powers.
03179         This can be done by subsequent merges.
03180 */
03181       r7 = r1;          /* The first object into which we merge. */
03182       r8 = r7 + r7[1];  /* The object that gets merged into r7.  */
03183       while ( *r8 ) {
03184         r2 = r8 + r8[1]; /* Next object */
03185         ArgDotproductMerge(r7,r8);
03186         r8 = r2;
03187       }
03188 /*
03189         Now we have to divide by the object in r7 and take it apart as factors.
03190         The division can be simple if there are no negative powers.
03191 */
03192       if ( r7[1] > 2 ) {
03193         r8 = r7+2;
03194         r2 = r7 + r7[1];
03195         act = 0;
03196         pow = 0;
03197         while ( r8 < r2 ) {
03198             if ( r8[2] < 0 ) { pow += -r8[2]*(ARGHEAD+9); }
03199             else             { pow +=  r8[2]*(ARGHEAD+9); }
03200             r8 += 3;
03201         }
03202 /*
03203         The amount of space we need to move r7 is given in pow
03204         For dotproducts we always need a new location
03205 */
03206         {
03207             argin3 = TermMalloc("TakeArgContent3");
03208 /*
03209             We have to multiply the inverse of r7 into argin
03210             The answer should go to argin2.
03211 */
03212             r5 = argin3; *r5++ = 0; *r5++ = 0; FILLARG(r5);
03213             t = argin2+ARGHEAD;
03214             while ( t < r3 ) {
03215                 rnext = t + *t;
03216                 GETSTOP(t,r6);
03217                 r9 = r5;
03218                 *r5++ = *t++ + r7[1];
03219                 while ( t < r6 ) *r5++ = *t++;
03220                 i = r7[1] - 2; r8 = r7+2;
03221                 *r5++ = r7[0]; *r5++ = r7[1];
03222                 while ( i > 0 ) { *r5++ = *r8++; *r5++ = *r8++; *r5++ = -*r8++; i -= 3; }
03223                 while ( t < rnext ) *r5++ = *t++;
03224                 Normalize(BHEAD r9);
03225                 r5 = r9 + *r9;
03226             }
03227             *r5 = 0;
03228             *argin3 = r5-argin3;
03229 /*
03230             We may have to sort the terms in argin3.
03231 */
03232             NewSort(BHEAD0);
03233             t = argin3+ARGHEAD;
03234             while ( *t ) {
03235                 StoreTerm(BHEAD t);
03236                 t += *t;
03237             }
03238             t = argin3+ARGHEAD;
03239             if ( EndSort(BHEAD t,0) < 0 ) goto Irreg;
03240             while ( *t ) t += *t;
03241             *argin3 = t - argin3;
03242             r3 = t;
03243 /*
03244             And now the factors that go to argout.
03245             First we have to move r7 out of the way.
03246 */
03247             r8 = r7+pow; i = r7[1];
03248             while ( --i >= 0 ) r8[i] = r7[i];
03249             r2 += pow;
03250             r8 += 2;
03251             while ( r8 < r2 ) {
03252                 for ( i = ABS(r8[2]); i > 0; i-- ) {
03253                     *r1++ = ARGHEAD+9; *r1++ = 0; FILLARG(r1);
03254                     *r1++ = 9; *r1++ = DOTPRODUCT; *r1++ = 5; *r1++ = *r8;
03255                     *r1++ = r8[1]; *r1++ = r8[2] < 0 ? -1: 1;
03256                     *r1++ = 1; *r1++ = 1; *r1++ = 3;
03257                 }
03258                 r8 += 3;
03259             }
03260             argout = r1;
03261         }
03262       }
03263 /*
03264             #] DOTPRODUCT :
```

```
03265
03266     We have now in argin3 the argument stripped of negative powers and
03267     common factors. The only thing left to deal with is to make the
03268     coefficients integer. For that we have to find the LCM of the denominators
03269     and the GCD of the numerators. And to start with, the sign.
03270     We force the sign of the first term to be positive.
03271 */
03272     t = argin3 + ARGHEAD; pow = 1;
03273     t += *t;
03274     if ( t[-1] < 0 ) {
03275         pow = 0;
03276         t[-1] = -t[-1];
03277         while ( t < r3 ) {
03278             t += *t; t[-1] = -t[-1];
03279         }
03280     }
03281 /*
03282     Now the GCD of the numerators and the LCM of the denominators:
03283 */
03284     argfree = TermMalloc("TakeArgContent1");
03285     if ( AN.cmod != 0 ) {
03286         r1 = MakeMod(BHEAD argin3,r1,argfree);
03287     }
03288     else {
03289         r1 = MakeInteger(BHEAD argin3,r1,argfree);
03290     }
03291     if ( pow == 0 ) {
03292         *r1++ = -SNUMBER;
03293         *r1++ = -1;
03294     }
03295     *r1 = 0;
03296 /*
03297     Cleanup
03298 */
03299     if ( argin3 != argin2 ) TermFree(argin3,"TakeArgContent3");
03300     if ( argin2 != argin  ) TermFree(argin2,"TakeArgContent2");
03301     return(argfree);
03302 Irreg:
03303 /* INTERNAL_ERROR_EXCL_START */
03304     MLOCK(ErrorMessageLock);
03305     MesPrint("!>Irregularity while sorting argument in TakeArgContent");
03306     MUNLOCK(ErrorMessageLock);
03307     if ( argin3 != argin2 ) TermFree(argin3,"TakeArgContent3");
03308     if ( argin2 != argin  ) TermFree(argin2,"TakeArgContent2");
03309     Terminate(-1);
03310     return(0);
03311 /* INTERNAL_ERROR_EXCL_STOP */
03312 }
03313
03314 /*
03315     #] TakeArgContent :
03316     #[ MakeInteger :
03317 */
03328 WORD *MakeInteger(PHEAD WORD *argin,WORD *argout,WORD *argfree)
03329 {
03330     UWORD *GCDbuffer, *GCDbuffer2, *LCMbuffer, *LCMb, *LCMc;
03331     WORD *r, *r1, *r2, *r3, *r4, *r5, *rnext, i, k, j;
03332     WORD kGCD, kLCM, kGCD2, kkLCM, jLCM, jGCD;
03333     GCDbuffer = NumberMalloc("MakeInteger");
03334     GCDbuffer2 = NumberMalloc("MakeInteger");
03335     LCMbuffer = NumberMalloc("MakeInteger");
03336     LCMb = NumberMalloc("MakeInteger");
03337     LCMc = NumberMalloc("MakeInteger");
03338     r4 = argin + *argin;
03339     r = argin + ARGHEAD;
03340 /*
03341     First take the first term to load up the LCM and the GCD
03342 */
03343     r2 = r + *r;
03344     j = r2[-1];
03345     r3 = r2 - ABS(j);
03346     k = REDLENG(j);
03347     if ( k < 0 ) k = -k;
03348     while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
03349     for ( kGCD = 0; kGCD < k; kGCD++ ) GCDbuffer[kGCD] = r3[kGCD];
03350     k = REDLENG(j);
03351     if ( k < 0 ) k = -k;
03352     r3 += k;
03353     while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
03354     for ( kLCM = 0; kLCM < k; kLCM++ ) LCMbuffer[kLCM] = r3[kLCM];
03355     r1 = r2;
03356 /*
03357     Now go through the rest of the terms in this argument.
03358 */
03359     while ( r1 < r4 ) {
03360         r2 = r1 + *r1;
03361         j = r2[-1];
```

```
03362         r3 = r2 - ABS(j);
03363         k = REDLENG(j);
03364         if ( k < 0 ) k = -k;
03365         while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
03366         if ( ( ( GCDbuffer[0] == 1 ) && ( kGCD == 1 ) ) ) {
03367 /*
03368             GCD is already 1
03369 */
03370         }
03371         else if ( ( ( k != 1 ) || ( r3[0] != 1 ) ) ) {
03372             if ( GcdLong(BHEAD GCDbuffer,kGCD,(UWORD *)r3,k,GCDbuffer2,&kGCD2) ) {
03373                 NumberFree(GCDbuffer,"MakeInteger");
03374                 NumberFree(GCDbuffer2,"MakeInteger");
03375                 NumberFree(LCMbuffer,"MakeInteger");
03376                 NumberFree(LCMb,"MakeInteger"); NumberFree(LCMc,"MakeInteger");
03377                 goto MakeIntegerErr;
03378             }
03379             kGCD = kGCD2;
03380             for ( i = 0; i < kGCD; i++ ) GCDbuffer[i] = GCDbuffer2[i];
03381         }
03382         else {
03383             kGCD = 1; GCDbuffer[0] = 1;
03384         }
03385         k = REDLENG(j);
03386         if ( k < 0 ) k = -k;
03387         r3 += k;
03388         while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
03389         if ( ( ( LCMbuffer[0] == 1 ) && ( kLCM == 1 ) ) ) {
03390             for ( kLCM = 0; kLCM < k; kLCM++ )
03391                 LCMbuffer[kLCM] = r3[kLCM];
03392         }
03393         else if ( ( ( k != 1 ) || ( r3[0] != 1 ) ) ) {
03394             if ( GcdLong(BHEAD LCMbuffer,kLCM,(UWORD *)r3,k,LCMb,&kkLCM) ) {
03395                 NumberFree(GCDbuffer,"MakeInteger"); NumberFree(GCDbuffer2,"MakeInteger");
03396                 NumberFree(LCMbuffer,"MakeInteger"); NumberFree(LCMb,"MakeInteger");
    NumberFree(LCMc,"MakeInteger");
03397                 goto MakeIntegerErr;
03398             }
03399             DivLong((UWORD *)r3,k,LCMb,kkLCM,LCMb,&kkLCM,LCMc,&jLCM);
03400             MulLong(LCMbuffer,kLCM,LCMb,kkLCM,LCMc,&jLCM);
03401             for ( kLCM = 0; kLCM < jLCM; kLCM++ )
03402                 LCMbuffer[kLCM] = LCMc[kLCM];
03403         }
03404         else {} /* LCM doesn't change */
03405         r1 = r2;
03406     }
03407 /*
03408     Now put the factor together: GCD/LCM
03409 */
03410     r3 = (WORD *)(GCDbuffer);
03411     if ( kGCD == kLCM ) {
03412         for ( jGCD = 0; jGCD < kGCD; jGCD++ )
03413             r3[jGCD+kGCD] = LCMbuffer[jGCD];
03414         k = kGCD;
03415     }
03416     else if ( kGCD > kLCM ) {
03417         for ( jGCD = 0; jGCD < kLCM; jGCD++ )
03418             r3[jGCD+kGCD] = LCMbuffer[jGCD];
03419         for ( jGCD = kLCM; jGCD < kGCD; jGCD++ )
03420             r3[jGCD+kGCD] = 0;
03421         k = kGCD;
03422     }
03423     else {
03424         for ( jGCD = kGCD; jGCD < kLCM; jGCD++ )
03425             r3[jGCD] = 0;
03426         for ( jGCD = 0; jGCD < kLCM; jGCD++ )
03427             r3[jGCD+kLCM] = LCMbuffer[jGCD];
03428         k = kLCM;
03429     }
03430     j = 2*k+1;
03431 /*
03432     Now we have to write this to argout
03433 */
03434     if ( ( j == 3 ) && ( r3[1] == 1 ) && ( (WORD)(r3[0]) > 0 ) ) {
03435         *argout = -SNUMBER;
03436         argout[1] = r3[0];
03437         r1 = argout+2;
03438     }
03439     else {
03440         r1 = argout;
03441         *r1++ = j+1+ARGHEAD; *r1++ = 0; FILLARG(r1);
03442         *r1++ = j+1; r2 = r3;
03443         for ( i = 0; i < k; i++ ) { *r1++ = *r2++; *r1++ = *r2++; }
03444         *r1++ = j;
03445     }
03446 /*
03447     Next we have to take the factor out from the argument.
```

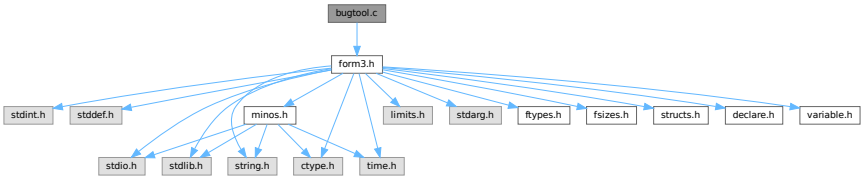
```
03448     This cannot be done in location, because the denominator stuff can make
03449     coefficients longer.
03450 */
03451     r2 = argfree + 2; FILLARG(r2)
03452     while ( r < r4 ) {
03453         rnext = r + *r;
03454         j = ABS(rnext[-1]);
03455         r5 = rnext - j;
03456         r3 = r2;
03457         while ( r < r5 ) *r2++ = *r++;
03458         j = (j-1)/2;    /* reduced length. Remember, k is the other red length */
03459         if ( DivRat(BHEAD (UWORD *)r5,j,GCDbuffer,k,(UWORD *)r2,&i) ) {
03460             goto MakeIntegerErr;
03461         }
03462         i = 2*i+1;
03463         r2 = r2 + i;
03464         if ( rnext[-1] < 0 ) r2[-1] = -i;
03465         else                 r2[-1] =  i;
03466         *r3 = r2-r3;
03467         r = rnext;
03468     }
03469     *r2 = 0;
03470     argfree[0] = r2-argfree;
03471     argfree[1] = 0;
03472 /*
03473     Cleanup
03474 */
03475     NumberFree(LCMc,"MakeInteger");
03476     NumberFree(LCMb,"MakeInteger");
03477     NumberFree(LCMbuffer,"MakeInteger");
03478     NumberFree(GCDbuffer2,"MakeInteger");
03479     NumberFree(GCDbuffer,"MakeInteger");
03480     return(r1);
03481
03482 MakeIntegerErr:
03483     MesCall("MakeInteger");
03484     Terminate(-1);
03485     return(0);
03486 }
03487
03488 /*
03489     #] MakeInteger :
03490     #[ MakeMod :
03491 */
03499 WORD *MakeMod(PHEAD WORD *argin,WORD *argout,WORD *argfree)
03500 {
03501     WORD *r, *instop, *r1, *m, x, xx, ix, ip;
03502     int i;
03503     r = argin; instop = r + *r; r += ARGHEAD;
03504     x = r[*r-3];
03505     if ( r[*r-1] < 0 ) x += AN.cmod[0];
03506     if ( GetModInverses(x,(WORD)(AN.cmod[0]),&ix,&ip) ) {
03507         Terminate(-1);
03508     }
03509     argout[0] = -SNUMBER;
03510     argout[1] = x;
03511     argout[2] = 0;
03512     r1 = argout+2;
03513 /*
03514     Now we have to multiply all coefficients by ix.
03515     This does not make things longer, but we should keep to the conventions
03516     of MakeInteger.
03517 */
03518     m = argfree + ARGHEAD;
03519     while ( r < instop ) {
03520         xx = r[*r-3]; if ( r[*r-1] < 0 ) xx += AN.cmod[0];
03521         xx = (WORD)((((LONG)xx)*ix) % AN.cmod[0]);
03522         if ( xx != 0 ) {
03523             i = *r; NCOPY(m,r,i);
03524             m[-3] = xx; m[-1] = 3;
03525         }
03526         else { r += *r; }
03527     }
03528     *m = 0;
03529     *argfree = m - argfree;
03530     argfree[1] = 0;
03531     argfree += 2; FILLARG(argfree);
03532     return(r1);
03533 }
03534
03535 /*
03536     #] MakeMod :
03537     #[ SortWeights :
03538 */
03544 void SortWeights(LONG *weights,LONG *extraspace,WORD number)
03545 {
03546     LONG w, *fill, *from1, *from2;
```

```
03547     int n1,n2,i;
03548     if ( number >= 4 ) {
03549         n1 = number/2; n2 = number - n1;
03550         SortWeights(weights,extraspace,n1);
03551         SortWeights(weights+n1,extraspace,n2);
03552 /*
03553         We copy the first patch to the extra space. Then we merge
03554         Note that a potential remaining n2 objects are already in place.
03555 */
03556         for ( i = 0; i < n1; i++ ) extraspace[i] = weights[i];
03557         fill = weights; from1 = extraspace; from2 = weights+n1;
03558         while ( n1 > 0 && n2 > 0 ) {
03559             if ( *from1 <= *from2 ) { *fill++ = *from1++; n1--; }
03560             else                    { *fill++ = *from2++; n2--; }
03561         }
03562         while ( n1 > 0 ) { *fill++ = *from1++; n1--; }
03563     }
03564 /*
03565     Special cases
03566 */
03567     else if ( number == 3 ) { /* 6 permutations of which one is trivial */
03568         if ( weights[0] > weights[1] ) {
03569             if ( weights[1] > weights[2] ) {
03570                 w = weights[0]; weights[0] = weights[2]; weights[2] = w;
03571             }
03572             else if ( weights[0] > weights[2] ) {
03573                 w = weights[0]; weights[0] = weights[1];
03574                 weights[1] = weights[2]; weights[2] = w;
03575             }
03576             else {
03577                 w = weights[0]; weights[0] = weights[1]; weights[1] = w;
03578             }
03579         }
03580         else if ( weights[0] > weights[2] ) {
03581             w = weights[0]; weights[0] = weights[2];
03582             weights[2] = weights[1]; weights[1] = w;
03583         }
03584         else if ( weights[1] > weights[2] ) {
03585             w = weights[1]; weights[1] = weights[2]; weights[2] = w;
03586         }
03587     }
03588     else if ( number == 2 ) {
03589         if ( weights[0] > weights[1] ) {
03590             w = weights[0]; weights[0] = weights[1]; weights[1] = w;
03591         }
03592     }
03593     return;
03594 }
03595
03596 /*
03597     #] SortWeights :
03598 */
```

4.3 bugtool.c File Reference

`#include "form3.h"`

Include dependency graph for bugtool.c:



Functions

- void ExprStatus (EXPRESSIONS e)

4.3.1 Detailed Description

Low level routines for debugging

Definition in file bugtool.c.

4.3.2 Function Documentation

4.3.2.1 ExprStatus()

```
void ExprStatus (
            EXPRESSIONS e )
```

Definition at line 66 of file bugtool.c.

4.4 bugtool.c

Go to the documentation of this file.

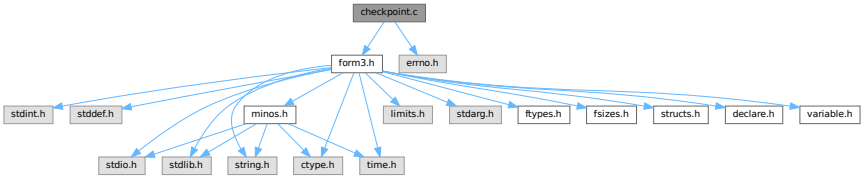
```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032
00033 /*
00034     #[ Includes :
00035 */
00036
00037 #include "form3.h"
00038
00039 /*
00040     #] Includes :
00041     #[ ExprStatus :
00042 */
00043
00044 static UBYTE *statusexpr[] = {
00045       (UBYTE *)"LOCALEXPRESSION"
00046     ,(UBYTE *)"SKIPLEXPRESSION"
00047     ,(UBYTE *)"DROPLEXPRESSION"
00048     ,(UBYTE *)"DROPPEDEXPRESSION"
00049     ,(UBYTE *)"GLOBALEXPRESSION"
00050     ,(UBYTE *)"SKIPGEXPRESSION"
00051     ,(UBYTE *)"DROPGEXPRESSION"
00052     ,(UBYTE *)"UNKNOWN"
00053     ,(UBYTE *)"STOREDEXPRESSION"
00054     ,(UBYTE *)"HIDDENLEXPRESSION"
00055     ,(UBYTE *)"HIDELEXPRESSION"
00056     ,(UBYTE *)"DROPHLEXPRESSION"
```

```
00057     ,(UBYTE *)"UNHIDELEXPRESSION"
00058     ,(UBYTE *)"HIDDENGEXPRESSION"
00059     ,(UBYTE *)"HIDEGEXPRESSION"
00060     ,(UBYTE *)"DROPHGEXPRESSION"
00061     ,(UBYTE *)"UNHIDEGEXPRESSION"
00062     ,(UBYTE *)"INTOHIDELEXPRESSION"
00063     ,(UBYTE *)"INTOHIDEGEXPRESSION"
00064 };
00065
00066 void ExprStatus(EXPRESSIONS e)
00067 {
00068     MesPrint("Expression %s(%d) has status %s(%d,%d). Buffer: %d, Position: %15p",
00069         AC.exprnames->namebuffer+e->name,(WORD)(e-Expressions),
00070         statusexpr[e->status],e->status,e->hidelevel,
00071         e->whichbuffer,&(e->onfile));
00072 }
00073
00074 /*
00075     #] ExprStatus :
00076 */
```

4.5 checkpoint.c File Reference

```
#include "form3.h"
#include <errno.h>
```

Include dependency graph for checkpoint.c:



Macros

- #define CACHED_SNAPSHOT
- #define CACHE_SIZE 4096
- #define R_FREE(ARG) if (ARG) M_free(ARG, #ARG);
- #define R_FREE_NAMETREE(ARG)
- #define R_FREE_STREAM(ARG)
- #define R_SET(VAR, TYPE) VAR = *((TYPE*)p); p = (unsigned char*)p + sizeof(TYPE);
- #define R_COPY_B(VAR, SIZE, CAST)
- #define S_WRITE_B(BUF, LEN) if (fwrite_cached(BUF, 1, LEN, fd) != (size_t)(LEN)) return(__LINE__);
- #define S_FLUSH_B if (flush_cache(fd) != 1) return(__LINE__);
- #define R_COPY_S(VAR, CAST)
- #define S_WRITE_S(STR)
- #define R_COPY_LIST(ARG)
- #define S_WRITE_LIST(LST)
- #define R_COPY_NAMETREE(ARG)
- #define S_WRITE_NAMETREE(ARG)
- #define S_WRITE_DOLLAR(ARG)
- #define ANNOUNCE(str)

Functions

- int CheckRecoveryFile (void)
- void DeleteRecoveryFile (void)
- char * RecoveryFilename (void)
- void InitRecovery (void)
- size_t fwrite_cached (const void *ptr, size_t size, size_t nmemb, FILE *fd)
- size_t flush_cache (FILE *fd)
- int DoRecovery (int *moduletype)
- void DoCheckpoint (int moduletype)

Variables

- unsigned char cache_buffer [CACHE_SIZE]
- size_t cache_fill = 0

4.5.1 Detailed Description

Contains all functions that deal with the recovery mechanism controlled and activated by the On Checkpoint switch.

The main function are DoCheckpoint, DoRecovery, and DoSnapshot. If the checkpoints are activated DoCheckpoint is called every time a module is finished executing. If the conditions for the creation of a recovery snapshot are met DoCheckpoint calls DoSnapshot. DoRecovery is called once when FORM starts up with the command line argument -R. Most of the other code contains debugging facilities that are only compiled if the macro PRINTDEBUG is defined.

The recovery mechanism is atomic, i.e. only if everything went well, the final recovery file is created (and the older one overwritten) in a single step (copying). If some errors occur, a warning is issued and the program continues without having created a new recovery file. The only situation in which the creation of the recovery data leads to a termination of the running program is if not enough disk or memory space is left.

For ParFORM each slave creates its own recovery file, sends it to the master and then it deletes the recovery file. The master stores all the recovery files and on recovery it feeds these files to the slaves. It is nearly impossible to recover after some MPI fault so ParFORM terminates on any recovery failure.

DoRecovery and DoSnapshot do the loading and saving of the recovery data, respectively. Every change in one functions needs to be accompanied by the appropriate change in the other function. The structure of both functions is quite similar. They handle the relevant global structs one after the other and then care about the copying of the hide and scratch files.

The names of the recovery, scratch and hide files are hard-coded in the variables in fold "filenames and system commands".

If the global structs AM,AP,AC,AR are changed, DoRecovery and DoSnapshot usually also have to be changed. Some structs are read/written as a whole (AP,AC), some are read/written only partly as a selection of their individual elements (AM,AR). If AM or AR have been changed by adding or removing an element that is important for the runtime status, then the reading/writing statements have to be added to or removed from DoRecovery and Do↩ Snapshot. If AP or AC are changed, then for non-pointer variables (in the case of a struct it also means that none of its elements is a pointer) nothing has to be changed in the functions here. If pointers are involved, extra code has to be added (or removed). See the comments of DoRecovery and DoSnapshot.

Definition in file checkpoint.c.

4.5.2 Macro Definition Documentation

4.5.2.1 CACHED_SNAPSHOT

```
#define CACHED_SNAPSHOT
```

Definition at line 1214 of file checkpoint.c.

4.5.2.2 CACHE_SIZE

```
#define CACHE_SIZE 4096
```

Definition at line 1216 of file checkpoint.c.

4.5.2.3 R_FREE

```
#define R_FREE(
              ARG )  if ( ARG ) M_free(ARG, #ARG);
```

Definition at line 1281 of file checkpoint.c.

4.5.2.4 R_FREE_NAMETREE

```
#define R_FREE_NAMETREE(
              ARG )
```

Value:

```
    R_FREE(ARG->namenode); \
    R_FREE(ARG->namebuffer); \
    R_FREE(ARG);
```

Definition at line 1284 of file checkpoint.c.

4.5.2.5 R_FREE_STREAM

```
#define R_FREE_STREAM(
              ARG )
```

Value:

```
    R_FREE(ARG.buffer); \
    R_FREE(ARG.FoldName); \
    R_FREE(ARG.name);
```

Definition at line 1289 of file checkpoint.c.

4.5.2.6 R_SET

```
#define R_SET(
              VAR,
              TYPE )  VAR = *((TYPE*)p); p = (unsigned char*)p + sizeof(TYPE);
```

Definition at line 1296 of file checkpoint.c.

4.5.2.7 R_COPY_B

```
#define R_COPY_B(
                VAR,
                SIZE,
                CAST )
```

Value:

```
    VAR = (CAST)Malloc1(SIZE,#VAR); \
    memcpy(VAR, p, SIZE); p = (unsigned char*)p + SIZE;
```

Definition at line 1301 of file checkpoint.c.

4.5.2.8 S_WRITE_B

```
#define S_WRITE_B(
                BUF,
                LEN )  if ( fwrite_cached(BUF, 1, LEN, fd) != (size_t)(LEN) ) return(__LINE__);
```

Definition at line 1305 of file checkpoint.c.

4.5.2.9 S_FLUSH_B

```
#define S_FLUSH_B  if ( flush_cache(fd) != 1 ) return(__LINE__);
```

Definition at line 1308 of file checkpoint.c.

4.5.2.10 R_COPY_S

```
#define R_COPY_S(
                VAR,
                CAST )
```

Value:

```
    if ( VAR ) { \
        VAR = (CAST)Malloc1(strlen(p)+1,"R_COPY_S"); \
        strcpy((char*)VAR, p); p = (unsigned char*)p + strlen(p) + 1; \
    }
```

Definition at line 1313 of file checkpoint.c.

4.5.2.11 S_WRITE_S

```
#define S_WRITE_S(
                STR )
```

Value:

```
    if ( STR ) { \
        l = strlen((char*)STR) + 1; \
        if ( fwrite_cached(STR, 1, l, fd) != (size_t)l ) return(__LINE__); \
    }
```

Definition at line 1319 of file checkpoint.c.

4.5.2.12 R_COPY_LIST

```
#define R_COPY_LIST(
                ARG )
```

Value:

```
    if ( ARG.maxnum ) { \
        R_COPY_B(ARG.lijst, ARG.size*ARG.maxnum, void*) \
    }
```

Definition at line 1327 of file checkpoint.c.

4.5.2.13 S_WRITE_LIST

```
#define S_WRITE_LIST(
                LST )
```

Value:

```
    if ( LST.maxnum ) { \
        S_WRITE_B((char*)LST.lijst, LST.maxnum*LST.size) \
    }
```

Definition at line 1332 of file checkpoint.c.

4.5.2.14 R_COPY_NAMETREE

```
#define R_COPY_NAMETREE(
                ARG )
```

Value:

```
    R_COPY_B(ARG, sizeof(NAMETREE), NAMETREE*); \
    if ( ARG->namenode ) { \
        R_COPY_B(ARG->namenode, ARG->nodesize*sizeof(NAMENODE), NAMENODE*); \
    } \
    if ( ARG->namebuffer ) { \
        R_COPY_B(ARG->namebuffer, ARG->namesize, UBYTE*); \
    }
```

Definition at line 1339 of file checkpoint.c.

4.5.2.15 S_WRITE_NAMETREE

```
#define S_WRITE_NAMETREE(
                ARG )
```

Value:

```
    S_WRITE_B(ARG, sizeof(NAMETREE)); \
    if ( ARG->namenode ) { \
        S_WRITE_B(ARG->namenode, ARG->nodesize*sizeof(struct NaMeNode)); \
    } \
    if ( ARG->namebuffer ) { \
        S_WRITE_B(ARG->namebuffer, ARG->namesize); \
    }
```

Definition at line 1348 of file checkpoint.c.

4.5.2.16 S_WRITE_DOLLAR

```
#define S_WRITE_DOLLAR(
              ARG )
```

Value:

```
    if ( ARG.size && ARG.where && ARG.where != &(AM.dollarzero) ) { \
        S_WRITE_B(ARG.where, ARG.size*sizeof(WORD)) \
    }
```

Definition at line 1359 of file checkpoint.c.

4.5.2.17 ANNOUNCE

```
#define ANNOUNCE(
              str )
```

Definition at line 1370 of file checkpoint.c.

4.5.3 Function Documentation

4.5.3.1 CheckRecoveryFile()

```
int CheckRecoveryFile (
             void  )
```

Checks whether a snapshot/recovery file exists. Returns 1 if it exists, 0 otherwise.

Definition at line 278 of file checkpoint.c.

References RecoveryFilename().

Here is the call graph for this function:

4.5.3.2 DeleteRecoveryFile()

```
void DeleteRecoveryFile (
             void  )
```

Deletes the recovery files. It is called by CleanUp() in the case of a successful completion.

Definition at line 333 of file checkpoint.c.

4.5.3.3 RecoveryFilename()

```
char * RecoveryFilename (
            void  )
```

Returns pointer to recovery filename.

Definition at line 364 of file checkpoint.c.

Referenced by CheckRecoveryFile(), and DoRecovery().

4.5.3.4 InitRecovery()

```
void InitRecovery (
            void  )
```

Sets up the strings for the filenames of the recovery files. This functions should only be called once to avoid memory leaks and after AM.TempDir has been initialized.

Definition at line 399 of file checkpoint.c.

4.5.3.5 fwrite_cached()

```
size_t fwrite_cached (
            const void * ptr,
            size_t size,
            size_t nmemb,
            FILE * fd )
```

Definition at line 1222 of file checkpoint.c.

4.5.3.6 flush_cache()

```
size_t flush_cache (
            FILE * fd )
```

Definition at line 1247 of file checkpoint.c.

4.5.3.7 DoRecovery()

```
int DoRecovery (
            int * moduletype )
```

Reads from the recovery file and restores all necessary variables and states in FORM, so that the execution can recommence in preprocessor() as if no restart of FORM had occurred.

The recovery file is read into memory as a whole. The pointer p then points into this memory at the next non-processed data. The macros by which variables are restored, like R_SET, automatically increase p appropriately.

If something goes wrong, the function returns with a non-zero value.

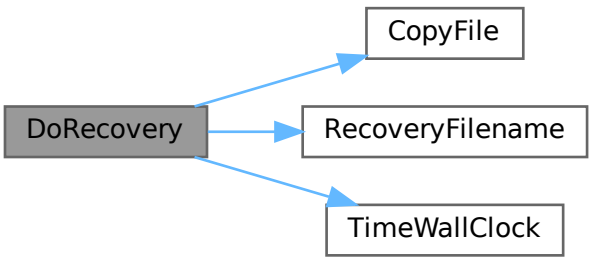
Allocated memory that would be lost when overwriting the global structs with data from the file is freed first. A major part of the code deals with the restoration of pointers. The idiom we use is to memorize the original pointer value (org), allocate new memory and copy the data from the file into this memory, calculate the offset between the old pointer value and the new allocated memory position (ofs), and then correct all affected pointers (+=ofs).

We rely on the fact that several variables (especially in AM) are already assigned the correct values by the startup functions. That means, in principle, that a change in the setup files between snapshot creation and recovery will be noticed.

Definition at line 1402 of file checkpoint.c.

References TaBlEs::argtail, TaBlEs::boomlijst, BrAcKeTiNfO::bracketbuffer, TaBlEs::buffers, TaBlEs::bufferssize, CopyFile(), TaBlEs::flags, ReNuMbEr::func, ReNuMbEr::funnum, VaRrEnUm::hi, BrAcKeTiNfO::indexbuffer, ReNuMbEr::indi, ReNuMbEr::indnum, VaRrEnUm::lo, TaBlEs::MaxTreeSize, TaBlEs::mm, TaBlEs::numind, TaBlEs::pattern, TaBlEs::prototype, TaBlEs::prototypeSize, RecoveryFilename(), ExPrEsSiOn::renumlists, TaBlEs::reserved, TaBlEs::spare, TaBlEs::sparse, VaRrEnUm::start, ReNuMbEr::symb, ReNuMbEr::symnum, TaBlEs::tablepointers, TimeWallClock(), TaBlEs::totind, ReNuMbEr::vecnum, and ReNuMbEr::vect.

Here is the call graph for this function:



4.5.3.8 DoCheckpoint()

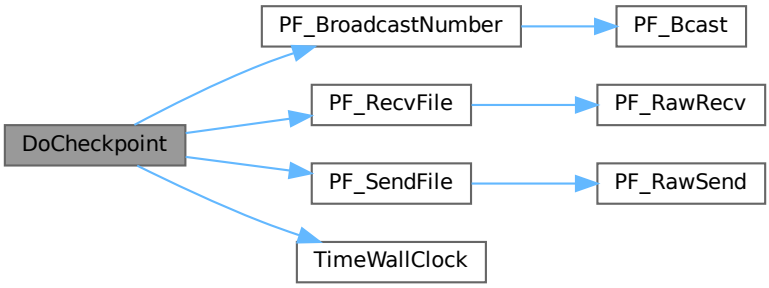
```
void DoCheckpoint (
            int moduletype )
```

Checks whether a snapshot should be done. Calls DoSnapshot() to create the snapshot.

Definition at line 3122 of file checkpoint.c.

References PF_BroadcastNumber(), PF_RecvFile(), PF_SendFile(), and TimeWallClock().

Here is the call graph for this function:



4.5.4 Variable Documentation

4.5.4.1 cache_buffer

```
unsigned char cache_buffer[CACHE_SIZE]
```

Definition at line 1219 of file checkpoint.c.

4.5.4.2 cache_fill

```
size_t cache_fill = 0
```

Definition at line 1220 of file checkpoint.c.

4.6 checkpoint.c

Go to the documentation of this file.

```
00001 /*
00002     #[ Explanations :
00003 */
00052 /*
00053     #] Explanations :
00054     #[ License :
00055  *
00056  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00057  *   When using this file you are requested to refer to the publication
00058  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00059  *   This is considered a matter of courtesy as the development was paid
00060  *   for by FOM the Dutch physics granting agency and we would like to
00061  *   be able to track its scientific use to convince FOM of its value
00062  *   for the community.
00063  *
00064  *   This file is part of FORM.
00065  *
00066  *   FORM is free software: you can redistribute it and/or modify it under the
00067  *   terms of the GNU General Public License as published by the Free Software
00068  *   Foundation, either version 3 of the License, or (at your option) any later
00069  *   version.
00070  *
00071  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00072  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00073  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00074  *   details.
00075  *
00076  *   You should have received a copy of the GNU General Public License along
00077  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00078  */
00079 /*
00080     #] License :
00081     #[ Includes :
00082 */
00083 /* UNFINISHED_FEATURE_EXCL_START */
00084 #include "form3.h"
00085
00086 #include <errno.h>
00087
00088 /*
00089 #define PRINTDEBUG
00090 */
00091
00092 /*
00093 #define PRINTTIMEMARKS
00094 */
00095
00096 /*
00097     #] Includes :
00098     #[ filenames and system commands :
00099 */
00100
00104 #ifdef WITHMPI
00105 #define BASENAME_FMT "%c%04dFORMrecv"
00111 static char BaseName[] = BASENAME_FMT;
00112 #else
00113 static char *BaseName = "FORMrecv";
00114 #endif
00118 static char *recoveryfile = 0;
00124 static char *intermedfile = 0;
00128 static char *sortfile = 0;
00132 static char *hidefile = 0;
00136 static char *storefile = 0;
00137
00142 static int done_snapshot = 0;
00143
00144 #ifdef WITHMPI
00149 static int PF_fmt_pos;
00150
00155 static const char *PF_recoveryfile(char prefix, int id, int intermed)
00156 {
00157     /*
00158      * Assume that InitRecovery() has been already called, namely
00159      * recoveryfile, intermedfile and PF_fmt_pos are already initialized.
00160      */
00161     static char *tmp_recovery = NULL;
00162     static char *tmp_intermed  = NULL;
00163     char *tmp, c;
00164     if ( tmp_recovery == NULL ) {
00165         if ( PF.numtasks > 9999 ) {  /* see BASENAME_FMT */
00166             MesPrint("Checkpoint: too many number of processors.");
00167             Terminate(-1);
```

```
00168         }
00169         tmp_recovery = (char *)Malloc1(strlen(recoveryfile) + strlen(intermedfile) + 2,
    "PF_recoveryfile");
00170         tmp_intermed = tmp_recovery + strlen(recoveryfile) + 1;
00171         strcpy(tmp_recovery, recoveryfile);
00172         strcpy(tmp_intermed, intermedfile);
00173     }
00174     tmp = intermed ? tmp_intermed : tmp_recovery;
00175     c = tmp[PF_fmt_pos + 13];  /* The magic number 13 comes from BASENAME_FMT. */
00176     sprintf(tmp + PF_fmt_pos, BASENAME_FMT, prefix, id);
00177     tmp[PF_fmt_pos + 13] = c;
00178     return tmp;
00179 }
00180 #endif
00181
00182 /*
00183     #] filenames and system commands :
00184     #[ CheckRecoveryFile :
00185 */
00186
00191 #ifdef WITHMPI
00192
00198 static int PF_CheckRecoveryFile()
00199 {
00200     int i,ret=0;
00201     FILE *fd;
00202     /* Check if the recovery file for the master exists. */
00203     if ( PF.me == MASTER ) {
00204         if ( (fd = fopen(recoveryfile, "r")) ) {
00205             fclose(fd);
00206             PF_BroadcastNumber(1);
00207         }
00208         else {
00209             PF_BroadcastNumber(0);
00210             return 0;
00211         }
00212     }
00213     else {
00214         if ( !PF_BroadcastNumber(0) )
00215             return 0;
00216     }
00217     /* Now the main part. */
00218     if (PF.me == MASTER){
00219         /*We have to have recovery files for the master and all the slaves:*/
00220         for(i=1; i<PF.numtasks;i++){
00221             const char *tmpnam = PF_recoveryfile('m', i, 0);
00222             if ( (fd = fopen(tmpnam, "r")) )
00223                 fclose(fd);
00224             else
00225                 break;
00226         }/*for(i=0; i<PF.numtasks;i++)*/
00227         if(i!=PF.numtasks){/*some files are absent*/
00228             int j;
00229             /*Send all slaves failure*/
00230             for(j=1; j<PF.numtasks;j++){
00231                 ret=PF_SendFile(j, NULL);
00232                 if(ret<0)
00233                     return(-1);
00234             }
00235             if(i==0)
00236                 return(0);/*Ok, no recovery files at all.*/
00237             /*The master recovery file exists but some slave files are absent*/
00238             MesPrint("The file %s exists but some of the slave recovery files are absent.",
00239                      RecoveryFilename());
00240             return(-1);
00241         }/*if(i!=PF.numtasks)*/
00242         /*All the files are here.*/
00243         /*Send all slaves success and files:*/
00244         for(i=1; i<PF.numtasks;i++){
00245             const char *tmpnam = PF_recoveryfile('m', i, 0);
00246             fd = fopen(tmpnam, "r");
00247             ret=PF_SendFile(i, fd);/*if fd==NULL, PF_SendFile seds to a slave the failure tag*/
00248             if(fd == NULL)
00249                 return(-1);
00250             else
00251                 fclose(fd);
00252             if(ret<0)
00253                 return(-1);
00254         }/*for(i=0; i<PF.numtasks;i++)*/
00255         return(1);
00256     }/*if (PF.me == MASTER)*/
00257     /*Slave:*/
00258     /*Get the answer from the master:*/
00259     fd=fopen(recoveryfile,"w");
00260     if(fd == NULL) {
00261         MesPrint("Failed to open %s in write mode in process %w", recoveryfile);
00262         return(-1);
```

```
00263     }
00264     ret=PF_RecvFile(MASTER,fd);
00265     if(ret<0)
00266         return(-1);
00267     fclose(fd);
00268     if(ret==0){
00269         /*Nothing is found by the master*/
00270         remove(recoveryfile);
00271         return(0);
00272     }
00273     /*Recovery file is successfully transferred*/
00274     return(1);
00275 }
00276 #endif
00277
00278 int CheckRecoveryFile(void)
00279 {
00280     int ret = 0;
00281 #ifdef WITHMPI
00282     ret = PF_CheckRecoveryFile();
00283 #else
00284     FILE *fd;
00285     if ( (fd = fopen(recoveryfile, "r")) ) {
00286         fclose(fd);
00287         ret = 1;
00288     }
00289 #endif
00290     if ( ret < 0 ){/*In ParFORM CheckRecoveryFile() may return a fatal error.*/
00291         MesPrint("Fail checking recovery file");
00292         Terminate(-1);
00293     }
00294     else if  ( ret > 0 ) {
00295         if ( AC.CheckpointFlag != -1 ) {
00296             /* recovery file exists but recovery option is not given */
00297 #ifdef WITHMPI
00298             if ( PF.me == MASTER ) {
00299 #endif
00300             MesPrint("The recovery file %s exists, but the recovery option -R has not been given!",
    RecoveryFilename());
00301             MesPrint("FORM will be terminated to avoid unintentional loss of data.");
00302             MesPrint("Delete the recovery file manually, if you want to start FORM without
    recovery.");
00303 #ifdef WITHMPI
00304             }
00305             if(PF.me != MASTER)
00306                 remove(RecoveryFilename());
00307 #endif
00308             Terminate(-1);
00309         }
00310     }
00311     else {
00312         if ( AC.CheckpointFlag == -1 ) {
00313             /* recovery option given but recovery file does not exist */
00314 #ifdef WITHMPI
00315             if ( PF.me == MASTER )
00316 #endif
00317             MesPrint("Option -R for recovery has been given, but the recovery file %s does not
    exist!", RecoveryFilename());
00318             Terminate(-1);
00319         }
00320     }
00321     return(ret);
00322 }
00323
00324 /*
00325     #] CheckRecoveryFile :
00326     #[ DeleteRecoveryFile :
00327 */
00328
00333 void DeleteRecoveryFile(void)
00334 {
00335     if ( done_snapshot ) {
00336         remove(recoveryfile);
00337 #ifdef WITHMPI
00338         if( PF.me == MASTER){
00339             int i;
00340             for(i=1; i<PF.numtasks;i++){
00341                 const char *tmpnam = PF_recoveryfile('m', i, 0);
00342                 remove(tmpnam);
00343             }/*for(i=1; i<PF.numtasks;i++)*/
00344             remove(storefile);
00345             remove(sortfile);
00346             remove(hidefile);
00347         }/*if( PF.me == MASTER)*/
00348 #else
00349         remove(storefile);
00350         remove(sortfile);
```

```
00351         remove(hidefile);
00352 #endif
00353     }
00354 }
00355
00356 /*
00357     #] DeleteRecoveryFile :
00358     #[ RecoveryFilename :
00359 */
00360
00364 char *RecoveryFilename(void)
00365 {
00366     return(recoveryfile);
00367 }
00368
00369 /*
00370     #] RecoveryFilename :
00371     #[ InitRecovery :
00372 */
00373
00377 static char *InitName(char *str, char *ext)
00378 {
00379     char *s, *d = str;
00380     if ( AM.TempDir ) {
00381         s = (char*)AM.TempDir;
00382         while ( *s ) { *d++ = *s++; }
00383         *d++ = SEPARATOR;
00384     }
00385     s = BaseName;
00386     while ( *s ) { *d++ = *s++; }
00387     *d++ = '.';
00388     s = ext;
00389     while ( *s ) { *d++ = *s++; }
00390     *d++ = 0;
00391     return d;
00392 }
00393
00399 void InitRecovery(void)
00400 {
00401     int lenpath = AM.TempDir ? strlen((char*)AM.TempDir)+1 : 0;
00402 #ifdef WITHMPI
00403     sprintf(BaseName,BASENAME_FMT,(PF.me == MASTER)?'m':'s',PF.me);
00404     /*Now BaseName has a form ?XXXXFORMrecv where ? == 'm' for master and 's' for slave,
00405         XXXX is a zero - padded PF.me*/
00406     PF_fmt_pos = lenpath;
00407 #endif
00408     recoveryfile = (char*)Malloc1(5*(lenpath+strlen(BaseName)+4+1),"InitRecovery");
00409     intermedfile = InitName(recoveryfile, "tmp");
00410     sortfile     = InitName(intermedfile, "XXX");
00411     hidefile     = InitName(sortfile, "out");
00412     storefile    = InitName(hidefile, "hid");
00413                    InitName(storefile, "str");
00414 }
00415
00416 /*
00417     #] InitRecovery :
00418     #[ Debugging :
00419 */
00420
00421 #ifdef PRINTDEBUG
00422
00423 void print_BYTE(void *p)
00424 {
00425     UBYTE h = (UBYTE)(*((UBYTE*)p) » 4);
00426     UBYTE l = (UBYTE)(*((UBYTE*)p) & 0x0F);
00427     if ( h > 9 ) h += 55; else h += 48;
00428     if ( l > 9 ) l += 55; else l += 48;
00429     printf("%c%c ", h, l);
00430 }
00431
00432 static void print_STR(UBYTE *p)
00433 {
00434     if ( p ) {
00435         MesPrint("%s", (char*)p);
00436     }
00437     else {
00438         MesPrint("NULL");
00439     }
00440 }
00441
00442 static void print_WORDB(WORD *buf, WORD *top)
00443 {
00444     LONG size = top-buf;
00445     int i;
00446     while ( size > 0 ) {
00447         if ( size > MAXPOSITIVE ) i = MAXPOSITIVE;
00448         else i = size;
```

```
00449         size -= i;
00450         MesPrint("%a",i,buf);
00451         buf += i;
00452     }
00453 }
00454
00455 static void print_VOIDP(void *p, size_t size)
00456 {
00457     int i;
00458     if ( p ) {
00459         while ( size > 0 ) {
00460             if ( size > MAXPOSITIVE ) i = MAXPOSITIVE;
00461             else i = size;
00462             size -= i;
00463             MesPrint("%b",i,(UBYTE *)p);
00464             p = ((UBYTE *)p)+i;
00465         }
00466     }
00467     else {
00468         MesPrint("NULL");
00469     }
00470 }
00471
00472 static void print_CHARS(UBYTE *p, size_t size)
00473 {
00474     int i;
00475     while ( size > 0 ) {
00476         if ( size > MAXPOSITIVE ) i = MAXPOSITIVE;
00477         else i = size;
00478         size -= i;
00479         MesPrint("%C",i,(char *)p);
00480         p += i;
00481     }
00482 }
00483
00484 static void print_WORDV(WORD *p, size_t size)
00485 {
00486     int i;
00487     if ( p ) {
00488         while ( size > 0 ) {
00489             if ( size > MAXPOSITIVE ) i = MAXPOSITIVE;
00490             else i = size;
00491             size -= i;
00492             MesPrint("%a",i,p);
00493             p += i;
00494         }
00495     }
00496     else {
00497         MesPrint("NULL");
00498     }
00499 }
00500
00501 static void print_INTV(int *p, size_t size)
00502 {
00503     int iarray[8];
00504     WORD i = 0;
00505     if ( p ) {
00506         while ( size > 0 ) {
00507             if ( i >= 8 ) {
00508                 MesPrint("%I",i,iarray);
00509                 i = 0;
00510             }
00511             iarray[i++] = *p++;
00512             size--;
00513         }
00514         if ( i > 0 ) MesPrint("%I",i,iarray);
00515     }
00516     else {
00517         MesPrint("NULL");
00518     }
00519 }
00520
00521 static void print_LONGV(LONG *p, size_t size)
00522 {
00523     LONG larray[8];
00524     WORD i = 0;
00525     if ( p ) {
00526         while ( size > 0 ) {
00527             if ( i >= 8 ) {
00528                 MesPrint("%I",i,larray);
00529                 i = 0;
00530             }
00531             larray[i++] = *p++;
00532             size--;
00533         }
00534         if ( i > 0 ) MesPrint("%I",i,larray);
00535     }
```

```
00536     else {
00537         MesPrint("NULL");
00538     }
00539 }
00540
00541 static void print_PRELOAD(PRELOAD *l)
00542 {
00543     if ( l->size ) {
00544         print_CHARS(l->buffer, l->size);
00545     }
00546     MesPrint("%ld", l->size);
00547 }
00548
00549 static void print_PREVAR(PREVAR *l)
00550 {
00551     MesPrint("%s", l->name);
00552     print_STR(l->value);
00553     if ( l->nargs ) print_STR(l->argnames);
00554     MesPrint("%d", l->nargs);
00555     MesPrint("%d", l->wildarg);
00556 }
00557
00558 static void print_DOLLARS(DOLLARS l)
00559 {
00560     print_VOIDP(l->where, l->size);
00561     MesPrint("%ld", l->size);
00562     MesPrint("%ld", l->name);
00563     MesPrint("%s", AC.dollarnames->namebuffer+l->name);
00564     MesPrint("%d", l->type);
00565     MesPrint("%d", l->node);
00566     MesPrint("%d", l->index);
00567     MesPrint("%d", l->zero);
00568     MesPrint("%d", l->numdummies);
00569     MesPrint("%d", l->nfactors);
00570 }
00571
00572 static void print_LIST(LIST *l)
00573 {
00574     print_VOIDP(l->lijst, l->size);
00575     MesPrint("%s", l->message);
00576     MesPrint("%d", l->num);
00577     MesPrint("%d", l->maxnum);
00578     MesPrint("%d", l->size);
00579     MesPrint("%d", l->numglobal);
00580     MesPrint("%d", l->numtemp);
00581     MesPrint("%d", l->numclear);
00582 }
00583
00584 static void print_DOLOOP(DOLOOP *l)
00585 {
00586     print_PRELOAD(&(l->p));
00587     print_STR(l->name);
00588     if ( l->type != NUMERICALLOOP ) {
00589         print_STR(l->vars);
00590     }
00591     print_STR(l->contents);
00592     if ( l->type != LISTEDLOOP && l->type != NUMERICALLOOP ) {
00593         print_STR(l->dollarname);
00594     }
00595     MesPrint("%l", l->startlinenumber);
00596     MesPrint("%l", l->firstnum);
00597     MesPrint("%l", l->lastnum);
00598     MesPrint("%l", l->incnum);
00599     MesPrint("%d", l->type);
00600     MesPrint("%d", l->NoShowInput);
00601     MesPrint("%d", l->errorsinloop);
00602     MesPrint("%d", l->firstloopcall);
00603 }
00604
00605 static void print_PROCEDURE(PROCEDURE *l)
00606 {
00607     if ( l->loadmode != 1 ) {
00608         print_PRELOAD(&(l->p));
00609     }
00610     print_STR(l->name);
00611     MesPrint("%d", l->loadmode);
00612 }
00613
00614 static void print_NAMETREE(NAMETREE *t)
00615 {
00616     int i;
00617     for ( i=0; i<t->nodefill; ++i ) {
00618         MesPrint("%l %d %d %d %d %d %d\n", t->namenode[i].name,
00619             t->namenode[i].parent, t->namenode[i].left, t->namenode[i].right,
00620             t->namenode[i].balance, t->namenode[i].type, t->namenode[i].number );
00621     }
00622     print_CHARS(t->namebuffer, t->namefill);
```

```
00623     MesPrint("%l", t->namesize);
00624     MesPrint("%l", t->namefill);
00625     MesPrint("%l", t->nodesize);
00626     MesPrint("%l", t->nodefill);
00627     MesPrint("%l", t->oldnamefill);
00628     MesPrint("%l", t->oldnodefill);
00629     MesPrint("%l", t->globalnamefill);
00630     MesPrint("%l", t->globalnodefill);
00631     MesPrint("%l", t->clearnamefill);
00632     MesPrint("%l", t->clearnodefill);
00633     MesPrint("%d", t->headnode);
00634 }
00635
00636 void print_CBUF(CBUF *c)
00637 {
00638     int i;
00639     print_WORDV(c->Buffer, c->BufferSize);
00640     /*
00641     MesPrint("%x", c->Buffer);
00642     MesPrint("%x", c->lhs);
00643     MesPrint("%x", c->rhs);
00644     */
00645     for ( i=0; i<c->numlhs; ++i ) {
00646         if ( c->lhs[i]) MesPrint("%d", *(c->lhs[i]));
00647     }
00648     for ( i=0; i<c->numrhs; ++i ) {
00649         if ( c->rhs[i]) MesPrint("%d", *(c->rhs[i]));
00650     }
00651     MesPrint("%l", *c->CanCommu);
00652     MesPrint("%l", *c->NumTerms);
00653     MesPrint("%d", *c->numdum);
00654     for ( i=0; i<c->MaxTreeSize; ++i ) {
00655         MesPrint("%d %d %d %d %d", c->boomlijst[i].parent, c->boomlijst[i].left,
    c->boomlijst[i].right,
00656                 c->boomlijst[i].value, c->boomlijst[i].blnce);
00657     }
00658 }
00659
00660 static void print_STREAM(STREAM *t)
00661 {
00662     print_CHARS(t->buffer, t->inbuffer);
00663     MesPrint("%l", (LONG)(t->pointer-t->buffer));
00664     print_STR(t->FoldName);
00665     print_STR(t->name);
00666     if ( t->type == PREVARSTREAM || t->type == DOLLARSTREAM ) {
00667         print_STR(t->pname);
00668     }
00669     MesPrint("%l", (LONG)t->fileposition);
00670     MesPrint("%l", (LONG)t->linenumber);
00671     MesPrint("%l", (LONG)t->prevline);
00672     MesPrint("%l", t->buffersize);
00673     MesPrint("%l", t->bufferposition);
00674     MesPrint("%l", t->inbuffer);
00675     MesPrint("%d", t->previous);
00676     MesPrint("%d", t->handle);
00677     switch ( t->type ) {
00678         case FILESTREAM: MesPrint("%d == FILESTREAM", t->type); break;
00679         case PREVARSTREAM: MesPrint("%d == PREVARSTREAM", t->type); break;
00680         case PREREADSTREAM: MesPrint("%d == PREREADSTREAM", t->type); break;
00681         case PIPESTREAM: MesPrint("%d == PIPESTREAM", t->type); break;
00682         case PRECALCSTREAM: MesPrint("%d == PRECALCSTREAM", t->type); break;
00683         case DOLLARSTREAM: MesPrint("%d == DOLLARSTREAM", t->type); break;
00684         case PREREADSTREAM2: MesPrint("%d == PREREADSTREAM2", t->type); break;
00685         case EXTERNALCHANNELSTREAM: MesPrint("%d == EXTERNALCHANNELSTREAM", t->type); break;
00686         case PREREADSTREAM3: MesPrint("%d == PREREADSTREAM3", t->type); break;
00687         default: MesPrint("%d == UNKNOWN", t->type);
00688     }
00689 }
00690
00691 static void print_M()
00692 {
00693     MesPrint("%%%% M_const");
00694     MesPrint("%d", *AM.gcmod);
00695     MesPrint("%d", *AM.gpowmod);
00696     print_STR(AM.TempDir);
00697     print_STR(AM.TempSortDir);
00698     print_STR(AM.IncDir);
00699     print_STR(AM.InputFileName);
00700     print_STR(AM.LogFileName);
00701     print_STR(AM.OutBuffer);
00702     print_STR(AM.Path);
00703     print_STR(AM.SetupDir);
00704     print_STR(AM.SetupFile);
00705     MesPrint("--MARK  1");
00706     MesPrint("%l", (LONG)BASEPOSITION(AM.zeropos));
00707 #ifdef WITHPTHREADS
00708     MesPrint("%l", AM.ThreadScratSize);
```

```
00709     MesPrint("%l", AM.ThreadScratOutSize);
00710 #endif
00711     MesPrint("%l", AM.MaxTer);
00712     MesPrint("%l", AM.CompressSize);
00713     MesPrint("%l", AM.ScratSize);
00714     MesPrint("%l", AM.SizeStoreCache);
00715     MesPrint("%l", AM.MaxStreamSize);
00716     MesPrint("%l", AM.SIOsize);
00717     MesPrint("%l", AM.SLargeSize);
00718     MesPrint("%l", AM.SSmallEsize);
00719     MesPrint("%l", AM.SSmallSize);
00720     MesPrint("--MARK  2");
00721     MesPrint("%l", AM.STermsInSmall);
00722     MesPrint("%l", AM.MaxBracketBufferSize);
00723     MesPrint("%l", AM.hProcessBucketSize);
00724     MesPrint("%l", AM.gProcessBucketSize);
00725     MesPrint("%l", AM.shmWinSize);
00726     MesPrint("%l", AM.OldChildTime);
00727     MesPrint("%l", AM.OldSecTime);
00728     MesPrint("%l", AM.OldMilliTime);
00729     MesPrint("%l", AM.WorkSize);
00730     MesPrint("%l", AM.gThreadBucketSize);
00731     MesPrint("--MARK  3");
00732     MesPrint("%l", AM.ggThreadBucketSize);
00733     MesPrint("%d", AM.FileOnlyFlag);
00734     MesPrint("%d", AM.Interact);
00735     MesPrint("%d", AM.MaxParLevel);
00736     MesPrint("%d", AM.OutBufSize);
00737     MesPrint("%d", AM.SMaxFpatches);
00738     MesPrint("%d", AM.SMaxPatches);
00739     MesPrint("%d", AM.StdOut);
00740     MesPrint("%d", AM.ginsidefirst);
00741     MesPrint("%d", AM.gDefDim);
00742     MesPrint("%d", AM.gDefDim4);
00743     MesPrint("--MARK  4");
00744     MesPrint("%d", AM.NumFixedSets);
00745     MesPrint("%d", AM.NumFixedFunctions);
00746     MesPrint("%d", AM.rbufnum);
00747     MesPrint("%d", AM.dbufnum);
00748     MesPrint("%d", AM.SkipClears);
00749     MesPrint("%d", AM.gfunpowers);
00750     MesPrint("%d", AM.gStatsFlag);
00751     MesPrint("%d", AM.gNamesFlag);
00752     MesPrint("%d", AM.gCodesFlag);
00753     MesPrint("%d", AM.gTokensWriteFlag);
00754     MesPrint("%d", AM.gSortType);
00755     MesPrint("%d", AM.gproperorderflag);
00756     MesPrint("--MARK  5");
00757     MesPrint("%d", AM.hparallelflag);
00758     MesPrint("%d", AM.gparallelflag);
00759     MesPrint("%d", AM.totalnumberofthreads);
00760     MesPrint("%d", AM.gSizeCommuteInSet);
00761     MesPrint("%d", AM.gThreadStats);
00762     MesPrint("%d", AM.ggThreadStats);
00763     MesPrint("%d", AM.gFinalStats);
00764     MesPrint("%d", AM.ggFinalStats);
00765     MesPrint("%d", AM.gThreadsFlag);
00766     MesPrint("%d", AM.ggThreadsFlag);
00767     MesPrint("%d", AM.gThreadBalancing);
00768     MesPrint("%d", AM.ggThreadBalancing);
00769     MesPrint("%d", AM.gThreadSortFileSynch);
00770     MesPrint("%d", AM.ggThreadSortFileSynch);
00771     MesPrint("%d", AM.gProcessStats);
00772     MesPrint("%d", AM.ggProcessStats);
00773     MesPrint("%d", AM.gOldParallelStats);
00774     MesPrint("%d", AM.ggOldParallelStats);
00775     MesPrint("%d", AM.gWTimeStatsFlag);
00776     MesPrint("%d", AM.ggWTimeStatsFlag);
00777     MesPrint("%d", AM.maxFlevels);
00778     MesPrint("--MARK  6");
00779     MesPrint("%d", AM.resetTimeOnClear);
00780     MesPrint("%d", AM.gcNumDollars);
00781     MesPrint("%d", AM.MultiRun);
00782     MesPrint("%d", AM.gNoSpacesInNumbers);
00783     MesPrint("%d", AM.ggNoSpacesInNumbers);
00784     MesPrint("%d", AM.MaxTal);
00785     MesPrint("%d", AM.IndDum);
00786     MesPrint("%d", AM.DumInd);
00787     MesPrint("%d", AM.WilInd);
00788     MesPrint("%d", AM.gncmod);
00789     MesPrint("%d", AM.gnpowmod);
00790     MesPrint("%d", AM.gmodmode);
00791     MesPrint("--MARK  7");
00792     MesPrint("%d", AM.gUnitTrace);
00793     MesPrint("%d", AM.gOutputMode);
00794     MesPrint("%d", AM.gCnumpows);
00795     MesPrint("%d", AM.gOutputSpaces);
```

```
00796     MesPrint("%d", AM.gOutNumberType);
00797     MesPrint("%d %d %d %d", AM.gUniTrace[0], AM.gUniTrace[1], AM.gUniTrace[2], AM.gUniTrace[3]);
00798     MesPrint("%d", AM.MaxWildcards);
00799     MesPrint("%d", AM.mTraceDum);
00800     MesPrint("%d", AM.OffsetIndex);
00801     MesPrint("%d", AM.OffsetVector);
00802     MesPrint("%d", AM.RepMax);
00803     MesPrint("%d", AM.LogType);
00804     MesPrint("%d", AM.ggStatsFlag);
00805     MesPrint("%d", AM.gLineLength);
00806     MesPrint("%d", AM.qError);
00807     MesPrint("--MARK  8");
00808     MesPrint("%d", AM.FortranCont);
00809     MesPrint("%d", AM.HoldFlag);
00810     MesPrint("%d %d %d %d %d", AM.Ordering[0], AM.Ordering[1], AM.Ordering[2], AM.Ordering[3],
      AM.Ordering[4]);
00811     MesPrint("%d %d %d %d %d", AM.Ordering[5], AM.Ordering[6], AM.Ordering[7], AM.Ordering[8],
      AM.Ordering[9]);
00812     MesPrint("%d %d %d %d %d", AM.Ordering[10], AM.Ordering[11], AM.Ordering[12], AM.Ordering[13],
      AM.Ordering[14]);
00813     MesPrint("%d", AM.silent);
00814     MesPrint("%d", AM.tracebackflag);
00815     MesPrint("%d", AM.expnum);
00816     MesPrint("%d", AM.denomnum);
00817     MesPrint("%d", AM.facnum);
00818     MesPrint("%d", AM.invfacnum);
00819     MesPrint("%d", AM.sumnum);
00820     MesPrint("%d", AM.sumpnum);
00821     MesPrint("--MARK  9");
00822     MesPrint("%d", AM.OldOrderFlag);
00823     MesPrint("%d", AM.termfunnum);
00824     MesPrint("%d", AM.matchfunnum);
00825     MesPrint("%d", AM.countfunnum);
00826     MesPrint("%d", AM.gPolyFun);
00827     MesPrint("%d", AM.gPolyFunInv);
00828     MesPrint("%d", AM.gPolyFunType);
00829     MesPrint("%d", AM.gPolyFunExp);
00830     MesPrint("%d", AM.gPolyFunVar);
00831     MesPrint("%d", AM.gPolyFunPow);
00832     MesPrint("--MARK 10");
00833     MesPrint("%d", AM.dollarzero);
00834     MesPrint("%d", AM.atstartup);
00835     MesPrint("%d", AM.exitflag);
00836     MesPrint("%d", AM.NumStoreCaches);
00837     MesPrint("%d", AM.gIndentSpace);
00838     MesPrint("%d", AM.ggIndentSpace);
00839     MesPrint("%d", AM.gShortStatsMax);
00840     MesPrint("%d", AM.ggShortStatsMax);
00841     MesPrint("--MARK 11");
00842     MesPrint("%d", AM.FromStdin);
00843     MesPrint("%d", AM.IgnoreDeprecation);
00844     MesPrint("%%%% END M_const");
00845 /*  fflush(0); */
00846 }
00847
00848 static void print_P()
00849 {
00850     int i;
00851     MesPrint("%%%% P_const");
00852     print_LIST(&AP.DollarList);
00853     for ( i=0; i<AP.DollarList.num; ++i ) {
00854         print_DOLLARS(&(Dollars[i]));
00855     }
00856     MesPrint("--MARK  1");
00857     print_LIST(&AP.PreVarList);
00858     for ( i=0; i<AP.PreVarList.num; ++i ) {
00859         print_PREVAR(&(PreVar[i]));
00860     }
00861     MesPrint("--MARK  2");
00862     print_LIST(&AP.LoopList);
00863     for ( i=0; i<AP.LoopList.num; ++i ) {
00864         print_DOLOOP(&(DoLoops[i]));
00865     }
00866     MesPrint("--MARK  3");
00867     print_LIST(&AP.ProcList);
00868     for ( i=0; i<AP.ProcList.num; ++i ) {
00869         print_PROCEDURE(&(Procedures[i]));
00870     }
00871     MesPrint("--MARK  4");
00872     for ( i=0; i<=AP.PreSwitchLevel; ++i ) {
00873         print_STR(AP.PreSwitchStrings[i]);
00874     }
00875     MesPrint("%l", AP.preStop-AP.preStart);
00876     if ( AP.preFill ) MesPrint("%l", AP.preFill-AP.preStart);
00877     print_CHARS(AP.preStart, AP.pSize);
00878     MesPrint("%s", AP.procedureExtension);
00879     MesPrint("%s", AP.cprocedureExtension);
```

```
00880     print_INTV(AP.PreIfStack, AP.MaxPreIfLevel);
00881     print_INTV(AP.PreSwitchModes, AP.NumPreSwitchStrings+1);
00882     print_INTV(AP.PreTypes, AP.NumPreTypes+1);
00883     MesPrint("%d", AP.PreAssignFlag);
00884     MesPrint("--MARK  5");
00885     MesPrint("%d", AP.PreContinuation);
00886     MesPrint("%l", AP.InOutBuf);
00887     MesPrint("%l", AP.pSize);
00888     MesPrint("%d", AP.PreproFlag);
00889     MesPrint("%d", AP.iBufError);
00890     MesPrint("%d", AP.PreOut);
00891     MesPrint("%d", AP.PreSwitchLevel);
00892     MesPrint("%d", AP.NumPreSwitchStrings);
00893     MesPrint("%d", AP.MaxPreTypes);
00894     MesPrint("--MARK  6");
00895     MesPrint("%d", AP.NumPreTypes);
00896     MesPrint("%d", AP.DelayPrevar);
00897     MesPrint("%d", AP.AllowDelay);
00898     MesPrint("%d", AP.lhdollarerror);
00899     MesPrint("%d", AP.eat);
00900     MesPrint("%d", AP.gNumPre);
00901     MesPrint("%d", AP.PreDebug);
00902     MesPrint("--MARK  7");
00903     MesPrint("%d", AP.DebugFlag);
00904     MesPrint("%d", AP.preError);
00905     MesPrint("%C", 1, &(AP.ComChar));
00906     MesPrint("%C", 1, &(AP.cComChar));
00907     MesPrint("%%%% END P_const");
00908 /*  fflush(0); */
00909 }
00910
00911 static void print_C()
00912 {
00913     int i;
00914     UBYTE buf[40], *t;
00915     MesPrint("%%%% C_const");
00916     for ( i=0; i<32; ++i ) {
00917         t = buf;
00918         t = NumCopy((WORD)(AC.separators[i].bit_7),t);
00919         t = NumCopy((WORD)(AC.separators[i].bit_6),t);
00920         t = NumCopy((WORD)(AC.separators[i].bit_5),t);
00921         t = NumCopy((WORD)(AC.separators[i].bit_4),t);
00922         t = NumCopy((WORD)(AC.separators[i].bit_3),t);
00923         t = NumCopy((WORD)(AC.separators[i].bit_2),t);
00924         t = NumCopy((WORD)(AC.separators[i].bit_1),t);
00925         t = NumCopy((WORD)(AC.separators[i].bit_0),t);
00926         MesPrint("%s ",buf);
00927     }
00928     print_NAMETREE(AC.dollarnames);
00929     print_NAMETREE(AC.exprnames);
00930     print_NAMETREE(AC.varnames);
00931     MesPrint("--MARK  1");
00932     print_LIST(&AC.ChannelList);
00933     for ( i=0; i<AC.ChannelList.num; ++i ) {
00934         MesPrint("%s %d", channels[i].name, channels[i].handle);
00935     }
00936     MesPrint("--MARK  2");
00937     print_LIST(&AC.DubiousList);
00938     MesPrint("--MARK  3");
00939     print_LIST(&AC.FunctionList);
00940     for ( i=0; i<AC.FunctionList.num; ++i ) {
00941         if ( functions[i].tabl ) {
00942
00943         }
00944         MesPrint("%l", functions[i].symminfo);
00945         MesPrint("%l", functions[i].name);
00946         MesPrint("%d", functions[i].namesize);
00947     }
00948     MesPrint("--MARK  4");
00949     print_LIST(&AC.ExpressionList);
00950     print_LIST(&AC.IndexList);
00951     print_LIST(&AC.SetElementList);
00952     print_LIST(&AC.SetList);
00953     MesPrint("--MARK  5");
00954     print_LIST(&AC.SymbolList);
00955     print_LIST(&AC.VectorList);
00956     print_LIST(&AC.PotModDolList);
00957     print_LIST(&AC.ModOptDolList);
00958     print_LIST(&AC.TableBaseList);
00959
00960 /*
00961     print_LIST(&AC.cbufList);
00962     for ( i=0; i<AC.cbufList.num; ++i ) {
00963         MesPrint("cbufList.num == %d", i);
00964         print_CBUF(cbuf+i);
00965     }
00966     MesPrint("%d", AC.cbufnum);
```

```
00967 */
00968     MesPrint("--MARK  6");
00969 
00970     print_LIST(&AC.AutoSymbolList);
00971     print_LIST(&AC.AutoIndexList);
00972     print_LIST(&AC.AutoVectorList);
00973     print_LIST(&AC.AutoFunctionList);
00974 
00975     print_NAMETREE(AC.autonames);
00976     MesPrint("--MARK  7");
00977 
00978     print_LIST(AC.Symbols);
00979     print_LIST(AC.Indices);
00980     print_LIST(AC.Vectors);
00981     print_LIST(AC.Functions);
00982     MesPrint("--MARK  8");
00983 
00984     print_NAMETREE(*AC.activenames);
00985 
00986     MesPrint("--MARK  9");
00987 
00988     MesPrint("%d", AC.AutoDeclareFlag);
00989 
00990     for ( i=0; i<AC.NumStreams; ++i ) {
00991         MesPrint("Stream %d\n", i);
00992         print_STREAM(AC.Streams+i);
00993     }
00994     print_STREAM(AC.CurrentStream);
00995     MesPrint("--MARK 10");
00996 
00997     print_LONGV(AC.termstack, AC.maxtermlevel);
00998     print_LONGV(AC.termsortstack, AC.maxtermlevel);
00999     print_VOIDP(AC.cmod, AM.MaxTal*4*sizeof(UWORD));
01000     print_WORDV((WORD *)(AC.cmod), 1);
01001     print_WORDV((WORD *)(AC.powmod), 1);
01002     print_WORDV((WORD*)AC.modpowers, 1);
01003     print_WORDV((WORD*)AC.halfmod, 1);
01004     MesPrint("--MARK 10-2");
01005     /*
01006     print_WORDV(AC.ProtoType, AC.ProtoType[1]);
01007     print_WORDV(AC.WildC, 1);
01008     */
01009 
01010     MesPrint("--MARK 11");
01011     /* IfHeap ... Labels */
01012 
01013     print_CHARS((UBYTE*)AC.tokens, AC.toptokens-AC.tokens);
01014     MesPrint("%l", AC.endoftokens-AC.tokens);
01015     print_WORDV(AC.tokenarglevel, AM.MaxParLevel);
01016     print_WORDV((WORD*)AC.modinverses, ABS(AC.ncmod));
01017 #ifdef WITHPTHREADS
01018     print_LONGV(AC.inputnumbers, AC.sizepfirstnum+AC.sizepfirstnum*sizeof(WORD)/sizeof(LONG));
01019     print_WORDV(AC.pfirstnum, 1);
01020 #endif
01021     MesPrint("--MARK 12");
01022     print_LONGV(AC.argstack, MAXNEST);
01023     print_LONGV(AC.insidestack, MAXNEST);
01024     print_LONGV(AC.inexprstack, MAXNEST);
01025     MesPrint("%l", AC.iBufferSize);
01026     MesPrint("%l", AC.TransEname);
01027     MesPrint("%l", AC.ProcessBucketSize);
01028     MesPrint("%l", AC.mProcessBucketSize);
01029     MesPrint("%l", AC.CModule);
01030     MesPrint("%l", AC.ThreadBucketSize);
01031     MesPrint("%d", AC.NoShowInput);
01032     MesPrint("%d", AC.ShortStats);
01033     MesPrint("%d", AC.compiletype);
01034     MesPrint("%d", AC.firstconstindex);
01035     MesPrint("%d", AC.insidefirst);
01036     MesPrint("%d", AC.minsidefirst);
01037     MesPrint("%d", AC.wildflag);
01038     MesPrint("%d", AC.NumLabels);
01039     MesPrint("%d", AC.MaxLabels);
01040     MesPrint("--MARK 13");
01041     MesPrint("%d", AC.lDefDim);
01042     MesPrint("%d", AC.lDefDim4);
01043     MesPrint("%d", AC.NumWildcardNames);
01044     MesPrint("%d", AC.WildcardBufferSize);
01045     MesPrint("%d", AC.MaxIf);
01046     MesPrint("%d", AC.NumStreams);
01047     MesPrint("%d", AC.MaxNumStreams);
01048     MesPrint("%d", AC.firstctypemessage);
01049     MesPrint("%d", AC.tablecheck);
01050     MesPrint("%d", AC.idoption);
01051     MesPrint("%d", AC.BottomLevel);
01052     MesPrint("%d", AC.CompileLevel);
01053     MesPrint("%d", AC.TokensWriteFlag);
```

```
01054     MesPrint("%d", AC.UnsureDollarMode);
01055     MesPrint("%d", AC.outsidefun);
01056     MesPrint("%d", AC.funpowers);
01057     MesPrint("--MARK 14");
01058     MesPrint("%d", AC.WarnFlag);
01059     MesPrint("%d", AC.StatsFlag);
01060     MesPrint("%d", AC.NamesFlag);
01061     MesPrint("%d", AC.CodesFlag);
01062     MesPrint("%d", AC.TokensWriteFlag);
01063     MesPrint("%d", AC.SetupFlag);
01064     MesPrint("%d", AC.SortType);
01065     MesPrint("%d", AC.lSortType);
01066     MesPrint("%d", AC.ThreadStats);
01067     MesPrint("%d", AC.FinalStats);
01068     MesPrint("%d", AC.ThreadsFlag);
01069     MesPrint("%d", AC.ThreadBalancing);
01070     MesPrint("%d", AC.ThreadSortFileSynch);
01071     MesPrint("%d", AC.ProcessStats);
01072     MesPrint("%d", AC.OldParallelStats);
01073     MesPrint("%d", AC.WTimeStatsFlag);
01074     MesPrint("%d", AC.BracketNormalize);
01075     MesPrint("%d", AC.maxtermlevel);
01076     MesPrint("%d", AC.dumnumflag);
01077     MesPrint("--MARK 15");
01078     MesPrint("%d", AC.bracketindexflag);
01079     MesPrint("%d", AC.parallelflag);
01080     MesPrint("%d", AC.mparallelflag);
01081     MesPrint("%d", AC.properorderflag);
01082     MesPrint("%d", AC.vetofilling);
01083     MesPrint("%d", AC.tablefilling);
01084     MesPrint("%d", AC.vetotablebasefill);
01085     MesPrint("%d", AC.exprfillwarning);
01086     MesPrint("%d", AC.lhdollarflag);
01087     MesPrint("%d", AC.NoCompress);
01088 #ifdef WITHPTHREADS
01089     MesPrint("%d", AC.numpfirstnum);
01090     MesPrint("%d", AC.sizepfirstnum);
01091 #endif
01092     MesPrint("%d", AC.RepLevel);
01093     MesPrint("%d", AC.arglevel);
01094     MesPrint("%d", AC.insidelevel);
01095     MesPrint("%d", AC.inexprlevel);
01096     MesPrint("%d", AC.termlevel);
01097     MesPrint("--MARK 16");
01098     print_WORDV(AC.argsumcheck, MAXNEST);
01099     print_WORDV(AC.insidesumcheck, MAXNEST);
01100     print_WORDV(AC.inexprsumcheck, MAXNEST);
01101     MesPrint("%d", AC.MustTestTable);
01102     MesPrint("%d", AC.DumNum);
01103     MesPrint("%d", AC.ncmod);
01104     MesPrint("%d", AC.npowmod);
01105     MesPrint("%d", AC.modmode);
01106     MesPrint("%d", AC.nhalfmod);
01107     MesPrint("%d", AC.DirtPow);
01108     MesPrint("%d", AC.lUnitTrace);
01109     MesPrint("%d", AC.NwildC);
01110     MesPrint("%d", AC.ComDefer);
01111     MesPrint("%d", AC.CollectFun);
01112     MesPrint("%d", AC.AltCollectFun);
01113     MesPrint("--MARK 17");
01114     MesPrint("%d", AC.OutputMode);
01115     MesPrint("%d", AC.Cnumpows);
01116     MesPrint("%d", AC.OutputSpaces);
01117     MesPrint("%d", AC.OutNumberType);
01118     print_WORDV(AC.lUniTrace, 4);
01119     print_WORDV(AC.RepSumCheck, MAXREPEAT);
01120     MesPrint("%d", AC.DidClean);
01121     MesPrint("%d", AC.IfLevel);
01122     MesPrint("%d", AC.WhileLevel);
01123     print_WORDV(AC.IfSumCheck, (AC.MaxIf+1));
01124     MesPrint("%d", AC.LogHandle);
01125     MesPrint("%d", AC.LineLength);
01126     MesPrint("%d", AC.StoreHandle);
01127     MesPrint("%d", AC.HideLevel);
01128     MesPrint("%d", AC.lPolyFun);
01129     MesPrint("%d", AC.lPolyFunInv);
01130     MesPrint("%d", AC.lPolyFunType);
01131     MesPrint("%d", AC.lPolyFunExp);
01132     MesPrint("%d", AC.lPolyFunVar);
01133     MesPrint("%d", AC.lPolyFunPow);
01134     MesPrint("%d", AC.SymChangeFlag);
01135     MesPrint("%d", AC.CollectPercentage);
01136     MesPrint("%d", AC.ShortStatsMax);
01137     MesPrint("--MARK 18");
01138
01139     print_CHARS(AC.Commercial, COMMERCIALSIZE+2);
01140
```

```
01141     MesPrint("%", AC.CheckpointFlag);
01142     MesPrint("%l", AC.CheckpointStamp);
01143     print_STR((unsigned char*)AC.CheckpointRunAfter);
01144     print_STR((unsigned char*)AC.CheckpointRunBefore);
01145     MesPrint("%l", AC.CheckpointInterval);
01146 
01147     MesPrint("%%%% END C_const");
01148 /*  fflush(0); */
01149 }
01150 
01151 static void print_R()
01152 {
01153     GETIDENTITY
01154     int i;
01155     MesPrint("%%%% R_const");
01156     MesPrint("%l", (LONG)(AR.infile-AR.Fscr));
01157     MesPrint("%s", AR.infile->name);
01158     MesPrint("%l", (LONG)(AR.outfile-AR.Fscr));
01159     MesPrint("%s", AR.outfile->name);
01160     MesPrint("%l", AR.hidefile-AR.Fscr);
01161     MesPrint("%s", AR.hidefile->name);
01162     for ( i=0; i<3; ++i ) {
01163         MesPrint("FSCR %d", i);
01164         print_WORDB(AR.Fscr[i].PObuffer, AR.Fscr[i].POfull);
01165     }
01166     /* ... */
01167     MesPrint("%l", AR.OldTime);
01168     MesPrint("%l", AR.InInBuf);
01169     MesPrint("%l", AR.InHiBuf);
01170     MesPrint("%l", AR.pWorkSize);
01171     MesPrint("%l", AR.lWorkSize);
01172     MesPrint("%l", AR.posWorkSize);
01173     MesPrint("%d", AR.NoCompress);
01174     MesPrint("%d", AR.gzipCompress);
01175     MesPrint("%d", AR.Cnumlhs);
01176 #ifdef WITHPTHREADS
01177     MesPrint("%d", AR.exprtodo);
01178 #endif
01179     MesPrint("%d", AR.GetFile);
01180     MesPrint("%d", AR.KeptInHold);
01181     MesPrint("%d", AR.BracketOn);
01182     MesPrint("%d", AR.MaxBracket);
01183     MesPrint("%d", AR.CurDum);
01184     MesPrint("%d", AR.DeferFlag);
01185     MesPrint("%d", AR.TePos);
01186     MesPrint("%d", AR.sLevel);
01187     MesPrint("%d", AR.Stage4Name);
01188     MesPrint("%d", AR.GetOneFile);
01189     MesPrint("%d", AR.PolyFun);
01190     MesPrint("%d", AR.PolyFunInv);
01191     MesPrint("%d", AR.PolyFunType);
01192     MesPrint("%d", AR.PolyFunExp);
01193     MesPrint("%d", AR.PolyFunVar);
01194     MesPrint("%d", AR.PolyFunPow);
01195     MesPrint("%d", AR.Eside);
01196     MesPrint("%d", AR.MaxDum);
01197     MesPrint("%d", AR.level);
01198     MesPrint("%d", AR.expchanged);
01199     MesPrint("%d", AR.expflags);
01200     MesPrint("%d", AR.CurExpr);
01201     MesPrint("%d", AR.SortType);
01202     MesPrint("%d", AR.ShortSortCount);
01203     MesPrint("%%%% END R_const");
01204 /*  fflush(0); */
01205 }
01206 
01207 #endif /* ifdef PRINTDEBUG */
01208 
01209 /*
01210     #] Debugging :
01211     #[ Cached file operation functions :
01212 */
01213 
01214 #define CACHED_SNAPSHOT
01215 
01216 #define CACHE_SIZE 4096
01217 
01218 #ifdef CACHED_SNAPSHOT
01219 unsigned char cache_buffer[CACHE_SIZE];
01220 size_t cache_fill = 0;
01221 
01222 size_t fwrite_cached(const void *ptr, size_t size, size_t nmemb, FILE *fd)
01223 {
01224     size_t fullsize = size*nmemb;
01225     if ( fullsize+cache_fill >= CACHE_SIZE ) {
01226         size_t overlap = CACHE_SIZE-cache_fill;
01227         memcpy(cache_buffer+cache_fill, (unsigned char*)ptr, overlap);
```

```
01228         if ( fwrite(cache_buffer, 1, CACHE_SIZE, fd) != CACHE_SIZE ) return 0;
01229         fullsize -= overlap;
01230         if ( fullsize >= CACHE_SIZE ) {
01231             cache_fill = fullsize % CACHE_SIZE;
01232             if ( cache_fill ) memcpy(cache_buffer, (unsigned char*)ptr+overlap+fullsize-cache_fill,
      cache_fill);
01233             if ( fwrite((unsigned char*)ptr+overlap, 1, fullsize-cache_fill, fd) !=
      fullsize-cache_fill ) return 0;
01234         }
01235         else {
01236             memcpy(cache_buffer, (unsigned char*)ptr+overlap, fullsize);
01237             cache_fill = fullsize;
01238         }
01239     }
01240     else {
01241         memcpy(cache_buffer+cache_fill, (unsigned char*)ptr, fullsize);
01242         cache_fill += fullsize;
01243     }
01244     return nmemb;
01245 }
01246
01247 size_t flush_cache(FILE *fd)
01248 {
01249     if ( cache_fill ) {
01250         size_t retval = fwrite(cache_buffer, 1, cache_fill, fd);
01251         if ( retval != cache_fill ) {
01252             cache_fill = 0;
01253             return 0;
01254         }
01255         cache_fill = 0;
01256     }
01257     return 1;
01258 }
01259 #else
01260 size_t fwrite_cached(const void *ptr, size_t size, size_t nmemb, FILE *fd)
01261 {
01262     return fwrite(ptr, size, nmemb, fd);
01263 }
01264
01265 size_t flush_cache(FILE *fd)
01266 {
01267     DUMMYUSE(fd)
01268     return 1;
01269 }
01270 #endif
01271
01272 /*
01273     #] Cached file operation functions :
01274     #[ Helper Macros :
01275 */
01276
01277 /* some helper macros to streamline the code in DoSnapshot() and DoRecovery() */
01278
01279 /* freeing memory */
01280
01281 #define R_FREE(ARG) \
01282     if ( ARG ) M_free(ARG, #ARG);
01283
01284 #define R_FREE_NAMETREE(ARG) \
01285     R_FREE(ARG->namenode); \
01286     R_FREE(ARG->namebuffer); \
01287     R_FREE(ARG);
01288
01289 #define R_FREE_STREAM(ARG) \
01290     R_FREE(ARG.buffer); \
01291     R_FREE(ARG.FoldName); \
01292     R_FREE(ARG.name);
01293
01294 /* reading a single variable */
01295
01296 #define R_SET(VAR,TYPE) \
01297     VAR = *((TYPE*)p); p = (unsigned char*)p + sizeof(TYPE);
01298
01299 /* general buffer */
01300
01301 #define R_COPY_B(VAR,SIZE,CAST) \
01302     VAR = (CAST)Malloc1(SIZE,#VAR); \
01303     memcpy(VAR, p, SIZE); p = (unsigned char*)p + SIZE;
01304
01305 #define S_WRITE_B(BUF,LEN) \
01306     if ( fwrite_cached(BUF, 1, LEN, fd) != (size_t)(LEN) ) return(__LINE__);
01307
01308 #define S_FLUSH_B \
01309     if ( flush_cache(fd) != 1 ) return(__LINE__);
01310
01311 /* character strings */
01312
```

```
01313 #define R_COPY_S(VAR,CAST) \
01314     if ( VAR ) { \
01315         VAR = (CAST)Malloc1(strlen(p)+1,"R_COPY_S"); \
01316         strcpy((char*)VAR, p); p = (unsigned char*)p + strlen(p) + 1; \
01317     }
01318
01319 #define S_WRITE_S(STR) \
01320     if ( STR ) { \
01321         l = strlen((char*)STR) + 1; \
01322         if ( fwrite_cached(STR, 1, l, fd) != (size_t)l ) return(__LINE__); \
01323     }
01324
01325 /* LIST */
01326
01327 #define R_COPY_LIST(ARG) \
01328     if ( ARG.maxnum ) { \
01329         R_COPY_B(ARG.lijst, ARG.size*ARG.maxnum, void*) \
01330     }
01331
01332 #define S_WRITE_LIST(LST) \
01333     if ( LST.maxnum ) { \
01334         S_WRITE_B((char*)LST.lijst, LST.maxnum*LST.size) \
01335     }
01336
01337 /* NAMETREE */
01338
01339 #define R_COPY_NAMETREE(ARG) \
01340     R_COPY_B(ARG, sizeof(NAMETREE), NAMETREE*); \
01341     if ( ARG->namenode ) { \
01342         R_COPY_B(ARG->namenode, ARG->nodesize*sizeof(NAMENODE), NAMENODE*); \
01343     } \
01344     if ( ARG->namebuffer ) { \
01345         R_COPY_B(ARG->namebuffer, ARG->namesize, UBYTE*); \
01346     }
01347
01348 #define S_WRITE_NAMETREE(ARG) \
01349     S_WRITE_B(ARG, sizeof(NAMETREE)); \
01350     if ( ARG->namenode ) { \
01351         S_WRITE_B(ARG->namenode, ARG->nodesize*sizeof(struct NaMeNode)); \
01352     } \
01353     if ( ARG->namebuffer ) { \
01354         S_WRITE_B(ARG->namebuffer, ARG->namesize); \
01355     }
01356
01357 /* DOLLAR */
01358
01359 #define S_WRITE_DOLLAR(ARG) \
01360     if ( ARG.size && ARG.where && ARG.where != &(AM.dollarzero) ) { \
01361         S_WRITE_B(ARG.where, ARG.size*sizeof(WORD)) \
01362     }
01363
01364 /* Printing time marks with ANNOUNCE macro */
01365
01366 #ifdef PRINTTIMEMARKS
01367 time_t announce_time;
01368 #define ANNOUNCE(str) time(&announce_time); MesPrint("TIMEMARK %s  %s", ctime(&announce_time), #str);
01369 #else
01370 #define ANNOUNCE(str)
01371 #endif
01372
01373 /*
01374     #] Helper Macros :
01375     #[ DoRecovery :
01376 */
01377
01402 int DoRecovery(int *moduletype)
01403 {
01404     GETIDENTITY
01405     FILE *fd;
01406     POSITION pos;
01407     void *buf, *p;
01408     LONG size, l;
01409     int i, j;
01410     UBYTE *org;
01411     char *namebufout, *namebufhide;
01412     LONG ofs;
01413     void *oldAMdollarzero;
01414     LIST PotModDolListBackup;
01415     LIST ModOptDolListBackup;
01416     WORD oldLogHandle;
01417
01418     MesPrint("Recovering ... %"); fflush(0);
01419
01420     if ( !(fd = fopen(recoveryfile, "r")) ) return(__LINE__);
01421
01422     /* load the complete recovery file into a buffer */
01423     if ( fread(&pos, sizeof(POSITION), 1, fd) != 1 ) {
```

```
01424         fclose(fd);
01425         return(__LINE__);
01426     }
01427     size = BASEPOSITION(pos) - sizeof(POSITION);
01428     buf = Malloc1(size, "recovery buffer");
01429     if ( fread(buf, size, 1, fd) != 1 ) {
01430         fclose(fd);
01431         return(__LINE__);
01432     }
01433 
01434     /* pointer p will go through the buffer in the following */
01435     p = buf;
01436 
01437     /* read moduletype */
01438     R_SET(*moduletype, int);
01439 
01440     /*#[ AM : */
01441 
01442     /* only certain elements will be restored. the rest of AM should have gotten
01443      * the correct values at startup. */
01444 
01445     R_SET(AM.hparallelflag, int);
01446     R_SET(AM.gparallelflag, int);
01447     R_SET(AM.gCodesFlag, int);
01448     R_SET(AM.gNamesFlag, int);
01449     R_SET(AM.gStatsFlag, int);
01450     R_SET(AM.gTokensWriteFlag, int);
01451     R_SET(AM.gNoSpacesInNumbers, int);
01452     R_SET(AM.gIndentSpace, WORD);
01453     R_SET(AM.gUnitTrace, WORD);
01454     R_SET(AM.gDefDim, int);
01455     R_SET(AM.gDefDim4, int);
01456     R_SET(AM.gncmod, WORD);
01457     R_SET(AM.gnpowmod, WORD);
01458     R_SET(AM.gmodmode, WORD);
01459     R_SET(AM.gOutputMode, WORD);
01460     R_SET(AM.gCnumpows, WORD);
01461     R_SET(AM.gOutputSpaces, WORD);
01462     R_SET(AM.gOutNumberType, WORD);
01463     R_SET(AM.gfunpowers, int);
01464     R_SET(AM.gPolyFun, WORD);
01465     R_SET(AM.gPolyFunInv, WORD);
01466     R_SET(AM.gPolyFunType, WORD);
01467     R_SET(AM.gPolyFunExp, WORD);
01468     R_SET(AM.gPolyFunVar, WORD);
01469     R_SET(AM.gPolyFunPow, WORD);
01470     R_SET(AM.gProcessBucketSize, LONG);
01471     R_SET(AM.OldChildTime, LONG);
01472     R_SET(AM.OldSecTime, LONG);
01473     R_SET(AM.OldMilliTime, LONG);
01474     R_SET(AM.gproperorderflag, int);
01475     R_SET(AM.gThreadBucketSize, LONG);
01476     R_SET(AM.gSizeCommuteInSet, int);
01477     R_SET(AM.gThreadStats, int);
01478     R_SET(AM.gFinalStats, int);
01479     R_SET(AM.gThreadsFlag, int);
01480     R_SET(AM.gThreadBalancing, int);
01481     R_SET(AM.gThreadSortFileSynch, int);
01482     R_SET(AM.gProcessStats, int);
01483     R_SET(AM.gOldParallelStats, int);
01484     R_SET(AM.gSortType, int);
01485     R_SET(AM.gShortStatsMax, WORD);
01486     R_SET(AM.gIsFortran90, int);
01487     R_SET(oldAMdollarzero, void*);
01488     R_FREE(AM.gFortran90Kind);
01489     R_SET(AM.gFortran90Kind,UBYTE *);
01490     R_COPY_S(AM.gFortran90Kind,UBYTE *);
01491 
01492     R_COPY_S(AM.gextrasym,UBYTE *);
01493     R_COPY_S(AM.ggextrasym,UBYTE *);
01494 
01495     R_SET(AM.PrintTotalSize,int);
01496     R_SET(AM.fbuffersize,int);
01497     R_SET(AM.gOldFactArgFlag,int);
01498     R_SET(AM.ggOldFactArgFlag,int);
01499 
01500     R_SET(AM.gnumextrasym,int);
01501     R_SET(AM.ggnumextrasym,int);
01502     R_SET(AM.NumSpectatorFiles,int);
01503     R_SET(AM.SizeForSpectatorFiles,int);
01504     R_SET(AM.gOldGCDflag,int);
01505     R_SET(AM.ggOldGCDflag,int);
01506     R_SET(AM.gWTimeStatsFlag, int);
01507 
01508     R_FREE(AM.Path);
01509     R_SET(AM.Path,UBYTE *);
01510     R_COPY_S(AM.Path,UBYTE *);
```

```
01511
01512     R_SET(AM.FromStdin, BOOL);
01513     R_SET(AM.IgnoreDeprecation, BOOL);
01514
01515 #ifdef PRINTDEBUG
01516     print_M();
01517 #endif
01518
01519     /*#] AM : */
01520     /*#[ AC : */
01521
01522     /* #[ AC free pointers */
01523
01524     /* AC will be overwritten by data from the recovery file, therefore
01525      * dynamically allocated memory must be freed first. */
01526
01527     R_FREE_NAMETREE(AC.dollarnames);
01528     R_FREE_NAMETREE(AC.exprnames);
01529     R_FREE_NAMETREE(AC.varnames);
01530     for ( i=0; i<AC.ChannelList.num; ++i ) {
01531         R_FREE(channels[i].name);
01532     }
01533     R_FREE(AC.ChannelList.lijst);
01534     R_FREE(AC.DubiousList.lijst);
01535     for ( i=0; i<AC.FunctionList.num; ++i ) {
01536         TABLES T = functions[i].tabl;
01537         if ( T ) {
01538             R_FREE(T->buffers);
01539             R_FREE(T->mm);
01540             R_FREE(T->flags);
01541             R_FREE(T->prototype);
01542             R_FREE(T->tablepointers);
01543             if ( T->sparse ) {
01544                 R_FREE(T->boomlijst);
01545                 R_FREE(T->argtail);
01546             }
01547             if ( T->spare ) {
01548                 R_FREE(T->spare->buffers);
01549                 R_FREE(T->spare->mm);
01550                 R_FREE(T->spare->flags);
01551                 R_FREE(T->spare->tablepointers);
01552                 if ( T->spare->sparse ) {
01553                     R_FREE(T->spare->boomlijst);
01554                 }
01555                 R_FREE(T->spare);
01556             }
01557             R_FREE(T);
01558         }
01559     }
01560     R_FREE(AC.FunctionList.lijst);
01561     for ( i=0; i<AC.ExpressionList.num; ++i ) {
01562         if ( Expressions[i].renum ) {
01563             R_FREE(Expressions[i].renum->symb.lo);
01564             R_FREE(Expressions[i].renum);
01565         }
01566         if ( Expressions[i].bracketinfo ) {
01567             R_FREE(Expressions[i].bracketinfo->indexbuffer);
01568             R_FREE(Expressions[i].bracketinfo->bracketbuffer);
01569             R_FREE(Expressions[i].bracketinfo);
01570         }
01571         if ( Expressions[i].newbracketinfo ) {
01572             R_FREE(Expressions[i].newbracketinfo->indexbuffer);
01573             R_FREE(Expressions[i].newbracketinfo->bracketbuffer);
01574             R_FREE(Expressions[i].newbracketinfo);
01575         }
01576         if ( Expressions[i].renumlists != AN.dummyrenumlist ) {
01577             R_FREE(Expressions[i].renumlists);
01578         }
01579         R_FREE(Expressions[i].inmem);
01580     }
01581     R_FREE(AC.ExpressionList.lijst);
01582     R_FREE(AC.IndexList.lijst);
01583     R_FREE(AC.SetElementList.lijst);
01584     R_FREE(AC.SetList.lijst);
01585     R_FREE(AC.SymbolList.lijst);
01586     R_FREE(AC.VectorList.lijst);
01587     for ( i=0; i<AC.TableBaseList.num; ++i ) {
01588         R_FREE(tablebases[i].iblocks);
01589         R_FREE(tablebases[i].nblocks);
01590         R_FREE(tablebases[i].name);
01591         R_FREE(tablebases[i].fullname);
01592         R_FREE(tablebases[i].tablenames);
01593     }
01594     R_FREE(AC.TableBaseList.lijst);
01595     for ( i=0; i<AC.cbufList.num; ++i ) {
01596         R_FREE(cbuf[i].Buffer);
01597         R_FREE(cbuf[i].lhs);
```

```
01598         R_FREE(cbuf[i].rhs);
01599         R_FREE(cbuf[i].boomlijst);
01600     }
01601     R_FREE(AC.cbufList.lijst);
01602     R_FREE(AC.AutoSymbolList.lijst);
01603     R_FREE(AC.AutoIndexList.lijst);
01604     R_FREE(AC.AutoVectorList.lijst);
01605     /* Tables cannot be auto-declared, therefore no extra code here */
01606     R_FREE(AC.AutoFunctionList.lijst);
01607     R_FREE_NAMETREE(AC.autonames);
01608     for ( i=0; i<AC.NumStreams; ++i ) {
01609         R_FREE_STREAM(AC.Streams[i]);
01610     }
01611     R_FREE(AC.Streams);
01612     R_FREE(AC.termstack);
01613     R_FREE(AC.termsortstack);
01614     R_FREE(AC.cmod);
01615     R_FREE(AC.modpowers);
01616     R_FREE(AC.halfmod);
01617     R_FREE(AC.IfHeap);
01618     R_FREE(AC.IfCount);
01619     R_FREE(AC.iBuffer);
01620     for ( i=0; i<AC.NumLabels; ++i ) {
01621         R_FREE(AC.LabelNames[i]);
01622     }
01623     R_FREE(AC.LabelNames);
01624     R_FREE(AC.FixIndices);
01625     R_FREE(AC.termsumcheck);
01626     R_FREE(AC.WildcardNames);
01627     R_FREE(AC.tokens);
01628     R_FREE(AC.tokenarglevel);
01629     R_FREE(AC.modinverses);
01630     R_FREE(AC.Fortran90Kind);
01631 #ifdef WITHPTHREADS
01632     R_FREE(AC.inputnumbers);
01633 #endif
01634     R_FREE(AC.IfSumCheck);
01635     R_FREE(AC.CommuteInSet);
01636     R_FREE(AC.CheckpointRunAfter);
01637     R_FREE(AC.CheckpointRunBefore);
01638 
01639     /* #] AC free pointers */
01640 
01641     /* backup some lists in order to restore it to the initial setup */
01642     PotModDolListBackup = AC.PotModDolList;
01643     ModOptDolListBackup = AC.ModOptDolList;
01644     oldLogHandle = AC.LogHandle;
01645 
01646     /* first we copy AC as a whole and then restore the pointer structures step
01647        by step. */
01648 
01649     AC = *((struct C_const*)p); p = (unsigned char*)p + sizeof(struct C_const);
01650 
01651     R_COPY_NAMETREE(AC.dollarnames);
01652     R_COPY_NAMETREE(AC.exprnames);
01653     R_COPY_NAMETREE(AC.varnames);
01654 
01655     R_COPY_LIST(AC.ChannelList);
01656     for ( i=0; i<AC.ChannelList.num; ++i ) {
01657         R_COPY_S(channels[i].name,char*);
01658         channels[i].handle = ReOpenFile(channels[i].name);
01659     }
01660     AC.ChannelList.message = "channel buffer";
01661 
01662     AC.DubiousList.lijst = 0;
01663     AC.DubiousList.message = "ambiguous variable";
01664     AC.DubiousList.num =
01665     AC.DubiousList.maxnum =
01666     AC.DubiousList.numglobal =
01667     AC.DubiousList.numtemp =
01668     AC.DubiousList.numclear = 0;
01669 
01670     R_COPY_LIST(AC.FunctionList);
01671     for ( i=0; i<AC.FunctionList.num; ++i ) {
01672         if ( functions[i].tabl ) {
01673             TABLES tabl;
01674             R_COPY_B(tabl, sizeof(struct TaBlEs), TABLES);
01675             functions[i].tabl = tabl;
01676             if ( tabl->tablepointers ) {
01677                 if ( tabl->sparse ) {
01678                     R_COPY_B(tabl->tablepointers,
01679                         tabl->reserved*sizeof(WORD)*(tabl->numind+TABLEEXTENSION),
01680                         WORD*);
01681                 }
01682                 else {
01683                     R_COPY_B(tabl->tablepointers,
01684                         TABLEEXTENSION*sizeof(WORD)*(tabl->totind), WORD*);
```

```
01685                 }
01686             }
01687             org = (UBYTE*)tabl->prototype;
01688 #ifdef WITHPTHREADS
01689             R_COPY_B(tabl->prototype, tabl->prototypeSize, WORD**);
01690             ofs = (UBYTE*)tabl->prototype - org;
01691             for ( j=0; j<AM.totalnumberofthreads; ++j ) {
01692                 if ( tabl->prototype[j] ) {
01693                     tabl->prototype[j] = (WORD*)((UBYTE*)tabl->prototype[j] + ofs);
01694                 }
01695             }
01696             if ( tabl->pattern ) {
01697                 tabl->pattern = (WORD**)((UBYTE*)tabl->pattern + ofs);
01698                 for ( j=0; j<AM.totalnumberofthreads; ++j ) {
01699                     if ( tabl->pattern[j] ) {
01700                         tabl->pattern[j] = (WORD*)((UBYTE*)tabl->pattern[j] + ofs);
01701                     }
01702                 }
01703             }
01704 #else
01705             ofs = tabl->pattern - tabl->prototype;
01706             R_COPY_B(tabl->prototype, tabl->prototypeSize, WORD*);
01707             if ( tabl->pattern ) {
01708                 tabl->pattern = tabl->prototype + ofs;
01709             }
01710 #endif
01711             R_COPY_B(tabl->mm, tabl->numind*(LONG)sizeof(MINMAX), MINMAX*);
01712             R_COPY_B(tabl->flags, tabl->numind*(LONG)sizeof(WORD), WORD*);
01713             if ( tabl->sparse ) {
01714                 R_COPY_B(tabl->boomlijst, tabl->MaxTreeSize*(LONG)sizeof(COMPTREE), COMPTREE*);
01715                 R_COPY_S(tabl->argtail,UBYTE*);
01716             }
01717             R_COPY_B(tabl->buffers, tabl->bufferssize*(LONG)sizeof(WORD), WORD*);
01718             if ( tabl->spare ) {
01719                 TABLES spare;
01720                 R_COPY_B(spare, sizeof(struct TaBlEs), TABLES);
01721                 tabl->spare = spare;
01722                 if ( spare->tablepointers ) {
01723                     if ( spare->sparse ) {
01724                         R_COPY_B(spare->tablepointers,
01725                             spare->reserved*sizeof(WORD)*(spare->numind+TABLEEXTENSION),
01726                             WORD*);
01727                     }
01728                     else {
01729                         R_COPY_B(spare->tablepointers,
01730                             TABLEEXTENSION*sizeof(WORD)*(spare->totind), WORD*);
01731                     }
01732                 }
01733                 spare->prototype = tabl->prototype;
01734                 spare->pattern = tabl->pattern;
01735                 R_COPY_B(spare->mm, spare->numind*(LONG)sizeof(MINMAX), MINMAX*);
01736                 R_COPY_B(spare->flags, spare->numind*(LONG)sizeof(WORD), WORD*);
01737                 if ( tabl->sparse ) {
01738                     R_COPY_B(spare->boomlijst, spare->MaxTreeSize*(LONG)sizeof(COMPTREE), COMPTREE*);
01739                     spare->argtail = tabl->argtail;
01740                 }
01741                 spare->spare = tabl;
01742                 R_COPY_B(spare->buffers, spare->bufferssize*(LONG)sizeof(WORD), WORD*);
01743             }
01744         }
01745     }
01746     AC.FunctionList.message = "function";
01747
01748     R_COPY_LIST(AC.ExpressionList);
01749     for ( i=0; i<AC.ExpressionList.num; ++i ) {
01750         EXPRESSIONS ex = Expressions + i;
01751         if ( ex->renum ) {
01752             R_COPY_B(ex->renum, sizeof(struct ReNuMbEr), RENUMBER);
01753             org = (UBYTE*)ex->renum->symb.lo;
01754             R_SET(size, size_t);
01755             R_COPY_B(ex->renum->symb.lo, size, WORD*);
01756             ofs = (UBYTE*)ex->renum->symb.lo - org;
01757             ex->renum->symb.start = (WORD*)((UBYTE*)ex->renum->symb.start + ofs);
01758             ex->renum->symb.hi = (WORD*)((UBYTE*)ex->renum->symb.hi + ofs);
01759             ex->renum->indi.lo = (WORD*)((UBYTE*)ex->renum->indi.lo + ofs);
01760             ex->renum->indi.start = (WORD*)((UBYTE*)ex->renum->indi.start + ofs);
01761             ex->renum->indi.hi = (WORD*)((UBYTE*)ex->renum->indi.hi + ofs);
01762             ex->renum->vect.lo = (WORD*)((UBYTE*)ex->renum->vect.lo + ofs);
01763             ex->renum->vect.start = (WORD*)((UBYTE*)ex->renum->vect.start + ofs);
01764             ex->renum->vect.hi = (WORD*)((UBYTE*)ex->renum->vect.hi + ofs);
01765             ex->renum->func.lo = (WORD*)((UBYTE*)ex->renum->func.lo + ofs);
01766             ex->renum->func.start = (WORD*)((UBYTE*)ex->renum->func.start + ofs);
01767             ex->renum->func.hi = (WORD*)((UBYTE*)ex->renum->func.hi + ofs);
01768             ex->renum->symnum = (WORD*)((UBYTE*)ex->renum->symnum + ofs);
01769             ex->renum->indnum = (WORD*)((UBYTE*)ex->renum->indnum + ofs);
01770             ex->renum->vecnum = (WORD*)((UBYTE*)ex->renum->vecnum + ofs);
01771             ex->renum->funnum = (WORD*)((UBYTE*)ex->renum->funnum + ofs);
```

```
01772         }
01773         if ( ex->bracketinfo ) {
01774             R_COPY_B(ex->bracketinfo, sizeof(BRACKETINFO), BRACKETINFO*);
01775             R_COPY_B(ex->bracketinfo->indexbuffer,
      ex->bracketinfo->indexbuffersize*sizeof(BRACKETINDEX), BRACKETINDEX*);
01776             R_COPY_B(ex->bracketinfo->bracketbuffer, ex->bracketinfo->bracketbuffersize*sizeof(WORD),
      WORD*);
01777         }
01778         if ( ex->newbracketinfo ) {
01779             R_COPY_B(ex->newbracketinfo, sizeof(BRACKETINFO), BRACKETINFO*);
01780             R_COPY_B(ex->newbracketinfo->indexbuffer,
      ex->newbracketinfo->indexbuffersize*sizeof(BRACKETINDEX), BRACKETINDEX*);
01781             R_COPY_B(ex->newbracketinfo->bracketbuffer,
      ex->newbracketinfo->bracketbuffersize*sizeof(WORD), WORD*);
01782         }
01783 #ifdef WITHPTHREADS
01784         ex->renumlists = 0;
01785 #else
01786         ex->renumlists = AN.dummyrenumlist;
01787 #endif
01788         if ( ex->inmem ) {
01789             R_SET(size, size_t);
01790             R_COPY_B(ex->inmem, size, WORD*);
01791         }
01792     }
01793     AC.ExpressionList.message = "expression";
01794
01795     R_COPY_LIST(AC.IndexList);
01796     AC.IndexList.message = "index";
01797     R_COPY_LIST(AC.SetElementList);
01798     AC.SetElementList.message = "set element";
01799     R_COPY_LIST(AC.SetList);
01800     AC.SetList.message = "set";
01801     R_COPY_LIST(AC.SymbolList);
01802     AC.SymbolList.message = "symbol";
01803     R_COPY_LIST(AC.VectorList);
01804     AC.VectorList.message = "vector";
01805
01806     AC.PotModDolList = PotModDolListBackup;
01807     AC.ModOptDolList = ModOptDolListBackup;
01808
01809     R_COPY_LIST(AC.TableBaseList);
01810     for ( i=0; i<AC.TableBaseList.num; ++i ) {
01811         if ( tablebases[i].iblocks ) {
01812             R_COPY_B(tablebases[i].iblocks,
      tablebases[i].info.numberofindexblocks*sizeof(INDEXBLOCK*),INDEXBLOCK**);
01813             for ( j=0; j<tablebases[i].info.numberofindexblocks; ++j ) {
01814                 if ( tablebases[i].iblocks[j] ) {
01815                     R_COPY_B(tablebases[i].iblocks[j], sizeof(INDEXBLOCK), INDEXBLOCK*);
01816                 }
01817             }
01818         }
01819         if ( tablebases[i].nblocks ) {
01820             R_COPY_B(tablebases[i].nblocks,
      tablebases[i].info.numberofnamesblocks*sizeof(NAMESBLOCK*),NAMESBLOCK**);
01821             for ( j=0; j<tablebases[i].info.numberofindexblocks; ++j ) {
01822                 if ( tablebases[i].nblocks[j] ) {
01823                     R_COPY_B(tablebases[i].nblocks[j], sizeof(NAMESBLOCK), NAMESBLOCK*);
01824                 }
01825             }
01826         }
01827         /* reopen file */
01828         if ( ( tablebases[i].handle = fopen(tablebases[i].fullname, "r+b") ) == NULL ) {
01829             MesPrint("ERROR: Could not reopen tablebase %s!",tablebases[i].name);
01830             Terminate(-1);
01831         }
01832         R_COPY_S(tablebases[i].name,char*);
01833         R_COPY_S(tablebases[i].fullname,char*);
01834         R_COPY_S(tablebases[i].tablenames,char*);
01835     }
01836     AC.TableBaseList.message = "list of tablebases";
01837
01838     R_COPY_LIST(AC.cbufList);
01839     for ( i=0; i<AC.cbufList.num; ++i ) {
01840         org = (UBYTE*)cbuf[i].Buffer;
01841         R_COPY_B(cbuf[i].Buffer, cbuf[i].BufferSize*sizeof(WORD), WORD*);
01842         ofs = (UBYTE*)cbuf[i].Buffer - org;
01843         cbuf[i].Top = (WORD*)((UBYTE*)cbuf[i].Top + ofs);
01844         cbuf[i].Pointer = (WORD*)((UBYTE*)cbuf[i].Pointer + ofs);
01845         R_COPY_B(cbuf[i].lhs, cbuf[i].maxlhs*(LONG)sizeof(WORD*), WORD**);
01846         for ( j=1; j<=cbuf[i].numlhs; ++j ) {
01847             if ( cbuf[i].lhs[j] ) cbuf[i].lhs[j] = (WORD*)((UBYTE*)cbuf[i].lhs[j] + ofs);
01848         }
01849         org = (UBYTE*)cbuf[i].rhs;
01850         R_COPY_B(cbuf[i].rhs, cbuf[i].maxrhs*(LONG)(sizeof(WORD*)+2*sizeof(LONG)+2*sizeof(WORD)),
      WORD**);
01851         for ( j=1; j<=cbuf[i].numrhs; ++j ) {
```

```
01852             if ( cbuf[i].rhs[j] ) cbuf[i].rhs[j] = (WORD*)((UBYTE*)cbuf[i].rhs[j] + ofs);
01853         }
01854         ofs = (UBYTE*)cbuf[i].rhs - org;
01855         cbuf[i].CanCommu = (LONG*)((UBYTE*)cbuf[i].CanCommu + ofs);
01856         cbuf[i].NumTerms = (LONG*)((UBYTE*)cbuf[i].NumTerms + ofs);
01857         cbuf[i].numdum = (WORD*)((UBYTE*)cbuf[i].numdum + ofs);
01858         cbuf[i].dimension = (WORD*)((UBYTE*)cbuf[i].dimension + ofs);
01859         if ( cbuf[i].boomlijst ) {
01860             R_COPY_B(cbuf[i].boomlijst, cbuf[i].MaxTreeSize*sizeof(COMPTREE), COMPTREE*);
01861         }
01862     }
01863     AC.cbufList.message = "compiler buffer";
01864 
01865     R_COPY_LIST(AC.AutoSymbolList);
01866     AC.AutoSymbolList.message = "autosymbol";
01867     R_COPY_LIST(AC.AutoIndexList);
01868     AC.AutoIndexList.message = "autoindex";
01869     R_COPY_LIST(AC.AutoVectorList);
01870     AC.AutoVectorList.message = "autovector";
01871     R_COPY_LIST(AC.AutoFunctionList);
01872     AC.AutoFunctionList.message = "autofunction";
01873 
01874     R_COPY_NAMETREE(AC.autonames);
01875 
01876     AC.Symbols = &(AC.SymbolList);
01877     AC.Indices = &(AC.IndexList);
01878     AC.Vectors = &(AC.VectorList);
01879     AC.Functions = &(AC.FunctionList);
01880     AC.activenames = &(AC.varnames);
01881 
01882     org = (UBYTE*)AC.Streams;
01883     R_COPY_B(AC.Streams, AC.MaxNumStreams*(LONG)sizeof(STREAM), STREAM*);
01884     for ( i=0; i<AC.NumStreams; ++i ) {
01885         if ( AC.Streams[i].type != FILESTREAM ) {
01886             UBYTE *org2;
01887             org2 = AC.Streams[i].buffer;
01888             if ( AC.Streams[i].inbuffer ) {
01889                 R_COPY_B(AC.Streams[i].buffer, AC.Streams[i].inbuffer, UBYTE*);
01890             }
01891             ofs = AC.Streams[i].buffer - org2;
01892             AC.Streams[i].pointer += ofs;
01893             AC.Streams[i].top += ofs;
01894         }
01895         else {
01896             p = (unsigned char*)p + AC.Streams[i].inbuffer;
01897         }
01898         AC.Streams[i].buffersize = AC.Streams[i].inbuffer;
01899         R_COPY_S(AC.Streams[i].FoldName,UBYTE*);
01900         R_COPY_S(AC.Streams[i].name,UBYTE*);
01901         if ( AC.Streams[i].type == PREVARSTREAM || AC.Streams[i].type == DOLLARSTREAM ) {
01902             AC.Streams[i].pname = AC.Streams[i].name;
01903         }
01904         else if ( AC.Streams[i].type == FILESTREAM ) {
01905             UBYTE *org2;
01906             org2 = AC.Streams[i].buffer;
01907             AC.Streams[i].buffer = (UBYTE*)Malloc1(AC.Streams[i].buffersize, "buffer");
01908             ofs = AC.Streams[i].buffer - org2;
01909             AC.Streams[i].pointer += ofs;
01910             AC.Streams[i].top += ofs;
01911 
01912             /* open file except for already opened main input file */
01913             if ( i ) {
01914                 AC.Streams[i].handle = OpenFile((char *)(AC.Streams[i].name));
01915                 if ( AC.Streams[i].handle == -1 ) {
01916                     MesPrint("ERROR: Could not reopen stream %s!",AC.Streams[i].name);
01917                     Terminate(-1);
01918                 }
01919             }
01920 
01921             PUTZERO(pos);
01922             ADDPOS(pos, AC.Streams[i].bufferposition);
01923             SeekFile(AC.Streams[i].handle, &pos, SEEK_SET);
01924 
01925             AC.Streams[i].inbuffer = ReadFile(AC.Streams[i].handle, AC.Streams[i].buffer,
      AC.Streams[i].inbuffer);
01926 
01927             SETBASEPOSITION(pos, AC.Streams[i].fileposition);
01928             SeekFile(AC.Streams[i].handle, &pos, SEEK_SET);
01929         }
01930         /*
01931          * Ideally, we should check if we have a type PREREADSTREAM, PREREADSTREAM2, and
01932          * PRECALCSTREAM here. If so, we should free element name and point it
01933          * to the name element of the embracing stream's struct. In practice,
01934          * this is undoable without adding new data to STREAM. Since we create
01935          * only a small memory leak here (some few byte for each existing
01936          * stream of these types) and only once when we do a recovery, we
01937          * tolerate this leak and keep it STREAM as it is.
```

```
01938          */
01939     }
01940     ofs = (UBYTE*)AC.Streams - org;
01941     AC.CurrentStream = (STREAM*)((UBYTE*)AC.CurrentStream + ofs);
01942
01943     if ( AC.termstack ) {
01944         R_COPY_B(AC.termstack, AC.maxtermlevel*(LONG)sizeof(LONG), LONG*);
01945     }
01946
01947     if ( AC.termsortstack ) {
01948         R_COPY_B(AC.termsortstack, AC.maxtermlevel*(LONG)sizeof(LONG), LONG*);
01949     }
01950
01951     /* exception: here we also change values from struct AM */
01952     R_COPY_B(AC.cmod, AM.MaxTal*4*(LONG)sizeof(UWORD), UWORD*);
01953     AM.gcmod = AC.cmod + AM.MaxTal;
01954     AC.powmod = AM.gcmod + AM.MaxTal;
01955     AM.gpowmod = AC.powmod + AM.MaxTal;
01956
01957     AC.modpowers = 0;
01958     AC.halfmod = 0;
01959
01960     /* we don't care about AC.ProtoType/WildC */
01961
01962     if ( AC.IfHeap ) {
01963         ofs = AC.IfStack - AC.IfHeap;
01964         R_COPY_B(AC.IfHeap, (LONG)sizeof(LONG)*(AC.MaxIf+1), LONG*);
01965         AC.IfStack = AC.IfHeap + ofs;
01966         R_COPY_B(AC.IfCount, (LONG)sizeof(LONG)*(AC.MaxIf+1), LONG*);
01967     }
01968
01969     org = AC.iBuffer;
01970     size = AC.iStop - AC.iBuffer + 2;
01971     R_COPY_B(AC.iBuffer, size, UBYTE*);
01972     ofs = AC.iBuffer - org;
01973     AC.iPointer += ofs;
01974     AC.iStop += ofs;
01975
01976     if ( AC.LabelNames ) {
01977         org = (UBYTE*)AC.LabelNames;
01978         R_COPY_B(AC.LabelNames, AC.MaxLabels*(LONG)(sizeof(UBYTE*)+sizeof(WORD)), UBYTE**);
01979         for ( i=0; i<AC.NumLabels; ++i ) {
01980             R_COPY_S(AC.LabelNames[i],UBYTE*);
01981         }
01982         ofs = (UBYTE*)AC.LabelNames - org;
01983         AC.Labels = (int*)((UBYTE*)AC.Labels + ofs);
01984     }
01985
01986     R_COPY_B(AC.FixIndices, AM.OffsetIndex*(LONG)sizeof(WORD), WORD*);
01987
01988     if ( AC.termsumcheck ) {
01989         R_COPY_B(AC.termsumcheck, AC.maxtermlevel*(LONG)sizeof(WORD), WORD*);
01990     }
01991
01992     R_COPY_B(AC.WildcardNames, AC.WildcardBufferSize, UBYTE*);
01993
01994     if ( AC.tokens ) {
01995         size = AC.toptokens - AC.tokens;
01996         if ( size ) {
01997             org = (UBYTE*)AC.tokens;
01998             R_COPY_B(AC.tokens, size, SBYTE*);
01999             ofs = (UBYTE*)AC.tokens - org;
02000             AC.endoftokens += ofs;
02001             AC.toptokens += ofs;
02002         }
02003         else {
02004             AC.tokens = 0;
02005             AC.endoftokens = AC.tokens;
02006             AC.toptokens = AC.tokens;
02007         }
02008     }
02009
02010     R_COPY_B(AC.tokenarglevel, AM.MaxParLevel*(LONG)sizeof(WORD), WORD*);
02011
02012     AC.modinverses = 0;
02013
02014     R_COPY_S(AC.Fortran90Kind,UBYTE *);
02015
02016 #ifdef WITHPTHREADS
02017     if ( AC.inputnumbers ) {
02018         org = (UBYTE*)AC.inputnumbers;
02019         R_COPY_B(AC.inputnumbers, AC.sizepfirstnum*(LONG)(sizeof(WORD)+sizeof(LONG)), LONG*);
02020         ofs = (UBYTE*)AC.inputnumbers - org;
02021         AC.pfirstnum = (WORD*)((UBYTE*)AC.pfirstnum + ofs);
02022     }
02023     AC.halfmodlock = dummylock;
02024 #endif /* ifdef WITHPTHREADS */
```

```
02025
02026     if ( AC.IfSumCheck ) {
02027         R_COPY_B(AC.IfSumCheck, (LONG)sizeof(WORD)*(AC.MaxIf+1), WORD*);
02028     }
02029     if ( AC.CommuteInSet ) {
02030         R_COPY_B(AC.CommuteInSet, (LONG)sizeof(WORD)*(AC.SizeCommuteInSet+1), WORD*);
02031     }
02032
02033     AC.LogHandle = oldLogHandle;
02034
02035     R_COPY_S(AC.CheckpointRunAfter,char*);
02036     R_COPY_S(AC.CheckpointRunBefore,char*);
02037
02038     R_COPY_S(AC.extrasym,UBYTE *);
02039
02040 #ifdef PRINTDEBUG
02041     print_C();
02042 #endif
02043
02044     /*#] AC : */
02045     /*#[ AP : */
02046
02047     /* #[ AP free pointers */
02048
02049     /* AP will be overwritten by data from the recovery file, therefore
02050      * dynamically allocated memory must be freed first. */
02051
02052     for ( i=0; i<AP.DollarList.num; ++i ) {
02053         if ( Dollars[i].size ) {
02054             R_FREE(Dollars[i].where);
02055         }
02056         CleanDollarFactors(Dollars+i);
02057     }
02058     R_FREE(AP.DollarList.lijst);
02059
02060     for ( i=0; i<AP.PreVarList.num; ++i ) {
02061         R_FREE(PreVar[i].name);
02062     }
02063     R_FREE(AP.PreVarList.lijst);
02064
02065     for ( i=0; i<AP.LoopList.num; ++i ) {
02066         R_FREE(DoLoops[i].p.buffer);
02067         if ( DoLoops[i].dollarname ) {
02068             R_FREE(DoLoops[i].dollarname);
02069         }
02070     }
02071     R_FREE(AP.LoopList.lijst);
02072
02073     for ( i=0; i<AP.ProcList.num; ++i ) {
02074         R_FREE(Procedures[i].p.buffer);
02075         R_FREE(Procedures[i].name);
02076     }
02077     R_FREE(AP.ProcList.lijst);
02078
02079     for ( i=1; i<=AP.PreSwitchLevel; ++i ) {
02080         R_FREE(AP.PreSwitchStrings[i]);
02081     }
02082     R_FREE(AP.PreSwitchStrings);
02083     R_FREE(AP.preStart);
02084     R_FREE(AP.procedureExtension);
02085     R_FREE(AP.cprocedureExtension);
02086     R_FREE(AP.PreIfStack);
02087     R_FREE(AP.PreSwitchModes);
02088     R_FREE(AP.PreTypes);
02089
02090     /* #] AP free pointers */
02091
02092     /* first we copy AP as a whole and then restore the pointer structures step
02093        by step. */
02094
02095     AP = *((struct P_const*)p); p = (unsigned char*)p + sizeof(struct P_const);
02096 #ifdef WITHPTHREADS
02097     AP.PreVarLock = dummylock;
02098 #endif
02099
02100     R_COPY_LIST(AP.DollarList);
02101     for ( i=0; i<AP.DollarList.num; ++i ) {
02102         DOLLARS d = Dollars + i;
02103         size = d->size * sizeof(WORD);
02104         if ( size && d->where && d->where != oldAMdollarzero ) {
02105             R_COPY_B(d->where, size, void*);
02106         }
02107 #ifdef WITHPTHREADS
02108         INIRECLOCK(d->pthreadslock);
02109 #endif
02110         if ( d->nfactors > 1 ) {
02111             R_COPY_B(d->factors,sizeof(FACDOLLAR)*d->nfactors,FACDOLLAR*);
```

```
02112             for ( j = 0; j < d->nfactors; j++ ) {
02113                 if ( d->factors[j].size > 0 ) {
02114                     R_COPY_B(d->factors[i].where,sizeof(WORD)*(d->factors[j].size+1),WORD*);
02115                 }
02116             }
02117         }
02118     }
02119     AP.DollarList.message = "$-variable";
02120 
02121     R_COPY_LIST(AP.PreVarList);
02122     for ( i=0; i<AP.PreVarList.num; ++i ) {
02123         R_SET(size, size_t);
02124         org = PreVar[i].name;
02125         R_COPY_B(PreVar[i].name, size, UBYTE*);
02126         ofs = PreVar[i].name - org;
02127         if ( PreVar[i].value ) PreVar[i].value += ofs;
02128         if ( PreVar[i].argnames ) PreVar[i].argnames += ofs;
02129     }
02130     AP.PreVarList.message = "PreVariable";
02131 
02132     R_COPY_LIST(AP.LoopList);
02133     for ( i=0; i<AP.LoopList.num; ++i ) {
02134         org = DoLoops[i].p.buffer;
02135         R_COPY_B(DoLoops[i].p.buffer, DoLoops[i].p.size, UBYTE*);
02136         ofs = DoLoops[i].p.buffer - org;
02137         if ( DoLoops[i].name ) DoLoops[i].name += ofs;
02138         if ( DoLoops[i].vars ) DoLoops[i].vars += ofs;
02139         if ( DoLoops[i].contents ) DoLoops[i].contents += ofs;
02140         if ( DoLoops[i].type == ONEEXPRESSION ) {
02141             R_COPY_S(DoLoops[i].dollarname,UBYTE*);
02142         }
02143     }
02144     AP.LoopList.message = "doloop";
02145 
02146     R_COPY_LIST(AP.ProcList);
02147     for ( i=0; i<AP.ProcList.num; ++i ) {
02148         if ( Procedures[i].p.size ) {
02149             if ( Procedures[i].loadmode != 1 ) {
02150                 R_COPY_B(Procedures[i].p.buffer, Procedures[i].p.size, UBYTE*);
02151             }
02152             else {
02153                 R_SET(j, int);
02154                 Procedures[i].p.buffer = Procedures[j].p.buffer;
02155             }
02156         }
02157         R_COPY_S(Procedures[i].name,UBYTE*);
02158     }
02159     AP.ProcList.message = "procedure";
02160 
02161     size = (AP.NumPreSwitchStrings+1)*(LONG)sizeof(UBYTE*);
02162     R_COPY_B(AP.PreSwitchStrings, size, UBYTE**);
02163     for ( i=1; i<=AP.PreSwitchLevel; ++i ) {
02164         R_COPY_S(AP.PreSwitchStrings[i],UBYTE*);
02165     }
02166 
02167     org = AP.preStart;
02168     R_COPY_B(AP.preStart, AP.pSize, UBYTE*);
02169     ofs = AP.preStart - org;
02170     if ( AP.preFill ) AP.preFill += ofs;
02171     if ( AP.preStop ) AP.preStop += ofs;
02172 
02173     R_COPY_S(AP.procedureExtension,UBYTE*);
02174     R_COPY_S(AP.cprocedureExtension,UBYTE*);
02175 
02176     R_COPY_B(AP.PreAssignStack,AP.MaxPreAssignLevel*(LONG)sizeof(LONG),LONG*);
02177     R_COPY_B(AP.PreIfStack, AP.MaxPreIfLevel*(LONG)sizeof(int), int*);
02178     R_COPY_B(AP.PreSwitchModes, (AP.NumPreSwitchStrings+1)*(LONG)sizeof(int), int*);
02179     R_COPY_B(AP.PreTypes, (AP.MaxPreTypes+1)*(LONG)sizeof(int), int*);
02180 
02181 #ifdef PRINTDEBUG
02182     print_P();
02183 #endif
02184 
02185     /*#] AP : */
02186     /*#[ AR : */
02187 
02188     R_SET(ofs,LONG);
02189     if ( ofs ) {
02190         AR.infile = AR.Fscr+1;
02191         AR.outfile = AR.Fscr;
02192         AR.hidefile = AR.Fscr+2;
02193     }
02194     else {
02195         AR.infile = AR.Fscr;
02196         AR.outfile = AR.Fscr+1;
02197         AR.hidefile = AR.Fscr+2;
02198     }
```

```
02199
02200     /* #[ AR free pointers */
02201
02202     /* Parts of AR will be overwritten by data from the recovery file, therefore
02203      * dynamically allocated memory must be freed first. */
02204
02205     namebufout = AR.outfile->name;
02206     namebufhide = AR.hidefile->name;
02207     R_FREE(AR.outfile->POobuffer);
02208 #ifdef WITHZLIB
02209     R_FREE(AR.outfile->zsp);
02210     R_FREE(AR.outfile->ziobuffer);
02211 #endif
02212     namebufhide = AR.hidefile->name;
02213     R_FREE(AR.hidefile->POobuffer);
02214 #ifdef WITHZLIB
02215     R_FREE(AR.hidefile->zsp);
02216     R_FREE(AR.hidefile->ziobuffer);
02217 #endif
02218     /* no files should be opened -> nothing to do with handle */
02219
02220     /* #] AR free pointers */
02221
02222     /* outfile */
02223     R_SET(*AR.outfile, FILEHANDLE);
02224     org = (UBYTE*)AR.outfile->POobuffer;
02225     size = AR.outfile->POfull - AR.outfile->POobuffer;
02226     AR.outfile->POobuffer = (WORD*)Malloc1(AR.outfile->POsize, "POobuffer");
02227     if ( size ) {
02228         memcpy(AR.outfile->POobuffer, p, size*sizeof(WORD));
02229         p = (unsigned char*)p + size*sizeof(WORD);
02230     }
02231     ofs = (UBYTE*)AR.outfile->POobuffer - org;
02232     AR.outfile->POstop = (WORD*)((UBYTE*)AR.outfile->POstop + ofs);
02233     AR.outfile->POfill = (WORD*)((UBYTE*)AR.outfile->POfill + ofs);
02234     AR.outfile->POfull = (WORD*)((UBYTE*)AR.outfile->POfull + ofs);
02235     AR.outfile->name = namebufout;
02236 #ifdef WITHPTHREADS
02237     AR.outfile->wPObuffer = AR.outfile->POobuffer;
02238     AR.outfile->wPOstop = AR.outfile->POstop;
02239     AR.outfile->wPOfill = AR.outfile->POfill;
02240     AR.outfile->wPOfull = AR.outfile->POfull;
02241 #endif
02242 #ifdef WITHZLIB
02243     /* zsp and ziobuffer will be allocated when used */
02244     AR.outfile->zsp = 0;
02245     AR.outfile->ziobuffer = 0;
02246 #endif
02247     /* reopen old outfile */
02248 #ifdef WITHMPI
02249     if(PF.me==MASTER)
02250 #endif
02251     if ( AR.outfile->handle >= 0 ) {
02252         if ( CopyFile(sortfile, AR.outfile->name) ) {
02253             MesPrint("ERROR: Could not copy old output sort file %s!",sortfile);
02254             Terminate(-1);
02255         }
02256         AR.outfile->handle = ReOpenFile(AR.outfile->name);
02257         if ( AR.outfile->handle == -1 ) {
02258             MesPrint("ERROR: Could not reopen output sort file %s!",AR.outfile->name);
02259             Terminate(-1);
02260         }
02261         SeekFile(AR.outfile->handle, &AR.outfile->POposition, SEEK_SET);
02262     }
02263
02264     /* hidefile */
02265     R_SET(*AR.hidefile, FILEHANDLE);
02266     AR.hidefile->name = namebufhide;
02267     if ( AR.hidefile->POobuffer ) {
02268         org = (UBYTE*)AR.hidefile->POobuffer;
02269         size = AR.hidefile->POfull - AR.hidefile->POobuffer;
02270         AR.hidefile->POobuffer = (WORD*)Malloc1(AR.hidefile->POsize, "POobuffer");
02271         if ( size ) {
02272             memcpy(AR.hidefile->POobuffer, p, size*sizeof(WORD));
02273             p = (unsigned char*)p + size*sizeof(WORD);
02274         }
02275         ofs = (UBYTE*)AR.hidefile->POobuffer - org;
02276         AR.hidefile->POstop = (WORD*)((UBYTE*)AR.hidefile->POstop + ofs);
02277         AR.hidefile->POfill = (WORD*)((UBYTE*)AR.hidefile->POfill + ofs);
02278         AR.hidefile->POfull = (WORD*)((UBYTE*)AR.hidefile->POfull + ofs);
02279 #ifdef WITHPTHREADS
02280         AR.hidefile->wPObuffer = AR.hidefile->POobuffer;
02281         AR.hidefile->wPOstop = AR.hidefile->POstop;
02282         AR.hidefile->wPOfill = AR.hidefile->POfill;
02283         AR.hidefile->wPOfull = AR.hidefile->POfull;
02284 #endif
02285     }
```

```
02286 #ifdef WITHZLIB
02287         /* zsp and ziobuffer will be allocated when used */
02288         AR.hidefile->zsp = 0;
02289         AR.hidefile->ziobuffer = 0;
02290 #endif
02291     /* reopen old hidefile */
02292     if ( AR.hidefile->handle >= 0 ) {
02293         if ( CopyFile(hidefile, AR.hidefile->name) ) {
02294             MesPrint("ERROR: Could not copy old hide file %s!",hidefile);
02295             Terminate(-1);
02296         }
02297         AR.hidefile->handle = ReOpenFile(AR.hidefile->name);
02298         if ( AR.hidefile->handle == -1 ) {
02299             MesPrint("ERROR: Could not reopen hide file %s!",AR.hidefile->name);
02300             Terminate(-1);
02301         }
02302         SeekFile(AR.hidefile->handle, &AR.hidefile->POposition, SEEK_SET);
02303     }
02304
02305     /* store file */
02306     R_SET(pos, POSITION);
02307     if ( ISNOTZEROPOS(pos) ) {
02308         CloseFile(AR.StoreData.Handle);
02309         R_SET(AR.StoreData, FILEDATA);
02310         if ( CopyFile(storefile, FG.fname) ) {
02311             MesPrint("ERROR: Could not copy old store file %s!",storefile);
02312             Terminate(-1);
02313         }
02314         AR.StoreData.Handle = (WORD)ReOpenFile(FG.fname);
02315         SeekFile(AR.StoreData.Handle, &AR.StoreData.Position, SEEK_SET);
02316     }
02317
02318     R_SET(AR.DefPosition, POSITION);
02319     R_SET(AR.OldTime, LONG);
02320     R_SET(AR.InInBuf, LONG);
02321     R_SET(AR.InHiBuf, LONG);
02322
02323     R_SET(AR.NoCompress, int);
02324     R_SET(AR.gzipCompress, int);
02325
02326     R_SET(AR.outtohide, int);
02327
02328     R_SET(AR.GetFile, WORD);
02329     R_SET(AR.KeptInHold, WORD);
02330     R_SET(AR.BracketOn, WORD);
02331     R_SET(AR.MaxBracket, WORD);
02332     R_SET(AR.CurDum, WORD);
02333     R_SET(AR.DeferFlag, WORD);
02334     R_SET(AR.TePos, WORD);
02335     R_SET(AR.sLevel, WORD);
02336     R_SET(AR.Stage4Name, WORD);
02337     R_SET(AR.GetOneFile, WORD);
02338     R_SET(AR.PolyFun, WORD);
02339     R_SET(AR.PolyFunInv, WORD);
02340     R_SET(AR.PolyFunType, WORD);
02341     R_SET(AR.PolyFunExp, WORD);
02342     R_SET(AR.PolyFunVar, WORD);
02343     R_SET(AR.PolyFunPow, WORD);
02344     R_SET(AR.Eside, WORD);
02345     R_SET(AR.MaxDum, WORD);
02346     R_SET(AR.level, WORD);
02347     R_SET(AR.expchanged, WORD);
02348     R_SET(AR.expflags, WORD);
02349     R_SET(AR.CurExpr, WORD);
02350     R_SET(AR.SortType, WORD);
02351     R_SET(AR.ShortSortCount, WORD);
02352
02353     /* this is usually done in Process(), but sometimes FORM does not
02354        end up executing Process() before it uses the AR.CompressPointer,
02355        so we need to explicitly set it here. */
02356     AR.CompressPointer = AR.CompressBuffer;
02357
02358 #ifdef WITHPTHREADS
02359     for ( j = 0; j < AM.totalnumberofthreads; j++ ) {
02360         R_SET(AB[j]->R.wranfnpair1, int);
02361         R_SET(AB[j]->R.wranfnpair2, int);
02362         R_SET(AB[j]->R.wranfcall, int);
02363         R_SET(AB[j]->R.wranfseed, ULONG);
02364         R_SET(AB[j]->R.wranfia,ULONG*);
02365         if ( AB[j]->R.wranfia ) {
02366             R_COPY_B(AB[j]->R.wranfia, sizeof(ULONG)*AB[j]->R.wranfnpair2, ULONG*);
02367         }
02368     }
02369 #else
02370     R_SET(AR.wranfnpair1, int);
02371     R_SET(AR.wranfnpair2, int);
02372     R_SET(AR.wranfcall, int);
```

```
02373     R_SET(AR.wranfseed, ULONG);
02374     R_SET(AR.wranfia,ULONG*);
02375     if ( AR.wranfia ) {
02376         R_COPY_B(AR.wranfia, sizeof(ULONG)*AR.wranfnpair2, ULONG*);
02377     }
02378 #endif
02379
02380 #ifdef PRINTDEBUG
02381     print_R();
02382 #endif
02383
02384     /*#] AR : */
02385     /*#[ AO :*/
02386 /*
02387     We copy all non-pointer variables.
02388 */
02389     l = sizeof(A.O) - ((UBYTE *)(&(A.O.NumInBrack))-(UBYTE *)(&A.O));
02390     memcpy(&(A.O.NumInBrack), p, l); p = (unsigned char*)p + l;
02391 /*
02392     Now the variables in OptimizeResult
02393 */
02394     memcpy(&(A.O.OptimizeResult),p,sizeof(OPTIMIZERESULT));
02395         p = (unsigned char*)p + sizeof(OPTIMIZERESULT);
02396
02397     if ( A.O.OptimizeResult.codesize > 0 ) {
02398         R_COPY_B(A.O.OptimizeResult.code,A.O.OptimizeResult.codesize*sizeof(WORD),WORD *);
02399     }
02400     R_COPY_S(A.O.OptimizeResult.nameofexpr,UBYTE *);
02401 /*
02402     And now the dictionaries. We know how many there are. We also know
02403     how many elements the array AO.Dictionaries should have.
02404 */
02405     if ( AO.SizeDictionaries > 0 ) {
02406         AO.Dictionaries = (DICTIONARY **)Malloc1(AO.SizeDictionaries*sizeof(DICTIONARY *),
02407                     "Dictionaries");
02408         for ( i = 0; i < AO.NumDictionaries; i++ ) {
02409             R_SET(l,LONG)
02410             AO.Dictionaries[i] = DictFromBytes(p);
02411             p = (char *)p + l;
02412         }
02413     }
02414     /*#] AO :*/
02415 #ifdef WITHMPI
02416     /*#[ PF : */
02417     {/*Block*/
02418         int numtasks;
02419         R_SET(numtasks, int);
02420         if(numtasks!=PF.numtasks){
02421             MesPrint("%d number of tasks expected instead of %d; use mpirun -np %d",
02422                             numtasks,PF.numtasks,numtasks);
02423             if(PF.me!=MASTER)
02424                 remove(RecoveryFilename());
02425             Terminate(-1);
02426         }
02427     }/*Block*/
02428     R_SET(PF.rhsInParallel, int);
02429     R_SET(PF.exprbufsize, int);
02430     R_SET(PF.log, int);
02431     /*#] PF : */
02432 #endif
02433
02434 #ifdef WITHPTHREADS
02435     /* read timing information of individual threads */
02436     R_SET(i, int);
02437     for ( j=1; j<AM.totalnumberofthreads; ++j ) {
02438         /* ... and correcting OldTime */
02439         AB[j]->R.OldTime = -(*((LONG*)p+j));
02440     }
02441     WriteTimerInfo((LONG*)p,(LONG *)((unsigned char*)p + i*(LONG)sizeof(LONG)));
02442     p = (unsigned char*)p + 2*i*(LONG)sizeof(LONG);
02443 #endif /* ifdef WITHPTHREADS */
02444
02445     if ( fclose(fd) ) return(__LINE__);
02446
02447     M_free(buf,"recovery buffer");
02448
02449     /* cares about data in S_const */
02450     UpdatePositions();
02451     AT.SS = AT.S0;
02452 /*
02453     Set the checkpoint parameter right for the next checkpoint.
02454 */
02455     AC.CheckpointStamp = TimeWallClock(1);
02456
02457     done_snapshot = 1;
02458     MesPrint("done."); fflush(0);
02459
```

```
02460     return(0);
02461 }
02462
02463 /*
02464     #] DoRecovery :
02465     #[ DoSnapshot :
02466 */
02467
02483 static int DoSnapshot(int moduletype)
02484 {
02485     GETIDENTITY
02486     FILE *fd;
02487     POSITION pos = {0};
02488     int i, j;
02489     LONG l;
02490     WORD *w;
02491     void *adr;
02492 #ifdef WITHPTHREADS
02493     LONG *longp,*longpp;
02494 #endif /* ifdef WITHPTHREADS */
02495
02496     MesPrint("Saving recovery point ... %"); fflush(0);
02497 #ifdef PRINTTIMEMARKS
02498     MesPrint("\n");
02499 #endif
02500
02501     if ( !(fd = fopen(intermedfile, "wb")) ) return(__LINE__);
02502
02503     /* reserve space in the file for a length field */
02504     if ( fwrite(&pos, 1, sizeof(POSITION), fd) != sizeof(POSITION) ) {
02505         fclose(fd);
02506         return(__LINE__);
02507     }
02508
02509     /* write moduletype */
02510     if ( fwrite(&moduletype, 1, sizeof(int), fd) != sizeof(int) ) {
02511         fclose(fd);
02512         return(__LINE__);
02513     }
02514
02515     /*#[ AM :*/
02516
02517     /* since most values don't change during execution, AM doesn't need to be
02518      * written as a whole. all values will be correctly set when starting up
02519      * anyway. only the exceptions need to be taken care of. see MakeGlobal()
02520      * and PopVariables() in execute.c. */
02521
02522     ANNOUNCE(AM)
02523     S_WRITE_B(&AM.hparallelflag, sizeof(int));
02524     S_WRITE_B(&AM.gparallelflag, sizeof(int));
02525     S_WRITE_B(&AM.gCodesFlag, sizeof(int));
02526     S_WRITE_B(&AM.gNamesFlag, sizeof(int));
02527     S_WRITE_B(&AM.gStatsFlag, sizeof(int));
02528     S_WRITE_B(&AM.gTokensWriteFlag, sizeof(int));
02529     S_WRITE_B(&AM.gNoSpacesInNumbers, sizeof(int));
02530     S_WRITE_B(&AM.gIndentSpace, sizeof(WORD));
02531     S_WRITE_B(&AM.gUnitTrace, sizeof(WORD));
02532     S_WRITE_B(&AM.gDefDim, sizeof(int));
02533     S_WRITE_B(&AM.gDefDim4, sizeof(int));
02534     S_WRITE_B(&AM.gncmod, sizeof(WORD));
02535     S_WRITE_B(&AM.gnpowmod, sizeof(WORD));
02536     S_WRITE_B(&AM.gmodmode, sizeof(WORD));
02537     S_WRITE_B(&AM.gOutputMode, sizeof(WORD));
02538     S_WRITE_B(&AM.gCnumpows, sizeof(WORD));
02539     S_WRITE_B(&AM.gOutputSpaces, sizeof(WORD));
02540     S_WRITE_B(&AM.gOutNumberType, sizeof(WORD));
02541     S_WRITE_B(&AM.gfunpowers, sizeof(int));
02542     S_WRITE_B(&AM.gPolyFun, sizeof(WORD));
02543     S_WRITE_B(&AM.gPolyFunInv, sizeof(WORD));
02544     S_WRITE_B(&AM.gPolyFunType, sizeof(WORD));
02545     S_WRITE_B(&AM.gPolyFunExp, sizeof(WORD));
02546     S_WRITE_B(&AM.gPolyFunVar, sizeof(WORD));
02547     S_WRITE_B(&AM.gPolyFunPow, sizeof(WORD));
02548     S_WRITE_B(&AM.gProcessBucketSize, sizeof(LONG));
02549     S_WRITE_B(&AM.OldChildTime, sizeof(LONG));
02550     S_WRITE_B(&AM.OldSecTime, sizeof(LONG));
02551     S_WRITE_B(&AM.OldMilliTime, sizeof(LONG));
02552     S_WRITE_B(&AM.gproperorderflag, sizeof(int));
02553     S_WRITE_B(&AM.gThreadBucketSize, sizeof(LONG));
02554     S_WRITE_B(&AM.gSizeCommuteInSet, sizeof(int));
02555     S_WRITE_B(&AM.gThreadStats, sizeof(int));
02556     S_WRITE_B(&AM.gFinalStats, sizeof(int));
02557     S_WRITE_B(&AM.gThreadsFlag, sizeof(int));
02558     S_WRITE_B(&AM.gThreadBalancing, sizeof(int));
02559     S_WRITE_B(&AM.gThreadSortFileSynch, sizeof(int));
02560     S_WRITE_B(&AM.gProcessStats, sizeof(int));
02561     S_WRITE_B(&AM.gOldParallelStats, sizeof(int));
```

```
02562     S_WRITE_B(&AM.gSortType, sizeof(int));
02563     S_WRITE_B(&AM.gShortStatsMax, sizeof(WORD));
02564     S_WRITE_B(&AM.gIsFortran90, sizeof(int));
02565     adr = &AM.dollarzero;
02566     S_WRITE_B(&adr, sizeof(void*));
02567     S_WRITE_B(&AM.gFortran90Kind,sizeof(UBYTE *));
02568     S_WRITE_S(AM.gFortran90Kind);
02569
02570     S_WRITE_S(AM.gextrasym);
02571     S_WRITE_S(AM.ggextrasym);
02572
02573     S_WRITE_B(&AM.PrintTotalSize,sizeof(int));
02574     S_WRITE_B(&AM.fbuffersize,sizeof(int));
02575     S_WRITE_B(&AM.gOldFactArgFlag,sizeof(int));
02576     S_WRITE_B(&AM.ggOldFactArgFlag,sizeof(int));
02577
02578     S_WRITE_B(&AM.gnumextrasym,sizeof(int));
02579     S_WRITE_B(&AM.ggnumextrasym,sizeof(int));
02580     S_WRITE_B(&AM.NumSpectatorFiles,sizeof(int));
02581     S_WRITE_B(&AM.SizeForSpectatorFiles,sizeof(int));
02582     S_WRITE_B(&AM.gOldGCDflag,sizeof(int));
02583     S_WRITE_B(&AM.ggOldGCDflag,sizeof(int));
02584     S_WRITE_B(&AM.gWTimeStatsFlag, sizeof(int));
02585
02586     S_WRITE_B(&AM.Path,sizeof(UBYTE *));
02587     S_WRITE_S(AM.Path);
02588
02589     S_WRITE_B(&AM.FromStdin,sizeof(BOOL));
02590     S_WRITE_B(&AM.IgnoreDeprecation,sizeof(BOOL));
02591
02592     /*#] AM :*/
02593     /*#[ AC :*/
02594
02595     /* we write AC as a whole and then write all additional data step by step.
02596      * AC.DubiousList doesn't need to be treated, because it should be empty. */
02597
02598     ANNOUNCE(AC)
02599     S_WRITE_B(&AC, sizeof(struct C_const));
02600
02601     S_WRITE_NAMETREE(AC.dollarnames);
02602     S_WRITE_NAMETREE(AC.exprnames);
02603     S_WRITE_NAMETREE(AC.varnames);
02604
02605     S_WRITE_LIST(AC.ChannelList);
02606     for ( i=0; i<AC.ChannelList.num; ++i ) {
02607         S_WRITE_S(channels[i].name);
02608     }
02609
02610     ANNOUNCE(AC.FunctionList)
02611     S_WRITE_LIST(AC.FunctionList);
02612     for ( i=0; i<AC.FunctionList.num; ++i ) {
02613         /* if the function is a table */
02614         if ( functions[i].tabl ) {
02615             TABLES tabl = functions[i].tabl;
02616             S_WRITE_B(tabl, sizeof(struct TaBlEs));
02617             if ( tabl->tablepointers ) {
02618                 if ( tabl->sparse ) {
02619                     /* sparse tables. reserved holds number of allocated
02620                      * elements. the size of an element is numind plus
02621                      * TABLEEXTENSION times the size of WORD. */
02622                     S_WRITE_B(tabl->tablepointers,
02623                         tabl->reserved*sizeof(WORD)*(tabl->numind+TABLEEXTENSION));
02624                 }
02625                 else {
02626                     /* matrix like tables. */
02627                     S_WRITE_B(tabl->tablepointers,
02628                         TABLEEXTENSION*sizeof(WORD)*(tabl->totind));
02629                 }
02630             }
02631             S_WRITE_B(tabl->prototype, tabl->prototypeSize);
02632             S_WRITE_B(tabl->mm, tabl->numind*(LONG)sizeof(MINMAX));
02633             S_WRITE_B(tabl->flags, tabl->numind*(LONG)sizeof(WORD));
02634             if ( tabl->sparse ) {
02635                 S_WRITE_B(tabl->boomlijst, tabl->MaxTreeSize*(LONG)sizeof(COMPTREE));
02636                 S_WRITE_S(tabl->argtail);
02637             }
02638             S_WRITE_B(tabl->buffers, tabl->bufferssize*(LONG)sizeof(WORD));
02639             if ( tabl->spare ) {
02640                 TABLES spare = tabl->spare;
02641                 S_WRITE_B(spare, sizeof(struct TaBlEs));
02642                 if ( spare->tablepointers ) {
02643                     if ( spare->sparse ) {
02644                         /* sparse tables */
02645                         S_WRITE_B(spare->tablepointers,
02646                             spare->reserved*sizeof(WORD)*(spare->numind+TABLEEXTENSION));
02647                     }
02648                     else {
```

```
02649                       /* matrix like tables */
02650                       S_WRITE_B(spare->tablepointers,
02651                           TABLEEXTENSION*sizeof(WORD)*(spare->totind));
02652                   }
02653               }
02654               S_WRITE_B(spare->mm, spare->numind*(LONG)sizeof(MINMAX));
02655               S_WRITE_B(spare->flags, spare->numind*(LONG)sizeof(WORD));
02656               if ( spare->sparse ) {
02657                   S_WRITE_B(spare->boomlijst, spare->MaxTreeSize*(LONG)sizeof(COMPTREE));
02658               }
02659               S_WRITE_B(spare->buffers, spare->bufferssize*(LONG)sizeof(WORD));
02660           }
02661       }
02662   }
02663
02664   ANNOUNCE(AC.ExpressionList)
02665   S_WRITE_LIST(AC.ExpressionList);
02666   for ( i=0; i<AC.ExpressionList.num; ++i ) {
02667       EXPRESSIONS ex = Expressions + i;
02668       if ( ex->renum ) {
02669           S_WRITE_B(ex->renum, sizeof(struct ReNuMbEr));
02670           /* there is one dynamically allocated buffer for struct ReNuMbEr and
02671            * symb.lo points to its beginning. the size of the buffer is not
02672            * stored anywhere but we know it is 2*sizeof(WORD)*N, where N is
02673            * the number of all vectors, indices, functions and symbols. since
02674            * funum points into the buffer at a distance 2N-[Number of
02675            * functions] from symb.lo (see GetTable() in store.c), we can
02676            * calculate the buffer size by some pointer arithmetic. the size is
02677            * then written to the file. */
02678           l = ex->renum->funnum - ex->renum->symb.lo;
02679           l += ex->renum->funnum - ex->renum->func.lo;
02680           S_WRITE_B(&l, sizeof(size_t));
02681           S_WRITE_B(ex->renum->symb.lo, l);
02682       }
02683       if ( ex->bracketinfo ) {
02684           S_WRITE_B(ex->bracketinfo, sizeof(BRACKETINFO));
02685           S_WRITE_B(ex->bracketinfo->indexbuffer,
    ex->bracketinfo->indexbuffersize*sizeof(BRACKETINDEX));
02686           S_WRITE_B(ex->bracketinfo->bracketbuffer,
    ex->bracketinfo->bracketbuffersize*sizeof(WORD));
02687       }
02688       if ( ex->newbracketinfo ) {
02689           S_WRITE_B(ex->newbracketinfo, sizeof(BRACKETINFO));
02690           S_WRITE_B(ex->newbracketinfo->indexbuffer,
    ex->newbracketinfo->indexbuffersize*sizeof(BRACKETINDEX));
02691           S_WRITE_B(ex->newbracketinfo->bracketbuffer,
    ex->newbracketinfo->bracketbuffersize*sizeof(WORD));
02692       }
02693       /* don't need to write ex->renumlists */
02694       if ( ex->inmem ) {
02695           /* size of the inmem buffer has to be determined. we use the fact
02696            * that the end of an expression is marked by a zero. */
02697           w = ex->inmem;
02698           while ( *w++ ) ;
02699           l = w - ex->inmem;
02700           S_WRITE_B(&l, sizeof(size_t));
02701           S_WRITE_B(ex->inmem, l);
02702       }
02703   }
02704
02705   ANNOUNCE(AC.IndexList)
02706   S_WRITE_LIST(AC.IndexList);
02707   S_WRITE_LIST(AC.SetElementList);
02708   S_WRITE_LIST(AC.SetList);
02709   S_WRITE_LIST(AC.SymbolList);
02710   S_WRITE_LIST(AC.VectorList);
02711
02712   ANNOUNCE(AC.TableBaseList)
02713   S_WRITE_LIST(AC.TableBaseList);
02714   for ( i=0; i<AC.TableBaseList.num; ++i ) {
02715       /* see struct dbase in minos.h */
02716       if ( tablebases[i].iblocks ) {
02717           S_WRITE_B(tablebases[i].iblocks, tablebases[i].info.numberofindexblocks *
    sizeof(INDEXBLOCK*));
02718           for ( j=0; j<tablebases[i].info.numberofindexblocks; ++j ) {
02719               if ( tablebases[i].iblocks[j] ) {
02720                   S_WRITE_B(tablebases[i].iblocks[j], sizeof(INDEXBLOCK));
02721               }
02722           }
02723       }
02724       if ( tablebases[i].nblocks ) {
02725           S_WRITE_B(tablebases[i].nblocks, tablebases[i].info.numberofnamesblocks *
    sizeof(NAMESBLOCK*));
02726           for ( j=0; j<tablebases[i].info.numberofnamesblocks; ++j ) {
02727               if ( tablebases[i].nblocks[j] ) {
02728                   S_WRITE_B(tablebases[i].nblocks[j], sizeof(NAMESBLOCK));
02729               }
```

```
02730             }
02731         }
02732         S_WRITE_S(tablebases[i].name);
02733         S_WRITE_S(tablebases[i].fullname);
02734         S_WRITE_S(tablebases[i].tablenames);
02735     }
02736
02737     ANNOUNCE(AC.cbufList)
02738     S_WRITE_LIST(AC.cbufList);
02739     for ( i=0; i<AC.cbufList.num; ++i ) {
02740         S_WRITE_B(cbuf[i].Buffer, cbuf[i].BufferSize*sizeof(WORD));
02741         /* see inicbufs in comtool.c */
02742         S_WRITE_B(cbuf[i].lhs, cbuf[i].maxlhs*(LONG)sizeof(WORD*));
02743         S_WRITE_B(cbuf[i].rhs, cbuf[i].maxrhs*(LONG)(sizeof(WORD*)+2*sizeof(LONG)+2*sizeof(WORD)));
02744         if ( cbuf[i].boomlijst ) {
02745             S_WRITE_B(cbuf[i].boomlijst, cbuf[i].MaxTreeSize*(LONG)sizeof(COMPTREE));
02746         }
02747     }
02748
02749     S_WRITE_LIST(AC.AutoSymbolList);
02750     S_WRITE_LIST(AC.AutoIndexList);
02751     S_WRITE_LIST(AC.AutoVectorList);
02752     S_WRITE_LIST(AC.AutoFunctionList);
02753
02754     S_WRITE_NAMETREE(AC.autonames);
02755
02756     ANNOUNCE(AC.Streams)
02757     S_WRITE_B(AC.Streams, AC.MaxNumStreams*(LONG)sizeof(STREAM));
02758     for ( i=0; i<AC.NumStreams; ++i ) {
02759         if ( AC.Streams[i].inbuffer ) {
02760             S_WRITE_B(AC.Streams[i].buffer, AC.Streams[i].inbuffer);
02761         }
02762         S_WRITE_S(AC.Streams[i].FoldName);
02763         S_WRITE_S(AC.Streams[i].name);
02764     }
02765
02766     if ( AC.termstack ) {
02767         S_WRITE_B(AC.termstack, AC.maxtermlevel*(LONG)sizeof(LONG));
02768     }
02769
02770     if ( AC.termsortstack ) {
02771         S_WRITE_B(AC.termsortstack, AC.maxtermlevel*(LONG)sizeof(LONG));
02772     }
02773
02774     S_WRITE_B(AC.cmod, AM.MaxTal*4*(LONG)sizeof(UWORD));
02775
02776     if ( AC.IfHeap ) {
02777         S_WRITE_B(AC.IfHeap, (LONG)sizeof(LONG)*(AC.MaxIf+1));
02778         S_WRITE_B(AC.IfCount, (LONG)sizeof(LONG)*(AC.MaxIf+1));
02779     }
02780
02781     l = AC.iStop - AC.iBuffer + 2;
02782     S_WRITE_B(AC.iBuffer, l);
02783
02784     if ( AC.LabelNames ) {
02785         S_WRITE_B(AC.LabelNames, AC.MaxLabels*(LONG)(sizeof(UBYTE*)+sizeof(WORD)));
02786         for ( i=0; i<AC.NumLabels; ++i ) {
02787             S_WRITE_S(AC.LabelNames[i]);
02788         }
02789     }
02790
02791     S_WRITE_B(AC.FixIndices, AM.OffsetIndex*(LONG)sizeof(WORD));
02792
02793     if ( AC.termsumcheck ) {
02794         S_WRITE_B(AC.termsumcheck, AC.maxtermlevel*(LONG)sizeof(WORD));
02795     }
02796
02797     S_WRITE_B(AC.WildcardNames, AC.WildcardBufferSize);
02798
02799     if ( AC.tokens ) {
02800         l = AC.toptokens - AC.tokens;
02801         if ( l ) {
02802             S_WRITE_B(AC.tokens, l);
02803         }
02804     }
02805
02806     S_WRITE_B(AC.tokenarglevel, AM.MaxParLevel*(LONG)sizeof(WORD));
02807
02808     S_WRITE_S(AC.Fortran90Kind);
02809
02810 #ifdef WITHPTHREADS
02811     if ( AC.inputnumbers ) {
02812         S_WRITE_B(AC.inputnumbers, AC.sizepfirstnum*(LONG)(sizeof(WORD)+sizeof(LONG)));
02813     }
02814 #endif /* ifdef WITHPTHREADS */
02815
02816     if ( AC.IfSumCheck ) {
```

```
02817         S_WRITE_B(AC.IfSumCheck, (LONG)sizeof(WORD)*(AC.MaxIf+1));
02818     }
02819     if ( AC.CommuteInSet ) {
02820         S_WRITE_B(AC.CommuteInSet, (LONG)sizeof(WORD)*(AC.SizeCommuteInSet+1));
02821     }
02822
02823     S_WRITE_S(AC.CheckpointRunAfter);
02824     S_WRITE_S(AC.CheckpointRunBefore);
02825
02826     S_WRITE_S(AC.extrasym);
02827
02828     /*#] AC :*/
02829     /*#[ AP :*/
02830
02831     /* we write AP as a whole and then write all additional data step by step. */
02832
02833     ANNOUNCE(AP)
02834     S_WRITE_B(&AP, sizeof(struct P_const));
02835
02836     S_WRITE_LIST(AP.DollarList);
02837     for ( i=0; i<AP.DollarList.num; ++i ) {
02838         DOLLARS d = Dollars + i;
02839         S_WRITE_DOLLAR(Dollars[i]);
02840         if ( d->nfactors > 1 ) {
02841             S_WRITE_B(&(d->factors),sizeof(FACDOLLAR)*d->nfactors);
02842             for ( j = 0; j < d->nfactors; j++ ) {
02843                 if ( d->factors[j].size > 0 ) {
02844                     S_WRITE_B(&(d->factors[i].where),sizeof(WORD)*(d->factors[j].size+1));
02845                 }
02846             }
02847         }
02848     }
02849
02850     S_WRITE_LIST(AP.PreVarList);
02851     for ( i=0; i<AP.PreVarList.num; ++i ) {
02852         /* there is one dynamically allocated buffer in struct pReVaR holding
02853          * the strings name, value and several argnames. the size of the buffer
02854          * can be calculated by adding up their string lengths. */
02855         l = strlen((char*)PreVar[i].name) + 1;
02856         if ( PreVar[i].value ) {
02857             l += strlen((char*)(PreVar[i].name+l)) + 1;
02858         }
02859         for ( j=0; j<PreVar[i].nargs; ++j ) {
02860             l += strlen((char*)(PreVar[i].name+l)) + 1;
02861         }
02862         S_WRITE_B(&l, sizeof(size_t));
02863         S_WRITE_B(PreVar[i].name, l);
02864     }
02865
02866     ANNOUNCE(AP.LoopList)
02867     S_WRITE_LIST(AP.LoopList);
02868     for ( i=0; i<AP.LoopList.num; ++i ) {
02869         S_WRITE_B(DoLoops[i].p.buffer, DoLoops[i].p.size);
02870         if ( DoLoops[i].type == ONEEXPRESSION ) {
02871             /* do loops with an expression keep this expression in a dynamically
02872              * allocated buffer in dollarname */
02873             S_WRITE_S(DoLoops[i].dollarname);
02874         }
02875     }
02876
02877     S_WRITE_LIST(AP.ProcList);
02878     for ( i=0; i<AP.ProcList.num; ++i ) {
02879         if ( Procedures[i].p.size ) {
02880             if ( Procedures[i].loadmode != 1 ) {
02881                 S_WRITE_B(Procedures[i].p.buffer, Procedures[i].p.size);
02882             }
02883             else {
02884                 for ( j=0; j<AP.ProcList.num; ++j ) {
02885                     if ( Procedures[i].p.buffer == Procedures[j].p.buffer ) {
02886                         break;
02887                     }
02888                 }
02889                 if ( j == AP.ProcList.num ) {
02890                     MesPrint("Error writing procedures to recovery file!");
02891                 }
02892                 S_WRITE_B(&j, sizeof(int));
02893             }
02894         }
02895         S_WRITE_S(Procedures[i].name);
02896     }
02897
02898     S_WRITE_B(AP.PreSwitchStrings, (AP.NumPreSwitchStrings+1)*(LONG)sizeof(UBYTE*));
02899     for ( i=1; i<=AP.PreSwitchLevel; ++i ) {
02900         S_WRITE_S(AP.PreSwitchStrings[i]);
02901     }
02902
02903     S_WRITE_B(AP.preStart, AP.pSize);
```

```
02904
02905     S_WRITE_S(AP.procedureExtension);
02906     S_WRITE_S(AP.cprocedureExtension);
02907
02908     S_WRITE_B(AP.PreAssignStack, AP.MaxPreAssignLevel*(LONG)sizeof(LONG));
02909     S_WRITE_B(AP.PreIfStack, AP.MaxPreIfLevel*(LONG)sizeof(int));
02910     S_WRITE_B(AP.PreSwitchModes, (AP.NumPreSwitchStrings+1)*(LONG)sizeof(int));
02911     S_WRITE_B(AP.PreTypes, (AP.MaxPreTypes+1)*(LONG)sizeof(int));
02912
02913     /*#] AP :*/
02914     /*#[ AR :*/
02915
02916     ANNOUNCE(AR)
02917     /* to remember which entry in AR.Fscr corresponds to the infile */
02918     l = AR.infile - AR.Fscr;
02919     S_WRITE_B(&l, sizeof(LONG));
02920
02921     /* write the FILEHANDLEs */
02922     S_WRITE_B(AR.outfile, sizeof(FILEHANDLE));
02923     l = AR.outfile->POfull - AR.outfile->PObuffer;
02924     if ( l ) {
02925         S_WRITE_B(AR.outfile->PObuffer, l*sizeof(WORD));
02926     }
02927     S_WRITE_B(AR.hidefile, sizeof(FILEHANDLE));
02928     l = AR.hidefile->POfull - AR.hidefile->PObuffer;
02929     if ( l ) {
02930         S_WRITE_B(AR.hidefile->PObuffer, l*sizeof(WORD));
02931     }
02932
02933     S_WRITE_B(&AR.StoreData.Fill, sizeof(POSITION));
02934     if ( ISNOTZEROPOS(AR.StoreData.Fill) ) {
02935         S_WRITE_B(&AR.StoreData, sizeof(FILEDATA));
02936     }
02937
02938     S_WRITE_B(&AR.DefPosition, sizeof(POSITION));
02939
02940     l = TimeCPU(1); l = -l;
02941     S_WRITE_B(&l, sizeof(LONG));
02942
02943     ANNOUNCE(AR.InInBuf)
02944     S_WRITE_B(&AR.InInBuf, sizeof(LONG));
02945     S_WRITE_B(&AR.InHiBuf, sizeof(LONG));
02946
02947     S_WRITE_B(&AR.NoCompress, sizeof(int));
02948     S_WRITE_B(&AR.gzipCompress, sizeof(int));
02949
02950     S_WRITE_B(&AR.outtohide, sizeof(int));
02951
02952     S_WRITE_B(&AR.GetFile, sizeof(WORD));
02953     S_WRITE_B(&AR.KeptInHold, sizeof(WORD));
02954     S_WRITE_B(&AR.BracketOn, sizeof(WORD));
02955     S_WRITE_B(&AR.MaxBracket, sizeof(WORD));
02956     S_WRITE_B(&AR.CurDum, sizeof(WORD));
02957     S_WRITE_B(&AR.DeferFlag, sizeof(WORD));
02958     S_WRITE_B(&AR.TePos, sizeof(WORD));
02959     S_WRITE_B(&AR.sLevel, sizeof(WORD));
02960     S_WRITE_B(&AR.Stage4Name, sizeof(WORD));
02961     S_WRITE_B(&AR.GetOneFile, sizeof(WORD));
02962     S_WRITE_B(&AR.PolyFun, sizeof(WORD));
02963     S_WRITE_B(&AR.PolyFunInv, sizeof(WORD));
02964     S_WRITE_B(&AR.PolyFunType, sizeof(WORD));
02965     S_WRITE_B(&AR.PolyFunExp, sizeof(WORD));
02966     S_WRITE_B(&AR.PolyFunVar, sizeof(WORD));
02967     S_WRITE_B(&AR.PolyFunPow, sizeof(WORD));
02968     S_WRITE_B(&AR.Eside, sizeof(WORD));
02969     S_WRITE_B(&AR.MaxDum, sizeof(WORD));
02970     S_WRITE_B(&AR.level, sizeof(WORD));
02971     S_WRITE_B(&AR.expchanged, sizeof(WORD));
02972     S_WRITE_B(&AR.expflags, sizeof(WORD));
02973     S_WRITE_B(&AR.CurExpr, sizeof(WORD));
02974     S_WRITE_B(&AR.SortType, sizeof(WORD));
02975     S_WRITE_B(&AR.ShortSortCount, sizeof(WORD));
02976
02977 #ifdef WITHPTHREADS
02978     for ( j = 0; j < AM.totalnumberofthreads; j++ ) {
02979         S_WRITE_B(&(AB[j]->R.wranfnpair1), sizeof(int));
02980         S_WRITE_B(&(AB[j]->R.wranfnpair2), sizeof(int));
02981         S_WRITE_B(&(AB[j]->R.wranfcall), sizeof(int));
02982         S_WRITE_B(&(AB[j]->R.wranfseed), sizeof(ULONG));
02983         S_WRITE_B(&(AB[j]->R.wranfia),sizeof(ULONG *));
02984         if ( AB[j]->R.wranfia ) {
02985             S_WRITE_B(AB[j]->R.wranfia, sizeof(ULONG)*AB[j]->R.wranfnpair2);
02986         }
02987     }
02988 #else
02989     S_WRITE_B(&(AR.wranfnpair1), sizeof(int));
02990     S_WRITE_B(&(AR.wranfnpair2), sizeof(int));
```

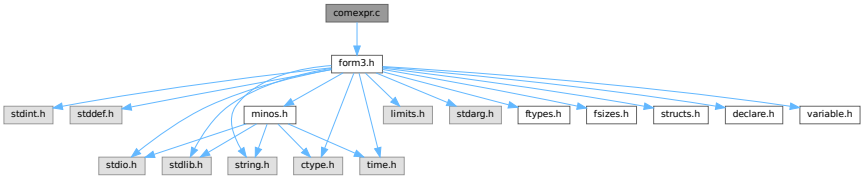
```
02991     S_WRITE_B(&(AR.wranfcall), sizeof(int));
02992     S_WRITE_B(&(AR.wranfseed), sizeof(ULONG));
02993     S_WRITE_B(&(AR.wranfia),sizeof(ULONG *));
02994     if ( AR.wranfia ) {
02995         S_WRITE_B(AR.wranfia, sizeof(ULONG)*AR.wranfnpair2);
02996     }
02997 #endif
02998 
02999     /*#] AR :*/
03000     /*#[ AO :*/
03001 /*
03002     We copy all non-pointer variables.
03003 */
03004     ANNOUNCE(AO)
03005     l = sizeof(A.O) - ((UBYTE *)(&(A.O.NumInBrack))-(UBYTE *)(&A.O));
03006     S_WRITE_B(&(A.O.NumInBrack),l);
03007 /*
03008     Now the variables in OptimizeResult
03009 */
03010     S_WRITE_B(&(A.O.OptimizeResult),sizeof(OPTIMIZERESULT));
03011     if ( A.O.OptimizeResult.codesize > 0 ) {
03012         S_WRITE_B(A.O.OptimizeResult.code,A.O.OptimizeResult.codesize*sizeof(WORD));
03013     }
03014     S_WRITE_S(A.O.OptimizeResult.nameofexpr);
03015 /*
03016     And now the dictionaries.
03017     We write each dictionary to a buffer and get the size of that buffer.
03018     Then we write the size and the buffer.
03019 */
03020     for ( i = 0; i < AO.NumDictionaries; i++ ) {
03021         l = DictToBytes(AO.Dictionaries[i],(UBYTE *)(AT.WorkPointer));
03022         S_WRITE_B(&l,sizeof(LONG));
03023         S_WRITE_B(AT.WorkPointer,l);
03024     }
03025 
03026     /*#] AO :*/
03027     /*#[ PF :*/
03028 #ifdef WITHMPI
03029     S_WRITE_B(&PF.numtasks, sizeof(int));
03030     S_WRITE_B(&PF.rhsInParallel, sizeof(int));
03031     S_WRITE_B(&PF.exprbufsize, sizeof(int));
03032     S_WRITE_B(&PF.log, sizeof(int));
03033 #endif
03034     /*#] PF :*/
03035 
03036 #ifdef WITHPTHREADS
03037 
03038     ANNOUNCE(GetTimerInfo)
03039 /*
03040     write timing information of individual threads
03041 */
03042     i = GetTimerInfo(&longp,&longpp);
03043     S_WRITE_B(&i, sizeof(int));
03044     S_WRITE_B(longp, i*(LONG)sizeof(LONG));
03045     S_WRITE_B(&i, sizeof(int));
03046     S_WRITE_B(longpp, i*(LONG)sizeof(LONG));
03047 
03048 #endif
03049 
03050     S_FLUSH_B /* because we will call fwrite() directly in the following code */
03051 
03052     /* save length of data at the beginning of the file */
03053     ANNOUNCE(file length)
03054     SETBASEPOSITION(pos, (ftell(fd)));
03055     fseek(fd, 0, SEEK_SET);
03056     if ( fwrite(&pos, 1, sizeof(POSITION), fd) != sizeof(POSITION) ) return(__LINE__);
03057     fseek(fd, BASEPOSITION(pos), SEEK_SET);
03058 
03059     ANNOUNCE(file close)
03060     if ( fclose(fd) ) return(__LINE__);
03061 #ifdef WITHMPI
03062     if ( PF.me == MASTER ) {
03063 #endif
03064 /*
03065         copy store file if necessary
03066 */
03067         ANNOUNCE(copy store file)
03068         if ( ISNOTZEROPOS(AR.StoreData.Fill) ) {
03069             if ( CopyFile(FG.fname, storefile) ) return(__LINE__);
03070         }
03071 /*
03072         copy sort file if necessary
03073 */
03074         ANNOUNCE(copy sort file)
03075         if ( AR.outfile->handle >= 0 ) {
03076             if ( CopyFile(AR.outfile->name, sortfile) ) return(__LINE__);
03077         }
```

```
03078 /*
03079         copy hide file if necessary
03080 */
03081         ANNOUNCE(copy hide file)
03082         if ( AR.hidefile->handle >= 0 ) {
03083             if ( CopyFile(AR.hidefile->name, hidefile) ) return(__LINE__);
03084         }
03085 #ifdef WITHMPI
03086     }
03087     /*
03088      * For ParFORM, the renaming will be performed after the master got
03089      * all recovery files from the slaves.
03090      */
03091 #else
03092 /*
03093     make the intermediate file the recovery file
03094 */
03095     ANNOUNCE(rename intermediate file)
03096     if ( rename(intermedfile, recoveryfile) ) return(__LINE__);
03097
03098     done_snapshot = 1;
03099
03100     MesPrint("done."); fflush(0);
03101 #endif
03102
03103 #ifdef PRINTDEBUG
03104     print_M();
03105     print_C();
03106     print_P();
03107     print_R();
03108 #endif
03109
03110     return(0);
03111 }
03112
03113 /*
03114     #] DoSnapshot :
03115     #[ DoCheckpoint :
03116 */
03117
03122 void DoCheckpoint(int moduletype)
03123 {
03124     int error;
03125     LONG timestamp = TimeWallClock(1);
03126 #ifdef WITHMPI
03127     if(PF.me == MASTER){
03128 #endif
03129     if ( timestamp - AC.CheckpointStamp >= AC.CheckpointInterval ) {
03130         char argbuf[20];
03131         int retvalue = 0;
03132         if ( AC.CheckpointRunBefore ) {
03133             size_t l, l2;
03134             char *str;
03135             l = strlen(AC.CheckpointRunBefore);
03136             NumToStr((UBYTE*)argbuf, AC.CModule);
03137             l2 = strlen(argbuf);
03138             str = (char*)Malloc1(l+l2+2, "callbefore");
03139             strcpy(str, AC.CheckpointRunBefore);
03140             *(str+l) = ' ';
03141             strcpy(str+l+1, argbuf);
03142             retvalue = system(str);
03143             M_free(str, "callbefore");
03144             if ( retvalue ) {
03145                 MesPrint("Script returned error -> no recovery file will be created.");
03146             }
03147         }
03148 #ifdef WITHMPI
03149         /* Confirm slaves to make snapshots. */
03150         PF_BroadcastNumber(retvalue == 0);
03151 #endif
03152         if ( retvalue == 0 ) {
03153             if ( (error = DoSnapshot(moduletype)) ) {
03154                 MesPrint("Error creating recovery files: %d", error);
03155             }
03156 #ifdef WITHMPI
03157             {
03158             int i;
03159             /*get recovery files from slaves:*/
03160             for(i=1; i<PF.numtasks;i++){
03161                 FILE *fd;
03162                 const char *tmpnam = PF_recoveryfile('m', i, 1);
03163                 fd = fopen(tmpnam, "w");
03164                 if(fd == NULL){
03165                     MesPrint("Error opening recovery file for slave %d",i);
03166                     Terminate(-1);
03167                 }/*if(fd == NULL)*/
03168                 retvalue=PF_RecvFile(i,fd);
```

```
03169                 if(retvalue<=0){
03170                     MesPrint("Error receiving recovery file from slave %d",i);
03171                     Terminate(-1);
03172                 }/*if(retvalue<=0)*/
03173                 fclose(fd);
03174             }/*for(i=0; i<PF.numtasks;i++)*/
03175             /*
03176              * Make the intermediate files the recovery files.
03177              */
03178             ANNOUNCE(rename intermediate file)
03179             for ( i = 0; i < PF.numtasks; i++ ) {
03180                 const char *src = PF_recoveryfile('m', i, 1);
03181                 const char *dst = PF_recoveryfile('m', i, 0);
03182                 if ( rename(src, dst) ) {
03183                     MesPrint("Error renaming recovery file %s -> %s", src, dst);
03184                 }
03185             }
03186             done_snapshot = 1;
03187             MesPrint("done."); fflush(0);
03188             }
03189 #endif
03190         }
03191         if ( AC.CheckpointRunAfter ) {
03192             size_t l, l2;
03193             char *str;
03194             l = strlen(AC.CheckpointRunAfter);
03195             NumToStr((UBYTE*)argbuf, AC.CModule);
03196             l2 = strlen(argbuf);
03197             str = (char*)Malloc1(l+l2+2, "callafter");
03198             strcpy(str, AC.CheckpointRunAfter);
03199             *(str+l) = ' ';
03200             strcpy(str+l+1, argbuf);
03201             retvalue = system(str);
03202             M_free(str, "callafter");
03203             if ( retvalue ) {
03204                 MesPrint("Error calling script after recovery.");
03205             }
03206         }
03207         AC.CheckpointStamp = TimeWallClock(1);
03208     }
03209 #ifdef WITHMPI
03210     else{/* timestamp - AC.CheckpointStamp < AC.CheckpointInterval*/
03211         /* The slaves don't need to make snapshots. */
03212         PF_BroadcastNumber(0);
03213     }
03214     }/*if(PF.me == MASTER)*/
03215     else{/*Slave*/
03216         int i;
03217         /* Check if the slave needs to make a snapshot. */
03218         if ( PF_BroadcastNumber(0) ) {
03219             error = DoSnapshot(moduletype);
03220             if(error == 0){
03221                 FILE *fd;
03222                 /*
03223                  * Send the recovery file to the master. Note that no renaming
03224                  * has been performed and what we have to send is actually sitting
03225                  * in the intermediate file.
03226                  */
03227                 fd = fopen(intermedfile, "r");
03228                 i=PF_SendFile(MASTER, fd);/*if fd==NULL, PF_SendFile seds to a slave the failure tag*/
03229                 if(fd == NULL)
03230                     Terminate(-1);
03231                 fclose(fd);
03232                 if(i<=0)
03233                     Terminate(-1);
03234                 /*Now the slave need not the recovery file so remove it:*/
03235                 remove(intermedfile);
03236             }
03237             else{
03238                 /*send the error tag to the master:*/
03239                 PF_SendFile(MASTER,NULL);/*if fd==NULL, PF_SendFile seds to a slave the failure tag*/
03240                 Terminate(-1);
03241             }
03242             done_snapshot = 1;
03243         }/*if(tag=PF_DATA_MSGTAG)*/
03244     }/*if(PF.me != MASTER)*/
03245 #endif
03246 }
03247 /* UNFINISHED_FEATURE_EXCL_STOP */
03248 /*
03249     #] DoCheckpoint :
03250 */
```

4.7 comexpr.c File Reference

`#include "form3.h"`
Include dependency graph for comexpr.c:



Data Structures

- struct **id_options**

Functions

- int CoLocal (UBYTE *inp)
- int CoGlobal (UBYTE *inp)
- int CoLocalFactorized (UBYTE *inp)
- int CoGlobalFactorized (UBYTE *inp)
- int DoExpr (UBYTE *inp, int type, int par)
- int CoIdOld (UBYTE *inp)
- int CoId (UBYTE *inp)
- int CoIdNew (UBYTE *inp)
- int CoDisorder (UBYTE *inp)
- int CoMany (UBYTE *inp)
- int CoMulti (UBYTE *inp)
- int CoIfMatch (UBYTE *inp)
- int CoIfNoMatch (UBYTE *inp)
- int CoOnce (UBYTE *inp)
- int CoOnly (UBYTE *inp)
- int CoSelect (UBYTE *inp)
- int CoIdExpression (UBYTE *inp, int type)
- int CoMultiply (UBYTE *inp)
- int CoFill (UBYTE *inp)
- int CoFillExpression (UBYTE *inp)
- int CoPrintTable (UBYTE *inp)
- int CoAssign (UBYTE *inp)
- int CoDeallocateTable (UBYTE *inp)

4.7.1 Detailed Description

Compiler routines for statements that involve algebraic expressions. These involve definitions, id-statements, the multiply statement and the fill statement.

Definition in file comexpr.c.

4.7.2 Function Documentation

4.7.2.1 CoLocal()

```
int CoLocal (
            UBYTE * inp )
```

Definition at line 66 of file comexpr.c.

4.7.2.2 CoGlobal()

```
int CoGlobal (
            UBYTE * inp )
```

Definition at line 73 of file comexpr.c.

4.7.2.3 CoLocalFactorized()

```
int CoLocalFactorized (
            UBYTE * inp )
```

Definition at line 80 of file comexpr.c.

4.7.2.4 CoGlobalFactorized()

```
int CoGlobalFactorized (
            UBYTE * inp )
```

Definition at line 87 of file comexpr.c.

4.7.2.5 DoExpr()

```
int DoExpr (
            UBYTE * inp,
            int type,
            int par )
```

Definition at line 96 of file comexpr.c.

4.7.2.6 CoIdOld()

```
int CoIdOld (
            UBYTE * inp )
```

Definition at line 387 of file comexpr.c.

4.7.2.7 CoId()

```
int CoId (
            UBYTE * inp )
```

Definition at line 398 of file comexpr.c.

4.7.2.8 CoIdNew()

```
int CoIdNew (
            UBYTE * inp )
```

Definition at line 409 of file comexpr.c.

4.7.2.9 CoDisorder()

```
int CoDisorder (
            UBYTE * inp )
```

Definition at line 420 of file comexpr.c.

4.7.2.10 CoMany()

```
int CoMany (
            UBYTE * inp )
```

Definition at line 431 of file comexpr.c.

4.7.2.11 CoMulti()

```
int CoMulti (
            UBYTE * inp )
```

Definition at line 442 of file comexpr.c.

4.7.2.12 CoIfMatch()

```
int CoIfMatch (
            UBYTE * inp )
```

Definition at line 453 of file comexpr.c.

4.7.2.13 CoIfNoMatch()

```
int CoIfNoMatch (
            UBYTE * inp )
```

Definition at line 464 of file comexpr.c.

4.7.2.14 CoOnce()

```
int CoOnce (
            UBYTE * inp )
```

Definition at line 475 of file comexpr.c.

4.7.2.15 CoOnly()

```
int CoOnly (
            UBYTE * inp )
```

Definition at line 486 of file comexpr.c.

4.7.2.16 CoSelect()

```
int CoSelect (
            UBYTE * inp )
```

Definition at line 497 of file comexpr.c.

4.7.2.17 CoIdExpression()

```
int CoIdExpression (
            UBYTE * inp,
            int type )
```

Definition at line 510 of file comexpr.c.

4.7.2.18 CoMultiply()

```
int CoMultiply (
            UBYTE * inp )
```

Definition at line 1034 of file comexpr.c.

4.7.2.19 CoFill()

```
int CoFill (
            UBYTE * inp )
```

Definition at line 1073 of file comexpr.c.

4.7.2.20 CoFillExpression()

```
int CoFillExpression (
            UBYTE * inp )
```

Definition at line 1359 of file comexpr.c.

4.7.2.21 CoPrintTable()

```
int CoPrintTable (
            UBYTE * inp )
```

Definition at line 1753 of file comexpr.c.

4.7.2.22 CoAssign()

```
int CoAssign (
            UBYTE * inp )
```

Definition at line 1929 of file comexpr.c.

4.7.2.23 CoDeallocateTable()

```
int CoDeallocateTable (
            UBYTE * inp )
```

Definition at line 1987 of file comexpr.c.

4.8 comexpr.c

Go to the documentation of this file.

```
00001
00008 /* #[ License : */
00009 /*
00010  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00011  *   When using this file you are requested to refer to the publication
00012  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013  *   This is considered a matter of courtesy as the development was paid
00014  *   for by FOM the Dutch physics granting agency and we would like to
00015  *   be able to track its scientific use to convince FOM of its value
00016  *   for the community.
00017  *
00018  *   This file is part of FORM.
00019  *
00020  *   FORM is free software: you can redistribute it and/or modify it under the
00021  *   terms of the GNU General Public License as published by the Free Software
00022  *   Foundation, either version 3 of the License, or (at your option) any later
00023  *   version.
00024  *
00025  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00027  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00028  *   details.
00029  *
00030  *   You should have received a copy of the GNU General Public License along
00031  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00032  */
00033 /* #] License : */
00034
00035 /*
00036     #[ Includes : compi2.c
00037
00038     File contains most of what has to do with compiling expressions.
00039     Main supporting file: token.c
00040 */
00041
00042 #include "form3.h"
00043
00044 static struct id_options {
00045     UBYTE *name;
00046     int code;
00047     int dummy;
00048 } IdOptions[] = {
```

```
00049       {(UBYTE *)"multi",       SUBMULTI     ,0}
00050      ,{(UBYTE *)"many",        SUBMANY      ,0}
00051      ,{(UBYTE *)"only",        SUBONLY      ,0}
00052      ,{(UBYTE *)"once",        SUBONCE      ,0}
00053      ,{(UBYTE *)"ifmatch",     SUBAFTER     ,0}
00054      ,{(UBYTE *)"ifnomatch",   SUBAFTERNOT  ,0}
00055      ,{(UBYTE *)"ifnotmatch", SUBAFTERNOT  ,0}
00056      ,{(UBYTE *)"disorder",    SUBDISORDER  ,0}
00057      ,{(UBYTE *)"select",      SUBSELECT    ,0}
00058      ,{(UBYTE *)"all",         SUBALL       ,0}
00059 };
00060
00061 /*
00062     #] Includes :
00063     #[ CoLocal :
00064 */
00065
00066 int CoLocal(UBYTE *inp) { return(DoExpr(inp,LOCALEXPRESSION,0)); }
00067
00068 /*
00069     #] CoLocal :
00070     #[ CoGlobal :
00071 */
00072
00073 int CoGlobal(UBYTE *inp) { return(DoExpr(inp,GLOBALEXPRESSION,0)); }
00074
00075 /*
00076     #] CoGlobal :
00077     #[ CoLocalFactorized :
00078 */
00079
00080 int CoLocalFactorized(UBYTE *inp) { return(DoExpr(inp,LOCALEXPRESSION,1)); }
00081
00082 /*
00083     #] CoLocalFactorized :
00084     #[ CoGlobalFactorized :
00085 */
00086
00087 int CoGlobalFactorized(UBYTE *inp) { return(DoExpr(inp,GLOBALEXPRESSION,1)); }
00088
00089 /*
00090     #] CoGlobalFactorized :
00091     #[ DoExpr:
00092
00093
00094 */
00095
00096 int DoExpr(UBYTE *inp, int type, int par)
00097 {
00098     GETIDENTITY
00099     int error = 0;
00100     UBYTE *p, *q, c;
00101     WORD *w, i, j = 0, c1, c2, *OldWork = AT.WorkPointer, osize;
00102     WORD jold = 0;
00103     POSITION pos;
00104     while ( *inp == ',' ) inp++;
00105     if ( par ) AC.ToBeInFactors = 1;
00106     else       AC.ToBeInFactors = 0;
00107     p = inp;
00108     while ( *p && *p != '=' ) {
00109         if ( *p == '(' ) SKIPBRA4(p)
00110         else if ( *p == '{' ) SKIPBRA5(p)
00111         else if ( *p == '[' ) SKIPBRA1(p)
00112         else p++;
00113     }
00114     if ( *p ) {     /* Variety with the = sign */
00115         q = SkipAName(inp);
00116         if ( *inp == '$' || q == 0 || q[-1] == '_' ) {
00117             MesPrint("&Illegal name for expression");
00118             error = 1;
00119             return(error);
00120         }
00121         else {
00122             c = *q; *q = 0;
00123             if ( GetVar(inp,&c1,&c2,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
00124                 if ( c1 == CEXPRESSION ) {
00125                     if ( Expressions[c2].status == STOREDEXPRESSION ) {
00126                         MesPrint("&Illegal attempt to overwrite a stored expression");
00127                         error = 1;
00128                     }
00129                     else {
00130                         HighWarning("Expression is replaced by new definition");
00131                         if ( AO.OptimizeResult.nameofexpr != NULL &&
00132                             StrCmp(inp,AO.OptimizeResult.nameofexpr) == 0 ) {
00133                             ClearOptimize();
00134                         }
00135                         if ( Expressions[c2].status != DROPPEDEXPRESSION ) {
```

```
00136                           w = &(Expressions[c2].status);
00137                           if ( *w == LOCALEXPRESSION || *w == SKIPLEXPRESSION )
00138                               *w = DROPLEXPRESSION;
00139                           else if ( *w == GLOBALEXPRESSION || *w == SKIPGEXPRESSION )
00140                               *w = DROPGEXPRESSION;
00141                           else if ( *w == HIDDENLEXPRESSION )
00142                               *w = DROPHLEXPRESSION;
00143                           else if ( *w == HIDDENGEXPRESSION )
00144                               *w = DROPHGEXPRESSION;
00145                       }
00146                       AC.TransEname = Expressions[c2].name;
00147                       j = EntVar(CEXPRESSION,0,type,0,0,0);
00148                       Expressions[j].node = Expressions[c2].node;
00149                       Expressions[c2].replace = j;
00150                   }
00151               }
00152               else {
00153                   MesPrint("&name of expression is also name of a variable");
00154                   error = 1;
00155                   j = EntVar(CEXPRESSION,inp,type,0,0,0);
00156               }
00157               jold = c2;
00158           }
00159           else {
00160 /*
00161               Here we have to worry about reuse of the expression in the
00162               same module. That will need AS.Oldvflags but that may not
00163               be defined or have the proper value.
00164 */
00165               j = EntVar(CEXPRESSION,inp,type,0,0,0);
00166               jold = j;
00167           }
00168           *q = c;
00169           OldWork = w = AT.WorkPointer;
00170           *w++ = TYPEEXPRESSION;
00171           *w++ = 3+SUBEXPSIZE;
00172           *w++ = j;
00173           AC.ProtoType = w;
00174           AR.CurExpr = j;             /* Block expression j */
00175           *w++ = SUBEXPRESSION;
00176           *w++ = SUBEXPSIZE;
00177           *w++ = j;
00178           *w++ = 1;
00179           *w++ = AC.cbufnum;
00180           FILLSUB(w)
00181
00182           if ( c == '(' ) {
00183               while ( *q == ',' || *q == '(' ) {
00184                   inp = q+1;
00185                   if ( ( q = SkipAName(inp) ) == 0 ) {
00186                       MesPrint("&Illegal name for expression argument");
00187                       error = 1;
00188                       q = p - 1;
00189                       break;
00190                   }
00191                   c = *q; *q = 0;
00192                   if ( GetVar(inp,&c1,&c2,ALLVARIABLES,WITHAUTO) < 0 ) c1 = -1;
00193                   switch ( c1 ) {
00194                       case CSYMBOL :
00195                           *w++ = SYMTOSYM; *w++ = 4; *w++ = c2; *w++ = 0;
00196                           break;
00197                       case CINDEX :
00198                           *w++ = INDTOIND; *w++ = 4;
00199                           *w++ = c2 + AM.OffsetIndex; *w++ = 0;
00200                           break;
00201                       case CVECTOR :
00202                           *w++ = VECTOVEC; *w++ = 4;
00203                           *w++ = c2 + AM.OffsetVector; *w++ = 0;
00204                           break;
00205                       case CFUNCTION :
00206                           *w++ = FUNTOFUN; *w++ = 4; *w++ = c2 + FUNCTION; *w++ = 0;
00207                           break;
00208                       default :
00209                           MesPrint("&Illegal expression parameter: %s",inp);
00210                           error = 1;
00211                           break;
00212                   }
00213                   *q = c;
00214               }
00215               if ( *q != ')' || q+1 != p ) {
00216                   MesPrint("&Illegal use of arguments for expression");
00217                   error = 1;
00218               }
00219               AC.ProtoType[1] = w - AC.ProtoType;
00220           }
00221           else if ( c != '=' ) {
00222 /*
```

```
00223                 The dummy accepted L F := RHS;
00224 */
00225                 MesPrint("&Illegal LHS for expression definition");
00226                 error = 1;
00227             }
00228             *w++ = 1;
00229             *w++ = 1;
00230             *w++ = 3;
00231             *w++ = 0;
00232             SeekScratch(AR.outfile,&pos);
00233             Expressions[j].counter = 1;
00234             Expressions[j].onfile = pos;
00235             Expressions[j].whichbuffer = 0;
00236 #ifdef PARALLELCODE
00237             Expressions[j].partodo = AC.inparallelflag;
00238 #endif
00239             OldWork[2] = w - OldWork - 3;
00240             AT.WorkPointer = w;
00241 /*
00242             Writing the expression prototype to disk and to the compiler
00243             buffer is done only after the RHS has been compiled because
00244             we don't know the number of the main level RHS yet.
00245 */
00246         }
00247         inp = p+1;
00248         ClearWildcardNames();
00249         osize = AC.ProtoType[1]; AC.ProtoType[1] = SUBEXPSIZE;
00250         PutInVflags(jold);
00251         if ( ( i = CompileAlgebra(inp,RHSIDE,AC.ProtoType) ) < 0 ) {
00252             AC.ProtoType[1] = osize;
00253             error = 1;
00254         }
00255         else if ( error == 0 ) {
00256             AC.ProtoType[1] = osize;
00257             AC.ProtoType[2] = i;
00258             if ( PutOut(BHEAD OldWork+2,&pos,AR.outfile,0) < 0 ) {
00259 /* INTERNAL_ERROR_EXCL_START */
00260                 MesPrint("!>Cannot create expression");
00261                 error = -1;
00262 /* INTERNAL_ERROR_EXCL_STOP */
00263             }
00264             else {
00265                 Expressions[j].sizeprototype = OldWork[2];
00266                 OldWork[2] = 4+SUBEXPSIZE;
00267                 OldWork[4] = SUBEXPSIZE;
00268                 OldWork[5] = i;
00269                 OldWork[SUBEXPSIZE+3] = 1;
00270                 OldWork[SUBEXPSIZE+4] = 1;
00271                 OldWork[SUBEXPSIZE+5] = 3;
00272                 OldWork[SUBEXPSIZE+6] = 0;
00273                 if ( PutOut(BHEAD OldWork+2,&pos,AR.outfile,0) < 0
00274                 || FlushOut(&pos,AR.outfile,0) ) {
00275 /* INTERNAL_ERROR_EXCL_START */
00276                     MesPrint("!>Cannot create expression");
00277                     error = -1;
00278 /* INTERNAL_ERROR_EXCL_STOP */
00279                 }
00280                 AR.outfile->POfull = AR.outfile->POfill;
00281             }
00282             OldWork[2] = j;
00283 /*
00284             Seems unnecessary (13-feb-2018)
00285
00286             AddNtoL(OldWork[1],OldWork);
00287 */
00288             AT.WorkPointer = OldWork;
00289             if ( AC.dumnumflag ) Add2Com(TYPEDETCURDUM)
00290         }
00291         AC.ToBeInFactors = 0;
00292     }
00293     else {  /* Variety in which expressions change property */
00294 /*
00295             This code got a major revision because it didn't
00296             take hidden expressions into account. (1-jun-2010 JV)
00297 */
00298         do {
00299             if ( ( q = SkipAName(inp) ) == 0 ) {
00300                 MesPrint("&Illegal name(s) for expression(s)");
00301                 return(1);
00302             }
00303             c = *q; *q = 0;
00304             if ( GetName(AC.exprnames,inp,&c2,NOAUTO) == NAMENOTFOUND ) {
00305                 MesPrint("&%s is not a valid expression",inp);
00306                 error = 1;
00307             }
00308             else {
00309                 w = &(Expressions[c2].status);
```

```
00310                 if ( type == LOCALEXPRESSION ) {
00311                     switch ( *w ) {
00312                         case GLOBALEXPRESSION:
00313                             *w = LOCALEXPRESSION;
00314                             break;
00315                         case SKIPGEXPRESSION:
00316                             *w = SKIPLEXPRESSION;
00317                             break;
00318                         case DROPGEXPRESSION:
00319                             *w = DROPLEXPRESSION;
00320                             break;
00321                         case HIDDENGEXPRESSION:
00322                             *w = HIDDENLEXPRESSION;
00323                             break;
00324                         case HIDEGEXPRESSION:
00325                             *w = HIDELEXPRESSION;
00326                             break;
00327                         case UNHIDEGEXPRESSION:
00328                             *w = UNHIDELEXPRESSION;
00329                             break;
00330                         case INTOHIDEGEXPRESSION:
00331                             *w = INTOHIDELEXPRESSION;
00332                             break;
00333                         case DROPHGEXPRESSION:
00334                             *w = DROPHLEXPRESSION;
00335                             break;
00336                     }
00337                 }
00338                 else if ( type == GLOBALEXPRESSION ) {
00339                     switch ( *w ) {
00340                         case LOCALEXPRESSION:
00341                             *w = GLOBALEXPRESSION;
00342                             break;
00343                         case SKIPLEXPRESSION:
00344                             *w = SKIPGEXPRESSION;
00345                             break;
00346                         case DROPLEXPRESSION:
00347                             *w = DROPGEXPRESSION;
00348                             break;
00349                         case HIDDENLEXPRESSION:
00350                             *w = HIDDENGEXPRESSION;
00351                             break;
00352                         case HIDELEXPRESSION:
00353                             *w = HIDEGEXPRESSION;
00354                             break;
00355                         case UNHIDELEXPRESSION:
00356                             *w = UNHIDEGEXPRESSION;
00357                             break;
00358                         case INTOHIDELEXPRESSION:
00359                             *w = INTOHIDEGEXPRESSION;
00360                             break;
00361                         case DROPHLEXPRESSION:
00362                             *w = DROPHGEXPRESSION;
00363                             break;
00364                     }
00365                 }
00366 /*
00367                 old code
00368                 if ( type != LOCALEXPRESSION || *w != STOREDEXPRESSION )
00369                         *w = type;
00370 */
00371             }
00372             *q = c; inp = q+1;
00373         } while ( c == ',' );
00374         if ( c ) {
00375             MesPrint("&Illegal object in local or global redefinition");
00376             error = 1;
00377         }
00378     }
00379     return(error);
00380 }
00381
00382 /*
00383     #] DoExpr:
00384     #[ CoIdOld :
00385 */
00386
00387 int CoIdOld(UBYTE *inp)
00388 {
00389     AC.idoption = 0;
00390     return(CoIdExpression(inp,TYPEIDOLD));
00391 }
00392
00393 /*
00394     #] CoIdOld :
00395     #[ CoId :
00396 */
```

```
00397
00398 int CoId(UBYTE *inp)
00399 {
00400     AC.idoption = 0;
00401     return(CoIdExpression(inp,TYPEIDNEW));
00402 }
00403
00404 /*
00405     #] CoId :
00406     #[ CoIdNew :
00407 */
00408
00409 int CoIdNew(UBYTE *inp)
00410 {
00411     AC.idoption = 0;
00412     return(CoIdExpression(inp,TYPEIDNEW));
00413 }
00414
00415 /*
00416     #] CoIdNew :
00417     #[ CoDisorder :
00418 */
00419
00420 int CoDisorder(UBYTE *inp)
00421 {
00422     AC.idoption = SUBDISORDER;
00423     return(CoIdExpression(inp,TYPEIDNEW));
00424 }
00425
00426 /*
00427     #] CoDisorder :
00428     #[ CoMany :
00429 */
00430
00431 int CoMany(UBYTE *inp)
00432 {
00433     AC.idoption = SUBMANY;
00434     return(CoIdExpression(inp,TYPEIDNEW));
00435 }
00436
00437 /*
00438     #] CoMany :
00439     #[ CoMulti :
00440 */
00441
00442 int CoMulti(UBYTE *inp)
00443 {
00444     AC.idoption = SUBMULTI;
00445     return(CoIdExpression(inp,TYPEIDNEW));
00446 }
00447
00448 /*
00449     #] CoMulti :
00450     #[ CoIfMatch :
00451 */
00452
00453 int CoIfMatch(UBYTE *inp)
00454 {
00455     AC.idoption = SUBAFTER;
00456     return(CoIdExpression(inp,TYPEIDNEW));
00457 }
00458
00459 /*
00460     #] CoIfMatch :
00461     #[ CoIfNoMatch :
00462 */
00463
00464 int CoIfNoMatch(UBYTE *inp)
00465 {
00466     AC.idoption = SUBAFTERNOT;
00467     return(CoIdExpression(inp,TYPEIDNEW));
00468 }
00469
00470 /*
00471     #] CoIfNoMatch :
00472     #[ CoOnce :
00473 */
00474
00475 int CoOnce(UBYTE *inp)
00476 {
00477     AC.idoption = SUBONCE;
00478     return(CoIdExpression(inp,TYPEIDNEW));
00479 }
00480
00481 /*
00482     #] CoOnce :
00483     #[ CoOnly :
```

```
00484 */
00485
00486 int CoOnly(UBYTE *inp)
00487 {
00488     AC.idoption = SUBONLY;
00489     return(CoIdExpression(inp,TYPEIDNEW));
00490 }
00491
00492 /*
00493     #] CoOnly :
00494     #[ CoSelect :
00495 */
00496
00497 int CoSelect(UBYTE *inp)
00498 {
00499     AC.idoption = SUBSELECT;
00500     return(CoIdExpression(inp,TYPEIDNEW));
00501 }
00502
00503 /*
00504     #] CoSelect :
00505     #[ CoIdExpression :
00506
00507     First finish dealing with secondary keywords
00508 */
00509
00510 int CoIdExpression(UBYTE *inp, int type)
00511 {
00512     GETIDENTITY
00513     int i, j, idhead, error = 0, MinusSign = 0, opt, retcode;
00514     WORD *w, *s, *m, *mm, *ww, *FirstWork, *OldWork, c1, numsets = 0,
00515          oldnumrhs, *ow, oldEside;
00516     UBYTE *p, *pp, c;
00517     CBUF *C = cbuf+AC.cbufnum;
00518     LONG oldcpointer, x;
00519     FirstWork = OldWork = AT.WorkPointer;
00520 /*
00521     Don't forget to change in StudyPattern if we change/add_to the
00522     following setup.
00523     if ( type == TYPEIF ) idhead = IDHEAD-1;
00524     else
00525 */
00526     idhead = IDHEAD;
00527     AR.CurExpr = -1;
00528     w = AT.WorkPointer;
00529     *w++ = type;
00530     *w++ = idhead + SUBEXPSIZE;
00531     w++;
00532     if ( idhead >= IDHEAD ) *w++ = -1;
00533 #if IDHEAD > 4
00534     for ( i = 4; i < idhead; i++ ) *w++ = 0;
00535 #endif
00536     while ( *inp == ',' ) inp++;
00537     p = inp;
00538     if ( AC.idoption == SUBSELECT ) {
00539         p--;
00540         goto findsets;
00541     }
00542     else if ( ( AC.idoption == SUBAFTER ) || ( AC.idoption == SUBAFTERNOT ) ) {
00543         while ( *p && *p != '=' && *p != ',' ) {
00544             if ( *p == '(' ) SKIPBRA4(p)
00545             else if ( *p == '{' ) SKIPBRA5(p)
00546             else if ( *p == '[' ) SKIPBRA1(p)
00547             else p++;
00548         }
00549         if ( *p == '=' || *inp != '-' || inp[1] != '>' ) {
00550             MesPrint("&Illegal use if if[no]match in id statement");
00551             error = 1; goto AllDone;
00552         }
00553         if ( *p == 0 ) {
00554             MesPrint("&id-statement without = sign");
00555             error = 1; goto AllDone;
00556         }
00557         inp += 2; pp = inp;
00558         goto readlabel;
00559     }
00560     for(;;) {
00561         while ( *p && *p != '=' && *p != ',' ) {
00562             if ( *p == '(' ) SKIPBRA4(p)
00563             else if ( *p == '{' ) SKIPBRA5(p)
00564             else if ( *p == '[' ) SKIPBRA1(p)
00565             else p++;
00566         }
00567         if ( *p == '=' ) break;
00568         if ( *p == 0 ) {
00569             MesPrint("&id-statement without = sign");
00570             error = 1; goto AllDone;
```

```
00571         }
00572 /*
00573         We have either a secondary option or a syntax error
00574 */
00575         pp = inp;
00576         while ( FG.cTable[*pp] == 0 ) pp++;
00577         c = *pp; *pp = 0;
00578         i = sizeof(IdOptions)/sizeof(struct id_options);
00579         while ( --i >= 0 ) {
00580             if ( StrICmp(inp,IdOptions[i].name) == 0 ) break;
00581         }
00582         if ( i < 0 ) {
00583             MesPrint("&Illegal option %s in id-statement",inp);
00584             *pp = c; error = 1; p++; inp = p; continue;
00585         }
00586         opt = IdOptions[i].code;
00587         *pp = c;
00588         inp = pp+1;
00589         switch ( opt ) {
00590             case SUBDISORDER:
00591                 if ( pp != p ) goto IllField;
00592                 AC.idoption |= SUBDISORDER;
00593                 p++; inp = p;
00594                 break;
00595             case SUBSELECT:
00596                 if ( p != pp ) goto IllField;
00597                 if ( ( AC.idoption & SUBMASK ) != 0 ) {
00598                     if ( AC.idoption == SUBMULTI && type == TYPEIF ) {}
00599                     else {
00600                         MesPrint("&Conflicting options in id-statement");
00601                         error = 1;
00602                     }
00603                 }
00604 findsets:;
00605 /*
00606                 Now we read the sets
00607 */
00608                 numsets = 0;
00609                 for(;;) {
00610                     inp = ++p;
00611                     while ( *p && *p != '=' && *p != ',' ) {
00612                         if ( *p == '(' ) SKIPBRA4(p)
00613                         else if ( *p == '{' ) SKIPBRA5(p)
00614                         else if ( *p == '[' ) SKIPBRA1(p)
00615                         else p++;
00616                     }
00617                     if ( *p == '=' ) break;
00618                     if ( *p == 0 ) {
00619                         MesPrint("&id-statement without = sign");
00620                         error = 1; goto AllDone;
00621                     }
00622 /*
00623                     We have a set at inp.
00624 */
00625                     if ( *inp == '{' ) {
00626                         if ( p[-1] != '}' ) {
00627                             c = *p; *p = 0;
00628                             MesPrint("&Illegal temporary set: %s",inp);
00629                             error = 1; *p = c;
00630                         }
00631                         else {
00632                             inp++;
00633                             c = p[-1]; p[-1] = 0;
00634                             c1 = DoTempSet(inp,p-1);
00635                             *w++ = c1;
00636                             p[-1] = c;
00637                             numsets++;
00638                             if ( w[-1] < 0 ) error = 1;
00639                         }
00640                     }
00641                     else {
00642                         c = *p; *p = 0;
00643                         if ( GetName(AC.varnames,inp,&c1,NOAUTO) != CSET ) {
00644                             MesPrint("&%s is not a set",inp);
00645                             error = 1;
00646                         }
00647                         else {
00648                             if ( c1 < AM.NumFixedSets ) {
00649                                 MesPrint("&Built in sets are not allowed in the select option");
00650                                 error = 1;
00651                             }
00652                             else if ( Sets[c1].type == CRANGE ) {
00653                                 MesPrint("&Ranged sets are not allowed in the select option");
00654                                 error = 1;
00655                             }
00656                             numsets++;
00657                             *w++ = c1;
```

```
00658                       }
00659                       *p = c;
00660                   }
00661               }
00662 /*
00663               Now exchange the positions a bit.
00664               Regular stuff at OldWork, numsets sets at FirstWork[idhead]
00665 */
00666               OldWork = w;
00667               for ( i = 0; i < idhead; i++ ) *w++ = FirstWork[i];
00668               AC.idoption = SUBSELECT;
00669               break;
00670           case SUBAFTER:
00671           case SUBAFTERNOT:
00672               if ( type == TYPEIF ) {
00673                   MesPrint("&The if[no]match->label option is not allowed in an if statement");
00674                   error = 1; goto AllDone;
00675               }
00676               if ( pp[0] != '-' || pp[1] != '>' ) goto IllField;
00677               pp += 2;    /* points now at the label */
00678               inp = pp;
00679               AC.idoption |= opt;
00680 readlabel:
00681               while ( FG.cTable[*pp] <= 1 ) pp++;
00682               if ( pp != p ) {
00683                   c = *p; *p = 0;
00684                   MesPrint("&Illegal label %s in if[no]match option of id-statement",inp);
00685                   *p = c; error = 1; inp = p+1; continue;
00686               }
00687               c = *p; *p = 0;
00688               OldWork[3] = GetLabel(inp);
00689               *p++ = c; inp = p;
00690               break;
00691           case SUBALL:
00692               x = 0;
00693               if ( *pp == '(' ) {
00694                   if ( FG.cTable[*inp] == 1 ) {
00695                       while ( *inp >= '0' && *inp <= '9' ) x = 10*x+*inp++ - '0';
00696                   }
00697                   else {
00698                       pp++;
00699                       while ( FG.cTable[*inp] == 0 ) inp++;
00700                       c = *inp; *inp = 0;
00701                       if ( StrICont(pp,(UBYTE *)"normalize") != 0 ) goto IllOpt;
00702                       *inp = c;
00703                       OldWork[4] |= NORMALIZEFLAG;
00704                   }
00705                   if ( *inp != ')' || inp+1 != p ) {
00706                       c = *inp; *inp = 0;
00707 IllOpt:
00708                       MesPrint("&Illegal ALL option in id-statement: ",pp);
00709                       *inp++ = c;
00710                       error = 1;
00711                       continue;
00712                   }
00713                   pp = inp;
00714                   inp = pp+1;
00715               }
00716 /*
00717               Note that the following statement limits x to
00718 */
00719               if ( x > MAXPOSITIVE ) {
00720                   MesPrint("&Requested maximum number of matches %l in ALL option in id-statement is
      greater than %l ",x,MAXPOSITIVE);
00721                   error = 1;
00722               }
00723               OldWork[5] = x;
00724               if ( type != TYPEIDNEW ) {
00725                 if ( type == TYPEIDOLD ) {
00726                   MesPrint("&Requested ALL option not allowed in idold/also statement.");
00727                   error = 1;
00728                 }
00729                 else if ( type == TYPEIF ) {
00730                   MesPrint("&Requested ALL option not allowed in if(match())");
00731                   error = 1;
00732                 }
00733                 else {
00734                   MesPrint("&ALL option only allowed in regular id-statement.");
00735                   error = 1;
00736                 }
00737               }
00738               p++; inp = p;
00739               AC.idoption = opt;
00740               break;
00741           default:
00742               if ( pp != p ) {
00743 IllField:         c = *p; *p = 0;
```

```
00744                     MesPrint("&Illegal optionfield %s in id-statement",inp);
00745                     *p = c; error = 1; inp = p+1; continue;
00746                 }
00747                 i = AC.idoption & SUBMASK;
00748                 if ( i && i != opt ) {
00749                     MesPrint("&Conflicting options in id-statement");
00750                     error = 1;  continue;
00751                 }
00752                 else AC.idoption |= opt;
00753                 while ( *p == ',' ) p++;
00754                 inp = p;
00755                 break;
00756         }
00757     }
00758     if ( ( AC.idoption & SUBMASK ) == 0 ) AC.idoption |= SUBMULTI;
00759     OldWork[2] = AC.idoption;
00760 /*
00761     Now we have a field till the = sign
00762     Now the subexpression prototype
00763 */
00764     AC.ProtoType = w;
00765     *w++ = SUBEXPRESSION;
00766     *w++ = SUBEXPSIZE;
00767     *w++ = C->numrhs+1;
00768     *w++ = 1;
00769     *w++ = AC.cbufnum;
00770     FILLSUB(w)
00771     AC.WildC = w;
00772     AC.NwildC = 0;
00773     AT.WorkPointer = s = w + 4*AM.MaxWildcards + 8;
00774 /*
00775     Now read the LHS
00776 */
00777     ClearWildcardNames();
00778     oldcpointer = AddLHS(AC.cbufnum) - C->Buffer;
00779
00780     *p = 0;
00781     oldnumrhs = C->numrhs;
00782     if ( ( retcode = CompileAlgebra(inp,LHSIDE,AC.ProtoType) ) < 0 ) { error = 1; }
00783     else AC.ProtoType[2] = retcode;
00784     *p = '='; inp = p+1;
00785     AT.WorkPointer = s;
00786     if ( AC.NwildC && SortWild(w,AC.NwildC) ) error = 1;
00787
00788         /* Make the LHS pointers ready */
00789
00790     OldWork[1] = AC.WildC-OldWork;
00791     OldWork[idhead+1] = OldWork[1] - idhead;
00792     w = AC.WildC;
00793     AT.WorkPointer = w;
00794     s = C->rhs[C->numrhs];
00795 /*
00796     Now check whether wildcards get converted to dollars (for PARALLEL)
00797 */
00798     {
00799         WORD *tw, *twstop;
00800         tw = AC.ProtoType; twstop = tw + tw[1]; tw += SUBEXPSIZE;
00801         while ( tw < twstop ) {
00802             if ( *tw == LOADDOLLAR ) {
00803                 AddPotModdollar(tw[2]);
00804             }
00805             tw += tw[1];
00806         }
00807     }
00808 /*
00809     We have the expression in the compiler buffers.
00810     The main level is at lhs[numlhs]
00811     The partial lhs (including ProtoType) is in OldWork (in WorkSpace)
00812     We need to load the result at w after the prototype
00813     Because these sort routines don't use the WorkSpace
00814     there should not be a conflict
00815 */
00816     if ( !error && *s == 0 ) {
00817 IllLeft:MesPrint("&Illegal LHS");
00818         AC.lhdollarflag = 0;
00819         return(1);
00820     }
00821     if ( !error && *(s+*s) != 0 ) {
00822         MesPrint("&LHS should be one term only");
00823         return(1);
00824     }
00825     if ( error == 0 ) {
00826         WORD oldpolyfun = AR.PolyFun;
00827         if ( NewSort(BHEAD0) || NewSort(BHEAD0) ) {
00828             if ( !error ) error = 1;
00829             return(error);
00830         }
```

```
00831         AN.RepPoint = AT.RepCount + 1;
00832         ow = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
00833         mm = s; ww = ow; i = *mm;
00834         while ( --i >= 0 ) {*ww++ = *mm++;} AT.WorkPointer = ww;
00835         AC.lhdollarflag = 0; oldEside = AR.Eside; AR.Eside = LHSIDE;
00836         AR.Cnumlhs = C->numlhs;
00837         AR.PolyFun = 0;
00838         if ( Generator(BHEAD ow,C->numlhs) ) {
00839             AR.Eside = oldEside;
00840             LowerSortLevel(); LowerSortLevel(); AR.PolyFun = oldpolyfun; goto IllLeft;
00841         }
00842         AR.Eside = oldEside;
00843         AT.WorkPointer = w;
00844         if ( EndSort(BHEAD w,0) < 0 ) { LowerSortLevel(); AR.PolyFun = oldpolyfun; goto IllLeft; }
00845         AR.PolyFun = oldpolyfun;
00846         if ( *w == 0 || *(w+*w) != 0 ) {
00847             MesPrint("&LHS must be one term");
00848             AC.lhdollarflag = 0;
00849             return(1);
00850         }
00851         LowerSortLevel();
00852         if ( AC.lhdollarflag ) MarkDirty(w,DIRTYFLAG);
00853     }
00854     AT.WorkPointer = w + *w;
00855     AC.DumNum = 0;
00856 /*
00857     Everything is now after OldWork. We can pop the compilerbuffer.
00858     Next test for illegal things like a coefficient
00859     At this point we have:
00860     w = the term of the LHS
00861 */
00862     C->Pointer = C->Buffer + oldcpointer;
00863     C->numrhs = oldnumrhs;
00864     C->numlhs--;
00865
00866     m = w + *w - 3;
00867     AC.vectorlikeLHS = 0;
00868     if ( !error ) {
00869       if ( m[2] != 3 || m[1] != 1 || *m != 1 ) {
00870         if ( *m == 1 && m[1] == 1 && m[2] == -3 ) {
00871             MinusSign = 1;
00872         }
00873         else {
00874             MesPrint("&Coefficient in LHS");
00875             error = 1;
00876             AC.DumNum = 0;
00877             *w -= ABS(m[2])-3;
00878         }
00879       }
00880       if ( *w == 7 && w[1] == INDEX && w[3] < 0 ) {
00881         if ( ( AC.idoption & SUBMASK ) != 0 &&  ( AC.idoption & SUBMASK ) !=
00882         SUBMULTI ) {
00883             MesPrint("&Illegal option for substitution of a vector");
00884             error = 1;
00885         }
00886         AC.DumNum = AM.IndDum;
00887         OldWork[2] = ( OldWork[2] - ( OldWork[2] & SUBMASK ) ) | SUBVECTOR;
00888         c1 = w[3];
00889             /* We overwrite the LHS */
00890         *w++ = INDTOIND;
00891         *w++ = 4;
00892         *w++ = AC.DumNum + WILDOFFSET;
00893         *w++ = 0;
00894         w[0] = 5;
00895         w[1] = VECTOR;
00896         w[2] = 4;
00897         w[3] = c1;
00898         w[4] = AC.DumNum + WILDOFFSET;
00899         OldWork[idhead+1] = w - OldWork - idhead;
00900         AC.vectorlikeLHS = 1;
00901       }
00902       else {
00903         AC.DumNum = 0;
00904         *w -= 3;
00905         i = OldWork[2] & SUBMASK;
00906         m = w + *w;
00907         if ( i == 0 || i == SUBMULTI ) {
00908             s = w+1;
00909             while ( s < m ) {
00910                 if ( *s == SYMBOL ) {
00911                     j = s[1]/2; s += 2;
00912                     while ( --j >= 0 ) {
00913                         if ( ABS(s[1]) > 2*MAXPOWER ) {
00914                             OldWork[2] = ( OldWork[2] - i ) | SUBONCE;
00915                             break;
00916                         }
00917                         s += 2;
```

```
00918                   }
00919                   if ( j >= 0 ) break;
00920               }
00921               else if ( *s == DOTPRODUCT ) {
00922                   j = s[1]/3; s += 2;
00923                   while ( --j >= 0 ) {
00924                       if ( ABS(s[2]) > 2*MAXPOWER ) {
00925                           OldWork[2] = ( OldWork[2] - i ) | SUBONCE;
00926                           break;
00927                       }
00928                       else if ( s[1] >= -(2*WILDOFFSET) || s[0] >= -(2*WILDOFFSET) ) {
00929                           OldWork[2] = ( OldWork[2] - i ) | SUBMANY;
00930                           i = SUBMANY;
00931                       }
00932                       s += 3;
00933                   }
00934                   if ( j >= 0 ) break;
00935               }
00936               else {
00937                   OldWork[2] = ( OldWork[2] - i ) | SUBMANY;
00938                   break;
00939               }
00940           }
00941         }
00942         if ( ( OldWork[2] & SUBMASK ) == 0 ) OldWork[2] |= SUBMULTI;
00943       }
00944       if ( ( OldWork[2] & SUBMASK ) == SUBSELECT ) {
00945 /*
00946         Paste the SETSET information after the pattern.
00947         Important note: We will still get function information for the
00948         smart patternmatching after it. To distinguish them we need to have
00949         that SETSET != m*n+1 in which m is the number of words per function
00950         and n the number of functions. Currently (29-may-1997) m = 4.
00951 */
00952         *m++ = SETSET;
00953         *m++ = numsets+2;
00954         s = FirstWork + idhead;
00955         while ( --numsets >= 0 ) *m++ = *s++;
00956       }
00957       else {
00958         m = w + *w;
00959       }
00960     }
00961 /*
00962     We keep the whole thing in OldWork for the moment.
00963     We still have to add the number of the RHS expression.
00964     There is also some opportunity now to be smart about the pattern.
00965     This is needed for complicated wildcarding with symmetric functions.
00966     We do this in a special routine during compile time to make sure
00967     that we loose as little time as possible (during running) if there
00968     is no need to be smart.
00969 */
00970     *m++ = 0;
00971     OldWork[1] = m - OldWork;
00972     AC.ProtoType = OldWork+idhead;
00973     if ( !error ) {
00974         if ( StudyPattern(OldWork) ) error = 1;
00975     }
00976     AT.WorkPointer = OldWork + OldWork[1];
00977     if ( AC.lhdollarflag ) OldWork[4] |= DOLLARFLAG;
00978     AC.lhdollarflag = 0;
00979 /*
00980     Test whether the id/idold configuration is fine.
00981 */
00982     if ( type == TYPEIDOLD ) {
00983         WORD ci = C->numlhs;
00984         while ( ci >= 1 ) {
00985             if ( C->lhs[ci][0] == TYPEIDNEW ) {
00986                 if ( (C->lhs[ci][2] & SUBMASK) == SUBALL ) {
00987                     MesPrint("&Idold/also cannot follow an id,all statement.");
00988                     error = 1;
00989                 }
00990                 break;
00991             }
00992             else if ( C->lhs[ci][0] == TYPEDETCURDUM ) { ci--; continue; }
00993             else if ( C->lhs[ci][0] == TYPEIDOLD ) { ci--; continue; }
00994             else ci = 0;
00995         }
00996         if ( ci < 1 ) {
00997             MesPrint("&Idold/also should follow an id/idnew statement.");
00998             error = 1;
00999         }
01000     }
01001 /*
01002     Now the right hand side.
01003 */
01004     if ( type != TYPEIF ) {
```

```
01005         if ( ( retcode = CompileAlgebra(inp,RHSIDE,AC.ProtoType) ) < 0 ) error = 1;
01006         else {
01007             AC.ProtoType[2] = retcode;
01008             AC.DumNum = 0;
01009             if ( MinusSign ) {  /* Flip the sign of the RHS */
01010                 w = C->rhs[retcode];
01011                 while ( *w ) { w += *w; w[-1] = -w[-1]; }
01012             }
01013             if ( AC.dumnumflag ) Add2Com(TYPEDETCURDUM)
01014         }
01015     }
01016 /*
01017     Actual adding happens only now after numrhs insertion
01018 */
01019     if ( !error ) { AddNtoL(OldWork[1],OldWork); }
01020 AllDone:
01021     AC.lhdollarflag = 0;
01022     AT.WorkPointer = FirstWork;
01023     return(error);
01024 }
01025
01026 /*
01027     #] CoIdExpression :
01028     #[ CoMultiply :
01029 */
01030
01031 static WORD mularray[13] = { TYPEMULT, SUBEXPSIZE+3, 0, SUBEXPRESSION,
01032         SUBEXPSIZE, 0, 1, 0, 0, 0, 0, 0, 0 };
01033
01034 int CoMultiply(UBYTE *inp)
01035 {
01036     UBYTE *p;
01037     int error = 0, RetCode;
01038     mularray[2] = 0;        /* right multiply is default */
01039     while ( *inp == ',' ) inp++;
01040 /*  if ( inp[-1] == '-' || inp[-1] == '+' ) inp--; */
01041     p = SkipField(inp,0);
01042     if ( *p ) {
01043         *p = 0;
01044         if ( StrICont(inp,(UBYTE *)"left") == 0 )       mularray[2] = 1;
01045         else if ( StrICont(inp,(UBYTE *)"right") == 0 ) mularray[2] = 0;
01046         else {
01047             MesPrint("&Illegal option in multiply statement or ; forgotten.");
01048             return(1);
01049         }
01050         *p = ',';
01051         inp = p + 1;
01052     }
01053     ClearWildcardNames();
01054     while ( *inp == ',' ) inp++;
01055     AC.ProtoType = mularray+3;
01056     mularray[7] = AC.cbufnum;
01057     if ( ( RetCode = CompileAlgebra(inp,RHSIDE,AC.ProtoType) ) < 0 ) error = 1;
01058     else {
01059         mularray[5] = RetCode;
01060         AddNtoL(SUBEXPSIZE+3,mularray);
01061         if ( AC.dumnumflag ) Add2Com(TYPEDETCURDUM)
01062     }
01063     return(error);
01064 }
01065
01066 /*
01067     #] CoMultiply :
01068     #[ CoFill :
01069
01070     Special additions for tablebase-like tables added 12-aug-2002
01071 */
01072
01073 int CoFill(UBYTE *inp)
01074 {
01075     GETIDENTITY
01076     WORD error = 0, x, xx, funnum, type, *oldwp = AT.WorkPointer;
01077     int i, oldcbufnum = AC.cbufnum, nofill = 0, numover, redef = 0;
01078     WORD *w, *wold, *Tprototype;
01079     UBYTE *p = inp, c, *inp1;
01080     TABLES T = 0, oldT;
01081     LONG newreservation, sum = 0;
01082     UBYTE *p1, *p2, *p3, *p4, *fake = 0;
01083     int tablestub = 0;
01084     if ( AC.exprfillwarning == 1 ) AC.exprfillwarning = 0;
01085 /*
01086     Read the name of the function and test that it is in the table.
01087 */
01088     p1 = inp;
01089     if ( ( p = SkipAName(inp) ) == 0 ) return(1);
01090     p2 = p;
01091     c = *p; *p = 0;
```

```
01092     if ( ( GetVar(inp,&type,&funnum,CFUNCTION,WITHAUTO) == NAMENOTFOUND )
01093     || ( T = functions[funnum].tabl ) == 0 || ( T->numind > 0 && c != '(' ) ) {
01094         MesPrint("&%s should be a table with argument(s)",inp);
01095         *p = c; return(1);
01096     }
01097     oldT = T;
01098     *p++ = c;
01099     if ( T->numind == 0 ) {
01100         if ( c == '(' ) {
01101             if ( *p != ')' ) {
01102                 c = *p; *p = 0;
01103                 MesPrint("&%s should be a table without arguments",inp);
01104                 *p = c; return(1);
01105             }
01106             else { p++; }
01107         }
01108         else { p--; }
01109         sum = 0;
01110         p3 = p;
01111         goto andagain;
01112     }
01113     w = oldwp;
01114     if ( T->numind < 0 ) { /* Pick up the first index */
01115         ParseSignedNumber(xx,p);
01116         if ( FG.cTable[p[-1]] != 1 || *p != ',' || xx < 1 || ( xx > ( -T->numind - 1 ) ) ) {
01117             MesPrint("&No valid number of table indices in *-table fill statement.");
01118             return(1);
01119         }
01120         *w++ = xx;
01121         p++;
01122     }
01123     else { xx = T->numind; }
01124     for ( sum = 0, i = 0; i < xx; i++ ) {
01125         ParseSignedNumber(x,p);
01126         if ( FG.cTable[p[-1]] != 1 || ( *p != ',' && *p != ')' ) ) {
01127             MesPrint("&Table arguments in fill statement should be numbers");
01128             return(1);
01129         }
01130         if ( T->sparse ) *w++ = x;
01131         else if ( x < T->mm[i].mini || x > T->mm[i].maxi ) {
01132             MesPrint("&Value %d for argument %d of table out of bounds",x,i+1);
01133             error = 1; nofill = 1;
01134         }
01135         else sum += ( x - T->mm[i].mini ) * T->mm[i].size;
01136         if ( *p == ')' ) break;
01137         p++;
01138     }
01139     p3 = p;
01140     if ( T->numind < 0 ) {
01141         for ( ; i < ABS(T->numind)-1; i++ ) *w++ = 0;
01142         xx = -T->numind;
01143     }
01144     if ( *p != ')' || i < ( xx - 1 ) ) {
01145         MesPrint("&Incorrect number of table arguments in fill statement. Should be %d"
01146         ,T->numind);
01147         error = 1; nofill = 1;
01148     }
01149     AT.WorkPointer = w;
01150     if ( T->sparse == 0 ) sum *= TABLEEXTENSION;
01151 andagain:;
01152     AC.cbufnum = T->bufnum;
01153     if ( T->sparse ) {
01154         i = FindTableTree(T,oldwp,1);
01155         if ( i >= 0 ) {
01156             sum = i + ABS(T->numind);
01157             if ( tablestub == 0 && ( ( T->sparse & 2 ) == 2 ) && ( T->mode != 0 )
01158             && ( AC.vetotablebasefill == 0 ) ) {
01159 /*
01160                 This redefinition does not need a new stub
01161 */
01162                 functions[funnum].tabl = T = T->spare;
01163                 tablestub = 1;
01164                 goto andagain;
01165             }
01166             redef = 1;
01167             goto redef;
01168         }
01169         if ( T->totind >= T->reserved ) {
01170             if ( T->reserved == 0 ) newreservation = 20;
01171             else newreservation = T->reserved;
01172             while ( T->totind >= newreservation && newreservation < MAXTABLECOMBUF )
01173                 newreservation = 2*newreservation;
01174             if ( newreservation > MAXTABLECOMBUF ) newreservation = MAXTABLECOMBUF;
01175             if ( T->totind >= newreservation ) {
01176                 MesPrint("@More than %ld elements in sparse table",MAXTABLECOMBUF);
01177                 AC.cbufnum = oldcbufnum;
01178                 Terminate(-1);
```

```
01179             }
01180             wold = (WORD *)Malloc1(newreservation*sizeof(WORD)*
01181                                    (ABS(T->numind)+TABLEEXTENSION),"tablepointers");
01182             for ( i = T->reserved*(ABS(T->numind)+TABLEEXTENSION)-1; i >= 0; i-- )
01183                 wold[i] = T->tablepointers[i];
01184             if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
01185             T->tablepointers = wold;
01186             T->reserved = newreservation;
01187         }
01188         w = oldwp;
01189         for ( sum = T->totind*(ABS(T->numind)+TABLEEXTENSION), i = 0; i < ABS(T->numind); i++ ) {
01190             T->tablepointers[sum++] = *w++;
01191         }
01192         InsTableTree(T,T->tablepointers+sum-ABS(T->numind));
01193 #if TABLEEXTENSION == 2
01194         T->tablepointers[sum+TABLEEXTENSION-1] = -1;  /* New element! */
01195 #else
01196         T->tablepointers[sum+1] = T->bufnum;
01197         T->tablepointers[sum+2] = -1;
01198         T->tablepointers[sum+3] = -1;
01199         T->tablepointers[sum+4] = 0;
01200         T->tablepointers[sum+5] = 0;
01201 #endif
01202     }
01203     else {
01204         if ( !nofill && T->tablepointers[sum] >= 0 ) {
01205 redef:;
01206             if ( AC.vetofilling ) nofill = 1;
01207             else {
01208                 Warning("Table element was already defined. New definition will be used");
01209             }
01210         }
01211 #if TABLEEXTENSION == 2
01212         T->tablepointers[sum+TABLEEXTENSION-1] = -1;  /* New element! */
01213 #else
01214         T->tablepointers[sum+1] = T->bufnum;
01215         T->tablepointers[sum+2] = -1;
01216         T->tablepointers[sum+3] = -1;
01217         T->tablepointers[sum+4] = 0;
01218         T->tablepointers[sum+5] = 0;
01219 #endif
01220     }
01221     if ( T->numind ) { p++; }
01222     if ( *p != '=' ) {
01223         MesPrint("&Fill statement misses = sign after the table element");
01224         AC.cbufnum = oldcbufnum;
01225         AT.WorkPointer = oldwp;
01226         functions[funnum].tabl = oldT;
01227         return(1);
01228     }
01229     if ( tablestub == 0 && T->mode == 1 && AC.vetotablebasefill == 0 ) {
01230 /*
01231         Here we construct a righthandside from the indices and the wildcards
01232 */
01233         int numfake;
01234         tablestub = 1;
01235         p4 = T->argtail;
01236         while ( *p4 ) p4++;
01237         numfake = (p4-T->argtail)+(p3-p1)+10;
01238
01239         fake = (UBYTE *)Malloc1(numfake*sizeof(UBYTE),"Fill fake rhs");
01240         p = fake;
01241         *p++ = 't'; *p++ = 'b'; *p++ = 'l'; *p++ = '_'; *p++ = '(';
01242         p4 = p1; while ( p4 < p2 ) *p++ = *p4++; *p++ = ',';
01243         p4 = p2+1; while ( p4 < p3 ) *p++ = *p4++;
01244         if ( T->argtail ) {
01245             p4 = T->argtail + 1;
01246             while ( FG.cTable[*p4] == 1 ) p4++;
01247             while ( *p4 ) {
01248                 if ( *p4 == '?' && p[-1] != ',' ) {
01249                     p4++;
01250                     if ( FG.cTable[*p4] == 0 || *p4 == '$' || *p4 == '[' ) {
01251                         p4 = SkipAName(p4);
01252                         if ( *p4 == '[' ) {
01253                             SKIPBRA1(p4);
01254                         }
01255                     }
01256                     else if ( *p4 == '{' ) {
01257                         SKIPBRA2(p4);
01258                     }
01259                     else if ( *p4 ) { *p++ = *p4++; continue; }
01260                 }
01261                 else *p++ = *p4++;
01262             }
01263         }
01264         *p++ = ')';
01265         *p = 0;
```

```
01266         inp1 = fake;
01267     }
01268     else {
01269         inp1 = ++p;
01270     }
01271     c = 0;
01272 /*
01273     Now we have the indices and p points to the rhs.
01274 */
01275     numover = 0;
01276     AC.tablefilling = funnum;
01277     while ( *inp1 ) {
01278         p = SkipField(inp1,0);
01279         c = *p; *p = 0;
01280 #ifdef WITHPTHREADS
01281         Tprototype = T->prototype[0];
01282 #else
01283         Tprototype = T->prototype;
01284 #endif
01285         if ( ( i = CompileAlgebra(inp1,RHSIDE,Tprototype) ) < 0 ) { error = 1; i = 0; }
01286         if ( !nofill ) {
01287             T->tablepointers[sum] = i;
01288             T->tablepointers[sum+1] = T->bufnum;
01289         }
01290         AC.DumNum = 0;
01291         *p = c;
01292         if ( T->sparse || c == 0 ) break;
01293         inp1 = ++p;
01294 #if ( TABLEEXTENSION == 2 )
01295         sum++;
01296 #else
01297         sum += 2;
01298 #endif
01299         if ( !nofill && T->tablepointers[sum] >= 0 ) numover++;
01300 #if ( TABLEEXTENSION == 2 )
01301         sum++;
01302 #else
01303         sum += TABLEEXTENSION-2;
01304 #endif
01305     }
01306     if ( AC.exprfillwarning == 1 ) {
01307         AC.exprfillwarning = 2;
01308         Warning("Use of expressions and/or $variables in Fill statements is potentially very
      dangerous.");
01309     }
01310     AC.tablefilling = 0;
01311     if ( T->sparse && c != 0 ) {
01312         MesPrint("&In sparse tables one can fill only one element at a time");
01313         error = 1;
01314     }
01315     else if ( numover ) {
01316         if ( numover == 1 )
01317             Warning("one element was overwritten. New definition will be used");
01318         else if ( AC.WarnFlag )
01319             MesPrint("&Warning: %d elements were overwritten. New definitions will be used",numover);
01320     }
01321     if ( T->sparse ) {
01322         if ( redef == 0 ) T->totind++;
01323     }
01324     else T->defined++;
01325 /*
01326     NumSets = AC.SetList.numtemp;
01327     NumSetElements = AC.SetElementList.numtemp;
01328 */
01329     if ( fake ) {
01330         M_free(fake,"Fill fake rhs");
01331         fake = 0;
01332         functions[funnum].tabl = T = T->spare;
01333         p = p3;
01334         goto andagain;
01335     }
01336     AC.cbufnum = oldcbufnum;
01337     AC.SymChangeFlag = 1;
01338     AT.WorkPointer = oldwp;
01339     functions[funnum].tabl = oldT;
01340     return(error);
01341 }
01342
01343 /*
01344     #] CoFill :
01345     #[ CoFillExpression :
01346
01347     Syntax: FillExpression table = expression(x1,...,xn);
01348     The arguments should have been bracketed. Each corresponds to one
01349     of the dimensions of the table. Then the bracket with x1^2*x3^4
01350     will fill the (2,0,4) element of the table (if n=3 of course).
01351     Brackets that don't fit will be skipped. It just gives a warning.
```

```
01352
01353     New option (13-jul-2005)
01354     Syntax: FillExpression table = expression(f);
01355     The table indices are arguments of the function f which should
01356     have been bracketed before.
01357 */
01358
01359 int CoFillExpression(UBYTE *inp)
01360 {
01361     GETIDENTITY
01362     UBYTE *p, c;
01363     WORD type, funnum, expnum, symnum, numsym = 0, *oldwork = AT.WorkPointer;
01364     WORD *brackets, *term, brasize, *b, *m, *w, *pw, *tstop, zero = 0;
01365     WORD oldcbuf = AC.cbufnum, curelement = 0;
01366     int weneedit, i, j, numzero, pow, numfirst;
01367     TABLES T = 0;
01368     LONG newreservation, numcommu, sum;
01369     POSITION oldposition;
01370     FILEHANDLE *fi;
01371     CBUF *C;
01372     WORD numdummies;
01373
01374     AN.IndDum = AM.IndDum;
01375     if ( ( p = SkipAName(inp) ) == 0 ) return(1);
01376     c = *p; *p = 0;
01377     if ( ( GetVar(inp,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
01378     || ( T = functions[funnum].tabl ) == 0 ) {
01379         MesPrint("&%s should be a previously declared table",inp);
01380         *p = c; return(1);
01381     }
01382     *p++ = c;
01383     if ( T->spare ) T = T->spare;
01384     C = cbuf + T->bufnum;
01385     if ( c != '=' ) {
01386         MesPrint("&No = sign in FillExpression statement");
01387         return(1);
01388     }
01389     inp = p;
01390     if ( ( p = SkipAName(inp) ) == 0 ) return(1);
01391     c = *p; *p = 0;
01392     if ( ( type = GetName(AC.exprnames,inp,&expnum,NOAUTO) ) == NAMENOTFOUND
01393     || c != '(' || (
01394         Expressions[expnum].status != LOCALEXPRESSION &&
01395         Expressions[expnum].status != SKIPLEXPRESSION &&
01396         Expressions[expnum].status != DROPLEXPRESSION &&
01397         Expressions[expnum].status != GLOBALEXPRESSION &&
01398         Expressions[expnum].status != SKIPGEXPRESSION &&
01399         Expressions[expnum].status != DROPGEXPRESSION ) ) {
01400         MesPrint("&%s should be an active expression with arguments",inp);
01401         *p = c; return(1);
01402     }
01403     if ( Expressions[expnum].inmem ) {
01404         MesPrint("&%s cannot be used in a FillExpression statement in the same %n\
01405         module that it has been redefined",inp);
01406         *p = c; return(1);
01407     }
01408     *p++ = c;
01409     while ( *p ) {
01410         inp = p;
01411         if ( ( p = SkipAName(inp) ) == 0 ) return(1);
01412         c = *p; *p = 0;
01413
01414         if ( GetVar(inp,&type,&symnum,-1,NOAUTO) == NAMENOTFOUND ) {
01415             MesPrint("&%s should be a previously declared symbol or function",inp);
01416             *p = c; return(1);
01417         }
01418         else if ( type == CSYMBOL ) {
01419             *p++ = c;
01420             *AT.WorkPointer++ = symnum;
01421             numsym++;
01422         }
01423         else if ( type == CFUNCTION ) {
01424             numsym = -1;
01425             *p++ = c;
01426             if ( c != ')' ) {
01427                 MesPrint("&Argument should be a single function or a list of symbols");
01428                 return(1);
01429             }
01430             symnum += FUNCTION;
01431             *AT.WorkPointer++ = symnum;
01432         }
01433         else {
01434             MesPrint("&%s should be a previously declared symbol or function",inp);
01435             *p = c; return(1);
01436         }
01437         if ( c == ')' ) break;
01438         if ( c != ',' ) {
```

```
01439             MesPrint("&Illegal separator in FillExpression statement");
01440             goto noway;
01441         }
01442     }
01443     if ( *p ) {
01444         MesPrint("&Illegal end of FillExpression statement");
01445         goto noway;
01446     }
01447 /*
01448     We have the number of the table in funnum.
01449     The number of the expression in expnum, the table struct in T
01450     and either the numbers of the symbols in oldwork (there are numsym of them)
01451     or the number of the function in oldwork (just one and numsym = -1).
01452     We don't sort them!!!!
01453 */
01454     if ( ( numsym > 0 ) && ( ABS(T->numind) != numsym ) ) {
01455         MesPrint("&This table needs %d symbols for its array indices");
01456         goto noway;
01457     }
01458     EXCHINOUT
01459 #ifdef WITHMPI
01460     /*
01461      * The workers can't access to the data of the input expression. We need to
01462      * broadcast it to all the workers.
01463      */
01464     PF_BroadcastExpr(&Expressions[expnum], AR.infile);
01465     if ( PF.me == MASTER ) {
01466         /*
01467          * Restore the file position on the master.
01468          */
01469         POSITION pos;
01470         SetEndScratch(AR.infile, &pos);
01471     }
01472 #endif
01473     fi = AR.infile;
01474     if ( fi->handle >= 0 ) {
01475         PUTZERO(oldposition);
01476         SeekFile(fi->handle,&oldposition,SEEK_CUR);
01477         SetScratch(fi,&(Expressions[expnum].onfile));
01478         if ( ISNEGPOS(Expressions[expnum].onfile) ) {
01479 /* INTERNAL_ERROR_EXCL_START */
01480             MesPrint("!>File error in FillExpression");
01481             BACKINOUT
01482             goto noway;
01483 /* INTERNAL_ERROR_EXCL_STOP */
01484         }
01485     }
01486     else {
01487 /*
01488         Note: Because everything fits inside memory we never get problems
01489         with excessive file sizes.
01490 */
01491         SETBASEPOSITION(oldposition,(UBYTE *)(fi->POfill)-(UBYTE *)(fi->PObuffer));
01492         fi->POfill = (WORD *)((UBYTE *)(fi->PObuffer) + BASEPOSITION(Expressions[expnum].onfile));
01493     }
01494     pw = AT.WorkPointer;
01495     if ( numsym < 0 ) { brackets = pw + 1; }
01496     else { brackets = pw + numsym; }
01497     brasize = -1; weneedit = 0; /* stands for we need it */
01498     term = (WORD *)(((UBYTE *)(brackets)) + AM.MaxTer);
01499     AT.WorkPointer = (WORD *)(((UBYTE *)(term)) + AM.MaxTer);
01500     AC.cbufnum = T->bufnum;
01501     AC.tablefilling = funnum;
01502     if ( GetTerm(BHEAD term) > 0 ) {            /* Skip prototype */
01503         while ( GetTerm(BHEAD term) > 0 ) {
01504             GETSTOP(term,tstop);
01505             w = m = term + 1;
01506             while ( m < tstop && *m != HAAKJE ) m += m[1];
01507             if ( *m != HAAKJE ) {
01508                 MesPrint("&Illegal attempt to put an expression without brackets in a table");
01509                 BACKINOUT
01510                 goto noway;
01511             }
01512             if ( brasize == m - w ) {
01513                 b = brackets;
01514                 while ( *b == *w && w < m ) { b++; w++; }
01515                 if ( w == m ) { /* Same as current bracket. Copy. */
01516                     if ( weneedit ) {
01517                         m += m[1] - 1;
01518                         *m = *term - (m-term);
01519                         AddNtoC(AC.cbufnum,*m,m,3);
01520                         numdummies = DetCurDum(BHEAD term) - AM.IndDum;
01521                         if ( numdummies > T->numdummies ) T->numdummies = numdummies;
01522                     }
01523                     continue; /* Next term */
01524                 }
01525             }
```

```
01526             if ( weneedit ) {
01527                 AddNtoC(AC.cbufnum,1,&zero,4);   /* Terminate old bracket */
01528                 numcommu = numcommute(C->rhs[curelement],&(C->NumTerms[curelement]));
01529                 C->CanCommu[curelement] = numcommu;
01530             }
01531             b = brackets; w = term + 1;
01532             if ( numsym < 0 ) pw = oldwork + 1;
01533             else              pw = oldwork + numsym;
01534             while ( w < m ) *b++ = *w++;
01535             brasize = b - brackets;
01536 /*
01537             Now compute the element. See whether we need it
01538 */
01539             if ( numsym < 0 ) {
01540                 WORD *bb, bnum;
01541                 if ( *brackets != symnum || brasize != brackets[1] ) {
01542                     weneedit = 0; continue; /* Cannot work! */
01543                 }
01544 /*
01545                     Now count the number of arguments and whether they are numbers
01546 */
01547                 b = brackets + FUNHEAD;
01548                 bb = brackets+brackets[1];
01549                 i = 0;
01550                 if ( T->numind < 0 ) {
01551                     bnum = b[1]+1;
01552                     if ( bnum > -T->numind ) {
01553                         weneedit = 0; continue; /* Cannot work! */
01554                     }
01555                 }
01556                 else bnum = T->numind;
01557                 while ( b < bb ) {
01558                     if ( *b != -SNUMBER ) break;
01559                     i++;
01560                     b += 2;
01561                 }
01562                 if ( b < bb || i != bnum ) {
01563                     weneedit = 0; continue; /* Cannot work! */
01564                 }
01565             }
01566             else if ( brasize > 0 && ( *brackets != SYMBOL
01567             || brackets[1] < brasize || (brackets[1]-2) > numsym*2 ) ) {
01568                 weneedit = 0; continue; /* Cannot work! */
01569             }
01570             numzero = 0; sum = 0;
01571             numfirst = 0;
01572             if ( numsym > 0 ) {
01573               for ( i = 0; i < numsym; i++ ) {
01574                 if ( brasize > 0 ) {
01575                     b = brackets + 2; j = brackets[1]-2;
01576                     while ( j > 0 ) {
01577                         if ( *b == oldwork[i] ) break;
01578                         j -= 2; b += 2;
01579                     }
01580                     if ( j <= 0 ) {  /* it was not there */
01581                         numzero++; pow = 0;
01582                         if ( 2*numzero+brackets[1]-2 > numsym*2 ) {
01583                             weneedit = 0; goto nextterm;
01584                         }
01585                     }
01586                     else pow = b[1];
01587                 }
01588                 else pow = 0;
01589                 if ( T->sparse ) {
01590                     if ( T->numind < 0 ) {
01591                         if ( i == 0 ) {
01592                             numfirst = pow;
01593                             if ( pow > -T->numind ) {
01594                                 weneedit = 0; goto nextterm;
01595                             }
01596                         }
01597                         else if ( i > pow ) {
01598                             weneedit = 0; goto nextterm;
01599                         }
01600                     }
01601                     *pw++ = pow;
01602                 }
01603                 else if ( pow < T->mm[i].mini || pow > T->mm[i].maxi ) {
01604                     weneedit = 0; goto nextterm;
01605                 }
01606                 else sum += ( pow - T->mm[i].mini ) * T->mm[i].size;
01607               }
01608             }
01609             else {
01610               WORD xx;
01611               b = brackets + FUNHEAD;
01612               sum = 0;
```

```
01613 /*
01614               Now scan the arguments of the function.
01615               We did check already the number and type of the arguments.
01616 */
01617               xx = (brackets[1]-FUNHEAD)/2;
01618               for ( i = 0; i < xx; i++ ) {
01619                 pow = b[1];
01620                 b += 2;
01621                 if ( T->sparse ) {
01622                     if ( T->numind < 0 ) {
01623                         if ( i == 0 ) {
01624                             numfirst = pow;
01625                             if ( pow >= -T->numind ) {
01626                                 weneedit = 0; goto nextterm;
01627                             }
01628                         }
01629                     }
01630                     *pw++ = pow;
01631                 }
01632                 else if ( pow < T->mm[i].mini || pow > T->mm[i].maxi ) {
01633                     weneedit = 0; goto nextterm;
01634                 }
01635                 else sum += ( pow - T->mm[i].mini ) * T->mm[i].size;
01636               }
01637             }
01638             if ( T->numind < 0 ) {
01639                 for ( i = numfirst+1; i < -T->numind; i++ ) *pw++ = 0;
01640             }
01641             weneedit = 1;
01642             if ( T->sparse ) {
01643                 if ( numsym < 0 ) pw = oldwork + 1;
01644                 else              pw = oldwork + ABS(T->numind);
01645                 i = FindTableTree(T,pw,1);
01646                 if ( i >= 0 ) {
01647                     sum = i+ABS(T->numind);
01648 /*
01649 Wrong!!!!           C->rhs[T->tablepointers[sum]] = C->Pointer;
01650 */
01651                     C->Pointer--; /* Back up over the zero */
01652                     goto newentry;
01653                 }
01654                 if ( T->totind >= T->reserved ) {
01655                     if ( T->reserved == 0 ) newreservation = 20;
01656                     else newreservation = T->reserved;
01657 /*---Copied from Fill--------------------------*/
01658                     while ( T->totind >= newreservation && newreservation < MAXTABLECOMBUF )
01659                         newreservation = 2*newreservation;
01660                     if ( newreservation > MAXTABLECOMBUF ) newreservation = MAXTABLECOMBUF;
01661                     if ( T->totind >= newreservation ) {
01662                         MesPrint("@More than %ld elements in sparse table",MAXTABLECOMBUF);
01663                         AC.cbufnum = oldcbuf;
01664                         AT.WorkPointer = oldwork;
01665                         Terminate(-1);
01666                     }
01667 /*---Copied from Fill--------------------------*/
01668                     if ( T->totind >= newreservation ) {
01669                         MesPrint("@More than %ld elements in sparse table",MAXTABLECOMBUF);
01670                         AC.cbufnum = oldcbuf;
01671                         AT.WorkPointer = oldwork;
01672                         Terminate(-1);
01673                     }
01674                     w = (WORD *)Malloc1(newreservation*sizeof(WORD)*
01675                             (ABS(T->numind)+TABLEEXTENSION),"tablepointers");
01676                     for ( i = T->reserved*(ABS(T->numind)+TABLEEXTENSION)-1; i >= 0; i-- )
01677                         w[i] = T->tablepointers[i];
01678                     if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
01679                     T->tablepointers = w;
01680                     T->reserved = newreservation;
01681                 }
01682                 if ( numsym < 0 ) pw = oldwork + 1;
01683                 else              pw = oldwork + numsym;
01684                 for ( sum = T->totind*(ABS(T->numind)+TABLEEXTENSION), i = 0; i < ABS(T->numind); i++
      ) {
01685                     T->tablepointers[sum++] = *pw++;
01686                 }
01687                 InsTableTree(T,T->tablepointers+sum-ABS(T->numind));
01688                 (T->totind)++;
01689             }
01690 #if ( TABLEEXTENSION != 2 )
01691             else {
01692                 sum *= TABLEEXTENSION;
01693             }
01694 #endif
01695 /*
01696             Start a new entry. Copy the element.
01697 */
01698             AddRHS(T->bufnum,0);
```

```
01699             T->tablepointers[sum] = C->numrhs;
01700 #if ( TABLEEXTENSION == 2 )
01701             T->tablepointers[sum+TABLEEXTENSION-1] = -1;
01702 #else
01703             T->tablepointers[sum+1] = T->bufnum;
01704             T->tablepointers[sum+2] = -1;
01705             T->tablepointers[sum+3] = -1;
01706             T->tablepointers[sum+4] = 0;
01707             T->tablepointers[sum+5] = 0;
01708 #endif
01709 newentry:   if ( *m == HAAKJE ) { m += m[1] - 1; }
01710             else m--;
01711             *m = *term - (m-term);
01712             AddNtoC(AC.cbufnum,*m,m,5);
01713             curelement = T->tablepointers[sum];
01714 nextterm:;
01715         }
01716         if ( weneedit ) {
01717             AddNtoC(AC.cbufnum,1,&zero,6);   /* Terminate old bracket */
01718             numcommu = numcommute(C->rhs[curelement],&(C->NumTerms[curelement]));
01719             C->CanCommu[curelement] = numcommu;
01720         }
01721     }
01722     if ( fi->handle >= 0 ) {
01723         SetScratch(fi,&(oldposition));
01724     }
01725     else {
01726         fi->POfill = (WORD *)((UBYTE *)(fi->PObuffer) + BASEPOSITION(oldposition));
01727     }
01728     BACKINOUT
01729     AC.cbufnum = oldcbuf;
01730     AC.tablefilling = 0;
01731     AT.WorkPointer = oldwork;
01732     return(0);
01733 noway:
01734     BACKINOUT
01735     AC.cbufnum = oldcbuf;
01736     AC.tablefilling = 0;
01737     AT.WorkPointer = oldwork;
01738     return(1);
01739 }
01740
01741 /*
01742     #] CoFillExpression :
01743     #[ CoPrintTable :
01744
01745     Syntax
01746         PrintTable [+f] [+s] tablename [>[>] file];
01747     All defined elements are written with individual Fill statements.
01748     If a file is specified, the result is written to file only.
01749     The flags of the print statement apply as much as possible.
01750     We make use of the regular write routines.
01751 */
01752
01753 int CoPrintTable(UBYTE *inp)
01754 {
01755     GETIDENTITY
01756     int fflag = 0, sflag = 0, addflag = 0, error = 0, sum, i, j;
01757     UBYTE *filename, *p, c, buffer[100], *s, *oldoutputline = AO.OutputLine;
01758     WORD type, funnum, *expr, *m, num;
01759     TABLES T = 0;
01760     WORD oldSkip = AO.OutSkip, oldMode = AC.OutputMode, oldHandle = AC.LogHandle;
01761     WORD oldType = AO.PrintType, *oldwork = AT.WorkPointer;
01762     UBYTE *oldFill = AO.OutFill, *oldLine = AO.OutputLine;
01763 #ifdef WITHMPI
01764     if ( PF.me != MASTER ) return 0;
01765 #endif
01766 /*
01767     First the flags
01768 */
01769     while ( *inp == '+' ) {
01770         inp++;
01771         if ( *inp == 'f' || *inp == 'F' ) { fflag = 1; inp++; }
01772         else if ( *inp == 's' || *inp == 'S' ) { sflag = PRINTONETERM; inp++; }
01773         else {
01774             MesPrint("&Illegal + option in PrintTable statement");
01775             error = 1; inp++;
01776         }
01777         while ( *inp != ',' && *inp && *inp != '+' ) {
01778             if ( !error ) {
01779                 if ( *inp ) {
01780                     MesPrint("&Illegal + option in PrintTable statement");
01781                     inp++;
01782                 }
01783                 else {
01784                     MesPrint("&Unfinished PrintTable statement");
01785                     return(1);
```

```
01786                 }
01787                 error = 1;
01788             }
01789             inp++;
01790         }
01791         if ( *inp == ',' ) inp++;
01792     }
01793 /*
01794     Now the name of the table
01795 */
01796     if ( ( p = SkipAName(inp) ) == 0 ) return(1);
01797     c = *p; *p = 0;
01798     if ( ( GetVar(inp,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
01799     || ( T = functions[funnum].tabl ) == 0 ) {
01800         MesPrint("&%s should be a previously declared table",inp);
01801         *p = c; return(1);
01802     }
01803     if ( T->spare && T->mode == 1 ) T = T->spare;
01804     *p++ = c;
01805 /*
01806     Check for a filename. Runs to the end of the statement.
01807 */
01808     filename = 0;
01809     if ( c == '>' ) {
01810         if ( *p == '>' ) { addflag = 1; p++; }
01811         filename = p;
01812     }
01813     else filename = 0;
01814
01815     if ( filename ) {
01816         if ( addflag ) AC.LogHandle = OpenAddFile((char *)filename);
01817         else           AC.LogHandle = CreateFile((char *)filename);
01818         if ( AC.LogHandle < 0 ) {
01819             MesPrint("&Cannot open file '%s' properly",filename);
01820             error = 1; goto finally;
01821         }
01822         AO.PrintType = PRINTLFILE;
01823     }
01824     else if ( fflag && AC.LogHandle >= 0 ) {
01825         AO.PrintType = PRINTLFILE;
01826     }
01827     AO.OutFill = AO.OutputLine = (UBYTE *)AT.WorkPointer;
01828     AT.WorkPointer += 2*AC.LineLength;
01829
01830     AO.PrintType |= sflag;
01831     AC.OutputMode = 0;
01832     AO.IsBracket = 0;
01833     AO.OutSkip = 0;
01834     AR.DeferFlag = 0;
01835     AC.outsidefun = 1;
01836     if ( AC.LogHandle == oldHandle ) FiniLine();
01837     AO.OutputLine = AO.OutFill = (UBYTE *)Malloc1(AC.LineLength+20,"PrintTable");
01838     AO.OutStop = AO.OutFill + AC.LineLength;
01839     for ( i = 0; i < T->totind; i++ ) {
01840         if ( !T->sparse && T->tablepointers[i*TABLEEXTENSION] < 0 ) continue;
01841         TokenToLine((UBYTE *)"Fill ");
01842         TokenToLine((UBYTE *)(VARNAME(functions,funnum)));
01843         TokenToLine((UBYTE *)"(");
01844         AO.OutSkip = 3;
01845         if ( T->sparse ) {
01846             sum = i * ( T->numind + TABLEEXTENSION );
01847             for ( j = 0; j < T->numind; j++, sum++ ) {
01848                 if ( j > 0 ) TokenToLine((UBYTE *)",");
01849                 num = T->tablepointers[sum];
01850                 s = buffer; s = NumCopy(num,s);
01851                 TokenToLine(buffer);
01852             }
01853             expr = cbuf[T->bufnum].rhs[T->tablepointers[sum]];
01854         }
01855         else {
01856             for ( j = 0; j < T->numind; j++ ) {
01857                 if ( j > 0 ) {
01858                     TokenToLine((UBYTE *)",");
01859                     num = T->mm[j].mini + ( i % T->mm[j-1].size ) / T->mm[j].size;
01860                 }
01861                 else {
01862                     num = T->mm[j].mini + i / T->mm[j].size;
01863                 }
01864                 s = buffer; s = NumCopy(num,s);
01865                 TokenToLine(buffer);
01866             }
01867             expr = cbuf[T->bufnum].rhs[T->tablepointers[TABLEEXTENSION*i]];
01868         }
01869         TOKENTOLINE(") =")=");
01870         if ( sflag ) {
01871             FiniLine();
01872             if ( AC.OutputSpaces != NOSPACEFORMAT ) TokenToLine((UBYTE *)"   ");
```

```
01873         }
01874         m = expr;
01875 /*
01876         WORD lbrac, first;
01877         lbrac = 0; first = 1;
01878         while ( *m ) {
01879             if ( WriteTerm(m,&lbrac,first,1,0) ) {
01880                 MesPrint("Error while writing table");
01881                 error = 1;
01882                 goto finally;
01883             }
01884             first = 0;
01885             m += *m;
01886         }
01887         if ( first ) { TOKENTOLINE(" 0","0") }
01888         else if ( lbrac ) { TOKENTOLINE(" )",")") }
01889 */
01890         while ( *m ) m += *m;
01891         if ( m > expr ) {
01892             if ( WriteExpression(expr,(LONG)(m-expr)) ) { error = 1; goto finally; }
01893             AO.OutSkip = 0;
01894         }
01895         else {
01896             TokenToLine((UBYTE *)"0");
01897         }
01898         TokenToLine((UBYTE *)";");
01899         FiniLine();
01900     }
01901     M_free(AO.OutputLine,"PrintTable");
01902     AO.OutputLine = AO.OutFill = oldoutputline;
01903 /*
01904     Reset the file pointers and parameters if any. Close file if needed.
01905 */
01906 finally:
01907     AO.OutSkip     = oldSkip;
01908     AC.OutputMode  = oldMode;
01909     AC.LogHandle   = oldHandle;
01910     AO.PrintType   = oldType;
01911     AO.OutFill     = oldFill;
01912     AO.OutputLine  = oldLine;
01913     AT.WorkPointer = oldwork;
01914     AC.outsidefun  = 0;
01915     return(error);
01916 }
01917
01918 /*
01919     #] CoPrintTable :
01920     #[ CoAssign :
01921
01922     This statement has an easy syntax:
01923         $name = expression
01924 */
01925
01926 static WORD AssignLHS[14] = { TYPEASSIGN, 3+SUBEXPSIZE, 0,
01927                               SUBEXPRESSION, SUBEXPSIZE, 0, 1, 0, 0,0,0,0,0 };
01928
01929 int CoAssign(UBYTE *inp)
01930 {
01931     int error = 0, retcode;
01932     UBYTE *name, c;
01933     WORD number;
01934     if ( *inp != '$' ) {
01935 nolhs:  MesPrint("&assign statement should have a dollar variable in the LHS");
01936         return(1);
01937     }
01938     inp++; name = inp;
01939     if ( FG.cTable[*inp] != 0 ) goto nolhs;
01940     while ( FG.cTable[*inp] < 2 ) inp++;
01941     if ( AP.PreAssignFlag == 2 ) {
01942         if ( *inp == '_' ) inp++;
01943     }
01944     if ( ( *inp == ',' && inp[1] != '=' ) && ( *inp != '=' ) ) {
01945         MesPrint("&assign statement should have only a dollar variable in the LHS");
01946         return(1);
01947     }
01948     c = *inp;
01949     *inp = 0;
01950     if ( GetName(AC.dollarnames,name,&number,NOAUTO) == NAMENOTFOUND ) {
01951         number = AddDollar(name,DOLUNDEFINED,0,0);
01952     }
01953     *inp = c;
01954     if ( c == ',' ) inp++;
01955     *inp++ = '=';
01956     if ( *inp == ',' ) inp++;
01957 /*
01958     Fake a Prototype and read the RHS
01959 */
```

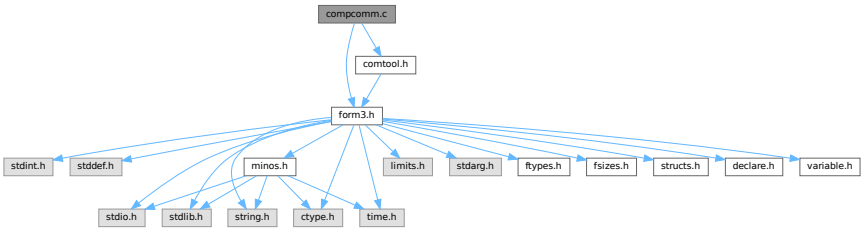
```
01960     AssignLHS[7] = AC.cbufnum;
01961     retcode = CompileAlgebra(inp,RHSIDE,(AssignLHS+3));
01962     if ( retcode < 0 ) error = 1;
01963     AC.DumNum = 0;
01964 /*
01965     Now add the LHS
01966 */
01967     AssignLHS[2] = number;
01968     AssignLHS[5] = retcode;
01969     AddNtoL(AssignLHS[1],AssignLHS);
01970 /*
01971     Add to the list of potentially modified dollars (for PARALLEL)
01972 */
01973     AddPotModdollar(number);
01974     return(error);
01975 }
01976
01977 /*
01978     #] CoAssign :
01979     #[ CoDeallocateTable :
01980
01981     Syntax: DeallocateTable tablename(s);
01982     Should work only for sparse tables.
01983     Action: Cleans all definitions of elements of a table as if there have
01984             never been any fill statements.
01985 */
01986
01987 int CoDeallocateTable(UBYTE *inp)
01988 {
01989     UBYTE *p, c;
01990     TABLES T = 0;
01991     WORD type, funnum, i;
01992     c = *inp;
01993     while ( c ) {
01994         while ( *inp == ',' ) inp++;
01995         if ( *inp == 0 ) break;
01996         if ( ( p = SkipAName(inp) ) == 0 ) return(1);
01997         c = *p; *p = 0;
01998         if ( ( GetVar(inp,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
01999         || ( T = functions[funnum].tabl ) == 0 ) {
02000             MesPrint("&%s should be a previously declared table",inp);
02001             *p = c; return(1);
02002         }
02003         if ( T->sparse == 0 ) {
02004             MesPrint("&%s should be a sparse table",inp);
02005             *p = c; return(1);
02006         }
02007         if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
02008         ClearTableTree(T);
02009         for (i = 0; i < T->buffersfill; i++ ) { /* was <= */
02010             finishcbuf(T->buffers[i]);
02011         }
02012         T->bufnum = inicbufs();
02013         T->buffersfill = 0;
02014         T->buffers[T->buffersfill++] = T->bufnum;
02015         T->tablepointers = 0;
02016         T->boomlijst = 0;
02017         T->totind = 0;
02018         T->reserved = 0;
02019
02020         if ( T->spare ) {
02021             TABLES TT = T->spare;
02022             if ( TT->tablepointers ) M_free(TT->tablepointers,"tablepointers");
02023             ClearTableTree(TT);
02024             for (i = 0; i < TT->buffersfill; i++ ) { /* was <= */
02025                 finishcbuf(TT->buffers[i]);
02026             }
02027             TT->bufnum = inicbufs();
02028             TT->buffersfill = 0;
02029             TT->buffers[T->buffersfill++] = T->bufnum;
02030             TT->tablepointers = 0;
02031             TT->boomlijst = 0;
02032             TT->totind = 0;
02033             TT->reserved = 0;
02034         }
02035         *p++ = c;
02036         inp = p;
02037     }
02038     return(0);
02039 }
02040
02041 /*
02042     #] CoDeallocateTable :
02043     #[ CoFactorCache :
02044 */
02054 /*
02055 int CoFactorCache(UBYTE *inp)
```

```
02056 {
02057     Code to be added in due time
02058     We need to read 'expression', get its terms through Generator and sort them.
02059     We store the result in the WorkSpace in argument notation.
02060     This will be argin.
02061     Then we do the same with the sequence of factors. They form argout.
02062     The whole is put in the buffer with the call
02063         InsertArg(BHEAD argin,argout,1)
02064     return(0);
02065 }
02066 */
02067
02068 /*
02069     #] CoFactorCache :
02070 */
```

4.9 compcomm.c File Reference

```
#include "form3.h"
#include "comtool.h"
```

Include dependency graph for compcomm.c:



Functions

- int CoFormat (UBYTE *s)
- int CoCollect (UBYTE *s)
- int setonoff (UBYTE *s, int *flag, int onvalue, int offvalue)
- int CoCompress (UBYTE *s)
- int CoFlags (UBYTE *s, int value)
- int CoOff (UBYTE *s)
- int CoOn (UBYTE *s)
- int CoInsideFirst (UBYTE *s)
- int CoProperCount (UBYTE *s)
- int CoDelete (UBYTE *s)
- int CoKeep (UBYTE *s)
- int CoFixIndex (UBYTE *s)
- int CoMetric (UBYTE *s)
- int DoPrint (UBYTE *s, int par)
- int CoPrint (UBYTE *s)
- int CoPrintB (UBYTE *s)
- int CoNPrint (UBYTE *s)
- int CoPushHide (UBYTE *s)
- int CoPopHide (UBYTE *s)
- int SetExprCases (int par, int setunset, int val)
- int SetExpr (UBYTE *s, int setunset, int par)
- int CoDrop (UBYTE *s)

- int CoNoDrop (UBYTE *s)
- int CoSkip (UBYTE *s)
- int CoNoSkip (UBYTE *s)
- int CoHide (UBYTE *inp)
- int CoIntoHide (UBYTE *inp)
- int CoNoIntoHide (UBYTE *inp)
- int CoNoHide (UBYTE *inp)
- int CoUnHide (UBYTE *inp)
- int CoNoUnHide (UBYTE *inp)
- void AddToCom (int n, WORD *array)
- int AddComString (int n, WORD *array, UBYTE *thestring, int par)
- int Add2ComStrings (int n, WORD *array, UBYTE *string1, UBYTE *string2)
- int CoDiscard (UBYTE *s)
- int CoContract (UBYTE *s)
- int CoGoTo (UBYTE *inp)
- int CoLabel (UBYTE *inp)
- int DoArgument (UBYTE *s, int par)
- int CoArgument (UBYTE *s)
- int CoEndArgument (UBYTE *s)
- int CoInside (UBYTE *s)
- int CoEndInside (UBYTE *s)
- int CoNormalize (UBYTE *s)
- int CoMakeInteger (UBYTE *s)
- int CoSplitArg (UBYTE *s)
- int CoSplitFirstArg (UBYTE *s)
- int CoSplitLastArg (UBYTE *s)
- int CoFactArg (UBYTE *s)
- int DoSymmetrize (UBYTE *s, int par)
- int CoSymmetrize (UBYTE *s)
- int CoAntiSymmetrize (UBYTE *s)
- int CoCycleSymmetrize (UBYTE *s)
- int CoRCycleSymmetrize (UBYTE *s)
- int CoWrite (UBYTE *s)
- int CoNWrite (UBYTE *s)
- int CoRatio (UBYTE *s)
- int CoRedefine (UBYTE *s)
- int CoRenumber (UBYTE *s)
- int CoSum (UBYTE *s)
- int CoToTensor (UBYTE *s)
- int CoToVector (UBYTE *s)
- int CoTrace4 (UBYTE *s)
- int CoTraceN (UBYTE *s)
- int CoChisholm (UBYTE *s)
- int DoChain (UBYTE *s, int option)
- int CoChainin (UBYTE *s)
- int CoChainout (UBYTE *s)
- int CoExit (UBYTE *s)
- int CoInParallel (UBYTE *s)
- int CoNotInParallel (UBYTE *s)
- int DoInParallel (UBYTE *s, int par)
- int CoInExpression (UBYTE *s)
- int CoEndInExpression (UBYTE *s)
- int CoSetExitFlag (UBYTE *s)
- int CoTryReplace (UBYTE *p)
- int CoModulus (UBYTE *inp)

- int CoRepeat (UBYTE *inp)
- int CoEndRepeat (UBYTE *inp)
- int DoBrackets (UBYTE *inp, int par)
- int CoBracket (UBYTE *inp)
- int CoAntiBracket (UBYTE *inp)
- int CoMultiBracket (UBYTE *inp)
- WORD * CountComp (UBYTE *inp, WORD *to)
- int CoIf (UBYTE *inp)
- int CoElse (UBYTE *p)
- int CoElseIf (UBYTE *inp)
- int CoEndIf (UBYTE *inp)
- int CoWhile (UBYTE *inp)
- int CoEndWhile (UBYTE *inp)
- int DoFindLoop (UBYTE *inp, int mode)
- int CoFindLoop (UBYTE *inp)
- int CoReplaceLoop (UBYTE *inp)
- int CoFunPowers (UBYTE *inp)
- int CoUnitTrace (UBYTE *s)
- int CoTerm (UBYTE *s)
- int CoEndTerm (UBYTE *s)
- int CoSort (UBYTE *s)
- int CoPolyFun (UBYTE *s)
- int CoPolyRatFun (UBYTE *s)
- int CoMerge (UBYTE *inp)
- int CoStuffle (UBYTE *inp)
- int CoProcessBucket (UBYTE *s)
- int CoThreadBucket (UBYTE *s)
- int DoArgPlode (UBYTE *s, int par)
- int CoArgExplode (UBYTE *s)
- int CoArgImplode (UBYTE *s)
- int CoClearTable (UBYTE *s)
- int CoDenominators (UBYTE *s)
- int CoDropCoefficient (UBYTE *s)
- int CoDropSymbols (UBYTE *s)
- int CoToPolynomial (UBYTE *inp)
- int CoFromPolynomial (UBYTE *inp)
- int CoArgToExtraSymbol (UBYTE *s)
- int CoExtraSymbols (UBYTE *inp)
- WORD * GetIfDollarFactor (UBYTE **inp, WORD *w)
- UBYTE * GetDoParam (UBYTE *inp, WORD **wp, int par)
- int CoDo (UBYTE *inp)
- int CoEndDo (UBYTE *inp)
- int CoFactDollar (UBYTE *inp)
- int CoFactorize (UBYTE *s)
- int CoNFactorize (UBYTE *s)
- int CoUnFactorize (UBYTE *s)
- int CoNUnFactorize (UBYTE *s)
- int DoFactorize (UBYTE *s, int par)
- int CoOptimizeOption (UBYTE *s)
- int CoPutInside (UBYTE *inp)
- int CoAntiPutInside (UBYTE *inp)
- int DoPutInside (UBYTE *inp, int par)
- int CoSwitch (UBYTE *s)
- int CoCase (UBYTE *s)
- int CoBreak (UBYTE *s)

- int CoDefault (UBYTE *s)
- int CoEndSwitch (UBYTE *s)
- int CoSetUserFlag (UBYTE *s)
- int CoClearUserFlag (UBYTE *s)
- int CoCreateAllLoops (UBYTE *s)
- int CoCreateAllPaths (UBYTE *s)
- int CoCreateAll (UBYTE *s)

4.9.1 Detailed Description

Compiler routines for most statements that don't involve algebraic expressions. Exceptions: all routines involving declarations are in the file names.c When making new statements one can add the compiler routines here and have a look whether there is already a routine that is similar. In that case one can make a copy and modify it.

Definition in file compcomm.c.

4.9.2 Function Documentation

4.9.2.1 CoFormat()

```
int CoFormat (
            UBYTE * s )
```

Definition at line 157 of file compcomm.c.

4.9.2.2 CoCollect()

```
int CoCollect (
            UBYTE * s )
```

Definition at line 437 of file compcomm.c.

4.9.2.3 setonoff()

```
int setonoff (
            UBYTE * s,
            int * flag,
            int onvalue,
            int offvalue )
```

Definition at line 499 of file compcomm.c.

4.9.2.4 CoCompress()

```
int CoCompress (
            UBYTE * s )
```

Definition at line 515 of file compcomm.c.

4.9.2.5 CoFlags()

```
int CoFlags (
            UBYTE * s,
            int value )
```

Definition at line 564 of file compcomm.c.

4.9.2.6 CoOff()

```
int CoOff (
            UBYTE * s )
```

Definition at line 599 of file compcomm.c.

4.9.2.7 CoOn()

```
int CoOn (
            UBYTE * s )
```

Definition at line 665 of file compcomm.c.

4.9.2.8 CoInsideFirst()

```
int CoInsideFirst (
            UBYTE * s )
```

Definition at line 937 of file compcomm.c.

4.9.2.9 CoProperCount()

```
int CoProperCount (
            UBYTE * s )
```

Definition at line 944 of file compcomm.c.

4.9.2.10 CoDelete()

```
int CoDelete (
            UBYTE * s )
```

Definition at line 951 of file compcomm.c.

4.9.2.11 CoKeep()

```
int CoKeep (
            UBYTE * s )
```

Definition at line 998 of file compcomm.c.

4.9.2.12 CoFixIndex()

```
int CoFixIndex (
            UBYTE * s )
```

Definition at line 1010 of file compcomm.c.

4.9.2.13 CoMetric()

```
int CoMetric (
            UBYTE * s )
```

Definition at line 1044 of file compcomm.c.

4.9.2.14 DoPrint()

```
int DoPrint (
            UBYTE * s,
            int par )
```

Definition at line 1052 of file compcomm.c.

4.9.2.15 CoPrint()

```
int CoPrint (
            UBYTE * s )
```

Definition at line 1268 of file compcomm.c.

4.9.2.16 CoPrintB()

```
int CoPrintB (
            UBYTE * s )
```

Definition at line 1275 of file compcomm.c.

4.9.2.17 CoNPrint()

```
int CoNPrint (
            UBYTE * s )
```

Definition at line 1282 of file compcomm.c.

4.9.2.18 CoPushHide()

```
int CoPushHide (
            UBYTE * s )
```

Definition at line 1289 of file compcomm.c.

4.9.2.19 CoPopHide()

```
int CoPopHide (
            UBYTE * s )
```

Definition at line 1333 of file compcomm.c.

4.9.2.20 SetExprCases()

```
int SetExprCases (
            int par,
            int setunset,
            int val )
```

Definition at line 1368 of file compcomm.c.

4.9.2.21 SetExpr()

```
int SetExpr (
            UBYTE * s,
            int setunset,
            int par )
```

Definition at line 1523 of file compcomm.c.

4.9.2.22 CoDrop()

```
int CoDrop (
            UBYTE * s )
```

Definition at line 1568 of file compcomm.c.

4.9.2.23 CoNoDrop()

```
int CoNoDrop (
            UBYTE * s )
```

Definition at line 1575 of file compcomm.c.

4.9.2.24 CoSkip()

```
int CoSkip (
            UBYTE * s )
```

Definition at line 1582 of file compcomm.c.

4.9.2.25 CoNoSkip()

```
int CoNoSkip (
            UBYTE * s )
```

Definition at line 1589 of file compcomm.c.

4.9.2.26 CoHide()

```
int CoHide (
            UBYTE * inp )
```

Definition at line 1596 of file compcomm.c.

4.9.2.27 CoIntoHide()

```
int CoIntoHide (
            UBYTE * inp )
```

Definition at line 1614 of file compcomm.c.

4.9.2.28 CoNoIntoHide()

```
int CoNoIntoHide (
            UBYTE * inp )
```

Definition at line 1632 of file compcomm.c.

4.9.2.29 CoNoHide()

```
int CoNoHide (
            UBYTE * inp )
```

Definition at line 1639 of file compcomm.c.

4.9.2.30 CoUnHide()

```
int CoUnHide (
            UBYTE * inp )
```

Definition at line 1646 of file compcomm.c.

4.9.2.31 CoNoUnHide()

```
int CoNoUnHide (
            UBYTE * inp )
```

Definition at line 1653 of file compcomm.c.

4.9.2.32 AddToCom()

```
void AddToCom (
             int n,
             WORD * array )
```

Definition at line 1660 of file compcomm.c.

4.9.2.33 AddComString()

```
int AddComString (
             int n,
             WORD * array,
             UBYTE * thestring,
             int par )
```

Definition at line 1675 of file compcomm.c.

4.9.2.34 Add2ComStrings()

```
int Add2ComStrings (
             int n,
             WORD * array,
             UBYTE * string1,
             UBYTE * string2 )
```

Definition at line 1734 of file compcomm.c.

4.9.2.35 CoDiscard()

```
int CoDiscard (
             UBYTE * s )
```

Definition at line 1775 of file compcomm.c.

4.9.2.36 CoContract()

```
int CoContract (
             UBYTE * s )
```

Definition at line 1798 of file compcomm.c.

4.9.2.37 CoGoTo()

```
int CoGoTo (
             UBYTE * inp )
```

Definition at line 1827 of file compcomm.c.

4.9.2.38 CoLabel()

```
int CoLabel (
            UBYTE * inp )
```

Definition at line 1846 of file compcomm.c.

4.9.2.39 DoArgument()

```
int DoArgument (
            UBYTE * s,
            int par )
```

Definition at line 1873 of file compcomm.c.

4.9.2.40 CoArgument()

```
int CoArgument (
            UBYTE * s )
```

Definition at line 2124 of file compcomm.c.

4.9.2.41 CoEndArgument()

```
int CoEndArgument (
            UBYTE * s )
```

Definition at line 2131 of file compcomm.c.

4.9.2.42 CoInside()

```
int CoInside (
            UBYTE * s )
```

Definition at line 2157 of file compcomm.c.

4.9.2.43 CoEndInside()

```
int CoEndInside (
            UBYTE * s )
```

Definition at line 2164 of file compcomm.c.

4.9.2.44 CoNormalize()

```
int CoNormalize (
            UBYTE * s )
```

Definition at line 2190 of file compcomm.c.

4.9.2.45 CoMakeInteger()

```
int CoMakeInteger (
            UBYTE * s )
```

Definition at line 2197 of file compcomm.c.

4.9.2.46 CoSplitArg()

```
int CoSplitArg (
            UBYTE * s )
```

Definition at line 2204 of file compcomm.c.

4.9.2.47 CoSplitFirstArg()

```
int CoSplitFirstArg (
            UBYTE * s )
```

Definition at line 2211 of file compcomm.c.

4.9.2.48 CoSplitLastArg()

```
int CoSplitLastArg (
            UBYTE * s )
```

Definition at line 2218 of file compcomm.c.

4.9.2.49 CoFactArg()

```
int CoFactArg (
            UBYTE * s )
```

Definition at line 2225 of file compcomm.c.

4.9.2.50 DoSymmetrize()

```
int DoSymmetrize (
            UBYTE * s,
            int par )
```

Definition at line 2245 of file compcomm.c.

4.9.2.51 CoSymmetrize()

```
int CoSymmetrize (
            UBYTE * s )
```

Definition at line 2365 of file compcomm.c.

4.9.2.52 CoAntiSymmetrize()

```
int CoAntiSymmetrize (
            UBYTE * s )
```

Definition at line 2372 of file compcomm.c.

4.9.2.53 CoCycleSymmetrize()

```
int CoCycleSymmetrize (
            UBYTE * s )
```

Definition at line 2379 of file compcomm.c.

4.9.2.54 CoRCycleSymmetrize()

```
int CoRCycleSymmetrize (
            UBYTE * s )
```

Definition at line 2386 of file compcomm.c.

4.9.2.55 CoWrite()

```
int CoWrite (
            UBYTE * s )
```

Definition at line 2393 of file compcomm.c.

4.9.2.56 CoNWrite()

```
int CoNWrite (
            UBYTE * s )
```

Definition at line 2418 of file compcomm.c.

4.9.2.57 CoRatio()

```
int CoRatio (
            UBYTE * s )
```

Definition at line 2445 of file compcomm.c.

4.9.2.58 CoRedefine()

```
int CoRedefine (
            UBYTE * s )
```

Definition at line 2485 of file compcomm.c.

4.9.2.59 CoRenumber()

```
int CoRenumber (
            UBYTE * s )
```

Definition at line 2590 of file compcomm.c.

4.9.2.60 CoSum()

```
int CoSum (
            UBYTE * s )
```

Definition at line 2611 of file compcomm.c.

4.9.2.61 CoToTensor()

```
int CoToTensor (
            UBYTE * s )
```

Definition at line 2731 of file compcomm.c.

4.9.2.62 CoToVector()

```
int CoToVector (
            UBYTE * s )
```

Definition at line 2899 of file compcomm.c.

4.9.2.63 CoTrace4()

```
int CoTrace4 (
            UBYTE * s )
```

Definition at line 2969 of file compcomm.c.

4.9.2.64 CoTraceN()

```
int CoTraceN (
            UBYTE * s )
```

Definition at line 3056 of file compcomm.c.

4.9.2.65 CoChisholm()

```
int CoChisholm (
            UBYTE * s )
```

Definition at line 3116 of file compcomm.c.

4.9.2.66 DoChain()

```
int DoChain (
            UBYTE * s,
            int option )
```

Definition at line 3198 of file compcomm.c.

4.9.2.67 CoChainin()

```
int CoChainin (
            UBYTE * s )
```

Definition at line 3242 of file compcomm.c.

4.9.2.68 CoChainout()

```
int CoChainout (
            UBYTE * s )
```

Definition at line 3254 of file compcomm.c.

4.9.2.69 CoExit()

```
int CoExit (
            UBYTE * s )
```

Definition at line 3264 of file compcomm.c.

4.9.2.70 CoInParallel()

```
int CoInParallel (
            UBYTE * s )
```

Definition at line 3291 of file compcomm.c.

4.9.2.71 CoNotInParallel()

```
int CoNotInParallel (
            UBYTE * s )
```

Definition at line 3301 of file compcomm.c.

4.9.2.72 DoInParallel()

```
int DoInParallel (
            UBYTE * s,
            int par )
```

Definition at line 3316 of file compcomm.c.

4.9.2.73 CoInExpression()

```
int CoInExpression (
            UBYTE * s )
```

Definition at line 3385 of file compcomm.c.

4.9.2.74 CoEndInExpression()

```
int CoEndInExpression (
            UBYTE * s )
```

Definition at line 3438 of file compcomm.c.

4.9.2.75 CoSetExitFlag()

```
int CoSetExitFlag (
            UBYTE * s )
```

Definition at line 3464 of file compcomm.c.

4.9.2.76 CoTryReplace()

```
int CoTryReplace (
            UBYTE * p )
```

Definition at line 3478 of file compcomm.c.

4.9.2.77 CoModulus()

```
int CoModulus (
            UBYTE * inp )
```

Definition at line 3581 of file compcomm.c.

4.9.2.78 CoRepeat()

```
int CoRepeat (
            UBYTE * inp )
```

Definition at line 3719 of file compcomm.c.

4.9.2.79 CoEndRepeat()

```
int CoEndRepeat (
            UBYTE * inp )
```

Definition at line 3742 of file compcomm.c.

4.9.2.80 DoBrackets()

```
int DoBrackets (
            UBYTE * inp,
            int par )
```

Definition at line 3780 of file compcomm.c.

4.9.2.81 CoBracket()

```
int CoBracket (
            UBYTE * inp )
```

Definition at line 3907 of file compcomm.c.

4.9.2.82 CoAntiBracket()

```
int CoAntiBracket (
            UBYTE * inp )
```

Definition at line 3915 of file compcomm.c.

4.9.2.83 CoMultiBracket()

```
int CoMultiBracket (
            UBYTE * inp )
```

Definition at line 3926 of file compcomm.c.

4.9.2.84 CountComp()

```
WORD * CountComp (
            UBYTE * inp,
            WORD * to )
```

Definition at line 4058 of file compcomm.c.

4.9.2.85 CoIf()

```
int CoIf (
            UBYTE * inp )
```

Definition at line 4248 of file compcomm.c.

4.9.2.86 CoElse()

```
int CoElse (
            UBYTE * p )
```

Definition at line 4895 of file compcomm.c.

4.9.2.87 CoElseIf()

```
int CoElseIf (
            UBYTE * inp )
```

Definition at line 4922 of file compcomm.c.

4.9.2.88 CoEndIf()

```
int CoEndIf (
            UBYTE * inp )
```

Definition at line 4949 of file compcomm.c.

4.9.2.89 CoWhile()

```
int CoWhile (
            UBYTE * inp )
```

Definition at line 4994 of file compcomm.c.

4.9.2.90 CoEndWhile()

```
int CoEndWhile (
            UBYTE * inp )
```

Definition at line 5015 of file compcomm.c.

4.9.2.91 DoFindLoop()

```
int DoFindLoop (
            UBYTE * inp,
            int mode )
```

Definition at line 5043 of file compcomm.c.

4.9.2.92 CoFindLoop()

```
int CoFindLoop (
            UBYTE * inp )
```

Definition at line 5160 of file compcomm.c.

4.9.2.93 CoReplaceLoop()

```
int CoReplaceLoop (
            UBYTE * inp )
```

Definition at line 5168 of file compcomm.c.

4.9.2.94 CoFunPowers()

```
int CoFunPowers (
            UBYTE * inp )
```

Definition at line 5188 of file compcomm.c.

4.9.2.95 CoUnitTrace()

```
int CoUnitTrace (
            UBYTE * s )
```

Definition at line 5215 of file compcomm.c.

4.9.2.96 CoTerm()

```
int CoTerm (
            UBYTE * s )
```

Definition at line 5250 of file compcomm.c.

4.9.2.97 CoEndTerm()

```
int CoEndTerm (
            UBYTE * s )
```

Definition at line 5298 of file compcomm.c.

4.9.2.98 CoSort()

```
int CoSort (
            UBYTE * s )
```

Definition at line 5325 of file compcomm.c.

4.9.2.99 CoPolyFun()

```
int CoPolyFun (
            UBYTE * s )
```

Definition at line 5364 of file compcomm.c.

4.9.2.100 CoPolyRatFun()

```
int CoPolyRatFun (
            UBYTE * s )
```

Definition at line 5411 of file compcomm.c.

4.9.2.101 CoMerge()

```
int CoMerge (
            UBYTE * inp )
```

Definition at line 5581 of file compcomm.c.

4.9.2.102 CoStuffle()

```
int CoStuffle (
            UBYTE * inp )
```

Definition at line 5637 of file compcomm.c.

4.9.2.103 CoProcessBucket()

```
int CoProcessBucket (
            UBYTE * s )
```

Definition at line 5692 of file compcomm.c.

4.9.2.104 CoThreadBucket()

```
int CoThreadBucket (
            UBYTE * s )
```

Definition at line 5710 of file compcomm.c.

4.9.2.105 DoArgPlode()

```
int DoArgPlode (
            UBYTE * s,
            int par )
```

Definition at line 5740 of file compcomm.c.

4.9.2.106 CoArgExplode()

```
int CoArgExplode (
            UBYTE * s )
```

Definition at line 5796 of file compcomm.c.

4.9.2.107 CoArgImplode()

```
int CoArgImplode (
            UBYTE * s )
```

Definition at line 5803 of file compcomm.c.

4.9.2.108 CoClearTable()

```
int CoClearTable (
            UBYTE * s )
```

Definition at line 5810 of file compcomm.c.

4.9.2.109 CoDenominators()

```
int CoDenominators (
            UBYTE * s )
```

Definition at line 5892 of file compcomm.c.

4.9.2.110 CoDropCoefficient()

```
int CoDropCoefficient (
            UBYTE * s )
```

Definition at line 5921 of file compcomm.c.

4.9.2.111 CoDropSymbols()

```
int CoDropSymbols (
            UBYTE * s )
```

Definition at line 5935 of file compcomm.c.

4.9.2.112 CoToPolynomial()

```
int CoToPolynomial (
            UBYTE * inp )
```

Definition at line 5958 of file compcomm.c.

4.9.2.113 CoFromPolynomial()

```
int CoFromPolynomial (
            UBYTE * inp )
```

Definition at line 6033 of file compcomm.c.

4.9.2.114 CoArgToExtraSymbol()

```
int CoArgToExtraSymbol (
            UBYTE * s )
```

Definition at line 6059 of file compcomm.c.

4.9.2.115 CoExtraSymbols()

```
int CoExtraSymbols (
            UBYTE * inp )
```

Definition at line 6109 of file compcomm.c.

4.9.2.116 GetIfDollarFactor()

```
WORD * GetIfDollarFactor (
            UBYTE ** inp,
            WORD * w )
```

Definition at line 6182 of file compcomm.c.

4.9.2.117 GetDoParam()

```
UBYTE * GetDoParam (
            UBYTE * inp,
            WORD ** wp,
            int par )
```

Definition at line 6240 of file compcomm.c.

4.9.2.118 CoDo()

```
int CoDo (
            UBYTE * inp )
```

Definition at line 6317 of file compcomm.c.

4.9.2.119 CoEndDo()

```
int CoEndDo (
            UBYTE * inp )
```

Definition at line 6420 of file compcomm.c.

4.9.2.120 CoFactDollar()

```
int CoFactDollar (
            UBYTE * inp )
```

Definition at line 6451 of file compcomm.c.

4.9.2.121 CoFactorize()

```
int CoFactorize (
            UBYTE * s )
```

Definition at line 6480 of file compcomm.c.

4.9.2.122 CoNFactorize()

```
int CoNFactorize (
            UBYTE * s )
```

Definition at line 6487 of file compcomm.c.

4.9.2.123 CoUnFactorize()

```
int CoUnFactorize (
            UBYTE * s )
```

Definition at line 6494 of file compcomm.c.

4.9.2.124 CoNUnFactorize()

```
int CoNUnFactorize (
            UBYTE * s )
```

Definition at line 6501 of file compcomm.c.

4.9.2.125 DoFactorize()

```
int DoFactorize (
            UBYTE * s,
            int par )
```

Definition at line 6508 of file compcomm.c.

4.9.2.126 CoOptimizeOption()

```
int CoOptimizeOption (
            UBYTE * s )
```

Definition at line 6641 of file compcomm.c.

4.9.2.127 CoPutInside()

```
int CoPutInside (
            UBYTE * inp )
```

Definition at line 7077 of file compcomm.c.

4.9.2.128 CoAntiPutInside()

```
int CoAntiPutInside (
            UBYTE * inp )
```

Definition at line 7078 of file compcomm.c.

4.9.2.129 DoPutInside()

```
int DoPutInside (
            UBYTE * inp,
            int par )
```

Definition at line 7080 of file compcomm.c.

4.9.2.130 CoSwitch()

```
int CoSwitch (
            UBYTE * s )
```

Definition at line 7200 of file compcomm.c.

4.9.2.131 CoCase()

```
int CoCase (
            UBYTE * s )
```

Definition at line 7246 of file compcomm.c.

4.9.2.132 CoBreak()

```
int CoBreak (
            UBYTE * s )
```

Definition at line 7296 of file compcomm.c.

4.9.2.133 CoDefault()

```
int CoDefault (
            UBYTE * s )
```

Definition at line 7322 of file compcomm.c.

4.9.2.134 CoEndSwitch()

```
int CoEndSwitch (
            UBYTE * s )
```

Definition at line 7349 of file compcomm.c.

4.9.2.135 CoSetUserFlag()

```
int CoSetUserFlag (
            UBYTE * s )
```

Definition at line 7417 of file compcomm.c.

4.9.2.136 CoClearUserFlag()

```
int CoClearUserFlag (
            UBYTE * s )
```

Definition at line 7445 of file compcomm.c.

4.9.2.137 CoCreateAllLoops()

```
int CoCreateAllLoops (
            UBYTE * s )
```

Definition at line 7482 of file compcomm.c.

4.9.2.138 CoCreateAllPaths()

```
int CoCreateAllPaths (
            UBYTE * s )
```

Definition at line 7602 of file compcomm.c.

4.9.2.139 CoCreateAll()

```
int CoCreateAll (
            UBYTE * s )
```

Definition at line 7730 of file compcomm.c.

4.10 compcomm.c

Go to the documentation of this file.

```
00001
00010 /* #[ License : */
00011 /*
00012  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00013  *   When using this file you are requested to refer to the publication
00014  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00015  *   This is considered a matter of courtesy as the development was paid
00016  *   for by FOM the Dutch physics granting agency and we would like to
00017  *   be able to track its scientific use to convince FOM of its value
00018  *   for the community.
00019  *
00020  *   This file is part of FORM.
00021  *
00022  *   FORM is free software: you can redistribute it and/or modify it under the
00023  *   terms of the GNU General Public License as published by the Free Software
00024  *   Foundation, either version 3 of the License, or (at your option) any later
00025  *   version.
00026  *
00027  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00028  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00029  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00030  *   details.
00031  *
00032  *   You should have received a copy of the GNU General Public License along
00033  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00034  */
00035 /* #] License : */
00036 /*
00037     #[ includes :
00038 */
00039
00040 #include "form3.h"
00041 #include "comtool.h"
00042 #ifdef WITHFLOAT
00043 #include <gmp.h>
00044 #include <math.h>
00045 #endif
00046
00047 static KEYWORD formatoptions[] = {
00048      {"allfloat",        (TFUN)0,    ALLINTEGERDOUBLE,   0}
00049     ,{"c",               (TFUN)0,    CMODE,              0}
00050     ,{"doublefortran",   (TFUN)0,    DOUBLEFORTRANMODE,  0}
00051     ,{"float",           (TFUN)0,    0,                  2}
00052 #ifdef WITHFLOAT
00053     ,{"floatprecision",  (TFUN)0,    0,                  5}
00054 #endif
00055     ,{"fortran",         (TFUN)0,    FORTRANMODE,        0}
00056     ,{"fortran90",       (TFUN)0,    FORTRANMODE,        4}
00057     ,{"maple",           (TFUN)0,    MAPLEMODE,          0}
00058     ,{"mathematica",     (TFUN)0,    MATHEMATICAMODE,    0}
00059     ,{"normal",          (TFUN)0,    NORMALFORMAT,       1}
00060     ,{"nospaces",        (TFUN)0,    NOSPACEFORMAT,      3}
00061     ,{"pfortran",        (TFUN)0,    PFORTRANMODE,       0}
00062     ,{"quadfortran",     (TFUN)0,    QUADRUPLEFORTRANMODE,   0}
00063     ,{"quadruplefortran",   (TFUN)0,    QUADRUPLEFORTRANMODE,   0}
00064     ,{"rational",        (TFUN)0,    RATIONALMODE,       1}
00065     ,{"reduce",          (TFUN)0,    REDUCEMODE,         0}
00066     ,{"spaces",          (TFUN)0,    NORMALFORMAT,       3}
00067     ,{"vortran",         (TFUN)0,    VORTRANMODE,        0}
00068 };
00069
00070 static KEYWORD trace4options[] = {
00071      {"contract",     (TFUN)0,   CHISHOLM,       0          }
00072     ,{"nocontract",   (TFUN)0,   0,              CHISHOLM   }
00073     ,{"nosymmetrize",(TFUN)0,    0,              ALSOREVERSE}
00074     ,{"notrick",      (TFUN)0,   NOTRICK,        0          }
00075     ,{"symmetrize",   (TFUN)0,   ALSOREVERSE,    0          }
00076     ,{"trick",        (TFUN)0,   0,              NOTRICK    }
00077 };
00078
00079 static KEYWORD chisoptions[] = {
00080      {"nosymmetrize",(TFUN)0,    0,              ALSOREVERSE}
00081     ,{"symmetrize",   (TFUN)0,   ALSOREVERSE,    0          }
00082 };
00083
00084 static KEYWORDV writeoptions[] = {
00085      {"stats",          &(AC.StatsFlag),    1,      0}
00086     ,{"statistics",     &(AC.StatsFlag),    1,      0}
00087     ,{"shortstats",     &(AC.ShortStats),   1,      0}
00088     ,{"shortstatistics",&(AC.ShortStats),   1,      0}
00089     ,{"warnings",       &(AC.WarnFlag), 1,      0}
00090     ,{"allwarnings",    &(AC.WarnFlag), 2,      0}
```

```
00091     ,{"setup",           &(AC.SetupFlag),     1,      0}
00092     ,{"names",           &(AC.NamesFlag),     1,      0}
00093     ,{"allnames",        &(AC.NamesFlag),     2,      0}
00094     ,{"codes",           &(AC.CodesFlag),     1,      0}
00095     ,{"highfirst",       &(AC.SortType), SORTHIGHFIRST,      SORTLOWFIRST}
00096     ,{"lowfirst",        &(AC.SortType), SORTLOWFIRST,       SORTHIGHFIRST}
00097     ,{"powerfirst",      &(AC.SortType), SORTPOWERFIRST,     SORTHIGHFIRST}
00098     ,{"tokens",          &(AC.TokensWriteFlag),1,    0}
00099 };
00100
00101 static KEYWORDV onoffoptions[] = {
00102      {"compress",        &(AC.NoCompress),  0,  1}
00103     ,{"checkpoint",      &(AC.CheckpointFlag),  1,  0}
00104     ,{"insidefirst",     &(AC.insidefirst), 1,  0}
00105     ,{"propercount",     &(AC.BottomLevel), 1,  0}
00106     ,{"stats",           &(AC.StatsFlag),     1,  0}
00107     ,{"statistics",      &(AC.StatsFlag),     1,  0}
00108     ,{"shortstats",      &(AC.ShortStats),    1,  0}
00109     ,{"shortstatistics",&(AC.ShortStats),    1,  0}
00110     ,{"names",           &(AC.NamesFlag),     1,  0}
00111     ,{"allnames",        &(AC.NamesFlag),     2,  0}
00112     ,{"warnings",        &(AC.WarnFlag), 1,  0}
00113     ,{"allwarnings",     &(AC.WarnFlag), 2,  0}
00114     ,{"highfirst",       &(AC.SortType), SORTHIGHFIRST,  SORTLOWFIRST}
00115     ,{"lowfirst",        &(AC.SortType), SORTLOWFIRST,   SORTHIGHFIRST}
00116     ,{"powerfirst",      &(AC.SortType), SORTPOWERFIRST, SORTHIGHFIRST}
00117     ,{"setup",           &(AC.SetupFlag),     1,  0}
00118     ,{"codes",           &(AC.CodesFlag),     1,  0}
00119     ,{"tokens",          &(AC.TokensWriteFlag),1,0}
00120     ,{"properorder",     &(AC.properorderflag),1,0}
00121     ,{"threadloadbalancing",&(AC.ThreadBalancing),1,    0}
00122     ,{"threads",         &(AC.ThreadsFlag),1,    0}
00123     ,{"threadsortfilesynch",&(AC.ThreadSortFileSynch),1,  0}
00124     ,{"threadstats",     &(AC.ThreadStats),1,    0}
00125     ,{"finalstats",      &(AC.FinalStats),1, 0}
00126     ,{"fewerstats",      &(AC.ShortStatsMax),    10,     0}
00127     ,{"fewerstatistics",&(AC.ShortStatsMax),    10,     0}
00128     ,{"processstats",    &(AC.ProcessStats),1,   0}
00129     ,{"oldparallelstats",&(AC.OldParallelStats),1,0}
00130     ,{"parallel",        &(AC.parallelflag),PARALLELFLAG,NOPARALLEL_USER}
00131     ,{"nospacesinnumbers",&(AO.NoSpacesInNumbers),1,0}
00132     ,{"indentspace",     &(AO.IndentSpace),INDENTSPACE,0}
00133     ,{"totalsize",       &(AM.PrintTotalSize),   1,  0}
00134     ,{"flag",            (int *)&(AC.debugFlags),    1,  0}
00135     ,{"oldfactarg",      &(AC.OldFactArgFlag),   1,  0}
00136     ,{"memdebugflag",    &(AC.MemDebugFlag), 1,  0}
00137     ,{"oldgcd",          &(AC.OldGCDflag),   1,  0}
00138     ,{"oldprfsign",          &(AC.OldPRFSignFlag),   1,  0}
00139     ,{"innertest",       &(AC.InnerTest),  1,  0}
00140     ,{"wtimestats",      &(AC.WTimeStatsFlag),  1,  0}
00141     ,{"sortreallocate", &(AC.SortReallocateFlag), 1, 0}
00142     ,{"backtrace",       &(AC.PrintBacktraceFlag), 1, 0}
00143     ,{"flint",           &(AC.FlintPolyFlag), 1, 0}
00144     ,{"humanstats", &(AC.HumanStatsFlag), 1, 0}
00145     ,{"humanstatistics",    &(AC.HumanStatsFlag), 1, 0}
00146     ,{"grccverbose", &(AC.GrccVerbose), 1, 0}
00147     ,{"sortverbose", &(AC.SortVerbose), 1, 0}
00148 };
00149
00150 static WORD one = 1;
00151
00152 /*
00153     #] includes :
00154     #[ CoFormat :
00155 */
00156
00157 int CoFormat(UBYTE *s)
00158 {
00159     int error = 0, x;
00160     KEYWORD *key;
00161     UBYTE *ss;
00162     while ( *s == ' ' || *s == ',' ) s++;
00163     if ( *s == 0 ) {
00164         AC.OutputMode = 72;
00165         AC.OutputSpaces = NORMALFORMAT;
00166         return(error);
00167     }
00168 /*
00169     First the optimization level
00170 */
00171     if ( *s == 'O' || *s == 'o' ) {
00172         if ( ( FG.cTable[s[1]] == 1 ) ||
00173              ( s[1] == '=' && FG.cTable[s[2]] == 1 ) ) {
00174             s++; if ( *s == '=' ) s++;
00175             x = 0;
00176             while ( *s >= '0' && *s <= '9' ) x = 10*x + *s++ - '0';
00177             while ( *s == ',' ) s++;
```

```
00178             AO.OptimizationLevel = x;
00179             AO.Optimize.greedytimelimit = 0;
00180             AO.Optimize.mctstimelimit = 0;
00181             AO.Optimize.printstats = 0;
00182             AO.Optimize.debugflags = 0;
00183             AO.Optimize.schemeflags = 0;
00184             AO.Optimize.mctsdecaymode = 1; /* default is decreasing C_p with iteration number */
00185             if ( AO.inscheme ) {
00186                 M_free(AO.inscheme,"Horner input scheme");
00187                 AO.inscheme = 0; AO.schemenum = 0;
00188             }
00189             switch ( x ) {
00190                 case 0:
00191                     break;
00192                 case 1:
00193                     AO.Optimize.mctsconstant.fval = -1.0;
00194                     AO.Optimize.horner = O_OCCURRENCE;
00195                     AO.Optimize.hornerdirection = O_FORWARDORBACKWARD;
00196                     AO.Optimize.method = O_CSE;
00197                     break;
00198                 case 2:
00199                     AO.Optimize.horner = O_OCCURRENCE;
00200                     AO.Optimize.hornerdirection = O_FORWARDORBACKWARD;
00201                     AO.Optimize.method = O_GREEDY;
00202                     AO.Optimize.greedyminnum = 10;
00203                     AO.Optimize.greedymaxperc = 5;
00204                     break;
00205                 case 3:
00206                     AO.Optimize.mctsconstant.fval = 1.0;
00207                     AO.Optimize.horner = O_MCTS;
00208                     AO.Optimize.hornerdirection = O_FORWARDORBACKWARD;
00209                     AO.Optimize.method = O_GREEDY;
00210                     AO.Optimize.mctsnumexpand = 1000;
00211                     AO.Optimize.mctsnumkeep = 10;
00212                     AO.Optimize.mctsnumrepeat = 1;
00213                     AO.Optimize.greedyminnum = 10;
00214                     AO.Optimize.greedymaxperc = 5;
00215                     break;
00216                 case 4:
00217                     AO.Optimize.horner = O_SIMULATED_ANNEALING;
00218                     AO.Optimize.saIter = 1000;
00219                     AO.Optimize.saMaxT.fval = 2000;
00220                     AO.Optimize.saMinT.fval = 1;
00221                     break;
00222                 default:
00223                     error = 1;
00224                     MesPrint("&Illegal optimization specification in format statement");
00225                     break;
00226             }
00227             if ( error == 0 && *s != 0 && x > 0 ) return(CoOptimizeOption(s));
00228             return(error);
00229         }
00230 #ifdef EXPOPT
00231         { UBYTE c;
00232         ss = s;
00233         while ( FG.cTable[*s] == 0 ) s++;
00234         c = *s; *s = 0;
00235         if ( StrICont(ss,(UBYTE *)"optimize") == 0 ) {
00236             *s = c;
00237             while ( *s == ',' ) s++;
00238             if ( *s == '=' ) s++;
00239             AO.OptimizationLevel = 3;
00240             AO.Optimize.mctsconstant.fval = 1.0;
00241             AO.Optimize.horner = O_MCTS;
00242             AO.Optimize.hornerdirection = O_FORWARDORBACKWARD;
00243             AO.Optimize.method = O_GREEDY;
00244             AO.Optimize.mctstimelimit = 0;
00245             AO.Optimize.mctsnumexpand = 1000;
00246             AO.Optimize.mctsnumkeep = 10;
00247             AO.Optimize.mctsnumrepeat = 1;
00248             AO.Optimize.greedytimelimit = 0;
00249             AO.Optimize.greedyminnum = 10;
00250             AO.Optimize.greedymaxperc = 5;
00251             AO.Optimize.printstats = 0;
00252             AO.Optimize.debugflags = 0;
00253             AO.Optimize.schemeflags = 0;
00254             AO.Optimize.mctsdecaymode = 1;
00255             if ( AO.inscheme ) {
00256                 M_free(AO.inscheme,"Horner input scheme");
00257                 AO.inscheme = 0; AO.schemenum = 0;
00258             }
00259             return(CoOptimizeOption(s));
00260         }
00261         else {
00262             error = 1;
00263             MesPrint("&Illegal optimization specification in format statement");
00264             return(error);
```

```
00265         }
00266         }
00267 #endif
00268     }
00269     else if ( FG.cTable[*s] == 1 ) {
00270         x = 0;
00271         while ( FG.cTable[*s] == 1 ) x = 10*x + *s++ - '0';
00272         if ( x <= 0 || x >= MAXLINELENGTH ) {
00273             error = 1;
00274             MesPrint("&Illegal value for linesize: %d",x);
00275             x = 72;
00276         }
00277         if ( x < 39 ) {
00278             MesPrint(" ... Too small value for linesize corrected to 39");
00279             x = 39;
00280         }
00281         AO.DoubleFlag = 0;
00282 /*
00283         The next line resets the mode to normal. Because the special modes
00284         reset the line length we have a little problem with the special modes
00285         and customized line length. We try to improve by removing the next line
00286 */
00287 /*      AC.OutputMode = 0;  */
00288         AC.LineLength = x;
00289         if ( *s != 0 ) {
00290             error = 1;
00291             MesPrint("&Illegal linesize field in format statement");
00292         }
00293     }
00294     else {
00295         key = FindKeyWord(s,formatoptions,
00296             sizeof(formatoptions)/sizeof(KEYWORD));
00297         if ( key ) {
00298             if ( key->type == FORTRANMODE || key->type == PFORTRANMODE || key->type ==
      DOUBLEFORTRANMODE
00299                 || key->type == QUADRUPLEFORTRANMODE || key->type == VORTRANMODE ) {
00300                 if (AC.LineLength > 72) AC.LineLength = 72;
00301             }
00302             if ( key->flags == 0 ) {
00303                 if ( key->type == FORTRANMODE || key->type == PFORTRANMODE
00304                 || key->type == DOUBLEFORTRANMODE || key->type == ALLINTEGERDOUBLE
00305                 || key->type == QUADRUPLEFORTRANMODE || key->type == VORTRANMODE ) {
00306                     AC.IsFortran90 = ISNOTFORTRAN90;
00307                     if ( AC.Fortran90Kind ) {
00308                         M_free(AC.Fortran90Kind,"Fortran90 Kind");
00309                         AC.Fortran90Kind = 0;
00310                     }
00311                 }
00312                 if ( ( key->type == ALLINTEGERDOUBLE ) && AO.DoubleFlag != 0 ) {
00313                     AO.DoubleFlag |= 4;
00314                 }
00315                 else {
00316                     AO.DoubleFlag = 0;
00317                     AC.OutputMode = key->type & NODOUBLEMASK;
00318                     if ( ( key->type & DOUBLEPRECISIONFLAG ) != 0 ) {
00319                         AO.DoubleFlag = 1;
00320                     }
00321                     else if ( ( key->type & QUADRUPLEPRECISIONFLAG ) != 0 ) {
00322                         AO.DoubleFlag = 2;
00323                     }
00324                 }
00325             }
00326             else if ( key->flags == 1 ) {
00327                 AC.OutputMode = AC.OutNumberType = key->type;
00328             }
00329             else if ( key->flags == 2 ) {
00330                 while ( FG.cTable[*s] == 0 ) s++;
00331                 if ( *s == 0 ) AC.OutNumberType = 10;
00332                 else if ( *s == ',' ) {
00333                     s++;
00334                     x = 0;
00335                     while ( FG.cTable[*s] == 1 ) x = 10*x + *s++ - '0';
00336                     if ( *s != 0 ) {
00337                         error = 1;
00338                         MesPrint("&Illegal float format specifier");
00339                     }
00340                     else {
00341                         if ( x < 3 ) {
00342                             x = 3;
00343                             MesPrint("& ... float format value corrected to 3");
00344                         }
00345                         if ( x > 100 ) {
00346                             x = 100;
00347                             MesPrint("& ... float format value corrected to 100");
00348                         }
00349                         AC.OutNumberType = x;
00350                     }
```

```
00351                 }
00352             }
00353             else if ( key->flags == 3 ) {
00354                 AC.OutputSpaces = key->type;
00355             }
00356             else if ( key->flags == 4 ) {
00357                 AC.IsFortran90 = ISFORTRAN90;
00358                 if ( AC.Fortran90Kind ) {
00359                     M_free(AC.Fortran90Kind,"Fortran90 Kind");
00360                     AC.Fortran90Kind = 0;
00361                 }
00362                 while ( FG.cTable[*s] <= 1 ) s++;
00363                 if ( *s == ',' ) {
00364                     s++; ss = s;
00365                     while ( *ss && *ss != ',' ) ss++;
00366                     if ( *ss == ',' ) {
00367                         MesPrint("&No white space or comma's allowed in Fortran90 option: %s",s);
      error = 1;
00368                     }
00369                     else {
00370                         AC.Fortran90Kind = strDup1(s,"Fortran90 Kind");
00371                     }
00372                 }
00373                 AO.DoubleFlag = 0;
00374                 AC.OutputMode = key->type & NODOUBLEMASK;
00375             }
00376 #ifdef WITHFLOAT
00377             else if ( key->flags == 5 ) {
00378 /*
00379                 Syntax: Format FloatPrecision [precision];
00380                         Format FloatPrecision off;
00381 */
00382                 while ( FG.cTable[*s] == 0 ) s++;
00383                 while ( *s == ' ' || *s == '\t' || *s == ',' ) s++;
00384                 if ( *s == 0 ) {
00385                     AO.FloatPrec = 0;
00386                 }
00387                 else if ( tolower(*s) == 'o' && tolower(s[1]) == 'f'
00388                 && tolower(s[2]) == 'f' ) {
00389                     ss = s;
00390                     s += 3;
00391                     while ( *s == ' ' || *s == '\t' || *s == ',' ) s++;
00392                     if ( *s ) { s = ss; goto WrongOption; }
00393                     AO.FloatPrec = -1;
00394                 }
00395                 else if ( FG.cTable[*s] == 1 ) {
00396                     ss = s;
00397                     ParseNumber(AO.FloatPrec,s)
00398 /*
00399                     The precision can either be in digits or bits.
00400                     AO.FloatPrec is in digits.
00401 */
00402                     if ( tolower(*s) == 'd' ) { s++; }
00403                     else if ( tolower(*s) == 'b' ) { AO.FloatPrec = AO.FloatPrec*log10(2.0); s++; }
00404                     else { s = ss; goto WrongOption; }
00405                     while ( *s == ' ' || *s == '\t' || *s == ',' ) s++;
00406                     if ( *s ) { s = ss; goto WrongOption; }
00407                 }
00408                 else {
00409 WrongOption:        MesPrint("&Illegal option in Format FloatPrecision: %s",s);
00410                     error = 1;
00411                 }
00412             }
00413 #endif
00414         }
00415         else if ( ( *s == 'c' || *s == 'C' ) && ( FG.cTable[s[1]] == 1 ) ) {
00416             UBYTE *ss = s+1;
00417             WORD x = 0;
00418             while ( *ss >= '0' && *ss <= '9' ) x = 10*x + *ss++ - '0';
00419             if ( *ss != 0 ) goto Unknown;
00420             AC.OutputMode = CMODE;
00421             AC.Cnumpows = x;
00422         }
00423         else {
00424 Unknown:    MesPrint("&Unknown option: %s",s); error = 1;
00425         }
00426     }
00427     return(error);
00428 }
00429
00430 /*
00431     #] CoFormat :
00432     #[ CoCollect :
00433
00434     Collect,functionname
00435 */
00436
```

```
00437 int CoCollect(UBYTE *s)
00438 {
00439 /*  --------------change 17-feb-2003 Added percentage */
00440     WORD numfun;
00441     int type,x = 0;
00442     UBYTE *t = SkipAName(s), *t1, *t2;
00443     AC.AltCollectFun = 0;
00444     if ( t == 0 ) goto syntaxerror;
00445     t1 = t; while ( *t1 == ',' || *t1 == ' ' || *t1 == '\t' ) t1++;
00446     *t = 0; t = t1;
00447     if ( *t1 && ( FG.cTable[*t1] == 0 || *t1 == '[' ) ) {
00448         t2 = SkipAName(t1);
00449         if ( t2 == 0 ) goto syntaxerror;
00450         t = t2;
00451         while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
00452         *t2 = 0;
00453     }
00454     else t1 = 0;
00455     if ( *t && FG.cTable[*t] == 1 ) {
00456         while ( *t >= '0' && *t <= '9' ) x = 10*x + *t++ - '0';
00457         if ( x > 100 ) x = 100;
00458         while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
00459         if ( *t ) goto syntaxerror;
00460     }
00461     else {
00462         if ( *t ) goto syntaxerror;
00463         x = 100;
00464     }
00465     if ( ( ( type = GetName(AC.varnames,s,&numfun,WITHAUTO) ) != CFUNCTION )
00466     || ( functions[numfun].spec != 0 ) ) {
00467         MesPrint("&%s should be a regular function",s);
00468         if ( type < 0 ) {
00469             if ( GetName(AC.exprnames,s,&numfun,NOAUTO) == NAMENOTFOUND )
00470                 AddFunction(s,0,0,0,0,0,-1,-1);
00471         }
00472         return(1);
00473     }
00474     AC.CollectFun = numfun+FUNCTION;
00475     AC.CollectPercentage = (WORD)x;
00476     if ( t1 ) {
00477         if ( ( ( type = GetName(AC.varnames,t1,&numfun,WITHAUTO) ) != CFUNCTION )
00478         || ( functions[numfun].spec != 0 ) ) {
00479             MesPrint("&%s should be a regular function",t1);
00480             if ( type < 0 ) {
00481                 if ( GetName(AC.exprnames,t1,&numfun,NOAUTO) == NAMENOTFOUND )
00482                     AddFunction(t1,0,0,0,0,0,-1,-1);
00483             }
00484             return(1);
00485         }
00486         AC.AltCollectFun = numfun+FUNCTION;
00487     }
00488     return(0);
00489 syntaxerror:
00490     MesPrint("&Collect statement needs one or two functions (and a percentage) for its argument(s)");
00491     return(1);
00492 }
00493
00494 /*
00495     #] CoCollect :
00496     #[ setonoff :
00497 */
00498
00499 int setonoff(UBYTE *s, int *flag, int onvalue, int offvalue)
00500 {
00501     if ( StrICmp(s,(UBYTE *)"on") == 0 ) *flag = onvalue;
00502     else if ( StrICmp(s,(UBYTE *)"off") == 0 ) *flag = offvalue;
00503     else {
00504         MesPrint("&Unknown option: %s, on or off expected",s);
00505         return(1);
00506     }
00507     return(0);
00508 }
00509
00510 /*
00511     #] setonoff :
00512     #[ CoCompress :
00513 */
00514
00515 int CoCompress(UBYTE *s)
00516 {
00517     GETIDENTITY
00518     UBYTE *t, c;
00519     if ( StrICmp(s,(UBYTE *)"on") == 0 ) {
00520         AC.NoCompress = 0;
00521         AR.gzipCompress = 0;
00522     }
00523     else if ( StrICmp(s,(UBYTE *)"off") == 0 ) {
```

```
00524         AC.NoCompress = 1;
00525         AR.gzipCompress = 0;
00526     }
00527     else {
00528         t = s; while ( FG.cTable[*t] <= 1 ) t++;
00529         c = *t; *t = 0;
00530         if ( StrICmp(s,(UBYTE *)"gzip") == 0 ) {
00531 #ifndef WITHZLIB
00532             Warning("gzip compression not supported on this platform");
00533 #endif
00534             s = t; *s = c;
00535             if ( *s == 0 ) {
00536                 AR.gzipCompress = GZIPDEFAULT;  /* Normally should be 6 */
00537                 return(0);
00538             }
00539             while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00540             t = s;
00541             if ( FG.cTable[*s] == 1 ) {
00542                 AR.gzipCompress = *s - '0';
00543                 s++;
00544                 while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00545                 if ( *s == 0 ) return(0);
00546             }
00547             MesPrint("&Unknown gzip option: %s, a digit was expected",t);
00548             return(1);
00549 
00550         }
00551         else {
00552             MesPrint("&Unknown option: %s, on, off or gzip expected",s);
00553             return(1);
00554         }
00555     }
00556     return(0);
00557 }
00558 
00559 /*
00560     #] CoCompress :
00561     #[ CoFlags :
00562 */
00563 
00564 int CoFlags(UBYTE *s,int value)
00565 {
00566     int i, error = 0;
00567     if ( *s != ',' ) {
00568         MesPrint("&Proper syntax is: On/Off Flag,number[s];");
00569         error = 1;
00570     }
00571     while ( *s == ',' ) {
00572         do { s++; } while ( *s == ',' );
00573         i = 0;
00574         if ( FG.cTable[*s] != 1 ) {
00575             MesPrint("&Proper syntax is: On/Off Flag,number[s];");
00576             error = 1;
00577             break;
00578         }
00579         while ( FG.cTable[*s] == 1 ) { i = 10*i + *s++ - '0'; }
00580         if ( i <= 0 || i > MAXFLAGS ) {
00581             MesPrint("&The number of a flag in On/Off Flag should be in the range
    0-%d",(int)MAXFLAGS);
00582             error = 1;
00583             break;
00584         }
00585         AC.debugFlags[i] = value;
00586     }
00587     if ( *s ) {
00588         MesPrint("&Proper syntax is: On/Off Flag,number[s];");
00589         error = 1;
00590     }
00591     return(error);
00592 }
00593 
00594 /*
00595     #] CoFlags :
00596     #[ CoOff :
00597 */
00598 
00599 int CoOff(UBYTE *s)
00600 {
00601     GETIDENTITY
00602     UBYTE *t, c;
00603     int i, num = sizeof(onoffoptions)/sizeof(KEYWORD);
00604     for (;;) {
00605         while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00606         if ( *s == 0 ) return(0);
00607         if ( chartype[*s] != 0 ) {
00608             MesPrint("&Illegal character or option encountered in OFF statement");
00609             return(-1);
```

```
00610         }
00611         t = s;  while ( chartype[*s] == 0 ) s++;
00612         c = *s; *s = 0;
00613         for ( i = 0; i < num; i++ ) {
00614             if ( StrICont(t,(UBYTE *)(onoffoptions[i].name)) == 0 ) break;
00615         }
00616         if ( i >= num ) {
00617             MesPrint("&Unrecognized option in OFF statement: %s",t);
00618             *s = c; return(-1);
00619         }
00620         else if ( StrICont(t,(UBYTE *)"compress") == 0 ) {
00621             AR.gzipCompress = 0;
00622         }
00623         else if ( StrICont(t,(UBYTE *)"checkpoint") == 0 ) {
00624             PrintDeprecation("the checkpoint mechanism", "issues/626");
00625             AC.CheckpointInterval = 0;
00626             if ( AC.CheckpointRunBefore ) { free(AC.CheckpointRunBefore); AC.CheckpointRunBefore =
      NULL; }
00627             if ( AC.CheckpointRunAfter ) { free(AC.CheckpointRunAfter); AC.CheckpointRunAfter = NULL;
     }
00628             if ( AC.NoShowInput == 0 ) MesPrint("Checkpoints deactivated.");
00629         }
00630         else if ( StrICont(t,(UBYTE *)"threads") == 0 ) {
00631             AS.MultiThreaded = 0;
00632         }
00633         else if ( StrICont(t,(UBYTE *)"flag") == 0 ) {
00634             *s = c;
00635             return(CoFlags(s,0));
00636         }
00637         else if ( StrICont(t,(UBYTE *)"innertest") == 0 ) {
00638             *s = c;
00639             AC.InnerTest = 0;
00640             if ( AC.TestValue ) {
00641                 M_free(AC.TestValue,"InnerTest");
00642                 AC.TestValue = 0;
00643             }
00644         }
00645         else if ( StrICont(t,(UBYTE *)"sortreallocate") == 0 ) {
00646             if ( AC.SortReallocateFlag == 2 ) {
00647                 /* The flag has been set by #sortreallocate, and it was off before. Leave it as 2,
00648                    so that the reallocation still happens in the current module. It will be turned
00649                    off after the reallocation is done. */
00650                 return(0);
00651             }
00652         }
00653         *s = c;
00654         *onoffoptions[i].var = onoffoptions[i].flags;
00655         AR.SortType = AC.SortType;
00656         AC.mparallelflag = AC.parallelflag | AM.hparallelflag;
00657     }
00658 }
00659
00660 /*
00661     #] CoOff :
00662     #[ CoOn :
00663 */
00664
00665 int CoOn(UBYTE *s)
00666 {
00667     GETIDENTITY
00668     UBYTE *t, c;
00669     int i, num = sizeof(onoffoptions)/sizeof(KEYWORD);
00670     LONG interval;
00671     for (;;) {
00672         while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00673         if ( *s == 0 ) return(0);
00674         if ( chartype[*s] != 0 ) {
00675             MesPrint("&Illegal character or option encountered in ON statement");
00676             return(-1);
00677         }
00678         t = s;  while ( chartype[*s] == 0 ) s++;
00679         c = *s; *s = 0;
00680         for ( i = 0; i < num; i++ ) {
00681             if ( StrICont(t,(UBYTE *)(onoffoptions[i].name)) == 0 ) break;
00682         }
00683         if ( i >= num ) {
00684             MesPrint("&Unrecognized option in ON statement: %s",t);
00685             *s = c; return(-1);
00686         }
00687         if ( StrICont(t,(UBYTE *)"backtrace") == 0 ) {
00688 #ifndef ENABLE_BACKTRACE
00689             Warning("backtrace not supported on this platform");
00690 #endif
00691         }
00692         else if ( StrICont(t,(UBYTE *)"compress") == 0 ) {
00693             AR.gzipCompress = 0;
00694             *s = c;
```

```
00695             while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00696             if ( *s ) {
00697                 t = s;
00698                 while ( FG.cTable[*s] <= 1 ) s++;
00699                 c = *s; *s = 0;
00700                 if ( StrICmp(t,(UBYTE *)"gzip") == 0 ) {
00701 #ifndef WITHZLIB
00702                     Warning("gzip compression not supported on this platform");
00703 #endif
00704 #ifdef WITHZSTD
00705                     /* If gzip is specified, turn off zstd compression. zlib still goes via the
    wrapper. */
00706                     ZWRAP_useZSTDcompression(0);
00707 #endif
00708                 }
00709                 else if ( StrICmp(t,(UBYTE *)"zstd") == 0 ) {
00710 #ifdef WITHZSTD
00711                     ZWRAP_useZSTDcompression(1);
00712 #else
00713                     Warning("zstd compression not supported on this platform");
00714 #endif
00715                 }
00716                 else {
00717                     MesPrint("&Unrecognized option in ON compress statement: %s",t);
00718                     return(-1);
00719                 }
00720                 /* Whether we are using zlib or zstd, accept and use a compression level. */
00721                 *s = c;
00722                 while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00723                 if ( FG.cTable[*s] == 1 ) {
00724                     AR.gzipCompress = *s++ - '0';
00725                     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00726                     if ( *s ) {
00727                         MesPrint("&Unrecognized option in ON compress gzip/zstd statement: %s",t);
00728                         return(-1);
00729                     }
00730                 }
00731                 else if ( *s == 0 ) {
00732                     AR.gzipCompress = GZIPDEFAULT;
00733                 }
00734                 else {
00735                     MesPrint("&Unrecognized option in ON compress gzip/zstd statement: %s, single
    digit expected",t);
00736                     return(-1);
00737                 }
00738             }
00739         }
00740         else if ( StrICont(t,(UBYTE *)"checkpoint") == 0 ) {
00741             PrintDeprecation("the checkpoint mechanism", "issues/626");
00742             AC.CheckpointInterval = 0;
00743             if ( AC.CheckpointRunBefore ) { free(AC.CheckpointRunBefore); AC.CheckpointRunBefore =
    NULL; }
00744             if ( AC.CheckpointRunAfter ) { free(AC.CheckpointRunAfter); AC.CheckpointRunAfter = NULL;
    }
00745             *s = c;
00746             while ( *s ) {
00747                 while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00748                 if ( FG.cTable[*s] == 1 ) {
00749                     interval = 0;
00750                     t = s;
00751                     do { interval = 10*interval + *s++ - '0'; } while ( FG.cTable[*s] == 1 );
00752                     if ( *s == 's' || *s == 'S' ) {
00753                         s++;
00754                     }
00755                     else if ( *s == 'm' || *s == 'M' ) {
00756                         interval *= 60; s++;
00757                     }
00758                     else if ( *s == 'h' || *s == 'H' ) {
00759                         interval *= 3600; s++;
00760                     }
00761                     else if ( *s == 'd' || *s == 'D' ) {
00762                         interval *= 86400; s++;
00763                     }
00764                     if ( *s != ',' && FG.cTable[*s] != 6 && FG.cTable[*s] != 10 ) {
00765                         MesPrint("&Unrecognized time interval in ON Checkpoint statement: %s", t);
00766                         return(-1);
00767                     }
00768                     AC.CheckpointInterval = interval * 100; /* in 1/100 of seconds */
00769                 }
00770                 else if ( FG.cTable[*s] == 0 ) {
00771                     int type;
00772                     t = s;
00773                     while ( FG.cTable[*s] == 0 ) s++;
00774                     c = *s; *s = 0;
00775                     if ( StrICmp(t,(UBYTE *)"run") == 0 ) {
00776                         type = 3;
00777                     }
```

```
00778                   else if ( StrICmp(t,(UBYTE *)"runafter") == 0 ) {
00779                       type = 2;
00780                   }
00781                   else if ( StrICmp(t,(UBYTE *)"runbefore") == 0 ) {
00782                       type = 1;
00783                   }
00784                   else {
00785                       MesPrint("&Unrecognized option in ON Checkpoint statement: %s", t);
00786                       *s = c; return(-1);
00787                   }
00788                   *s = c;
00789                   if ( *s != '=' && FG.cTable[*(s+1)] != 9 ) {
00790                       MesPrint("&Unrecognized option in ON Checkpoint statement: %s", t);
00791                       return(-1);
00792                   }
00793                   ++s;
00794                   t = ++s;
00795                   while ( *s ) {
00796                       if ( FG.cTable[*s] == 9 ) {
00797                           c = *s; *s = 0;
00798                           if ( type & 1 ) {
00799                               if ( AC.CheckpointRunBefore ) {
00800                                   free(AC.CheckpointRunBefore); AC.CheckpointRunBefore = NULL;
00801                               }
00802                               if ( s-t > 0 ) {
00803                                   AC.CheckpointRunBefore = Malloc1(s-t+1, "AC.CheckpointRunBefore");
00804                                   StrCopy(t, (UBYTE*)AC.CheckpointRunBefore);
00805                               }
00806                           }
00807                           if ( type & 2 ) {
00808                               if ( AC.CheckpointRunAfter ) {
00809                                   free(AC.CheckpointRunAfter); AC.CheckpointRunAfter = NULL;
00810                               }
00811                               if ( s-t > 0 ) {
00812                                   AC.CheckpointRunAfter = Malloc1(s-t+1, "AC.CheckpointRunAfter");
00813                                   StrCopy(t, (UBYTE*)AC.CheckpointRunAfter);
00814                               }
00815                           }
00816                           *s = c;
00817                           break;
00818                       }
00819                       ++s;
00820                   }
00821                   if ( FG.cTable[*s] != 9 ) {
00822                       MesPrint("&Unrecognized option in ON Checkpoint statement: %s", t);
00823                       return(-1);
00824                   }
00825                   ++s;
00826               }
00827           }
00828 /*
00829           if ( AC.NoShowInput == 0 ) {
00830               MesPrint("Checkpoints activated.");
00831               if ( AC.CheckpointInterval ) {
00832                   MesPrint("-> Minimum saving interval: %l seconds.", AC.CheckpointInterval/100);
00833               }
00834               else {
00835                   MesPrint("-> No minimum saving interval given. Saving after EVERY module.");
00836               }
00837               if ( AC.CheckpointRunBefore ) {
00838                   MesPrint("-> Calling script \"%s\" before saving.", AC.CheckpointRunBefore);
00839               }
00840               if ( AC.CheckpointRunAfter ) {
00841                   MesPrint("-> Calling script \"%s\" after saving.", AC.CheckpointRunAfter);
00842               }
00843           }
00844 */
00845       }
00846       else if ( StrICont(t,(UBYTE *)"indentspace") == 0 ) {
00847           *s = c;
00848           while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00849           if ( *s ) {
00850               i = 0;
00851               while ( FG.cTable[*s] == 1 ) { i = 10*i + *s++ - '0'; }
00852               if ( *s ) {
00853                   MesPrint("&Unrecognized option in ON IndentSpace statement: %s",t);
00854                   return(-1);
00855               }
00856               if ( i > 40 ) {
00857                   Warning("IndentSpace parameter adjusted to 40");
00858                   i = 40;
00859               }
00860               AO.IndentSpace = i;
00861           }
00862           else {
00863               AO.IndentSpace = AM.ggIndentSpace;
00864           }
```

```
00865             return(0);
00866         }
00867         else if ( ( StrICont(t,(UBYTE *)"fewerstats") == 0 ) ||
00868                   ( StrICont(t,(UBYTE *)"fewerstatistics") == 0 ) ) {
00869             *s = c;
00870             while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00871             if ( *s ) {
00872                 i = 0;
00873                 while ( FG.cTable[*s] == 1 ) { i = 10*i + *s++ - '0'; }
00874                 if ( *s ) {
00875                     MesPrint("&Unrecognized option in ON FewerStatistics statement: %s",t);
00876                     return(-1);
00877                 }
00878                 if ( i > AM.S0->MaxPatches ) {
00879                     if ( AC.WarnFlag )
00880                     MesPrint("&Warning: FewerStatistics parameter greater than MaxPatches(=%d).
      Adjusted to %d"
00881                     ,AM.S0->MaxPatches,(AM.S0->MaxPatches+1)/2);
00882                     i = (AM.S0->MaxPatches+1)/2;
00883                 }
00884                 AC.ShortStatsMax = i;
00885             }
00886             else {
00887                 AC.ShortStatsMax = 10; /* default value */
00888             }
00889             return(0);
00890         }
00891         else if ( StrICont(t,(UBYTE *)"threads") == 0 ) {
00892             if ( AM.totalnumberofthreads > 1 ) AS.MultiThreaded = 1;
00893         }
00894         else if ( StrICont(t,(UBYTE *)"flag") == 0 ) {
00895             *s = c;
00896             return(CoFlags(s,1));
00897         }
00898         else if ( StrICont(t,(UBYTE *)"innertest") == 0 ) {
00899             UBYTE *t;
00900             *s = c;
00901             while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00902             if ( *s ) {
00903                 t = s; while ( *t ) t++;
00904                 while ( t[-1] == ' ' || t[-1] == '\t' ) t--;
00905                 c = *t; *t = 0;
00906                 if ( AC.TestValue ) M_free(AC.TestValue,"InnerTest");
00907                 AC.TestValue = strDup1(s,"InnerTest");
00908                 *t = c;
00909                 s = t;
00910                 while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00911             }
00912             else {
00913                 if ( AC.TestValue ) {
00914                     M_free(AC.TestValue,"InnerTest");
00915                     AC.TestValue = 0;
00916                 }
00917             }
00918         }
00919         else if ( StrICont(t,(UBYTE *)"flint") == 0 ) {
00920 #ifndef WITHFLINT
00921             MesPrint("&Warning: FORM was not built with FLINT support.");
00922             MesPrint("Statement has no effect.");
00923 #endif
00924         }
00925         else { *s = c; }
00926         *onoffoptions[i].var = onoffoptions[i].type;
00927         AR.SortType = AC.SortType;
00928         AC.mparallelflag = AC.parallelflag | AM.hparallelflag;
00929     }
00930 }
00931
00932 /*
00933     #] CoOn :
00934     #[ CoInsideFirst :
00935 */
00936
00937 int CoInsideFirst(UBYTE *s) { return(setonoff(s,&AC.insidefirst,1,0)); }
00938
00939 /*
00940     #] CoInsideFirst :
00941     #[ CoProperCount :
00942 */
00943
00944 int CoProperCount(UBYTE *s) { return(setonoff(s,&AC.BottomLevel,1,0)); }
00945
00946 /*
00947     #] CoProperCount :
00948     #[ CoDelete :
00949 */
00950
```

```
00951 int CoDelete(UBYTE *s)
00952 {
00953     int error = 0;
00954     if ( StrICmp(s,(UBYTE *)"storage") == 0 ) {
00955         if ( DeleteStore(1) < 0 ) {
00956 /* INTERNAL_ERROR_EXCL_START */
00957             MesPrint("!>Cannot restart storage file");
00958             error = 1;
00959 /* INTERNAL_ERROR_EXCL_STOP */
00960         }
00961     }
00962     else {
00963         UBYTE *t = s, c;
00964         while ( *t && *t != ',' && *t != '>' ) t++;
00965         c = *t; *t = 0;
00966         if ( ( StrICmp(s,(UBYTE *)"extrasymbols") == 0 )
00967         || ( StrICmp(s,(UBYTE *)"extrasymbol") == 0 ) ) {
00968             WORD x = 0;
00969 /*
00970             Either deletes all extra symbols or deletes above a given number
00971 */
00972             *t = c; s = t;
00973             if ( *s == '>' ) {
00974                 s++;
00975                 if ( FG.cTable[*s] != 1 ) goto unknown;
00976                 while ( *s <= '9' && *s >= '0' ) x = 10*x + *s++ - '0';
00977                 if ( *s ) goto unknown;
00978             }
00979             else if ( *s ) goto unknown;
00980             if ( x < AM.gnumextrasym ) x = AM.gnumextrasym;
00981             PruneExtraSymbols(x);
00982         }
00983         else {
00984             *t = c;
00985 unknown:
00986             MesPrint("&Unknown option: %s",s);
00987             error = 1;
00988         }
00989     }
00990     return(error);
00991 }
00992
00993 /*
00994     #] CoDelete :
00995     #[ CoKeep :
00996 */
00997
00998 int CoKeep(UBYTE *s)
00999 {
01000     if ( StrICmp(s,(UBYTE *)"brackets") == 0 ) AC.ComDefer = 1;
01001     else { MesPrint("&Unknown option: '%s'",s); return(1); }
01002     return(0);
01003 }
01004
01005 /*
01006     #] CoKeep :
01007     #[ CoFixIndex :
01008 */
01009
01010 int CoFixIndex(UBYTE *s)
01011 {
01012     int x, y, error = 0;
01013     while ( *s ) {
01014         if ( FG.cTable[*s] != 1 ) {
01015 proper:     MesPrint("&Proper syntax is: FixIndex,number:value[,number,value];");
01016             return(1);
01017         }
01018         ParseNumber(x,s)
01019         if ( *s != ':' ) goto proper;
01020         s++;
01021         if ( *s != '-' && *s != '+' && FG.cTable[*s] != 1 ) goto proper;
01022         ParseSignedNumber(y,s)
01023         if ( *s && *s != ',' ) goto proper;
01024         while ( *s == ',' ) s++;
01025         if ( x >= AM.OffsetIndex ) {
01026             MesPrint("&Fixed index out of allowed range. Change ConstIndex in setup file?");
01027             MesPrint("&Current value of ConstIndex = %d",AM.OffsetIndex-1);
01028             error = 1;
01029         }
01030         if ( y != (int)((WORD)y) ) {
01031             MesPrint("&Value of d_(%d,%d) outside range for this computer",x,x);
01032             error = 1;
01033         }
01034         if ( error == 0 ) AC.FixIndices[x] = y;
01035     }
01036     return(error);
01037 }
```

```
01038
01039 /*
01040     #] CoFixIndex :
01041     #[ CoMetric :
01042 */
01043
01044 int CoMetric(UBYTE *s)
01045 { DUMMYUSE(s); MesPrint("&The metric statement does not do anything yet"); return(1); }
01046
01047 /*
01048     #] CoMetric :
01049     #[ DoPrint :
01050 */
01051
01052 int DoPrint(UBYTE *s, int par)
01053 {
01054     int i, error = 0, numdol = 0, type;
01055     WORD handle = -1;
01056     UBYTE *name, c, *t;
01057     EXPRESSIONS e;
01058     WORD numexpr, tofile = 0, *w, par2 = 0;
01059     CBUF *C = cbuf + AC.cbufnum;
01060     while ( *s == ',' ) s++;
01061     if ( ( *s == '+' || *s == '-' ) && ( s[1] == 'f' || s[1] == 'F' ) ) {
01062         t = s + 2; while ( *t == ' ' || *t == ',' ) t++;
01063         if ( *t == '"' ) {
01064             if ( *s == '+' ) { tofile = 1; handle = AC.LogHandle; }
01065             s = t;
01066         }
01067     }
01068     else if ( *s == '<' ) {
01069         UBYTE *filename;
01070         s++; filename = s;
01071         while ( *s && *s != '>' ) s++;
01072         if ( *s == 0 ) {
01073             MesPrint("&Improper filename in print statement");
01074             return(1);
01075         }
01076         *s++ = 0;
01077         tofile = 1;
01078         if ( ( handle = GetChannel((char *)filename,1) ) < 0 ) return(1);
01079         SKIPBLANKS(s) if ( *s == ',' ) s++; SKIPBLANKS(s)
01080         if ( *s == '+' && ( s[1] == 's' || s[1] == 'S' ) ) {
01081             s += 2;
01082             par2 |= PRINTONETERM;
01083             if ( *s == 's' || *s == 'S' ) {
01084                 s++;
01085                 par2 |= PRINTONEFUNCTION;
01086                 if ( *s == 's' || *s == 'S' ) {
01087                     s++;
01088                     par2 |= PRINTALL;
01089                 }
01090             }
01091             SKIPBLANKS(s) if ( *s == ',' ) s++; SKIPBLANKS(s)
01092         }
01093     }
01094     if ( par == PRINTON && *s == '"' ) {
01095         WORD code[3];
01096         if ( tofile == 1 ) code[0] = TYPEFPRINT;
01097         else code[0] = TYPEPRINT;
01098         code[1] = handle;
01099         code[2] = par2;
01100         s++; name = s;
01101         while ( *s && *s != '"' ) {
01102             if ( *s == '\\' ) s++;
01103             if ( *s == '%' && s[1] == '$' ) numdol++;
01104             s++;
01105         }
01106         if ( *s != '"' ) {
01107             MesPrint("&String in print statement should be enclosed in \"");
01108             return(1);
01109         }
01110         *s = 0;
01111         AddComString(3,code,name,1);
01112         *s++ = '"';
01113         while ( *s == ',' ) {
01114             s++;
01115             if ( *s == '$' ) {
01116                 s++; name = s; while ( FG.cTable[*s] <= 1 ) s++;
01117                 c = *s; *s = 0;
01118                 type = GetName(AC.dollarnames,name,&numexpr,NOAUTO);
01119                 if ( type == NAMENOTFOUND ) {
01120                     MesPrint("&$ variable %s not (yet) defined",name);
01121                     error = 1;
01122                 }
01123                 else {
01124                     C->lhs[C->numlhs][1] += 2;
```

```
01125                       *(C->Pointer)++ = DOLLAREXPRESSION;
01126                       *(C->Pointer)++ = numexpr;
01127                       numdol--;
01128                   }
01129               }
01130               else {
01131                   MesPrint("&Illegal object in print statement");
01132                   error = 1;
01133                   return(error);
01134               }
01135               *s = c;
01136               if ( c == '[' ) {
01137                   w = C->Pointer;
01138                   s++;
01139                   s = GetDoParam(s,&(C->Pointer),-1);
01140                   if ( s == 0 ) return(1);
01141                   if ( *s != ']' ) {
01142                       MesPrint("&unmatched [] in $ factor");
01143                       return(1);
01144                   }
01145                   C->lhs[C->numlhs][1] += C->Pointer - w;
01146                   s++;
01147               }
01148           }
01149           if ( *s != 0 ) {
01150               MesPrint("&Illegal object in print statement");
01151               error = 1;
01152           }
01153           if ( numdol > 0 ) {
01154               MesPrint("&More $ variables asked for than provided");
01155               error = 1;
01156           }
01157           *(C->Pointer)++ = 0;
01158           return(error);
01159       }
01160       if ( *s == 0 ) {    /* All active expressions */
01161   AllExpr:
01162           for ( e = Expressions, i = NumExpressions; i > 0; i--, e++ ) {
01163               if ( e->status == LOCALEXPRESSION || e->status ==
01164               GLOBALEXPRESSION || e->status == UNHIDELEXPRESSION
01165               || e->status == UNHIDEGEXPRESSION ) e->printflag = par;
01166           }
01167           return(error);
01168       }
01169       while ( *s ) {
01170           if ( *s == '+' ) {
01171               s++;
01172               if ( tolower(*s) == 'f' ) par |= PRINTLFILE;
01173               else if ( tolower(*s) == 's' ) {
01174                   if ( tolower(s[1]) == 's' ) {
01175                       if ( tolower(s[2]) == 's' ) {
01176                           par |= PRINTONEFUNCTION | PRINTONETERM | PRINTALL;
01177                           s++;
01178                       }
01179                       else if ( ( par & 3 ) < 2 ) par |= PRINTONEFUNCTION | PRINTONETERM;
01180                       s++;
01181                   }
01182                   else {
01183                       if ( ( par & 3 ) < 2 ) par |= PRINTONETERM;
01184                   }
01185               }
01186               else {
01187   illeg:              MesPrint("&Illegal option in (n)print statement");
01188                   error = 1;
01189               }
01190               s++;
01191               if ( *s == 0 ) goto AllExpr;
01192           }
01193           else if ( *s == '-' ) {
01194               s++;
01195               if ( tolower(*s) == 'f' ) par &= ~PRINTLFILE;
01196               else if ( tolower(*s) == 's' ) {
01197                   if ( tolower(s[1]) == 's' ) {
01198                       if ( tolower(s[2]) == 's' ) {
01199                           par &= ~PRINTALL;
01200                           s++;
01201                       }
01202                       else if ( ( par & 3 ) < 2 ) {
01203                           par &= ~PRINTONEFUNCTION;
01204                           par &= ~PRINTALL;
01205                       }
01206                       s++;
01207                   }
01208                   else {
01209                       if ( ( par & 3 ) < 2 ) {
01210                           par &= ~PRINTONETERM;
01211                           par &= ~PRINTONEFUNCTION;
```

```
01212                       par &= ~PRINTALL;
01213                   }
01214               }
01215           }
01216           else goto illeg;
01217           s++;
01218           if ( *s == 0 ) goto AllExpr;
01219       }
01220       else if ( FG.cTable[*s] == 0 || *s == '[' ) {
01221           name = s;
01222           if ( ( s = SkipAName(s) ) == 0 ) {
01223               MesPrint("&Improper name in (n)print statement");
01224               return(1);
01225           }
01226           c = *s; *s = 0;
01227           if ( ( GetName(AC.exprnames,name,&numexpr,NOAUTO) == CEXPRESSION )
01228           && ( Expressions[numexpr].status == LOCALEXPRESSION
01229           || Expressions[numexpr].status == GLOBALEXPRESSION ) ) {
01230 FoundExpr:;
01231               if ( c == '[' && s[1] == ']' ) {
01232                   Expressions[numexpr].printflag = par | PRINTCONTENTS;
01233                   *s++ = c; c = *++s;
01234               }
01235               else
01236                   Expressions[numexpr].printflag = par;
01237           }
01238           else if ( GetLastExprName(name,&numexpr)
01239           && ( Expressions[numexpr].status == LOCALEXPRESSION
01240           || Expressions[numexpr].status == GLOBALEXPRESSION
01241           || Expressions[numexpr].status == UNHIDELEXPRESSION
01242           || Expressions[numexpr].status == UNHIDEGEXPRESSION
01243           ) ) {
01244               goto FoundExpr;
01245           }
01246           else {
01247               MesPrint("&%s is not the name of an active expression",name);
01248               error = 1;
01249           }
01250           *s++ = c;
01251           if ( c == 0 ) return(0);
01252           if ( c == '-' || c == '+' ) s--;
01253       }
01254       else if ( *s == ',' ) s++;
01255       else {
01256           MesPrint("&Illegal object in (n)print statement");
01257           return(1);
01258       }
01259   }
01260   return(0);
01261 }
01262
01263 /*
01264     #] DoPrint :
01265     #[ CoPrint :
01266 */
01267
01268 int CoPrint(UBYTE *s) { return(DoPrint(s,PRINTON)); }
01269
01270 /*
01271     #] CoPrint :
01272     #[ CoPrintB :
01273 */
01274
01275 int CoPrintB(UBYTE *s) { return(DoPrint(s,PRINTCONTENT)); }
01276
01277 /*
01278     #] CoPrintB :
01279     #[ CoNPrint :
01280 */
01281
01282 int CoNPrint(UBYTE *s) { return(DoPrint(s,PRINTOFF)); }
01283
01284 /*
01285     #] CoNPrint :
01286     #[ CoPushHide :
01287 */
01288
01289 int CoPushHide(UBYTE *s)
01290 {
01291     GETIDENTITY
01292     WORD *ScratchBuf;
01293     int i;
01294     if ( AR.Fscr[2].PObuffer == 0 ) {
01295         ScratchBuf = (WORD *)Malloc1(AM.HideSize*sizeof(WORD),"hidesize");
01296         AR.Fscr[2].POsize = AM.HideSize * sizeof(WORD);
01297         AR.Fscr[2].POfull = AR.Fscr[2].POfill = AR.Fscr[2].PObuffer = ScratchBuf;
01298         AR.Fscr[2].POstop = AR.Fscr[2].PObuffer + AM.HideSize;
```

```
01299         PUTZERO(AR.Fscr[2].POposition);
01300     }
01301     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
01302     AC.HideLevel += 2;
01303     if ( *s ) {
01304         MesPrint("&PushHide statement should have no arguments");
01305         return(-1);
01306     }
01307     for ( i = 0; i < NumExpressions; i++ ) {
01308         switch ( Expressions[i].status ) {
01309             case DROPLEXPRESSION:
01310             case SKIPLEXPRESSION:
01311             case LOCALEXPRESSION:
01312                 Expressions[i].status = HIDELEXPRESSION;
01313                 Expressions[i].hidelevel = AC.HideLevel-1;
01314                 break;
01315             case DROPGEXPRESSION:
01316             case SKIPGEXPRESSION:
01317             case GLOBALEXPRESSION:
01318                 Expressions[i].status = HIDEGEXPRESSION;
01319                 Expressions[i].hidelevel = AC.HideLevel-1;
01320                 break;
01321             default:
01322                 break;
01323         }
01324     }
01325     return(0);
01326 }
01327
01328 /*
01329     #] CoPushHide :
01330     #[ CoPopHide :
01331 */
01332
01333 int CoPopHide(UBYTE *s)
01334 {
01335     int i;
01336     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
01337     if ( AC.HideLevel <= 0 ) {
01338         MesPrint("&PopHide statement without corresponding PushHide statement");
01339         return(-1);
01340     }
01341     AC.HideLevel -= 2;
01342     if ( *s ) {
01343         MesPrint("&PopHide statement should have no arguments");
01344         return(-1);
01345     }
01346     for ( i = 0; i < NumExpressions; i++ ) {
01347         switch ( Expressions[i].status ) {
01348             case HIDDENLEXPRESSION:
01349                 if ( Expressions[i].hidelevel > AC.HideLevel )
01350                     Expressions[i].status = UNHIDELEXPRESSION;
01351                 break;
01352             case HIDDENGEXPRESSION:
01353                 if ( Expressions[i].hidelevel > AC.HideLevel )
01354                     Expressions[i].status = UNHIDEGEXPRESSION;
01355                 break;
01356             default:
01357                 break;
01358         }
01359     }
01360     return(0);
01361 }
01362
01363 /*
01364     #] CoPopHide :
01365     #[ SetExprCases :
01366 */
01367
01368 int SetExprCases(int par, int setunset, int val)
01369 {
01370     switch ( par ) {
01371         case SKIP:
01372             switch ( val ) {
01373                 case SKIPLEXPRESSION:
01374                     if ( !setunset ) val = LOCALEXPRESSION;
01375                     break;
01376                 case SKIPGEXPRESSION:
01377                     if ( !setunset ) val = GLOBALEXPRESSION;
01378                     break;
01379                 case LOCALEXPRESSION:
01380                     if ( setunset ) val = SKIPLEXPRESSION;
01381                     break;
01382                 case GLOBALEXPRESSION:
01383                     if ( setunset ) val = SKIPGEXPRESSION;
01384                     break;
01385                 case INTOHIDEGEXPRESSION:
```

```
01386                 case INTOHIDELEXPRESSION:
01387                 default:
01388                     break;
01389             }
01390             break;
01391         case DROP:
01392             switch ( val ) {
01393                 case SKIPLEXPRESSION:
01394                 case LOCALEXPRESSION:
01395                 case HIDELEXPRESSION:
01396                     if ( setunset ) val = DROPLEXPRESSION;
01397                     break;
01398                 case DROPLEXPRESSION:
01399                     if ( !setunset ) val = LOCALEXPRESSION;
01400                     break;
01401                 case SKIPGEXPRESSION:
01402                 case GLOBALEXPRESSION:
01403                 case HIDEGEXPRESSION:
01404                     if ( setunset ) val = DROPGEXPRESSION;
01405                     break;
01406                 case DROPGEXPRESSION:
01407                     if ( !setunset ) val = GLOBALEXPRESSION;
01408                     break;
01409                 case HIDDENLEXPRESSION:
01410                 case UNHIDELEXPRESSION:
01411                     if ( setunset ) val = DROPHLEXPRESSION;
01412                     break;
01413                 case HIDDENGEXPRESSION:
01414                 case UNHIDEGEXPRESSION:
01415                     if ( setunset ) val = DROPHGEXPRESSION;
01416                     break;
01417                 case DROPHLEXPRESSION:
01418                     if ( !setunset ) val = HIDDENLEXPRESSION;
01419                     break;
01420                 case DROPHGEXPRESSION:
01421                     if ( !setunset ) val = HIDDENGEXPRESSION;
01422                     break;
01423                 case INTOHIDEGEXPRESSION:
01424                 case INTOHIDELEXPRESSION:
01425                 default:
01426                     break;
01427             }
01428             break;
01429         case HIDE:
01430             switch ( val ) {
01431                 case DROPLEXPRESSION:
01432                 case SKIPLEXPRESSION:
01433                 case LOCALEXPRESSION:
01434                     if ( setunset ) val = HIDELEXPRESSION;
01435                     break;
01436                 case HIDELEXPRESSION:
01437                     if ( !setunset ) val = LOCALEXPRESSION;
01438                     break;
01439                 case DROPGEXPRESSION:
01440                 case SKIPGEXPRESSION:
01441                 case GLOBALEXPRESSION:
01442                     if ( setunset ) val = HIDEGEXPRESSION;
01443                     break;
01444                 case HIDEGEXPRESSION:
01445                     if ( !setunset ) val = GLOBALEXPRESSION;
01446                     break;
01447                 case INTOHIDEGEXPRESSION:
01448                 case INTOHIDELEXPRESSION:
01449                 default:
01450                     break;
01451             }
01452             break;
01453         case UNHIDE:
01454             switch ( val ) {
01455                 case HIDDENLEXPRESSION:
01456                 case DROPHLEXPRESSION:
01457                     if ( setunset ) val = UNHIDELEXPRESSION;
01458                     break;
01459                 case UNHIDELEXPRESSION:
01460                     if ( !setunset ) val = HIDDENLEXPRESSION;
01461                     break;
01462                 case HIDDENGEXPRESSION:
01463                 case DROPHGEXPRESSION:
01464                     if ( setunset ) val = UNHIDEGEXPRESSION;
01465                     break;
01466                 case UNHIDEGEXPRESSION:
01467                     if ( !setunset ) val = HIDDENGEXPRESSION;
01468                     break;
01469                 case INTOHIDEGEXPRESSION:
01470                 case INTOHIDELEXPRESSION:
01471                 default:
01472                     break;
```

```
01473             }
01474             break;
01475         case INTOHIDE:
01476             switch ( val ) {
01477                 case HIDDENLEXPRESSION:
01478                 case HIDDENGEXPRESSION:
01479                     MesPrint("&Expression is already hidden");
01480                     return(-1);
01481                 case DROPHLEXPRESSION:
01482                 case DROPHGEXPRESSION:
01483                 case UNHIDELEXPRESSION:
01484                 case UNHIDEGEXPRESSION:
01485                     if ( setunset ) {
01486                         MesPrint("&Cannot unhide/drop and put intohide expression in the same
      module");
01487                         return(-1);
01488                     }
01489                     break;
01490                 case LOCALEXPRESSION:
01491                 case DROPLEXPRESSION:
01492                 case SKIPLEXPRESSION:
01493                 case HIDELEXPRESSION:
01494                     if ( setunset ) val = INTOHIDELEXPRESSION;
01495                     break;
01496                 case GLOBALEXPRESSION:
01497                 case DROPGEXPRESSION:
01498                 case SKIPGEXPRESSION:
01499                 case HIDEGEXPRESSION:
01500                     if ( setunset ) val = INTOHIDEGEXPRESSION;
01501                     break;
01502                 case INTOHIDELEXPRESSION:
01503                     if ( !setunset ) val = LOCALEXPRESSION;
01504                     break;
01505                 case INTOHIDEGEXPRESSION:
01506                     if ( !setunset ) val = GLOBALEXPRESSION;
01507                     break;
01508                 default:
01509                     break;
01510             }
01511             break;
01512         default:
01513             break;
01514     }
01515     return(val);
01516 }
01517
01518 /*
01519     #] SetExprCases :
01520     #[ SetExpr :
01521 */
01522
01523 int SetExpr(UBYTE *s, int setunset, int par)
01524 {
01525     WORD *w, numexpr;
01526     int error = 0, i;
01527     UBYTE *name, c;
01528     if ( *s == 0 ) {
01529         for ( i = 0; i < NumExpressions; i++ ) {
01530             w = &(Expressions[i].status);
01531             *w = SetExprCases(par,setunset,*w);
01532             if ( *w < 0 ) error = 1;
01533             if ( ( par == HIDE || par == INTOHIDE ) && setunset == 1 )
01534                 Expressions[i].hidelevel = AC.HideLevel;
01535         }
01536         return(0);
01537     }
01538     while ( *s ) {
01539         if ( *s == ',' ) { s++; continue; }
01540         if ( *s == '0' ) { s++; continue; }
01541         name = s;
01542         if ( ( s = SkipAName(s) ) == 0 ) {
01543             MesPrint("&Improper name for an expression: '%s'",name);
01544             return(1);
01545         }
01546         c = *s; *s = 0;
01547         if ( GetName(AC.exprnames,name,&numexpr,NOAUTO) == CEXPRESSION ) {
01548             w = &(Expressions[numexpr].status);
01549             *w = SetExprCases(par,setunset,*w);
01550             if ( *w < 0 ) error = 1;
01551             if ( ( par == HIDE || par == INTOHIDE ) && setunset == 1 )
01552                 Expressions[numexpr].hidelevel = AC.HideLevel;
01553         }
01554         else if ( GetName(AC.varnames,name,&numexpr,NOAUTO) != NAMENOTFOUND ) {
01555             MesPrint("&%s is not an expression",name);
01556             error = 1;
01557         }
01558         *s = c;
```

```
01559     }
01560     return(error);
01561 }
01562
01563 /*
01564     #] SetExpr :
01565     #[ CoDrop :
01566 */
01567
01568 int CoDrop(UBYTE *s) { return(SetExpr(s,1,DROP)); }
01569
01570 /*
01571     #] CoDrop :
01572     #[ CoNoDrop :
01573 */
01574
01575 int CoNoDrop(UBYTE *s) { return(SetExpr(s,0,DROP)); }
01576
01577 /*
01578     #] CoNoDrop :
01579     #[ CoSkip :
01580 */
01581
01582 int CoSkip(UBYTE *s) { return(SetExpr(s,1,SKIP)); }
01583
01584 /*
01585     #] CoSkip :
01586     #[ CoNoSkip :
01587 */
01588
01589 int CoNoSkip(UBYTE *s) { return(SetExpr(s,0,SKIP)); }
01590
01591 /*
01592     #] CoNoSkip :
01593     #[ CoHide :
01594 */
01595
01596 int CoHide(UBYTE *inp) {
01597     GETIDENTITY
01598     WORD *ScratchBuf;
01599     if ( AR.Fscr[2].PObuffer == 0 ) {
01600         ScratchBuf = (WORD *)Malloc1(AM.HideSize*sizeof(WORD),"hidesize");
01601         AR.Fscr[2].POsize = AM.HideSize * sizeof(WORD);
01602         AR.Fscr[2].POfull = AR.Fscr[2].POfill = AR.Fscr[2].POstart = ScratchBuf;
01603         AR.Fscr[2].POstop = AR.Fscr[2].POstart + AM.HideSize;
01604         PUTZERO(AR.Fscr[2].POposition);
01605     }
01606     return(SetExpr(inp,1,HIDE));
01607 }
01608
01609 /*
01610     #] CoHide :
01611     #[ CoIntoHide :
01612 */
01613
01614 int CoIntoHide(UBYTE *inp) {
01615     GETIDENTITY
01616     WORD *ScratchBuf;
01617     if ( AR.Fscr[2].PObuffer == 0 ) {
01618         ScratchBuf = (WORD *)Malloc1(AM.HideSize*sizeof(WORD),"hidesize");
01619         AR.Fscr[2].POsize = AM.HideSize * sizeof(WORD);
01620         AR.Fscr[2].POfull = AR.Fscr[2].POfill = AR.Fscr[2].POstart = ScratchBuf;
01621         AR.Fscr[2].POstop = AR.Fscr[2].POstart + AM.HideSize;
01622         PUTZERO(AR.Fscr[2].POposition);
01623     }
01624     return(SetExpr(inp,1,INTOHIDE));
01625 }
01626
01627 /*
01628     #] CoIntoHide :
01629     #[ CoNoIntoHide :
01630 */
01631
01632 int CoNoIntoHide(UBYTE *inp) { return(SetExpr(inp,0,INTOHIDE)); }
01633
01634 /*
01635     #] CoNoIntoHide :
01636     #[ CoNoHide :
01637 */
01638
01639 int CoNoHide(UBYTE *inp) { return(SetExpr(inp,0,HIDE)); }
01640
01641 /*
01642     #] CoNoHide :
01643     #[ CoUnHide :
01644 */
01645
```

```
01646 int CoUnHide(UBYTE *inp) { return(SetExpr(inp,1,UNHIDE)); }
01647
01648 /*
01649     #] CoUnHide :
01650     #[ CoNoUnHide :
01651 */
01652
01653 int CoNoUnHide(UBYTE *inp) { return(SetExpr(inp,0,UNHIDE)); }
01654
01655 /*
01656     #] CoNoUnHide :
01657     #[ AddToCom :
01658 */
01659
01660 void AddToCom(int n, WORD *array)
01661 {
01662     CBUF *C = cbuf+AC.cbufnum;
01663 #ifdef COMPBUFDEBUG
01664     MesPrint("  %a",n,array);
01665 #endif
01666     while ( C->Pointer+n >= C->Top ) DoubleCbuffer(AC.cbufnum,C->Pointer,18);
01667     while ( --n >= 0 ) *(C->Pointer)++ = *array++;
01668 }
01669
01670 /*
01671     #] AddToCom :
01672     #[ AddComString :
01673 */
01674
01675 int AddComString(int n, WORD *array, UBYTE *thestring, int par)
01676 {
01677     CBUF *C = cbuf+AC.cbufnum;
01678     UBYTE *s = thestring, *w;
01679 #ifdef COMPBUFDEBUG
01680     WORD *cc;
01681     UBYTE *ww;
01682 #endif
01683     int i, numchars = 0, size, zeroes;
01684     while ( *s ) {
01685         if ( *s == '\\' ) s++;
01686         else if ( par == 1 &&
01687         ( ( *s == '%' && s[1] != 't' && s[1] != 'T' && s[1] != '$' &&
01688          s[1] != 'w' && s[1] != 'W' && s[1] != 'r' && s[1] != 0 ) || *s == '#'
01689         || *s == '@' || *s == '&' ) ) {
01690             numchars++;
01691         }
01692         s++; numchars++;
01693     }
01694     AddLHS(AC.cbufnum);
01695     size = numchars/sizeof(WORD)+1;
01696     while ( C->Pointer+size+n+2 >= C->Top ) DoubleCbuffer(AC.cbufnum,C->Pointer,19);
01697 #ifdef COMPBUFDEBUG
01698     cc = C->Pointer;
01699 #endif
01700     *(C->Pointer)++ = array[0];
01701     *(C->Pointer)++ = size+n+2;
01702     for ( i = 1; i < n; i++ ) *(C->Pointer)++ = array[i];
01703     *(C->Pointer)++ = size;
01704 #ifdef COMPBUFDEBUG
01705     ww =
01706 #endif
01707     w = (UBYTE *)(C->Pointer);
01708     zeroes = size*sizeof(WORD)-numchars;
01709     s = thestring;
01710     while ( *s ) {
01711         if ( *s == '\\' ) s++;
01712         else if ( par == 1 && ( ( *s == '%' &&
01713         s[1] != 't' && s[1] != 'T' && s[1] != '$' &&
01714         s[1] != 'w' && s[1] != 'W' && s[1] != 'r' && s[1] != 0 ) || *s == '#'
01715         || *s == '@' || *s == '&' ) ) {
01716             *w++ = '%';
01717         }
01718         *w++ = *s++;
01719     }
01720     while ( --zeroes >= 0 ) *w++ = 0;
01721     C->Pointer += size;
01722 #ifdef COMPBUFDEBUG
01723     MesPrint("LH: %a",size+1+n,cc);
01724     MesPrint("        %s",thestring);
01725 #endif
01726     return(0);
01727 }
01728
01729 /*
01730     #] AddComString :
01731     #[ Add2ComStrings :
01732 */
```

```
01733
01734 int Add2ComStrings(int n, WORD *array, UBYTE *string1, UBYTE *string2)
01735 {
01736     CBUF *C = cbuf+AC.cbufnum;
01737     UBYTE *s1 = string1, *s2 = string2, *w;
01738     int i, num1chars = 0, num2chars = 0, size1, size2, zeroes1, zeroes2;
01739     AddLHS(AC.cbufnum);
01740     while ( *s1 ) { s1++; num1chars++; }
01741     size1 = num1chars/sizeof(WORD)+1;
01742     if ( s2 ) {
01743         while ( *s2 ) { s2++; num2chars++; }
01744         size2 = num2chars/sizeof(WORD)+1;
01745     }
01746     else size2 = 0;
01747     while ( C->Pointer+size1+size2+n+3 >= C->Top ) DoubleCbuffer(AC.cbufnum,C->Pointer,20);
01748     *(C->Pointer)++ = array[0];
01749     *(C->Pointer)++ = size1+size2+n+3;
01750     for ( i = 1; i < n; i++ ) *(C->Pointer)++ = array[i];
01751     *(C->Pointer)++ = size1;
01752     w = (UBYTE *)(C->Pointer);
01753     zeroes1 = size1*sizeof(WORD)-num1chars;
01754     s1 = string1;
01755     while ( *s1 ) { *w++ = *s1++; }
01756     while ( --zeroes1 >= 0 ) *w++ = 0;
01757     C->Pointer += size1;
01758     *(C->Pointer)++ = size2;
01759     if ( size2 ) {
01760         w = (UBYTE *)(C->Pointer);
01761         zeroes2 = size2*sizeof(WORD)-num2chars;
01762         s2 = string2;
01763         while ( *s2 ) { *w++ = *s2++; }
01764         while ( --zeroes2 >= 0 ) *w++ = 0;
01765         C->Pointer += size2;
01766     }
01767     return(0);
01768 }
01769
01770 /*
01771     #] Add2ComStrings :
01772     #[ CoDiscard :
01773 */
01774
01775 int CoDiscard(UBYTE *s)
01776 {
01777     if ( *s == 0 ) {
01778         Add2Com(TYPEDISCARD)
01779         return(0);
01780     }
01781     MesPrint("&Illegal argument in discard statement: '%s'",s);
01782     return(1);
01783 }
01784
01785 /*
01786     #] CoDiscard :
01787     #[ CoContract :
01788
01789     Syntax:
01790         Contract
01791         Contract:#
01792         Contract #
01793         Contract:#,#
01794 */
01795
01796 static WORD ccarray[5] = { TYPEOPERATION,5,CONTRACT,0,0 };
01797
01798 int CoContract(UBYTE *s)
01799 {
01800     int x;
01801     if ( *s == ':' ) {
01802         s++;
01803         ParseNumber(x,s)
01804         if ( *s != ',' && *s ) {
01805 proper:     MesPrint("&Illegal number in contract statement");
01806             return(1);
01807         }
01808         if ( *s ) s++;
01809         ccarray[4] = x;
01810     }
01811     else ccarray[4] = 0;
01812     if ( FG.cTable[*s] == 1 ) {
01813         ParseNumber(x,s)
01814         if ( *s ) goto proper;
01815         ccarray[3] = x;
01816     }
01817     else if ( *s ) goto proper;
01818     else ccarray[3] = -1;
01819     return(AddNtoL(5,ccarray));
```

```
01820 }
01821
01822 /*
01823     #] CoContract :
01824     #[ CoGoTo :
01825 */
01826
01827 int CoGoTo(UBYTE *inp)
01828 {
01829     UBYTE *s = inp;
01830     int x;
01831     while ( FG.cTable[*s] <= 1 ) s++;
01832     if ( *s ) {
01833         MesPrint("&Label should be an alpha-numeric string");
01834         return(1);
01835     }
01836     x = GetLabel(inp);
01837     Add3Com(TYPEGOTO,x);
01838     return(0);
01839 }
01840
01841 /*
01842     #] CoGoTo :
01843     #[ CoLabel :
01844 */
01845
01846 int CoLabel(UBYTE *inp)
01847 {
01848     UBYTE *s = inp;
01849     int x;
01850     while ( FG.cTable[*s] <= 1 ) s++;
01851     if ( *s ) {
01852         MesPrint("&Label should be an alpha-numeric string");
01853         return(1);
01854     }
01855     x = GetLabel(inp);
01856     if ( AC.Labels[x] >= 0 ) {
01857         MesPrint("&Label %s defined more than once",AC.LabelNames[x]);
01858         return(1);
01859     }
01860     AC.Labels[x] = cbuf[AC.cbufnum].numlhs;
01861     return(0);
01862 }
01863
01864 /*
01865     #] CoLabel :
01866     #[ DoArgument :
01867
01868     Layout:
01869         par,full size,numlhs(+1),par,scale
01870         scale is for normalize
01871 */
01872
01873 int DoArgument(UBYTE *s, int par)
01874 {
01875     GETIDENTITY
01876     UBYTE *name, *t, *v, c;
01877     WORD *oldworkpointer = AT.WorkPointer, *w, *ww, number, *scale;
01878     int error = 0, zeroflag, type, x;
01879     AC.lhdollarflag = 0;
01880     while ( *s == ',' ) s++;
01881     w = AT.WorkPointer;
01882     *w++ = par;
01883     w++;
01884     switch ( par ) {
01885         case TYPEARG:
01886             if ( AC.arglevel >= MAXNEST ) {
01887                 MesPrint("@Nesting of argument statements more than %d levels"
01888                 ,(WORD)MAXNEST);
01889                 return(-1);
01890             }
01891             AC.argsumcheck[AC.arglevel] = NestingChecksum();
01892             AC.argstack[AC.arglevel] = cbuf[AC.cbufnum].Pointer
01893                                      - cbuf[AC.cbufnum].Buffer + 2;
01894             AC.arglevel++;
01895             *w++ = cbuf[AC.cbufnum].numlhs;
01896             break;
01897         case TYPENORM:
01898         case TYPENORM4:
01899         case TYPESPLITARG:
01900         case TYPESPLITFIRSTARG:
01901         case TYPESPLITLASTARG:
01902         case TYPEFACTARG:
01903         case TYPEARGTOEXTRASYMBOL:
01904             *w++ = cbuf[AC.cbufnum].numlhs+1;
01905             break;
01906     }
```

```
01907     *w++ = par;
01908     scale = w;
01909     *w++ = 1;
01910     *w++ = 0;
01911     if ( *s == '^' ) {
01912         s++; ParseSignedNumber(x,s)
01913         while ( *s == ',' ) s++;
01914         *scale = x;
01915     }
01916     if ( *s == '(' ) {
01917         t = s+1; SKIPBRA3(s)    /* We did check the brackets already */
01918         if ( par == TYPEARG ) {
01919             MesPrint("&Illegal () entry in argument statement");
01920             error = 1; s++; goto skipbracks;
01921         }
01922         else if ( par == TYPESPLITFIRSTARG ) {
01923             MesPrint("&Illegal () entry in splitfirstarg statement");
01924             error = 1; s++; goto skipbracks;
01925         }
01926         else if ( par == TYPESPLITLASTARG ) {
01927             MesPrint("&Illegal () entry in splitlastarg statement");
01928             error = 1; s++; goto skipbracks;
01929         }
01930         v = t;
01931         while ( v < s ) {
01932             if ( *v == '?' ) {
01933                 MesPrint("&Wildcarding not allowed in this type of statement");
01934                 error = 1; break;
01935             }
01936             v++;
01937         }
01938         v = s++;
01939         if ( *t == '(' && v[-1] == ')' ) {
01940             t++; v--;
01941             if ( par == TYPESPLITARG ) oldworkpointer[0] = TYPESPLITARG2;
01942             else if ( par == TYPEFACTARG ) oldworkpointer[0] = TYPEFACTARG2;
01943             else if ( par == TYPENORM4 ) oldworkpointer[0] = TYPENORM4;
01944             else if ( par == TYPENORM ) {
01945                 if ( *t == '-' ) { oldworkpointer[0] = TYPENORM3; t++; }
01946                 else             { oldworkpointer[0] = TYPENORM2; *scale = 0; }
01947             }
01948         }
01949         if ( error == 0 ) {
01950             CBUF *C = cbuf+AC.cbufnum;
01951             WORD oldnumrhs = C->numrhs, oldnumlhs = C->numlhs;
01952             WORD prototype[SUBEXPSIZE+40]; /* Up to 10 nested sums! */
01953             WORD *m, *mm;
01954             int i, retcode;
01955             LONG oldpointer = C->Pointer - C->Buffer;
01956             *v = 0;
01957             prototype[0] = SUBEXPRESSION;
01958             prototype[1] = SUBEXPSIZE;
01959             prototype[2] = C->numrhs+1;
01960             prototype[3] = 1;
01961             prototype[4] = AC.cbufnum;
01962             AT.WorkPointer += TYPEARGHEADSIZE+1;
01963             AddLHS(AC.cbufnum);
01964             if ( ( retcode = CompileAlgebra(t,LHSIDE,prototype) ) < 0 )
01965                 error = 1;
01966             else {
01967                 prototype[2] = retcode;
01968                 ww = C->lhs[retcode];
01969                 AC.lhdollarflag = 0;
01970                 if ( *ww == 0 ) {
01971                     *w++ = -2; *w++ = 0;
01972                 }
01973                 else if ( ww[ww[0]] != 0 ) {
01974                     MesPrint("&There should be only one term between ()");
01975                     error = 1;
01976                 }
01977                 else if ( NewSort(BHEAD0) ) { if ( !error ) error = 1; }
01978                 else if ( NewSort(BHEAD0) ) {
01979                     LowerSortLevel();
01980                     if ( !error ) error = 1;
01981                 }
01982                 else {
01983                     AN.RepPoint = AT.RepCount + 1;
01984                     m = AT.WorkPointer;
01985                     mm = ww; i = *mm;
01986                     while ( --i >= 0 ) *m++ = *mm++;
01987                     mm = AT.WorkPointer; AT.WorkPointer = m;
01988                     AR.Cnumlhs = C->numlhs;
01989                     if ( Generator(BHEAD mm,C->numlhs) ) {
01990                         LowerSortLevel(); error = 1;
01991                     }
01992                     else if ( EndSort(BHEAD mm,0) < 0 ) {
01993                         error = 1;
```

```
01994                       AT.WorkPointer = mm;
01995                   }
01996                   else if ( *mm == 0 ) {
01997                       *w++ = -2; *w++ = 0;
01998                       AT.WorkPointer = mm;
01999                   }
02000                   else if ( mm[mm[0]] != 0 ) {
02001                       error = 1;
02002                       AT.WorkPointer = mm;
02003                   }
02004                   else {
02005                       AT.WorkPointer = mm;
02006                       m = mm+*mm;
02007                       if ( par == TYPEFACTARG ) {
02008                           if ( *mm != ABS(m[-1])+1 ) {
02009                               *mm -= ABS(m[-1]);  /* Strip coefficient */
02010                           }
02011                           mm[-1] = -*mm-1; w += *mm+1;
02012                       }
02013                       else {
02014                           *mm -= ABS(m[-1]);  /* Strip coefficient */
02015 /*
02016                           if ( *mm == 1 ) { *w++ = -2; *w++ = 0; }
02017                           else
02018 */
02019                           { mm[-1] = -*mm-1; w += *mm+1; }
02020                       }
02021                       oldworkpointer[1] = w - oldworkpointer;
02022                   }
02023                   LowerSortLevel();
02024               }
02025               oldworkpointer[5] = AC.lhdollarflag;
02026           }
02027           *v = ')';
02028           C->numrhs = oldnumrhs;
02029           C->numlhs = oldnumlhs;
02030           C->Pointer = C->Buffer + oldpointer;
02031       }
02032   }
02033 skipbracks:
02034   if ( *s == 0 ) { *w++ = 0; *w++ = 2; *w++ = 1; }
02035   else {
02036       do {
02037           if ( *s == ',' ) { s++; continue; }
02038           ww = w; *w++ = 0; w++;
02039           if ( FG.cTable[*s] > 1 && *s != '[' && *s != '{' ) {
02040               MesPrint("&Illegal parameters in statement");
02041               error = 1;
02042               break;
02043           }
02044           while ( FG.cTable[*s] == 0 || *s == '[' || *s == '{' ) {
02045               if ( *s == '{' ) {
02046                   name = s+1;
02047                   SKIPBRA2(s)
02048                   c = *s; *s = 0;
02049                   number = DoTempSet(name,s);
02050                   name--; *s++ = c; c = *s; *s = 0;
02051                   goto doset;
02052               }
02053               else {
02054                   name = s;
02055                   if ( ( s = SkipAName(s) ) == 0 ) {
02056                       MesPrint("&Illegal name '%s'",name);
02057                       return(1);
02058                   }
02059                   c = *s; *s = 0;
02060                   if ( ( type = GetName(AC.varnames,name,&number,WITHAUTO) ) == CSET ) {
02061 doset:                 if ( Sets[number].type != CFUNCTION ) goto nofun;
02062 #ifdef WITHFLOAT
02063                       WORD *r1, *r2;
02064                       r1 = SetElements + Sets[number].first;
02065                       r2 = SetElements + Sets[number].last;
02066                       while ( r1 < r2 ) {
02067                           if ( *r1++ == FLOATFUN ) {
02068                               MesPrint("&Illegal use of argument environment and float_.");
02069                               error = 1;
02070                           }
02071                       }
02072 #endif
02073                       *w++ = CSET; *w++ = number;
02074                   }
02075                   else if ( type == CFUNCTION ) {
02076 #ifdef WITHFLOAT
02077                       if ( (number + FUNCTION) == FLOATFUN ) {
02078                           MesPrint("&Illegal use of argument environment and float_.");
02079                           error = 1;
02080                       }
```

```
02081 #endif
02082                         *w++ = CFUNCTION; *w++ = number + FUNCTION;
02083                     }
02084                     else {
02085 nofun:                  MesPrint("&%s is not a function or a set of functions"
02086                         ,name);
02087                         error = 1;
02088                     }
02089                 }
02090                 *s = c;
02091                 while ( *s == ',' ) s++;
02092             }
02093             ww[1] = w - ww;
02094             ww = w; w++; zeroflag = 0;
02095             while ( FG.cTable[*s] == 1 ) {
02096                 ParseNumber(x,s)
02097                 if ( *s && *s != ',' ) {
02098                     MesPrint("&Illegal separator after number");
02099                     error = 1;
02100                     while ( *s && *s != ',' ) s++;
02101                 }
02102                 while ( *s == ',' ) s++;
02103                 if ( x == 0 ) zeroflag = 1;
02104                 if ( !zeroflag ) *w++ = (WORD)x;
02105             }
02106             *ww = w - ww;
02107         } while ( *s );
02108     }
02109     oldworkpointer[1] = w - oldworkpointer;
02110     if ( par == TYPEARG ) {  /* To make sure. The Pointer might move in the future */
02111         AC.argstack[AC.arglevel-1] = cbuf[AC.cbufnum].Pointer
02112                                    - cbuf[AC.cbufnum].Buffer + 2;
02113     }
02114     AddNtoL(oldworkpointer[1],oldworkpointer);
02115     AT.WorkPointer = oldworkpointer;
02116     return(error);
02117 }
02118
02119 /*
02120     #] DoArgument :
02121     #[ CoArgument :
02122 */
02123
02124 int CoArgument(UBYTE *s) { return(DoArgument(s,TYPEARG)); }
02125
02126 /*
02127     #] CoArgument :
02128     #[ CoEndArgument :
02129 */
02130
02131 int CoEndArgument(UBYTE *s)
02132 {
02133     CBUF *C = cbuf+AC.cbufnum;
02134     while ( *s == ',' ) s++;
02135     if ( *s ) {
02136         MesPrint("&Illegal syntax for EndArgument statement");
02137         return(1);
02138     }
02139     if ( AC.arglevel <= 0 ) {
02140         MesPrint("&EndArgument without corresponding Argument statement");
02141         return(1);
02142     }
02143     AC.arglevel--;
02144     cbuf[AC.cbufnum].Buffer[AC.argstack[AC.arglevel]] = C->numlhs;
02145     if ( AC.argsumcheck[AC.arglevel] != NestingChecksum() ) {
02146         MesNesting();
02147         return(1);
02148     }
02149     return(0);
02150 }
02151
02152 /*
02153     #] CoEndArgument :
02154     #[ CoInside :
02155 */
02156
02157 int CoInside(UBYTE *s) { return(ExecInside(s)); }
02158
02159 /*
02160     #] CoInside :
02161     #[ CoEndInside :
02162 */
02163
02164 int CoEndInside(UBYTE *s)
02165 {
02166     CBUF *C = cbuf+AC.cbufnum;
02167     while ( *s == ',' ) s++;
```

```
02168     if ( *s ) {
02169         MesPrint("&Illegal syntax for EndInside statement");
02170         return(1);
02171     }
02172     if ( AC.insidelevel <= 0 ) {
02173         MesPrint("&EndInside without corresponding Inside statement");
02174         return(1);
02175     }
02176     AC.insidelevel--;
02177     cbuf[AC.cbufnum].Buffer[AC.insidestack[AC.insidelevel]] = C->numlhs;
02178     if ( AC.insidesumcheck[AC.insidelevel] != NestingChecksum() ) {
02179         MesNesting();
02180         return(1);
02181     }
02182     return(0);
02183 }
02184
02185 /*
02186     #] CoEndInside :
02187     #[ CoNormalize :
02188 */
02189
02190 int CoNormalize(UBYTE *s) { return(DoArgument(s,TYPENORM)); }
02191
02192 /*
02193     #] CoNormalize :
02194     #[ CoMakeInteger :
02195 */
02196
02197 int CoMakeInteger(UBYTE *s) { return(DoArgument(s,TYPENORM4)); }
02198
02199 /*
02200     #] CoMakeInteger :
02201     #[ CoSplitArg :
02202 */
02203
02204 int CoSplitArg(UBYTE *s) { return(DoArgument(s,TYPESPLITARG)); }
02205
02206 /*
02207     #] CoSplitArg :
02208     #[ CoSplitFirstArg :
02209 */
02210
02211 int CoSplitFirstArg(UBYTE *s) { return(DoArgument(s,TYPESPLITFIRSTARG)); }
02212
02213 /*
02214     #] CoSplitFirstArg :
02215     #[ CoSplitLastArg :
02216 */
02217
02218 int CoSplitLastArg(UBYTE *s) { return(DoArgument(s,TYPESPLITLASTARG)); }
02219
02220 /*
02221     #] CoSplitLastArg :
02222     #[ CoFactArg :
02223 */
02224
02225 int CoFactArg(UBYTE *s) {
02226     if ( ( AC.topolynomialflag & TOPOLYNOMIALFLAG ) != 0 ) {
02227         MesPrint("&ToPolynomial statement and FactArg statement are not allowed in the same module");
02228         return(1);
02229     }
02230     AC.topolynomialflag |= FACTARGFLAG;
02231     return(DoArgument(s,TYPEFACTARG));
02232 }
02233
02234 /*
02235     #] CoFactArg :
02236     #[ DoSymmetrize :
02237
02238         Syntax:
02239         Symmetrize Fun[:[number]] [Fields]      -> par = 0;
02240         AntiSymmetrize Fun[:[number]] [Fields]  -> par = 1;
02241         CycleSymmetrize Fun[:[number]] [Fields] -> par = 2;
02242         RCycleSymmetrize Fun[:[number]] [Fields]-> par = 3;
02243 */
02244
02245 int DoSymmetrize(UBYTE *s, int par)
02246 {
02247     GETIDENTITY
02248     int extra = 0, error = 0, err, fix, x, groupsize, num, i;
02249     UBYTE *name, c;
02250     WORD funnum, *w, *ww, type;
02251     for(;;) {
02252         name = s;
02253         if ( ( s = SkipAName(s) ) == 0 ) {
02254             MesPrint("&Improper function name");
```

```
02255             return(1);
02256         }
02257         c = *s; *s = 0;
02258         if ( c != ',' || ( FG.cTable[s[1]] != 0 && s[1] != '[' ) ) break;
02259         if ( par <= 0 && StrICmp(name,(UBYTE *)"cyclic") == 0 ) extra = 2;
02260         else if ( par <= 0 && StrICmp(name,(UBYTE *)"rcyclic") == 0 ) extra = 6;
02261         else {
02262             MesPrint("&Illegal option: '%s'",name);
02263             error = 1;
02264         }
02265         *s++ = c;
02266     }
02267     if ( ( err = GetVar(name,&type,&funnum,CFUNCTION,WITHAUTO) ) == NAMENOTFOUND ) {
02268         MesPrint("&Undefined function: %s",name);
02269         AddFunction(name,0,0,0,0,0,-1,-1);
02270         *s++ = c;
02271         return(1);
02272     }
02273     funnum += FUNCTION;
02274     if ( err == -1 ) error = 1;
02275     *s = c;
02276     if ( *s == ':' ) {
02277         s++;
02278         if ( *s == ',' || *s == '(' || *s == 0 ) fix = -1;
02279         else if ( FG.cTable[*s] == 1 ) {
02280             ParseNumber(fix,s)
02281             if ( fix == 0 )
02282                 Warning("Restriction to zero arguments removed");
02283         }
02284         else {
02285             MesPrint("&Illegal character after :");
02286             return(1);
02287         }
02288     }
02289     else fix = 0;
02290     w = AT.WorkPointer;
02291     *w++ = TYPEOPERATION;
02292     w++;
02293     *w++ = SYMMETRIZE;
02294     *w++ = par | extra;
02295     *w++ = funnum;
02296     *w++ = fix;
02297 /*
02298     And now the argument lists. We have either ,#,#,... or (#,#,..,#),(#,...
02299 */
02300     w += 2; ww = w; groupsize = -1;
02301     while ( *s == ',' ) s++;
02302     while ( *s ) {
02303         if ( *s == '(' ) {
02304             s++; num = 0;
02305             while ( *s && *s != ')' ) {
02306                 if ( *s == ',' ) { s++; continue; }
02307                 if ( FG.cTable[*s] != 1 ) goto illarg;
02308                 ParseNumber(x,s)
02309                 if ( x <= 0 || ( fix > 0 && x > fix ) ) goto illnum;
02310                 num++;
02311                 *w++ = x-1;
02312             }
02313             if ( *s == 0 ) {
02314                 MesPrint("&Improper termination of statement");
02315                 return(1);
02316             }
02317             if ( groupsize < 0 ) groupsize = num;
02318             else if ( groupsize != num ) goto group;
02319             s++;
02320         }
02321         else if ( FG.cTable[*s] == 1 ) {
02322             if ( groupsize < 0 ) groupsize = 1;
02323             else if ( groupsize != 1 ) {
02324 group:          MesPrint("&All groups should have the same number of arguments");
02325                 return(1);
02326             }
02327             ParseNumber(x,s)
02328             if ( x <= 0 || ( fix > 0 && x > fix ) ) {
02329 illnum:         MesPrint("&Illegal argument number: %d",x);
02330                 return(1);
02331             }
02332             *w++ = x-1;
02333         }
02334         else {
02335 illarg:     MesPrint("&Illegal argument");
02336             return(1);
02337         }
02338         while ( *s == ',' ) s++;
02339     }
02340 /*
02341     Now the completion
```

```
02342 */
02343     if ( w == ww ) {
02344         ww[-1] = 1;
02345         ww[-2] = 0;
02346         if ( fix > 0 ) {
02347             for ( i = 0; i < fix; i++ ) *w++ = i;
02348             ww[-2] = fix; /* Bugfix 31-oct-2001. Reported by York Schroeder */
02349         }
02350     }
02351     else {
02352         ww[-1] = groupsize;
02353         ww[-2] = (w-ww)/groupsize;
02354     }
02355     AT.WorkPointer[1] = w - AT.WorkPointer;
02356     AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
02357     return(error);
02358 }
02359
02360 /*
02361     #] DoSymmetrize :
02362     #[ CoSymmetrize :
02363 */
02364
02365 int CoSymmetrize(UBYTE *s) { return(DoSymmetrize(s,SYMMETRIC)); }
02366
02367 /*
02368     #] CoSymmetrize :
02369     #[ CoAntiSymmetrize :
02370 */
02371
02372 int CoAntiSymmetrize(UBYTE *s) { return(DoSymmetrize(s,ANTISYMMETRIC)); }
02373
02374 /*
02375     #] CoAntiSymmetrize :
02376     #[ CoCycleSymmetrize :
02377 */
02378
02379 int CoCycleSymmetrize(UBYTE *s) { return(DoSymmetrize(s,CYCLESYMMETRIC)); }
02380
02381 /*
02382     #] CoCycleSymmetrize :
02383     #[ CoRCycleSymmetrize :
02384 */
02385
02386 int CoRCycleSymmetrize(UBYTE *s) { return(DoSymmetrize(s,RCYCLESYMMETRIC)); }
02387
02388 /*
02389     #] CoRCycleSymmetrize :
02390     #[ CoWrite :
02391 */
02392
02393 int CoWrite(UBYTE *s)
02394 {
02395     GETIDENTITY
02396     UBYTE *option;
02397     KEYWORDV *key;
02398     option = s;
02399     if ( ( ( s = SkipAName(s) ) == 0 ) || *s != 0 ) {
02400         MesPrint("&Proper use of write statement is: write option");
02401         return(1);
02402     }
02403     key = (KEYWORDV *)FindInKeyWord(option,(KEYWORD
      *)writeoptions,sizeof(writeoptions)/sizeof(KEYWORD));
02404     if ( key == 0 ) {
02405         MesPrint("&Unrecognized option in write statement");
02406         return(1);
02407     }
02408     *key->var = key->type;
02409     AR.SortType = AC.SortType;
02410     return(0);
02411 }
02412
02413 /*
02414     #] CoWrite :
02415     #[ CoNWrite :
02416 */
02417
02418 int CoNWrite(UBYTE *s)
02419 {
02420     GETIDENTITY
02421     UBYTE *option;
02422     KEYWORDV *key;
02423     option = s;
02424     if ( ( ( s = SkipAName(s) ) == 0 ) || *s != 0 ) {
02425         MesPrint("&Proper use of nwrite statement is: nwrite option");
02426         return(1);
02427     }
```

```
02428     key = (KEYWORDV *)FindInKeyWord(option,(KEYWORD
      *)writeoptions,sizeof(writeoptions)/sizeof(KEYWORD));
02429     if ( key == 0 ) {
02430         MesPrint("&Unrecognized option in nwrite statement");
02431         return(1);
02432     }
02433     *key->var = key->flags;
02434     AR.SortType = AC.SortType;
02435     return(0);
02436 }
02437
02438 /*
02439     #] CoNWrite :
02440     #[ CoRatio :
02441 */
02442
02443 static WORD ratstring[6] = { TYPEOPERATION, 6, RATIO, 0, 0, 0 };
02444
02445 int CoRatio(UBYTE *s)
02446 {
02447     UBYTE c, *t;
02448     int i, type, error = 0;
02449     WORD numsym, *rs;
02450     rs = ratstring+3;
02451     for ( i = 0; i < 3; i++ ) {
02452         if ( *s ) {
02453             t = s;
02454             s = SkipAName(s);
02455             c = *s; *s = 0;
02456             if ( ( ( type = GetName(AC.varnames,t,&numsym,WITHAUTO) ) != CSYMBOL )
02457             && type != CDUBIOUS ) {
02458                 MesPrint("&%s is not a symbol",t);
02459                 error = 4;
02460                 if ( type < 0 ) numsym = AddSymbol(t,-MAXPOWER,MAXPOWER,0,0);
02461             }
02462             *s = c;
02463             if ( *s == ',' ) s++;
02464         }
02465         else {
02466             if ( error == 0 )
02467                 MesPrint("&The ratio statement needs three symbols for its arguments");
02468             error++;
02469             numsym = 0;
02470         }
02471         *rs++ = numsym;
02472     }
02473     AddNtoL(6,ratstring);
02474     return(error);
02475 }
02476
02477 /*
02478     #] CoRatio :
02479     #[ CoRedefine :
02480
02481     We have a preprocessor variable and a (new) value for it.
02482     This value is inside a string that must be stored.
02483 */
02484
02485 int CoRedefine(UBYTE *s)
02486 {
02487     UBYTE *name, c, *args = 0;
02488     int numprevar;
02489     WORD code[2];
02490     name = s;
02491     if ( FG.cTable[*s] || ( s = SkipAName(s) ) == 0 || s[-1] == '_' ) {
02492         MesPrint("&Illegal name for preprocessor variable in redefine statement");
02493         return(1);
02494     }
02495     c = *s; *s = 0;
02496     for ( numprevar = NumPre-1; numprevar >= 0; numprevar-- ) {
02497         if ( StrCmp(name,PreVar[numprevar].name) == 0 ) break;
02498     }
02499     if ( numprevar < 0 ) {
02500         MesPrint("&There is no preprocessor variable with the name `%s'",name);
02501         *s = c;
02502         return(1);
02503     }
02504     *s = c;
02505 /*
02506     The next code worries about arguments.
02507     It is a direct copy of the code in TheDefine in the preprocessor.
02508 */
02509     if ( *s == '(' ) {  /* arguments. scan for correctness */
02510         s++; args = s;
02511         for (;;) {
02512             if ( chartype[*s] != 0 ) goto illarg;
02513             s++;
```

```
02514             while ( chartype[*s] <= 1 ) s++;
02515             while ( *s == ' ' || *s == '\t' ) s++;
02516             if ( *s == ')' ) break;
02517             if ( *s != ',' ) goto illargs;
02518             s++;
02519             while ( *s == ' ' || *s == '\t' ) s++;
02520         }
02521         *s++ = 0;
02522         while ( *s == ' ' || *s == '\t' ) s++;
02523     }
02524     while ( *s == ',' ) s++;
02525     if ( *s != '"' ) {
02526 encl:   MesPrint("&Value for %s should be enclosed in double quotes"
02527         ,PreVar[numprevar].name);
02528         return(1);
02529     }
02530     s++; name = s; /* actually name points to the new string */
02531     while ( *s && *s != '"' ) { if ( *s == '\\' ) s++; s++; }
02532     if ( *s != '"' ) goto encl;
02533     *s = 0;
02534     code[0] = TYPEREDEFPRE; code[1] = numprevar;
02535 /*
02536     AddComString(2,code,name,0);
02537 */
02538     Add2ComStrings(2,code,name,args);
02539     *s = '"';
02540 #ifdef PARALLELCODE
02541 /*
02542     Now we prepare the input numbering system for pthreads.
02543     We need a list of preprocessor variables that are redefined in this
02544     module.
02545 */
02546     {
02547       int j;
02548       WORD *newpf;
02549       LONG *newin;
02550       for ( j = 0; j < AC.numpfirstnum; j++ ) {
02551         if ( numprevar == AC.pfirstnum[j] ) break;
02552       }
02553       if ( j >= AC.numpfirstnum ) {  /* add to list */
02554         if ( j >= AC.sizepfirstnum ) {
02555             if ( AC.sizepfirstnum <= 0 ) { AC.sizepfirstnum = 10; }
02556             else { AC.sizepfirstnum = 2 * AC.sizepfirstnum; }
02557             newin = (LONG *)Malloc1(AC.sizepfirstnum*(sizeof(WORD)+sizeof(LONG)),"AC.pfirstnum");
02558             newpf = (WORD *)(newin+AC.sizepfirstnum);
02559             for ( j = 0; j < AC.numpfirstnum; j++ ) {
02560                 newpf[j] = AC.pfirstnum[j];
02561                 newin[j] = AC.inputnumbers[j];
02562             }
02563             if ( AC.inputnumbers ) M_free(AC.inputnumbers,"AC.pfirstnum");
02564             AC.inputnumbers = newin;
02565             AC.pfirstnum = newpf;
02566         }
02567         AC.pfirstnum[AC.numpfirstnum] = numprevar;
02568         AC.inputnumbers[AC.numpfirstnum] = -1;
02569         AC.numpfirstnum++;
02570       }
02571     }
02572 #endif
02573     return(0);
02574 illarg:;
02575     MesPrint("&Illegally formed name in argument of redefine statement");
02576     return(1);
02577 illargs:;
02578     MesPrint("&Illegally formed arguments in redefine statement");
02579     return(1);
02580 }
02581
02582 /*
02583     #] CoRedefine :
02584     #[ CoRenumber :
02585
02586     renumber    or renumber,0     Only exchanges (n^2 until no improvement)
02587     renumber,1                    All permutations (could be slow)
02588 */
02589
02590 int CoRenumber(UBYTE *s)
02591 {
02592     int x;
02593     UBYTE *inp;
02594     while ( *s == ',' ) s++;
02595     inp = s;
02596     if ( *s == 0 ) { x = 0; }
02597     else ParseNumber(x,s)
02598     if ( *s == 0 && x >= 0 && x <= 1 ) {
02599         Add3Com(TYPERENUMBER,x);
02600         return(0);
```

```
02601     }
02602     MesPrint("&Illegal argument in Renumber statement: '%s'",inp);
02603     return(1);
02604 }
02605
02606 /*
02607     #] CoRenumber :
02608     #[ CoSum :
02609 */
02610
02611 int CoSum(UBYTE *s)
02612 {
02613     CBUF *C = cbuf+AC.cbufnum;
02614     UBYTE *ss = 0, c, *t;
02615     int error = 0, i = 0, type, x;
02616     WORD numindex,number;
02617     while ( *s ) {
02618         t = s;
02619         if ( *s == '$' ) {
02620             t++; s++; while ( FG.cTable[*s] < 2 ) s++;
02621             c = *s; *s = 0;
02622             if ( ( number = GetDollar(t) ) < 0 ) {
02623                 MesPrint("&Undefined variable $%s",t);
02624                 if ( !error ) error = 1;
02625                 number = AddDollar(t,0,0,0);
02626             }
02627             numindex = -number;
02628         }
02629         else {
02630             if ( ( s = SkipAName(s) ) == 0 ) return(1);
02631             c = *s; *s = 0;
02632             if ( ( ( type = GetOName(AC.exprnames,t,&numindex,NOAUTO) ) != NAMENOTFOUND )
02633             || ( ( type = GetOName(AC.varnames,t,&numindex,WITHAUTO) ) != CINDEX ) ) {
02634                 if ( type != NAMENOTFOUND ) error = NameConflict(type,t);
02635                 else {
02636                     MesPrint("&%s should have been declared as an index",t);
02637                     error = 1;
02638                     numindex = AddIndex(s,AC.lDefDim,AC.lDefDim4) + AM.OffsetIndex;
02639                 }
02640             }
02641         }
02642         Add3Com(TYPESUM,numindex);
02643         i = 3; *s = c;
02644         if ( *s == 0 ) break;
02645         if ( *s != ',' ) {
02646             MesPrint("&Illegal separator between objects in sum statement.");
02647             return(1);
02648         }
02649         s++;
02650         if ( FG.cTable[*s] == 0 || *s == '[' || *s == '$' ) {
02651             while ( FG.cTable[*s] == 0 || *s == '[' || *s == '$' ) {
02652                 if ( *s == '$' ) {
02653                     s++;
02654                     ss = t = s;
02655                     while ( FG.cTable[*s] < 2 ) s++;
02656                     c = *s; *s = 0;
02657                     if ( ( number = GetDollar(t) ) < 0 ) {
02658                         MesPrint("&Undefined variable $%s",t);
02659                         if ( !error ) error = 1;
02660                         number = AddDollar(t,0,0,0);
02661                     }
02662                     numindex = -number;
02663                 }
02664                 else {
02665                     ss = t = s;
02666                     if ( ( s = SkipAName(s) ) == 0 ) return(1);
02667                     c = *s; *s = 0;
02668                     if ( ( ( type = GetOName(AC.exprnames,t,&numindex,NOAUTO) ) != NAMENOTFOUND )
02669                     || ( ( type = GetOName(AC.varnames,t,&numindex,WITHAUTO) ) != CINDEX ) ) {
02670                         if ( type != NAMENOTFOUND ) error = NameConflict(type,t);
02671                         else {
02672                             MesPrint("&%s should have been declared as an index",t);
02673                             error = 1;
02674                             numindex = AddIndex(s,AC.lDefDim,AC.lDefDim4) + AM.OffsetIndex;
02675                         }
02676                     }
02677                 }
02678                 AddToCB(C,numindex)
02679                 i++;
02680                 C->Pointer[-i+1] = i;
02681                 *s = c;
02682                 if ( *s == 0 ) return(error);
02683                 if ( *s != ',' ) {
02684                     MesPrint("&Illegal separator between objects in sum statement.");
02685                     return(1);
02686                 }
02687                 s++;
```

```
02688             }
02689             if ( FG.cTable[*s] == 1 ) {
02690                 C->Pointer[-i+1]--; C->Pointer--; s = ss;
02691             }
02692         }
02693         else if ( FG.cTable[*s] == 1 ) {
02694             while ( FG.cTable[*s] == 1 ) {
02695                 t = s;
02696                 x = *s++ - '0';
02697                 while( FG.cTable[*s] == 1 ) x = 10*x + *s++ - '0';
02698                 if ( *s && *s != ',' ) {
02699                     MesPrint("&%s is not a legal fixed index",t);
02700                     return(1);
02701                 }
02702                 else if ( x >= AM.OffsetIndex ) {
02703                     MesPrint("&%d is too large to be a fixed index",x);
02704                     error = 1;
02705                 }
02706                 else {
02707                     AddToCB(C,x)
02708                     i++;
02709                     C->Pointer[-i] = TYPESUMFIX;
02710                     C->Pointer[-i+1] = i;
02711                 }
02712                 if ( *s == 0 ) break;
02713                 s++;
02714             }
02715         }
02716         else {
02717             MesPrint("&Illegal object in sum statement");
02718             error = 1;
02719         }
02720     }
02721     return(error);
02722 }
02723
02724 /*
02725     #] CoSum :
02726     #[ CoToTensor :
02727 */
02728
02729 static WORD cttarray[7] = { TYPEOPERATION,7,TENVEC,0,0,1,0 };
02730
02731 int CoToTensor(UBYTE *s)
02732 {
02733     UBYTE c, *t;
02734     int type, j, nargs, error = 0;
02735     WORD number, dol[2] = { 0, 0 };
02736     cttarray[1] = 6;  /* length */
02737     cttarray[3] = 0;  /* tensor */
02738     cttarray[4] = 0;  /* vector */
02739     cttarray[5] = 1;  /* option flags */
02740 /*  cttarray[6] = 0;     set veto */
02741 /*
02742     Count the number of the arguments. The validity of them is not checked here.
02743 */
02744     nargs = 0;
02745     t = s;
02746     for (;;) {
02747         while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
02748         if ( *s == 0 ) break;
02749         if ( *s == '!' ) {
02750             s++;
02751             if ( *s == '{' ) {
02752                 SKIPBRA2(s)
02753                 s++;
02754             } else {
02755                 if ( ( s = SkipAName(s) ) == 0 ) goto syntax_error;
02756             }
02757         } else {
02758             if ( ( s = SkipAName(s) ) == 0 ) goto syntax_error;
02759         }
02760         nargs++;
02761     }
02762     if ( nargs < 2 ) goto not_enough_arguments;
02763     s = t;
02764 /*
02765     Parse options, which are given as the arguments except the last two.
02766 */
02767     for ( j = 2; j < nargs; j++ ) {
02768         while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
02769         if ( *s == '!' ) {
02770 /*
02771             Handle !set or !{vector,...}. Note: If two or more sets are
02772             specified, then only the last one is used.
02773 */
02774             s++;
```

```
02775             cttarray[1] = 7;
02776             cttarray[5] |= 8;
02777             if ( FG.cTable[*s] == 0 || *s == '[' || *s == '_' ) {
02778                 t = s;
02779                 if ( ( s = SkipAName(s) ) == 0 ) goto syntax_error;
02780                 c = *s; *s = 0;
02781                 type = GetName(AC.varnames,t,&number,WITHAUTO);
02782                 if ( type == CVECTOR ) {
02783 /*
02784                     As written in the manual, "!p" (without "{}") should work.
02785 */
02786                     cttarray[6] = DoTempSet(t,s);
02787                     *s = c;
02788                     goto check_tempset;
02789                 }
02790                 else if ( type != CSET ) {
02791                     MesPrint("&%s is not the name of a set or a vector",t);
02792                     error = 1;
02793                 }
02794                 *s = c;
02795                 cttarray[6] = number;
02796             }
02797             else if ( *s == '{' ) {
02798                 t = ++s; SKIPBRA2(s) *s = 0;
02799                 cttarray[6] = DoTempSet(t,s);
02800                 *s++ = '}';
02801 check_tempset:
02802                 if ( cttarray[6] < 0 ) {
02803                     error = 1;
02804                 }
02805                 if ( AC.wildflag ) {
02806                     MesPrint("&Improper use of wildcard(s) in set specification");
02807                     error = 1;
02808                 }
02809             }
02810         } else {
02811 /*
02812             Other options.
02813 */
02814             t = s;
02815             if ( ( s = SkipAName(s) ) == 0 ) goto syntax_error;
02816             c = *s; *s = 0;
02817             if ( StrICmp(t,(UBYTE *)"nosquare") == 0 ) cttarray[5] |= 2;
02818             else if ( StrICmp(t,(UBYTE *)"functions") == 0 ) cttarray[5] |= 4;
02819             else {
02820                 MesPrint("&Unrecognized option in ToTensor statement: '%s'",t);
02821                 *s = c;
02822                 return(1);
02823             }
02824             *s = c;
02825         }
02826     }
02827 /*
02828     Now parse a vector and a tensor. The ordering doesn't matter.
02829 */
02830     for ( j = 0; j < 2; j++ ) {
02831         while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
02832         t = s;
02833         if ( ( s = SkipAName(s) ) == 0 ) goto syntax_error;
02834         c = *s; *s = 0;
02835         if ( t[0] == '$' ) {
02836             dol[j] = GetDollar(t+1);
02837             if ( dol[j] < 0 ) dol[j] = AddDollar(t+1,DOLUNDEFINED,0,0);
02838         } else {
02839             type = GetName(AC.varnames,t,&number,WITHAUTO);
02840             if ( type == CVECTOR ) {
02841                 cttarray[4] = number + AM.OffsetVector;
02842             }
02843             else if ( type == CFUNCTION && ( functions[number].spec > 0 ) ) {
02844                 cttarray[3] = number + FUNCTION;
02845             }
02846             else {
02847                 MesPrint("&%s is not a vector or a tensor",t);
02848                 error = 1;
02849             }
02850         }
02851         *s = c;
02852     }
02853     if ( cttarray[3] == 0 || cttarray[4] == 0 ) {
02854         if ( dol[0] == 0 && dol[1] == 0 ) {
02855             goto not_enough_arguments;
02856         }
02857         else if ( cttarray[3] ) {
02858             if ( dol[1] )        cttarray[4] = dol[1];
02859             else if ( dol[0] ) { cttarray[4] = dol[0]; }
02860             else {
02861                 goto not_enough_arguments;
```

```
02862             }
02863         }
02864         else if ( cttarray[4] ) {
02865             if ( dol[1] )     { cttarray[3] = -dol[1]; }
02866             else if ( dol[0] ) cttarray[3] = -dol[0];
02867             else {
02868                 goto not_enough_arguments;
02869             }
02870         }
02871         else {
02872             if ( dol[0] == 0 || dol[1] == 0 ) {
02873                 goto not_enough_arguments;
02874             }
02875             else {
02876                 cttarray[3] = -dol[0]; cttarray[4] = dol[1];
02877             }
02878         }
02879     }
02880     AddNtoL(cttarray[1],cttarray);
02881     return(error);
02882
02883 syntax_error:
02884     MesPrint("&Syntax error in ToTensor statement");
02885     return(1);
02886
02887 not_enough_arguments:
02888     MesPrint("&ToTensor statement needs a vector and a tensor");
02889     return(1);
02890 }
02891
02892 /*
02893     #] CoToTensor :
02894     #[ CoToVector :
02895 */
02896
02897 static WORD ctvarray[6] = { TYPEOPERATION,6,TENVEC,0,0,0 };
02898
02899 int CoToVector(UBYTE *s)
02900 {
02901     UBYTE *t, c;
02902     int j, type, error = 0;
02903     WORD number, dol[2];
02904     dol[0] = dol[1] = 0;
02905     ctvarray[3] = ctvarray[4] = ctvarray[5] = 0;
02906     for ( j = 0; j < 2; j++ ) {
02907         t = s;
02908         if ( ( s = SkipAName(s) ) == 0 ) {
02909 proper:     MesPrint("&Arguments of ToVector statement should be a vector and a tensor");
02910             return(1);
02911         }
02912         c = *s; *s = 0;
02913         if ( *t == '$' ) {
02914             dol[j] = GetDollar(t+1);
02915             if ( dol[j] < 0 ) dol[j] = AddDollar(t+1,DOLUNDEFINED,0,0);
02916         }
02917         else if ( ( type = GetName(AC.varnames,t,&number,WITHAUTO) ) == CVECTOR )
02918             ctvarray[4] = number + AM.OffsetVector;
02919         else if ( type == CFUNCTION && ( functions[number].spec > 0 ) )
02920             ctvarray[3] = number+FUNCTION;
02921         else {
02922             MesPrint("&%s is not a vector or a tensor",t);
02923             error = 1;
02924         }
02925         *s = c; if ( *s && *s != ',' ) goto proper;
02926         if ( *s ) s++;
02927     }
02928     if ( *s != 0 ) goto proper;
02929     if ( ctvarray[3] == 0 || ctvarray[4] == 0 ) {
02930         if ( dol[0] == 0 && dol[1] == 0 ) {
02931             MesPrint("&ToVector statement needs a vector and a tensor");
02932             error = 1;
02933         }
02934         else if ( ctvarray[3] ) {
02935             if ( dol[1] )      ctvarray[4] = dol[1];
02936             else if ( dol[0] ) ctvarray[4] = dol[0];
02937             else {
02938                 MesPrint("&ToVector statement needs a vector and a tensor");
02939                 error = 1;
02940             }
02941         }
02942         else if ( ctvarray[4] ) {
02943             if ( dol[1] )      ctvarray[3] = -dol[1];
02944             else if ( dol[0] ) ctvarray[3] = -dol[0];
02945             else {
02946                 MesPrint("&ToVector statement needs a vector and a tensor");
02947                 error = 1;
02948             }
```

```
02949         }
02950         else {
02951             if ( dol[0] == 0 || dol[1] == 0 ) {
02952                 MesPrint("&ToVector statement needs a vector and a tensor");
02953                 error = 1;
02954             }
02955             else {
02956                 ctvarray[3] = -dol[0]; ctvarray[4] = dol[1];
02957             }
02958         }
02959     }
02960     AddNtoL(6,ctvarray);
02961     return(error);
02962 }
02963
02964 /*
02965     #] CoToVector :
02966     #[ CoTrace4 :
02967 */
02968
02969 int CoTrace4(UBYTE *s)
02970 {
02971     int error = 0, type, option = CHISHOLM;
02972     UBYTE *t, c;
02973     WORD numindex, one = 1;
02974     KEYWORD *key;
02975     for (;;) {
02976         t = s;
02977         if ( FG.cTable[*s] == 1 ) break;
02978         if ( ( s = SkipAName(s) ) == 0 ) {
02979 proper:     MesPrint("&Proper syntax for Trace4 is 'Trace4[,options],index;'");
02980             return(1);
02981         }
02982         if ( *s == 0 ) break;
02983         c = *s; *s = 0;
02984         if ( ( key = FindKeyWord(t,trace4options,
02985             sizeof(trace4options)/sizeof(KEYWORD)) ) == 0 ) break;
02986         else {
02987             option |=  key->type;
02988             option &= ~key->flags;
02989         }
02990         if ( ( *s++ = c ) != ',' ) {
02991             MesPrint("&Illegal separator in Trace4 statement");
02992             return(1);
02993         }
02994         if ( *s == 0 ) goto proper;
02995     }
02996     s = t;
02997     if ( FG.cTable[*s] == 1 ) {
02998 retry:
02999         ParseNumber(numindex,s)
03000         if ( *s != 0 ) {
03001             MesPrint("&Last argument of Trace4 should be an index");
03002             return(1);
03003         }
03004         if ( numindex >= AM.OffsetIndex ) {
03005             MesPrint("&fixed index >= %d. Change value of OffsetIndex in setup file"
03006             ,AM.OffsetIndex);
03007             return(1);
03008         }
03009     }
03010     else if ( *s == '$' ) {
03011         if ( ( type = GetName(AC.dollarnames,s+1,&numindex,NOAUTO) ) == CDOLLAR )
03012             numindex = -numindex;
03013         else {
03014             MesPrint("&%s is undefined",s);
03015             numindex = AddDollar(s+1,DOLINDEX,&one,1);
03016             return(1);
03017         }
03018 tests:  s = SkipAName(s);
03019         if ( *s != 0 ) {
03020             MesPrint("&Trace4 should have a single index or $variable for its argument");
03021             return(1);
03022         }
03023     }
03024     else if ( ( type = GetName(AC.varnames,s,&numindex,WITHAUTO) ) == CINDEX ) {
03025         numindex += AM.OffsetIndex;
03026         goto tests;
03027     }
03028     else if ( type != -1 ) {
03029         if ( type != CDUBIOUS ) {
03030             if ( ( FG.cTable[*s] != 0 ) && ( *s != '[' ) ) {
03031                 if ( *s == '+' && FG.cTable[s[1]] == 1 ) { s++; goto retry; }
03032                 goto proper;
03033             }
03034             NameConflict(type,s);
03035             type = MakeDubious(AC.varnames,s,&numindex);
```

```
03036         }
03037         return(1);
03038     }
03039     else {
03040         MesPrint("&%s is not an index",s);
03041         numindex = AddIndex(s,AC.lDefDim,AC.lDefDim4) + AM.OffsetIndex;
03042         return(1);
03043     }
03044     if ( error ) return(error);
03045     if ( ( option & CHISHOLM ) != 0 )
03046         Add4Com(TYPECHISHOLM,numindex,(option & ALSOREVERSE));
03047     Add5Com(TYPEOPERATION,TAKETRACE,4 + (option & NOTRICK),numindex);
03048     return(0);
03049 }
03050
03051 /*
03052     #] CoTrace4 :
03053     #[ CoTraceN :
03054 */
03055
03056 int CoTraceN(UBYTE *s)
03057 {
03058     WORD numindex, one = 1;
03059     int type;
03060     if ( FG.cTable[*s] == 1 ) {
03061 retry:
03062         ParseNumber(numindex,s)
03063         if ( *s != 0 ) {
03064 proper:     MesPrint("&TraceN should have a single index for its argument");
03065             return(1);
03066         }
03067         if ( numindex >= AM.OffsetIndex ) {
03068             MesPrint("&fixed index >= %d. Change value of OffsetIndex in setup file"
03069             ,AM.OffsetIndex);
03070             return(1);
03071         }
03072     }
03073     else if ( *s == '$' ) {
03074         if ( ( type = GetName(AC.dollarnames,s+1,&numindex,NOAUTO) ) == CDOLLAR )
03075             numindex = -numindex;
03076         else {
03077             MesPrint("&%s is undefined",s);
03078             numindex = AddDollar(s+1,DOLINDEX,&one,1);
03079             return(1);
03080         }
03081 tests:  s = SkipAName(s);
03082         if ( *s != 0 ) {
03083             MesPrint("&TraceN should have a single index or $variable for its argument");
03084             return(1);
03085         }
03086     }
03087     else if ( ( type = GetName(AC.varnames,s,&numindex,WITHAUTO) ) == CINDEX ) {
03088         numindex += AM.OffsetIndex;
03089         goto tests;
03090     }
03091     else if ( type != -1 ) {
03092         if ( type != CDUBIOUS ) {
03093             if ( ( FG.cTable[*s] != 0 ) && ( *s != '[' ) ) {
03094                 if ( *s == '+' && FG.cTable[s[1]] == 1 ) { s++; goto retry; }
03095                 goto proper;
03096             }
03097             NameConflict(type,s);
03098             type = MakeDubious(AC.varnames,s,&numindex);
03099         }
03100         return(1);
03101     }
03102     else {
03103         MesPrint("&%s is not an index",s);
03104         numindex = AddIndex(s,AC.lDefDim,AC.lDefDim4) + AM.OffsetIndex;
03105         return(1);
03106     }
03107     Add5Com(TYPEOPERATION,TAKETRACE,0,numindex);
03108     return(0);
03109 }
03110
03111 /*
03112     #] CoTraceN :
03113     #[ CoChisholm :
03114 */
03115
03116 int CoChisholm(UBYTE *s)
03117 {
03118     int error = 0, type, option = CHISHOLM;
03119     UBYTE *t, c;
03120     WORD numindex, one = 1;
03121     KEYWORD *key;
03122     for (;;) {
```

```
03123         t = s;
03124         if ( FG.cTable[*s] == 1 ) break;
03125         if ( ( s = SkipAName(s) ) == 0 ) {
03126 proper:     MesPrint("&Proper syntax for Chisholm is 'Chisholm[,options],index;'");
03127             return(1);
03128         }
03129         if ( *s == 0 ) break;
03130         c = *s; *s = 0;
03131         if ( ( key = FindKeyWord(t,chisoptions,
03132             sizeof(chisoptions)/sizeof(KEYWORD)) ) == 0 ) break;
03133         else {
03134             option |=  key->type;
03135             option &= ~key->flags;
03136         }
03137         if ( ( *s++ = c ) != ',' ) {
03138             MesPrint("&Illegal separator in Chisholm statement");
03139             return(1);
03140         }
03141         if ( *s == 0 ) goto proper;
03142     }
03143     s = t;
03144     if ( FG.cTable[*s] == 1 ) {
03145         ParseNumber(numindex,s)
03146         if ( *s != 0 ) {
03147             MesPrint("&Last argument of Chisholm should be an index");
03148             return(1);
03149         }
03150         if ( numindex >= AM.OffsetIndex ) {
03151             MesPrint("&fixed index >= %d. Change value of OffsetIndex in setup file"
03152             ,AM.OffsetIndex);
03153             return(1);
03154         }
03155     }
03156     else if ( *s == '$' ) {
03157         if ( ( type = GetName(AC.dollarnames,s+1,&numindex,NOAUTO) ) == CDOLLAR )
03158             numindex = -numindex;
03159         else {
03160             MesPrint("&%s is undefined",s);
03161             numindex = AddDollar(s+1,DOLINDEX,&one,1);
03162             return(1);
03163         }
03164 tests:  s = SkipAName(s);
03165         if ( *s != 0 ) {
03166             MesPrint("&Chisholm should have a single index or $variable for its argument");
03167             return(1);
03168         }
03169     }
03170     else if ( ( type = GetName(AC.varnames,s,&numindex,WITHAUTO) ) == CINDEX ) {
03171         numindex += AM.OffsetIndex;
03172         goto tests;
03173     }
03174     else if ( type != -1 ) {
03175         if ( type != CDUBIOUS ) {
03176             NameConflict(type,s);
03177             type = MakeDubious(AC.varnames,s,&numindex);
03178         }
03179         return(1);
03180     }
03181     else {
03182         MesPrint("&%s is not an index",s);
03183         numindex = AddIndex(s,AC.lDefDim,AC.lDefDim4) + AM.OffsetIndex;
03184         return(1);
03185     }
03186     if ( error ) return(error);
03187     Add4Com(TYPECHISHOLM,numindex,(option & ALSOREVERSE));
03188     return(0);
03189 }
03190
03191 /*
03192     #] CoChisholm :
03193     #[ DoChain :
03194
03195     Syntax: Chainxx functionname;
03196 */
03197
03198 int DoChain(UBYTE *s, int option)
03199 {
03200     WORD numfunc,type;
03201     if ( *s == '$' ) {
03202         if ( ( type = GetName(AC.dollarnames,s+1,&numfunc,NOAUTO) ) == CDOLLAR )
03203             numfunc = -numfunc;
03204         else {
03205             MesPrint("&%s is undefined",s);
03206             numfunc = AddDollar(s+1,DOLINDEX,&one,1);
03207             return(1);
03208         }
03209 tests:  s = SkipAName(s);
```

```
03210         if ( *s != 0 ) {
03211             MesPrint("&ChainIn/ChainOut should have a single function or $variable for its argument");
03212             return(1);
03213         }
03214     }
03215     else if ( ( type = GetName(AC.varnames,s,&numfunc,WITHAUTO) ) == CFUNCTION ) {
03216         numfunc += FUNCTION;
03217         goto tests;
03218     }
03219     else if ( type != -1 ) {
03220         if ( type != CDUBIOUS ) {
03221             NameConflict(type,s);
03222             type = MakeDubious(AC.varnames,s,&numfunc);
03223         }
03224         return(1);
03225     }
03226     else {
03227         MesPrint("&%s is not a function",s);
03228         numfunc = AddFunction(s,0,0,0,0,0,-1,-1) + FUNCTION;
03229         return(1);
03230     }
03231     Add3Com(option,numfunc);
03232     return(0);
03233 }
03234
03235 /*
03236     #] DoChain :
03237     #[ CoChainin :
03238
03239     Syntax: Chainin functionname;
03240 */
03241
03242 int CoChainin(UBYTE *s)
03243 {
03244     return(DoChain(s,TYPECHAININ));
03245 }
03246
03247 /*
03248     #] CoChainin :
03249     #[ CoChainout :
03250
03251     Syntax: Chainout functionname;
03252 */
03253
03254 int CoChainout(UBYTE *s)
03255 {
03256     return(DoChain(s,TYPECHAINOUT));
03257 }
03258
03259 /*
03260     #] CoChainout :
03261     #[ CoExit :
03262 */
03263
03264 int CoExit(UBYTE *s)
03265 {
03266     UBYTE *name;
03267     WORD code = TYPEEXIT;
03268     while ( *s == ',' ) s++;
03269     if ( *s == 0 ) {
03270         Add3Com(TYPEEXIT,0);
03271         return(0);
03272     }
03273     name = s+1;
03274     s++;
03275     while ( *s ) { if ( *s == '\\' ) s++; s++; }
03276     if ( name[-1] != '"' || s[-1] != '"' ) {
03277         MesPrint("&Illegal syntax for exit statement");
03278         return(1);
03279     }
03280     s[-1] = 0;
03281     AddComString(1,&code,name,0);
03282     s[-1] = '"';
03283     return(0);
03284 }
03285
03286 /*
03287     #] CoExit :
03288     #[ CoInParallel :
03289 */
03290
03291 int CoInParallel(UBYTE *s)
03292 {
03293     return(DoInParallel(s,1));
03294 }
03295
03296 /*
```

```
03297     #] CoInParallel :
03298     #[ CoNotInParallel :
03299 */
03300
03301 int CoNotInParallel(UBYTE *s)
03302 {
03303     return(DoInParallel(s,0));
03304 }
03305
03306 /*
03307     #] CoNotInParallel :
03308     #[ DoInParallel :
03309
03310     InParallel;
03311     InParallel,names;
03312     NotInParallel;
03313     NotInParallel,names;
03314 */
03315
03316 int DoInParallel(UBYTE *s, int par)
03317 {
03318 #ifdef PARALLELCODE
03319     EXPRESSIONS e;
03320     WORD i;
03321 #endif
03322     WORD number;
03323     UBYTE *t, c;
03324     int error = 0;
03325 #ifndef WITHPTHREADS
03326     DUMMYUSE(par);
03327 #endif
03328     if ( *s == 0 ) {
03329         AC.inparallelflag = par;
03330 #ifdef PARALLELCODE
03331         for ( i = NumExpressions-1; i >= 0; i-- ) {
03332             e = Expressions+i;
03333             if ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION
03334             || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION
03335             ) {
03336                 e->partodo = par;
03337             }
03338         }
03339 #endif
03340     }
03341     else {
03342         for(;;) {   /* Look for a (comma separated) list of variables */
03343             while ( *s == ',' ) s++;
03344             if ( *s == 0 ) break;
03345             if ( *s == '[' || FG.cTable[*s] == 0 ) {
03346                 t = s;
03347                 if ( ( s = SkipAName(s) ) == 0 ) {
03348                     MesPrint("&Improper name for an expression: '%s'",t);
03349                     return(1);
03350                 }
03351                 c = *s; *s = 0;
03352                 if ( GetName(AC.exprnames,t,&number,NOAUTO) == CEXPRESSION ) {
03353 #ifdef PARALLELCODE
03354                     e = Expressions+number;
03355                     if ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION
03356                     || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION
03357                     ) {
03358                         e->partodo = par;
03359                     }
03360 #endif
03361                 }
03362                 else if ( GetName(AC.varnames,t,&number,NOAUTO) != NAMENOTFOUND ) {
03363                     MesPrint("&%s is not an expression",t);
03364                     error = 1;
03365                 }
03366                 *s = c;
03367             }
03368             else {
03369                 MesPrint("&Illegal object in InParallel statement");
03370                 error = 1;
03371                 while ( *s && *s != ',' ) s++;
03372                 if ( *s == 0 ) break;
03373             }
03374         }
03375
03376     }
03377     return(error);
03378 }
03379
03380 /*
03381     #] DoInParallel :
03382     #[ CoInExpression :
03383 */
```

```
03384
03385 int CoInExpression(UBYTE *s)
03386 {
03387     GETIDENTITY
03388     UBYTE *t, c;
03389     WORD *w, number;
03390     int error = 0;
03391     w = AT.WorkPointer;
03392     if ( AC.inexprlevel >= MAXNEST ) {
03393         MesPrint("@Nesting of inexpression statements more than %d levels",(WORD)MAXNEST);
03394         return(-1);
03395     }
03396     AC.inexprsumcheck[AC.inexprlevel] = NestingChecksum();
03397     AC.inexprstack[AC.inexprlevel] = cbuf[AC.cbufnum].Pointer
03398                                    - cbuf[AC.cbufnum].Buffer + 2;
03399     AC.inexprlevel++;
03400     *w++ = TYPEINEXPRESSION;
03401     w++; w++;
03402     for(;;) {   /* Look for a (comma separated) list of variables */
03403         while ( *s == ',' ) s++;
03404         if ( *s == 0 ) break;
03405         if ( *s == '[' || FG.cTable[*s] == 0 ) {
03406             t = s;
03407             if ( ( s = SkipAName(s) ) == 0 ) {
03408                 MesPrint("&Improper name for an expression: '%s'",t);
03409                 return(1);
03410             }
03411             c = *s; *s = 0;
03412             if ( GetName(AC.exprnames,t,&number,NOAUTO) == CEXPRESSION ) {
03413                 *w++ = number;
03414             }
03415             else if ( GetName(AC.varnames,t,&number,NOAUTO) != NAMENOTFOUND ) {
03416                 MesPrint("&%s is not an expression",t);
03417                 error = 1;
03418             }
03419             *s = c;
03420         }
03421         else {
03422             MesPrint("&Illegal object in InExpression statement");
03423             error = 1;
03424             while ( *s && *s != ',' ) s++;
03425             if ( *s == 0 ) break;
03426         }
03427     }
03428     AT.WorkPointer[1] = w - AT.WorkPointer;
03429     AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
03430     return(error);
03431 }
03432
03433 /*
03434     #] CoInExpression :
03435     #[ CoEndInExpression :
03436 */
03437
03438 int CoEndInExpression(UBYTE *s)
03439 {
03440     CBUF *C = cbuf+AC.cbufnum;
03441     while ( *s == ',' ) s++;
03442     if ( *s ) {
03443         MesPrint("&Illegal syntax for EndInExpression statement");
03444         return(1);
03445     }
03446     if ( AC.inexprlevel <= 0 ) {
03447         MesPrint("&EndInExpression without corresponding InExpression statement");
03448         return(1);
03449     }
03450     AC.inexprlevel--;
03451     cbuf[AC.cbufnum].Buffer[AC.inexprstack[AC.inexprlevel]] = C->numlhs;
03452     if ( AC.inexprsumcheck[AC.inexprlevel] != NestingChecksum() ) {
03453         MesNesting();
03454         return(1);
03455     }
03456     return(0);
03457 }
03458
03459 /*
03460     #] CoEndInExpression :
03461     #[ CoSetExitFlag :
03462 */
03463
03464 int CoSetExitFlag(UBYTE *s)
03465 {
03466     if ( *s ) {
03467         MesPrint("&Illegal syntax for the SetExitFlag statement");
03468         return(1);
03469     }
03470     Add2Com(TYPESETEXIT);
```

```
03471     return(0);
03472 }
03473
03474 /*
03475     #] CoSetExitFlag :
03476     #[ CoTryReplace :
03477 */
03478 int CoTryReplace(UBYTE *p)
03479 {
03480     GETIDENTITY
03481     UBYTE *name, c;
03482     WORD *w, error = 0, i, which = -1, c1, minvec = 0;
03483     w = AT.WorkPointer;
03484     *w++ = TYPETRY;
03485     *w++ = 3;
03486     *w++ = 0;
03487     *w++ = REPLACEMENT;
03488     *w++ = FUNHEAD;
03489     FILLFUN(w)
03490 /*
03491     Now we have to read a function argument for the replace_ function.
03492     Current arguments that we allow involve only single arguments
03493     that do not expand further. No brackets!
03494 */
03495     while ( *p ) {
03496 /*
03497         No numbers yet
03498 */
03499         if ( *p == '-' && minvec == 0 && which == (CVECTOR+1) ) {
03500             minvec = 1; p++;
03501         }
03502         if ( *p == '[' || FG.cTable[*p] == 0 ) {
03503             name = p;
03504             if ( ( p = SkipAName(p) )  == 0 ) return(1);
03505             c = *p; *p = 0;
03506             i = GetName(AC.varnames,name,&c1,WITHAUTO);
03507             if ( which >= 0 && i >= 0 && i != CDUBIOUS && which != (i+1) ) {
03508                 MesPrint("&Illegal combination of objects in TryReplace");
03509                 error = 1;
03510             }
03511             else if ( minvec && i != CVECTOR && i != CDUBIOUS ) {
03512                 MesPrint("&Currently a - sign can be used only with a vector in TryReplace");
03513                 error = 1;
03514             }
03515             else switch ( i ) {
03516                 case CSYMBOL: *w++ = -SYMBOL; *w++ = c1; break;
03517                 case CVECTOR:
03518                     if ( minvec ) *w++ = -MINVECTOR;
03519                     else          *w++ = -VECTOR;
03520                     *w++ = c1 + AM.OffsetVector;
03521                     minvec = 0;
03522                     break;
03523                 case CINDEX: *w++ = -INDEX; *w++ = c1 + AM.OffsetIndex;
03524                     if ( c1 >= AM.WilInd && c == '?' ) { *p++ = c; c = *p; }
03525                     break;
03526                 case CFUNCTION: *w++ = -c1-FUNCTION; break;
03527                 case CDUBIOUS: minvec = 0; error = 1; break;
03528                 default:
03529                     MesPrint("&Illegal object type in TryReplace: %s",name);
03530                     error = 1;
03531                     i = 0;
03532                     break;
03533             }
03534             if ( which < 0 ) which = i+1;
03535             else which = -1;
03536             *p = c;
03537             if ( *p == ',' ) p++;
03538             continue;
03539         }
03540         else {
03541             MesPrint("&Illegal object in TryReplace");
03542             error = 1;
03543             while ( *p && *p != ',' ) {
03544                 if ( *p == '(' ) SKIPBRA3(p)
03545                 else if ( *p == '{' ) SKIPBRA2(p)
03546                 else if ( *p == '[' ) SKIPBRA1(p)
03547                 else p++;
03548             }
03549         }
03550         if ( *p == ',' ) p++;
03551         if ( which < 0 ) which = 0;
03552         else which = -1;
03553     }
03554     if ( which >= 0 ) {
03555         MesPrint("&Odd number of arguments in TryReplace");
03556         error = 1;
03557     }
```

```
03558     i = w - AT.WorkPointer;
03559     AT.WorkPointer[1] = i;
03560     AT.WorkPointer[2] = i - 3;
03561     AT.WorkPointer[4] = i - 3;
03562     AddNtoL((int)i,AT.WorkPointer);
03563     return(error);
03564 }
03565
03566 /*
03567     #] CoTryReplace :
03568     #[ CoModulus :
03569
03570     Old syntax:  Modulus [-] number [:number]
03571     New syntax:  Modulus [option(s)] number
03572         Options are: NoFunctions/CoefficientsOnly/AlsoFunctions
03573                      PlusMin/Positive
03574                      InverseTable
03575                      PrintPowersOf(number)
03576                      AlsoPowers/NoPowers
03577                      AlsoDollars/NoDollars
03578     Notice: We change the defaults. This may cause problems to some.
03579 */
03580
03581 int CoModulus(UBYTE *inp)
03582 {
03583     GETIDENTITY
03584     int Retval = 0, sign = 1;
03585     UBYTE *p, c;
03586     while ( *inp == ',' || *inp == ' ' || *inp == '\t' ) inp++;
03587     if ( *inp == 0 ) {
03588 SwitchOff:
03589         if ( AC.modpowers ) M_free(AC.modpowers,"AC.modpowers");
03590         AC.modpowers = 0;
03591         AN.ncmod = AC.ncmod = 0;
03592         if ( AC.halfmod ) M_free(AC.halfmod,"halfmod");
03593         AC.halfmod = 0; AC.nhalfmod = 0;
03594         if ( AC.modinverses ) M_free(AC.modinverses,"modinverses");
03595         AC.modinverses = 0;
03596         AC.modmode = 0;
03597         return(0);
03598     }
03599 #ifdef WITHFLOAT
03600     if ( AT.aux_ != 0 ) {
03601         MesPrint("&Simultaneous use of floating point and modulus arithmetic makes no sense.");
03602         Retval = 1;
03603     }
03604 #endif
03605     AC.modmode = 0;
03606     if ( *inp == '-' ) {
03607         sign = -1;
03608         inp++;
03609     }
03610     else {
03611       while ( FG.cTable[*inp] == 0 ) {
03612         p = inp;
03613         while ( FG.cTable[*inp] == 0 ) inp++;
03614         c = *inp; *inp = 0;
03615         if ( StrICmp(p,(UBYTE *)"nofunctions") == 0 ) {
03616             AC.modmode &= ~ALSOFUNARGS;
03617         }
03618         else if ( StrICmp(p,(UBYTE *)"alsofunctions") == 0 ) {
03619             AC.modmode |= ALSOFUNARGS;
03620         }
03621         else if ( StrICmp(p,(UBYTE *)"coefficientsonly") == 0 ) {
03622             AC.modmode &= ~ALSOFUNARGS;
03623             AC.modmode &= ~ALSOPOWERS;
03624             sign = -1;
03625         }
03626         else if ( StrICmp(p,(UBYTE *)"plusmin") == 0 ) {
03627             AC.modmode |= POSNEG;
03628         }
03629         else if ( StrICmp(p,(UBYTE *)"positive") == 0 ) {
03630             AC.modmode &= ~POSNEG;
03631         }
03632         else if ( StrICmp(p,(UBYTE *)"inversetable") == 0 ) {
03633             AC.modmode |= INVERSETABLE;
03634         }
03635         else if ( StrICmp(p,(UBYTE *)"noinversetable") == 0 ) {
03636             AC.modmode &= ~INVERSETABLE;
03637         }
03638         else if ( StrICmp(p,(UBYTE *)"nodollars") == 0 ) {
03639             AC.modmode &= ~ALSODOLLARS;
03640         }
03641         else if ( StrICmp(p,(UBYTE *)"alsodollars") == 0 ) {
03642             AC.modmode |= ALSODOLLARS;
03643         }
03644         else if ( StrICmp(p,(UBYTE *)"printpowersof") == 0 ) {
```

```
03645             *inp = c;
03646             if ( *inp != '(' ) {
03647 badsyntax:
03648                 MesPrint("&Bad syntax in argument of PrintPowersOf(number) in Modulus statement");
03649                 return(1);
03650             }
03651             while ( *inp == ',' || *inp == ' ' || *inp == '\t' ) inp++;
03652             inp++; p = inp;
03653             if ( FG.cTable[*inp] != 1 ) goto badsyntax;
03654             do { inp++; } while ( FG.cTable[*inp] == 1 );
03655             c = *inp; *inp = 0;
03656             if ( GetLong(p,(UWORD *)AC.powmod,&AC.npowmod) ) Retval = -1;
03657             if ( TakeModulus((UWORD *)AC.powmod,&AC.npowmod,AC.cmod,AC.ncmod,NOUNPACK) ) Retval = -1;
03658             if ( AC.npowmod == 0 ) {
03659                 MesPrint("&Improper value for generator");
03660                 Retval = -1;
03661             }
03662             if ( MakeModTable() ) Retval = -1;
03663             AC.DirtPow = 1;
03664             *inp = c;
03665             while ( *inp == ',' || *inp == ' ' || *inp == '\t' ) inp++;
03666             if ( *inp != ')' ) goto badsyntax;
03667             inp++;
03668             c = *inp;
03669         }
03670         else if ( StrICmp(p,(UBYTE *)"alsopowers") == 0 ) {
03671             AC.modmode |= ALSOPOWERS;
03672             sign =  1;
03673         }
03674         else if ( StrICmp(p,(UBYTE *)"nopowers") == 0 ) {
03675             AC.modmode &= ~ALSOPOWERS;
03676             sign = -1;
03677         }
03678         else {
03679             MesPrint("&Unrecognized option %s in Modulus statement",inp);
03680             return(1);
03681         }
03682         *inp = c;
03683         while ( *inp == ',' || *inp == ' ' || *inp == '\t' ) inp++;
03684         if ( *inp == 0 ) {
03685             MesPrint("&Modulus statement with no value!!!");
03686             return(1);
03687         }
03688       }
03689     }
03690     p = inp;
03691     if ( FG.cTable[*inp] != 1 ) {
03692         MesPrint("&Invalid value for modulus:%s",inp);
03693         if ( AC.modpowers ) M_free(AC.modpowers,"AC.modpowers");
03694         AC.modpowers = 0;
03695         AN.ncmod = AC.ncmod = 0;
03696         if ( AC.halfmod ) M_free(AC.halfmod,"halfmod");
03697         AC.halfmod = 0; AC.nhalfmod = 0;
03698         if ( AC.modinverses ) M_free(AC.modinverses,"modinverses");
03699         AC.modinverses = 0;
03700         return(1);
03701     }
03702     do { inp++; } while ( FG.cTable[*inp] == 1 );
03703     c = *inp; *inp = 0;
03704     Retval = GetLong(p,(UWORD *)AC.cmod,&AC.ncmod);
03705     if ( Retval == 0 && AC.ncmod == 0 ) goto SwitchOff;
03706     if ( sign < 0 ) AC.ncmod = -AC.ncmod;
03707     AN.ncmod = AC.ncmod;
03708     if ( ( AC.modmode & INVERSETABLE ) != 0 ) MakeInverses();
03709     if ( AC.halfmod ) M_free(AC.halfmod,"halfmod");
03710     AC.halfmod = 0; AC.nhalfmod = 0;
03711     return(Retval);
03712 }
03713
03714 /*
03715     #] CoModulus :
03716     #[ CoRepeat :
03717 */
03718
03719 int CoRepeat(UBYTE *inp)
03720 {
03721     int error = 0;
03722     AC.RepSumCheck[AC.RepLevel] = NestingChecksum();
03723     AC.RepLevel++;
03724     if ( AC.RepLevel > AM.RepMax ) {
03725         MesPrint("&Too many repeat levels. Maximum is %d",AM.RepMax);
03726         return(1);
03727     }
03728     Add3Com(TYPEREPEAT,-1)   /* Means indefinite */
03729     while ( *inp == ' ' || *inp == ',' || *inp == '\t' ) inp++;
03730     if ( *inp ) {
03731         error = CompileStatement(inp);
```

```
03732         if ( CoEndRepeat(inp) ) error = 1;
03733     }
03734     return(error);
03735 }
03736
03737 /*
03738     #] CoRepeat :
03739     #[ CoEndRepeat :
03740 */
03741
03742 int CoEndRepeat(UBYTE *inp)
03743 {
03744     CBUF *C = cbuf+AC.cbufnum;
03745     int level, error = 0, repeatlevel = 0;
03746     DUMMYUSE(inp);
03747     AC.RepLevel--;
03748     if ( AC.RepLevel < 0 ) {
03749         MesPrint("&EndRepeat without Repeat");
03750         AC.RepLevel = 0;
03751         return(1);
03752     }
03753     else if ( AC.RepSumCheck[AC.RepLevel] != NestingChecksum() ) {
03754         MesNesting();
03755         error = 1;
03756     }
03757     level = C->numlhs+1;
03758     while ( level > 0 ) {
03759         if ( C->lhs[--level][0] == TYPEREPEAT ) {
03760             if ( repeatlevel == 0 ) {
03761                 Add3Com(TYPEENDREPEAT,level)
03762                 return(error);
03763             }
03764             repeatlevel--;
03765         }
03766         else if ( C->lhs[level][0] == TYPEENDREPEAT ) repeatlevel++;
03767     }
03768     return(1);
03769 }
03770
03771 /*
03772     #] CoEndRepeat :
03773     #[ DoBrackets :
03774
03775         Reads in the bracket information.
03776         Storage is in the form of a regular term.
03777         No subterms and arguments are allowed.
03778 */
03779
03780 int DoBrackets(UBYTE *inp, int par)
03781 {
03782     GETIDENTITY
03783     UBYTE *p, *pp, c;
03784     WORD *to, i, type, *w, error = 0;
03785     WORD c1,c2, *WorkSave;
03786     int biflag;
03787     p = inp;
03788     WorkSave = to = AT.WorkPointer;
03789     to++;
03790     if ( AT.BrackBuf == 0 ) {
03791         AR.MaxBracket = 100;
03792         AT.BrackBuf = (WORD *)Malloc1(sizeof(WORD)*(AR.MaxBracket+1),"bracket buffer");
03793     }
03794     *AT.BrackBuf = 0;
03795     AR.BracketOn = 0;
03796     AC.bracketindexflag = 0;
03797     AT.bracketindexflag = 0;
03798     if ( *p == '+' || *p == '-' ) p++;
03799     if ( p[-1] == ',' && *p ) p--;
03800     if ( p[-1] == '+' && *p ) { biflag = 1;  if ( *p != ',' ) { *--p = ','; } }
03801     else if ( p[-1] == '-' && *p ) { biflag = -1; if ( *p != ',' ) { *--p = ','; } }
03802     else biflag = 0;
03803     while ( *p == ',' ) {
03804 redo:   AR.BracketOn++;
03805         while ( *p == ',' ) p++;
03806         if ( *p == 0 ) break;
03807         if ( *p == '0' ) {
03808             p++; while ( *p == '0' ) p++;
03809             continue;
03810         }
03811         inp = pp = p;
03812         p = SkipAName(p);
03813         if ( p == 0 ) return(1);
03814         c = *p;
03815         *p = 0;
03816         type = GetName(AC.varnames,inp,&c1,WITHAUTO);
03817         if ( c == '.' ) {
03818             if ( type == CVECTOR || type == CDUBIOUS ) {
```

```
03819                 *p++ = c;
03820                 inp = p;
03821                 p = SkipAName(p);
03822                 if ( p == 0 ) return(1);
03823                 c = *p;
03824                 *p = 0;
03825                 type = GetName(AC.varnames,inp,&c2,WITHAUTO);
03826                 if ( type != CVECTOR && type != CDUBIOUS ) {
03827                     MesPrint("&Not a vector in dotproduct in bracket statement: %s",inp);
03828                     error = 1;
03829                 }
03830                 else type = CDOTPRODUCT;
03831             }
03832             else {
03833                 MesPrint("&Illegal use of . after %s in bracket statement",inp);
03834                 error = 1;
03835                 *p++ = c;
03836                 goto redo;
03837             }
03838         }
03839         switch ( type ) {
03840             case CSYMBOL :
03841                 *to++ = SYMBOL; *to++ = 4; *to++ = c1; *to++ = 1; break;
03842             case CVECTOR :
03843                 *to++ = INDEX; *to++ = 3; *to++ = AM.OffsetVector + c1; break;
03844             case CFUNCTION :
03845                 *to++ = c1+FUNCTION; *to++ = FUNHEAD; *to++ = 0;
03846                 FILLFUN3(to)
03847                 break;
03848             case CDOTPRODUCT :
03849                 *to++ = DOTPRODUCT; *to++ = 5; *to++ = c1 + AM.OffsetVector;
03850                 *to++ = c2 + AM.OffsetVector; *to++ = 1; break;
03851             case CDELTA :
03852                 *to++ = DELTA; *to++ = 4; *to++ = EMPTYINDEX; *to++ = EMPTYINDEX; break;
03853             case CSET :
03854                 *to++ = SETSET; *to++ = 4; *to++ = c1; *to++ = Sets[c1].type; break;
03855             default :
03856                 MesPrint("&Illegal bracket request for %s",pp);
03857                 error = 1; break;
03858         }
03859         *p = c;
03860     }
03861     if ( *p ) {
03862         MesCerr("separator",p);
03863         AC.BracketNormalize = 0;
03864         AT.WorkPointer = WorkSave;
03865         error = 1;
03866         return(error);
03867     }
03868     *to++ = 1; *to++ = 1; *to++ = 3;
03869     *AT.WorkPointer = to - AT.WorkPointer;
03870     AT.WorkPointer = to;
03871     AC.BracketNormalize = 1;
03872     if ( BracketNormalize(BHEAD WorkSave) ) { error = 1; AR.BracketOn = 0; }
03873     else {
03874         w = WorkSave;
03875         if ( *w == 4 || !*w ) { AR.BracketOn = 0; }
03876         else {
03877             i = *(w+*w-1);
03878             if ( i < 0 ) i = -i;
03879             *w -= i;
03880             i = *w;
03881             if ( i > AR.MaxBracket ) {
03882                 WORD *newbuf;
03883                 newbuf = (WORD *)Malloc1(sizeof(WORD)*(i+1),"bracket buffer");
03884                 AR.MaxBracket = i;
03885                 if ( AT.BrackBuf != 0 ) M_free(AT.BrackBuf,"bracket buffer");
03886                 AT.BrackBuf = newbuf;
03887             }
03888             to = AT.BrackBuf;
03889             NCOPY(to,w,i);
03890         }
03891     }
03892     AC.BracketNormalize = 0;
03893     if ( par == 1 ) AR.BracketOn = -AR.BracketOn;
03894     if ( error == 0 ) {
03895         AC.bracketindexflag = biflag;
03896         AT.bracketindexflag = biflag;
03897     }
03898     AT.WorkPointer = WorkSave;
03899     return(error);
03900 }
03901
03902 /*
03903     #] DoBrackets :
03904     #[ CoBracket :
03905 */
```

```
03906
03907 int CoBracket(UBYTE *inp)
03908 { return(DoBrackets(inp,0)); }
03909
03910 /*
03911     #] CoBracket :
03912     #[ CoAntiBracket :
03913 */
03914
03915 int CoAntiBracket(UBYTE *inp)
03916 { return(DoBrackets(inp,1)); }
03917
03918 /*
03919     #] CoAntiBracket :
03920     #[ CoMultiBracket :
03921
03922     Syntax:
03923         MultiBracket:{A|B} bracketinfo:...:{A|B} bracketinfo;
03924 */
03925 /* UNFINISHED_FEATURE_EXCL_START */
03926 int CoMultiBracket(UBYTE *inp)
03927 {
03928     GETIDENTITY
03929     int i, error = 0, error1, type, num;
03930     UBYTE *s, c;
03931     WORD *to, *from;
03932
03933     if ( *inp != ':' ) {
03934         MesPrint("&Illegal Multiple Bracket separator: %s",inp);
03935         return(1);
03936     }
03937     inp++;
03938     if ( AC.MultiBracketBuf == 0 ) {
03939         AC.MultiBracketBuf = (WORD **)Malloc1(sizeof(WORD *)*MAXMULTIBRACKETLEVELS,"multi bracket
      buffer");
03940         for ( i = 0; i < MAXMULTIBRACKETLEVELS; i++ ) {
03941             AC.MultiBracketBuf[i] = 0;
03942         }
03943     }
03944     else {
03945       for ( i = 0; i < MAXMULTIBRACKETLEVELS; i++ ) {
03946         if ( AC.MultiBracketBuf[i] ) {
03947             M_free(AC.MultiBracketBuf[i],"bracket buffer i");
03948             AC.MultiBracketBuf[i] = 0;
03949         }
03950       }
03951       AC.MultiBracketLevels = 0;
03952     }
03953     AC.MultiBracketLevels = 0;
03954 /*
03955         Start with disabling the regular brackets.
03956 */
03957     if ( AT.BrackBuf == 0 ) {
03958         AR.MaxBracket = 100;
03959         AT.BrackBuf = (WORD *)Malloc1(sizeof(WORD)*(AR.MaxBracket+1),"bracket buffer");
03960     }
03961     *AT.BrackBuf = 0;
03962     AR.BracketOn = 0;
03963     AC.bracketindexflag = 0;
03964     AT.bracketindexflag = 0;
03965 /*
03966     Now loop through the various levels, separated by the colons.
03967 */
03968     for ( i = 0; i < MAXMULTIBRACKETLEVELS; i++ ) {
03969         if ( *inp == 0 ) goto RegEnd;
03970 /*
03971         1: skip to ':', determine bracket or antibracket
03972 */
03973         s = inp;
03974         while ( *s && *s != ':' ) {
03975             if ( *s == '[' ) { SKIPBRA1(s) s++; }
03976             else if ( *s == '{' ) { SKIPBRA2(s) s++; }
03977             else s++;
03978         }
03979         c = *s; *s = 0;
03980         if ( StrICont(inp,(UBYTE *)"antibrackets") == 0 ) { type = 1; }
03981         else if ( StrICont(inp,(UBYTE *)"brackets") == 0 ) { type = 0; }
03982         else {
03983             MesPrint("&Illegal (anti)bracket specification in MultiBracket statement");
03984             if ( error == 0 ) error = 1;
03985             goto NextLevel;
03986         }
03987         while ( FG.cTable[*inp] == 0 ) inp++;
03988         if ( *inp != ',' ) {
03989             MesPrint("&Illegal separator after (anti)bracket specification in MultiBracket
      statement");
03990             if ( error == 0 ) error = 1;
```

```
03991             goto NextLevel;
03992         }
03993         inp++;
03994 /*
03995         2: call DoBrackets.
03996 */
03997         error1 = DoBrackets(inp, type);
03998         if ( error < 0 ) return(error1);
03999         if ( error1 > error ) error = error1;
04000 /*
04001         3: copy bracket information to the multi bracket arrays
04002 */
04003         if ( AR.BracketOn ) {
04004             num = AT.BrackBuf[0];
04005             to = AC.MultiBracketBuf[i] = (WORD *)Malloc1((num+2)*sizeof(WORD),"bracket buffer i");
04006             from = AT.BrackBuf;
04007             *to++ = AR.BracketOn;
04008             NCOPY(to,from,num);
04009             *to = 0;
04010         }
04011 /*
04012         4: set ready for the next level
04013 */
04014 NextLevel:
04015         *s = c; if ( c == ':' ) s++;
04016         inp = s;
04017         *AT.BrackBuf = 0;
04018         AR.BracketOn = 0;
04019     }
04020     if ( *inp != 0 ) {
04021         MesPrint("&More than %d levels in MultiBracket statement",(WORD)MAXMULTIBRACKETLEVELS);
04022         if ( error == 0 ) error = 1;
04023     }
04024 RegEnd:
04025     AC.MultiBracketLevels = i;
04026     *AT.BrackBuf = 0;
04027     AR.BracketOn = 0;
04028     AC.bracketindexflag = 0;
04029     AT.bracketindexflag = 0;
04030     return(error);
04031 }
04032 /* UNFINISHED_FEATURE_EXCL_STOP */
04033 /*
04034     #] CoMultiBracket :
04035     #[ CountComp :
04036
04037         This routine reads the count statement. The syntax is:
04038         count minimum,object,size[,object,size]
04039         Objects can be:
04040             symbol
04041             dotproduct
04042             vector
04043             function
04044         Vectors can have the auxiliary flags:
04045             +v +f +d +?setname
04046
04047         Output for the compiler:
04048         TYPECOUNT,size,minimum,objects
04049         with the objects:
04050         SYMBOL,4,number,size
04051         DOTPRODUCT,5,v1,v2,size
04052         FUNCTION,4,number,size
04053         VECTOR,5,number,bits,size or VECTOR,6,number,bits,setnumber,size
04054
04055         Currently only used in the if statement
04056 */
04057
04058 WORD *CountComp(UBYTE *inp, WORD *to)
04059 {
04060     GETIDENTITY
04061     UBYTE *p, c;
04062     WORD *w, mini = 0, type, c1, c2;
04063     int error = 0;
04064     p = inp;
04065     w = to;
04066     AR.Eside = 2;
04067     *w++ = TYPECOUNT;
04068     *w++ = 0;
04069     *w++ = 0;
04070     while ( *p == ',' ) {
04071         p++; inp = p;
04072         if ( *p == '[' || FG.cTable[*p] == 0 ) {
04073             if ( ( p = SkipAName(inp) ) == 0 ) return(0);
04074             c = *p; *p = 0;
04075             type = GetName(AC.varnames,inp,&c1,WITHAUTO);
04076             if ( c == '.' ) {
04077                 if ( type == CVECTOR || type == CDUBIOUS ) {
```

```
04078                       *p++ = c;
04079                       inp = p;
04080                       p = SkipAName(p);
04081                       if ( p == 0 ) return(0);
04082                       c = *p;
04083                       *p = 0;
04084                       type = GetName(AC.varnames,inp,&c2,WITHAUTO);
04085                       if ( type != CVECTOR && type != CDUBIOUS ) {
04086                           MesPrint("&Not a vector in dotproduct in if statement: %s",inp);
04087                           error = 1;
04088                       }
04089                       else type = CDOTPRODUCT;
04090                   }
04091                   else {
04092                       MesPrint("&Illegal use of . after %s in if statement",inp);
04093                       if ( type == NAMENOTFOUND )
04094                           MesPrint("&%s is not a properly declared variable",inp);
04095                       error = 1;
04096                       *p++ = c;
04097                       while ( *p && *p != ')' && *p != ',' ) p++;
04098                       if ( *p == ',' && FG.cTable[p[1]] == 1 ) {
04099                           p++;
04100                           while ( *p && *p != ')' && *p != ',' ) p++;
04101                       }
04102                       continue;
04103                   }
04104               }
04105               *p = c;
04106               switch ( type ) {
04107                   case CSYMBOL:
04108                       *w++ = SYMBOL; *w++ = 4; *w++ = c1;
04109 Sgetnum:              if ( *p != ',' ) {
04110                           MesCerr("sequence",p);
04111                           while ( *p && *p != ')' && *p != ',' ) p++;
04112                           error = 1;
04113                       }
04114                       p++; inp = p;
04115                       ParseSignedNumber(mini,p)
04116                       if ( FG.cTable[p[-1]] != 1 || ( *p && *p != ')' && *p != ',' ) ) {
04117                           while ( *p && *p != ')' && *p != ',' ) p++;
04118                           error = 1;
04119                           c = *p; *p = 0;
04120                           MesPrint("&Improper value in count: %s",inp);
04121                           *p = c;
04122                           while ( *p && *p != ')' && *p != ',' ) p++;
04123                       }
04124                       *w++ = mini;
04125                       break;
04126                   case CFUNCTION:
04127                       *w++ = FUNCTION; *w++ = 4; *w++ = c1+FUNCTION; goto Sgetnum;
04128                   case CDOTPRODUCT:
04129                       *w++ = DOTPRODUCT; *w++ = 5;
04130                       *w++ = c2 + AM.OffsetVector;
04131                       *w++ = c1 + AM.OffsetVector;
04132                       goto Sgetnum;
04133                   case CVECTOR:
04134                       *w++ = VECTOR; *w++ = 5;
04135                       *w++ = c1 + AM.OffsetVector;
04136                       if ( *p == ',' ) {
04137                           *w++ = VECTBIT | DOTPBIT | FUNBIT;
04138                           goto Sgetnum;
04139                       }
04140                       else if ( *p == '+' ) {
04141                           p++;
04142                           *w = 0;
04143                           while ( *p && *p != ',' ) {
04144                               if ( *p == 'v' || *p == 'V' ) {
04145                                   *w |= VECTBIT; p++;
04146                               }
04147                               else if ( *p == 'd' || *p == 'D' ) {
04148                                   *w |= DOTPBIT; p++;
04149                               }
04150                               else if ( *p == 'f' || *p == 'F'
04151                               || *p == 't' || *p == 'T' ) {
04152                                   *w |= FUNBIT; p++;
04153                               }
04154                               else if ( *p == '?' ) {
04155                                   p++; inp = p;
04156                                   if ( *p == '{' ) { /* } */
04157                                       SKIPBRA2(p)
04158                                       if ( p == 0 ) return(0);
04159                                       if ( ( c1 = DoTempSet(inp+1,p) ) < 0 ) return(0);
04160                                       if ( Sets[c1].type != CFUNCTION ) {
04161                                           MesPrint("&set type conflict: Function expected");
04162                                           return(0);
04163                                       }
04164                                       type = CSET;
```

```
04165                                   c = *++p;
04166                               }
04167                               else {
04168                                   p = SkipAName(p);
04169                                   if ( p == 0 ) return(0);
04170                                   c = *p; *p = 0;
04171                                   type = GetName(AC.varnames,inp,&c1,WITHAUTO);
04172                               }
04173                               if ( type != CSET && type != CDUBIOUS ) {
04174                                   MesPrint("&%s is not a set",inp);
04175                                   error = 1;
04176                               }
04177                               w[-2] = 6;
04178                               *w++ |= SETBIT;
04179                               *w++ = c1;
04180                               *p = c;
04181                               goto Sgetnum;
04182                           }
04183                           else {
04184                               MesCerr("specifier for vector",p);
04185                               error = 1;
04186                           }
04187                       }
04188                       w++;
04189                       goto Sgetnum;
04190                   }
04191                   else {
04192                       MesCerr("specifier for vector",p);
04193                       while ( *p && *p != ')' && *p != ',' ) p++;
04194                       error = 1;
04195                       *w++ = VECTBIT | DOTPBIT | FUNBIT;
04196                       goto Sgetnum;
04197                   }
04198               case CDUBIOUS:
04199                   goto skipfield;
04200               default:
04201                   *p = 0;
04202                   MesPrint("&%s is not a symbol, function, vector or dotproduct",inp);
04203                   error = 1;
04204 skipfield:        while ( *p && *p != ')' && *p != ',' ) p++;
04205                   if ( *p && FG.cTable[p[1]] == 1 ) {
04206                       p++;
04207                       while ( *p && *p != ')' && *p != ',' ) p++;
04208                   }
04209                   break;
04210           }
04211       }
04212       else {
04213           MesCerr("name",p);
04214           while ( *p && *p != ',' ) p++;
04215           error = 1;
04216       }
04217   }
04218   to[1] = w-to;
04219   if ( *p == ')' ) p++;
04220   if ( *p ) { MesCerr("end of statement",p); return(0); }
04221   if ( error ) return(0);
04222   return(w);
04223 }
04224
04225 /*
04226     #] CountComp :
04227     #[ CoIf :
04228
04229         Reads the if statement: There must be a pair of parentheses.
04230         Much work is delegated to the routines in compi2 and CountComp.
04231         The goto is kept hanging as it is forward.
04232         The address in which the label must be written is pushed on
04233         the AC.IfStack.
04234
04235         Here we allow statements of the type
04236         if ( condition ) single statement;
04237         compile the if statement.
04238         test character at end
04239         if not ; or )
04240         copy the statement after the proper parenthesis to the
04241         beginning of the AC.iBuffer.
04242         Have it compiled.
04243         generate an endif statement.
04244 */
04245
04246 static UWORD *CIscratC = 0;
04247
04248 int CoIf(UBYTE *inp)
04249 {
04250     GETIDENTITY
04251     int error = 0, level;
```

```
04252     WORD *w, *ww, *u, *s, *OldWork, *OldSpace = AT.WorkSpace;
04253     WORD gotexp = 0;        /* Indicates whether there can be a condition */
04254     WORD lenpp, lenlev, ncoef, i, number;
04255     UBYTE *p, *pp, *ppp, c;
04256     CBUF *C = cbuf+AC.cbufnum;
04257     LONG x;
04258 #ifdef WITHFLOAT
04259 /* UNFINISHED_FEATURE_EXCL_START */
04260     int spec;
04261 /* UNFINISHED_FEATURE_EXCL_STOP */
04262 #endif
04263     if ( *inp == '(' && inp[1] == ',' ) inp += 2;
04264     else if ( *inp == '(' ) inp++;  /* Usually we enter at the bracket */
04265
04266     if ( CIscratC == 0 )
04267         CIscratC = (UWORD *)Malloc1((AM.MaxTal+2)*sizeof(UWORD),"CoIf");
04268     lenpp = 0;
04269     lenlev = 1;
04270     if ( AC.IfLevel >= AC.MaxIf ) DoubleIfBuffers();
04271     AC.IfCount[lenpp++] = 0;
04272 /*
04273     IfStack is used for organizing the 'go to' for the various if levels
04274 */
04275     *AC.IfStack++ = C->Pointer-C->Buffer+2;
04276 /*
04277     IfSumCheck is used to test for illegal nesting of if, argument or repeat.
04278 */
04279     AC.IfSumCheck[AC.IfLevel] = NestingChecksum();
04280     AC.IfLevel++;
04281     w = OldWork = AT.WorkPointer;
04282     *w++ = TYPEIF;
04283     w += 2;
04284     p = inp;
04285     for(;;) {
04286         inp = p;
04287         level = 0;
04288 ReDo:
04289         if ( FG.cTable[*p] == 1 ) {     /* Number */
04290             if ( gotexp == 1 ) { MesCerr("position for )",p); error = 1; }
04291 #ifdef WITHFLOAT
04292 /* UNFINISHED_FEATURE_EXCL_START */
04293             pp = CheckFloat(p,&spec);
04294             if ( pp > p ) { /* Got one */
04295 HaveFloat:
04296                 if ( spec == -1 ) {
04297                     MesPrint("&The floating point system has not been started: %s",p);
04298                     if ( !error ) error = 1;
04299                 }
04300                 else {
04301                     WORD *ow = AT.WorkPointer;
04302                     AT.WorkPointer = w;
04303                     c = *pp; *pp = 0;
04304                     ReadFloat((SBYTE *)p);  /* Is now at AT.WorkPointer */
04305                     *pp = c;
04306                     p = pp;
04307                     AT.WorkPointer[0] = IFFLOATNUMBER;
04308                     w = AT.WorkPointer + AT.WorkPointer[1];
04309                     AT.WorkPointer = ow;
04310                     if ( level ) w[FUNHEAD+3] = -w[FUNHEAD+3];
04311                 }
04312                 goto DoneWithNumber;
04313             }
04314 /*
04315             Notation: Same as FLOATFUN but FLOATFUN replaced by IFFLOATNUMBER.
04316 */
04317 /* UNFINISHED_FEATURE_EXCL_STOP */
04318 #endif
04319             u = w;
04320             *w++ = LONGNUMBER;
04321 /*
04322             Notation:
04323             LONGNUMBER,size,reducedsize*sign,numerator,denominator
04324             with the length of denominator and numerator equal to reducedsize
04325 */
04326             w += 2;
04327             if ( GetLong(p,(UWORD *)w,&ncoef) ) { ncoef = 1; error = 1; }
04328             w[-1] = ncoef;
04329             while ( FG.cTable[*++p] == 1 );
04330             if ( *p == '/' ) {
04331                 p++;
04332                 if ( FG.cTable[*p] != 1 ) {
04333                     MesCerr("sequence",p); error = 1; goto OnlyNum;
04334                 }
04335                 if ( GetLong(p,CIscratC,&ncoef) ) {
04336                     ncoef = 1; error = 1;
04337                 }
04338                 while ( FG.cTable[*++p] == 1 );
```

```
04339                 if ( ncoef == 0 ) {
04340                     MesPrint("&Division by zero!");
04341                     error = 1;
04342                 }
04343                 else {
04344                     if ( w[-1] != 0 ) {
04345                         if ( Simplify(BHEAD (UWORD *)w,(WORD *)(w-1),
04346                         CIscratC,&ncoef) ) error = 1;
04347                         else {
04348                             i = w[-1];
04349                             if ( i >= ncoef ) {
04350                                 i = w[-1];
04351                                 w += i;
04352                                 i -= ncoef;
04353                                 s = (WORD *)CIscratC;
04354                                 NCOPY(w,s,ncoef);
04355                                 while ( --i >= 0 ) *w++ = 0;
04356                             }
04357                             else {
04358                                 w += i;
04359                                 i = ncoef - i;
04360                                 while ( --i >= 0 ) *w++ = 0;
04361                                 s = (WORD *)CIscratC;
04362                                 NCOPY(w,s,ncoef);
04363                             }
04364                         }
04365                     }
04366                 }
04367             }
04368             else {
04369 OnlyNum:
04370                 w += ncoef;
04371                 if ( ncoef > 0 ) {
04372                     ncoef--; *w++ = 1;
04373                     while ( --ncoef >= 0 ) *w++ = 0;
04374                 }
04375             }
04376             u[1] = WORDDIF(w,u);
04377             u[2] = (u[1] - 3)/2;
04378             if ( level ) u[2] = -u[2];
04379 #ifdef WITHFLOAT
04380 /* UNFINISHED_FEATURE_EXCL_START */
04381 DoneWithNumber:
04382 /* UNFINISHED_FEATURE_EXCL_STOP */
04383 #endif
04384             gotexp = 1;
04385         }
04386         else if ( *p == '+' ) { p++; goto ReDo; }
04387         else if ( *p == '-' ) { level ^= 1; p++; goto ReDo; }
04388         else if ( *p == 'c' || *p == 'C' ) {    /* Count or Coefficient */
04389             if ( gotexp == 1 ) { MesCerr("position for )",p); error = 1; }
04390             while ( FG.cTable[*++p] == 0 );
04391             c = *p; *p = 0;
04392             if ( !StrICmp(inp,(UBYTE *)"count") ) {
04393                 *p = c;
04394                 if ( c != '(' ) {
04395                     MesPrint("&no ( after count");
04396                     error = 1;
04397                     goto endofif;
04398                 }
04399                 inp = p;
04400                 SKIPBRA4(p);
04401                 c = *++p; *p = 0; *inp = ',';
04402                 w = CountComp(inp,w);
04403                 *p = c; *inp = '(';
04404                 if ( w == 0 ) { error = 1; goto endofif; }
04405                 gotexp = 1;
04406             }
04407             else if ( ConWord(inp,(UBYTE *)"coefficient") && ( p - inp ) > 3 ) {
04408                 *w++ = COEFFI;
04409                 *w++ = 2;
04410                 *p = c;
04411                 gotexp = 1;
04412             }
04413             else goto NoGood;
04414             inp = p;
04415         }
04416         else if ( *p == 'm' || *p == 'M' ) {    /* match */
04417             if ( gotexp == 1 ) { MesCerr("position for )",p); error = 1; }
04418             while ( !FG.cTable[*++p] );
04419             c = *p; *p = 0;
04420             if ( !StrICmp(inp,(UBYTE *)"match") ) {
04421                 *p = c;
04422                 if ( c != '(' ) {
04423                     MesPrint("&no ( after match");
04424                     error = 1;
04425                     goto endofif;
```

```
04426                 }
04427                 p++; inp = p;
04428                 SKIPBRA4(p);
04429                 *p = '=';
04430 /*
04431                 Now we can call the reading of the lhs of an id statement.
04432                 This has to be modified in the future.
04433 */
04434                 AT.WorkSpace = AT.WorkPointer = w;
04435                 ppp = inp;
04436                 while ( FG.cTable[*ppp] == 0 && ppp < p ) ppp++;
04437                 if ( *ppp == ',' ) AC.idoption = 0;
04438                 else AC.idoption = SUBMULTI;
04439                 level = CoIdExpression(inp,TYPEIF);
04440                 AT.WorkSpace = OldSpace;
04441                 AT.WorkPointer = OldWork;
04442                 if ( level != 0 ) {
04443                     if ( level < 0 ) { error = -1; goto endofif; }
04444                     error = 1;
04445                 }
04446 /*
04447                 If we pop numlhs we are in good shape
04448 */
04449                 s = u = C->lhs[C->numlhs];
04450                 while ( u < C->Pointer ) *w++ = *u++;
04451                 C->numlhs--; C->Pointer = s;
04452                 *p++ = ')';
04453                 inp = p;
04454                 gotexp = 1;
04455             }
04456             else if ( !StrICmp(inp,(UBYTE *)"multipleof") ) {
04457                 if ( gotexp == 1 ) { MesCerr("position for )",p); error = 1; }
04458                 *p = c;
04459                 if ( c != '(' ) {
04460                     MesPrint("&no ( after multipleof");
04461                     error = 1; goto endofif;
04462                 }
04463                 p++;
04464                 if ( FG.cTable[*p] != 1 ) {
04465 Nomulof:            MesPrint("&multipleof needs a short positive integer argument");
04466                     error = 1; goto endofif;
04467                 }
04468                 ParseNumber(x,p)
04469                 if ( *p != ')' || x <= 0 || x > MAXPOSITIVE ) goto Nomulof;
04470                 p++;
04471                 *w++ = MULTIPLEOF; *w++ = 3; *w++ = (WORD)x;
04472                 inp = p;
04473                 gotexp = 1;
04474             }
04475             else {
04476 NoGood:         MesPrint("&Unrecognized word: %s",inp);
04477                 *p = c;
04478                 error = 1;
04479                 level = 0;
04480                 if ( c == '(' ) SKIPBRA4(p)
04481                 inp = ++p;
04482                 gotexp = 1;
04483             }
04484         }
04485         else if ( *p == 'f' || *p == 'F' ) {    /* FindLoop */
04486             if ( gotexp == 1 ) { MesCerr("position for )",p); error = 1; }
04487             while ( FG.cTable[*++p] == 0 );
04488             c = *p; *p = 0;
04489             if ( !StrICmp(inp,(UBYTE *)"findloop") ) {
04490                 *p = c;
04491                 if ( c != '(' ) {
04492                     MesPrint("&no ( after findloop");
04493                     error = 1;
04494                     goto endofif;
04495                 }
04496                 inp = p;
04497                 SKIPBRA4(p);
04498                 c = *++p; *p = 0; *inp = ',';
04499                 if ( CoFindLoop(inp) ) { error = 1; goto endofif; }
04500                 s = u = C->lhs[C->numlhs];
04501                 while ( u < C->Pointer ) *w++ = *u++;
04502                 C->numlhs--; C->Pointer = s;
04503                 *p = c; *inp = '(';
04504                 if ( w == 0 ) { error = 1; goto endofif; }
04505                 gotexp = 1;
04506             }
04507             else if ( !StrICmp(inp,(UBYTE *)"flag") ) {
04508                 UBYTE cc = c, *pppp;
04509                 *p = cc;
04510                 if ( cc != '(' ) {
04511                     MesPrint("&no ( after flag");
04512                     error = 1;
```

```
04513                   goto endofif;
04514               }
04515               inp = p;
04516               SKIPBRA4(p);
04517               cc = *++p; *p = 0; *inp = ',';  pppp = p;
04518               ww = w;
04519               *w++ = IFUSERFLAG; *w++ = 0;
04520               while ( *inp ) {
04521                   int x = 0;
04522                   while ( *inp == ',' ) inp++;
04523                   if ( *inp == 0 || *inp == ')' ) break;
04524                   while ( *inp >= '0' && *inp <= '9' ) x = 10*x+(*inp++-'0');
04525                   if ( x < 1 || x > BITSINWORD ) {
04526                       MesPrint("&Flag number %d outside the permitted range 1-%d.",BITSINWORD);
04527                       error = 1;
04528                   }
04529                   *w++ = x-1;
04530               }
04531               ww[1] = w-ww;
04532               p = pppp; *p = cc; *inp = '(';
04533               gotexp = 1;
04534               if ( ww[1] <= 2 ) {
04535                   MesPrint("&The userflag condition in the if statement needs arguments.");
04536                   error = 1;
04537               }
04538               inp = p;
04539               gotexp = 1;
04540           }
04541           else goto NoGood;
04542       }
04543       else if ( *p == 'e' || *p == 'E' ) { /* Expression */
04544           if ( gotexp == 1 ) { MesCerr("position for )",p); error = 1; }
04545           while ( FG.cTable[*++p] == 0 );
04546           c = *p; *p = 0;
04547           if ( !StrICmp(inp,(UBYTE *)"expression") ) {
04548               *p = c;
04549               if ( c != '(' ) {
04550                   MesPrint("&no ( after expression");
04551                   error = 1;
04552                   goto endofif;
04553               }
04554               p++; ww = w; *w++ = IFEXPRESSION; w++;
04555               while ( *p != ')' ) {
04556                   if ( *p == ',' ) { p++; continue; }
04557                   if ( *p == '[' || FG.cTable[*p] == 0 ) {
04558                       pp = p;
04559                       if ( ( p = SkipAName(p) ) == 0 ) {
04560                           MesPrint("&Improper name for an expression: '%s'",pp);
04561                           error = 1;
04562                           goto endofif;
04563                       }
04564                       c = *p; *p = 0;
04565                       if ( GetName(AC.exprnames,pp,&number,NOAUTO) == CEXPRESSION ) {
04566                           *w++ = number;
04567                       }
04568                       else if ( GetName(AC.varnames,pp,&number,NOAUTO) != NAMENOTFOUND ) {
04569                           MesPrint("&%s is not an expression",pp);
04570                           error = 1;
04571                           *w++ = number;
04572                       }
04573                       *p = c;
04574                   }
04575                   else {
04576                       MesPrint("&Illegal object in Expression in if-statement");
04577                       error = 1;
04578                       while ( *p && *p != ',' && *p != ')' ) p++;
04579                       if ( *p == 0 || *p == ')' ) break;
04580                   }
04581               }
04582               ww[1] = w - ww;
04583               p++;
04584               gotexp = 1;
04585           }
04586           else goto NoGood;
04587           inp = p;
04588       }
04589       else if ( *p == 'i' || *p == 'I' ) { /* IsFactorized */
04590           if ( gotexp == 1 ) { MesCerr("position for )",p); error = 1; }
04591           while ( FG.cTable[*++p] == 0 );
04592           c = *p; *p = 0;
04593           if ( !StrICmp(inp,(UBYTE *)"isfactorized") ) {
04594               *p = c;
04595               if ( c != '(' ) { /* No expression means current expression */
04596                 ww = w; *w++ = IFISFACTORIZED; w++;
04597               }
04598               else {
04599                 p++; ww = w; *w++ = IFISFACTORIZED; w++;
```

```
04600                   while ( *p != ')' ) {
04601                     if ( *p == ',' ) { p++; continue; }
04602                     if ( *p == '[' || FG.cTable[*p] == 0 ) {
04603                         pp = p;
04604                         if ( ( p = SkipAName(p) ) == 0 ) {
04605                             MesPrint("&Improper name for an expression: '%s'",pp);
04606                             error = 1;
04607                             goto endofif;
04608                         }
04609                         c = *p; *p = 0;
04610                         if ( GetName(AC.exprnames,pp,&number,NOAUTO) == CEXPRESSION ) {
04611                             *w++ = number;
04612                         }
04613                         else if ( GetName(AC.varnames,pp,&number,NOAUTO) != NAMENOTFOUND ) {
04614                             MesPrint("&%s is not an expression",pp);
04615                             error = 1;
04616                             *w++ = number;
04617                         }
04618                         *p = c;
04619                     }
04620                     else {
04621                         MesPrint("&Illegal object in IsFactorized in if-statement");
04622                         error = 1;
04623                         while ( *p && *p != ',' && *p != ')' ) p++;
04624                         if ( *p == 0 || *p == ')' ) break;
04625                     }
04626                   }
04627                   p++;
04628                 }
04629                 ww[1] = w - ww;
04630                 gotexp = 1;
04631             }
04632             else goto NoGood;
04633             inp = p;
04634         }
04635         else if ( *p == 'o' || *p == 'O' ) { /* Occurs */
04636 /*
04637             Tests whether variables occur inside a term.
04638             At the moment this is done one by one.
04639             If we want to do them in groups we should do the reading
04640             a bit different: each as a variable in a term, and then
04641             use Normalize to get the variables grouped and in order.
04642             That way FindVar (in if.c) can work more efficiently.
04643             Still to be done!!!
04644             TASK: Nice little task for someone to learn.
04645 */
04646             UBYTE cc;
04647             if ( gotexp == 1 ) { MesCerr("position for )",p); error = 1; }
04648             while ( FG.cTable[*++p] == 0 );
04649             c = cc = *p; *p = 0;
04650             if ( !StrICmp(inp,(UBYTE *)"occurs") ) {
04651                 WORD c1, c2, type;
04652                 *p = cc;
04653                 if ( cc != '(' ) {
04654                     MesPrint("&no ( after occurs");
04655                     error = 1;
04656                     goto endofif;
04657                 }
04658                 inp = p;
04659                 SKIPBRA4(p);
04660                 cc = *++p; *p = 0; *inp = ','; pp = p;
04661                 ww = w;
04662                 *w++ = IFOCCURS; *w++ = 0;
04663                 while ( *inp ) {
04664                     while ( *inp == ',' ) inp++;
04665                     if ( *inp == 0 || *inp == ')' ) break;
04666 /*
04667                     Now read a list of names
04668                     We can have symbols, vectors, dotproducts, indices, functions.
04669                     There could also be dummy indices and/or extra symbols.
04670 */
04671                     if ( *inp == '[' || FG.cTable[*inp] == 0 ) {
04672                         if ( ( p = SkipAName(inp) ) == 0 ) return(0);
04673                         c = *p; *p = 0;
04674                         type = GetName(AC.varnames,inp,&c1,WITHAUTO);
04675                         if ( c == '.' ) {
04676                             if ( type == CVECTOR || type == CDUBIOUS ) {
04677                                 *p++ = c;
04678                                 inp = p;
04679                                 p = SkipAName(p);
04680                                 if ( p == 0 ) return(0);
04681                                 c = *p;
04682                                 *p = 0;
04683                                 type = GetName(AC.varnames,inp,&c2,WITHAUTO);
04684                                 if ( type != CVECTOR && type != CDUBIOUS ) {
04685                                     MesPrint("&Not a vector in dotproduct in if statement: %s",inp);
04686                                     error = 1;
```

```
04687                                   }
04688                                   else type = CDOTPRODUCT;
04689                               }
04690                               else {
04691                                   MesPrint("&Illegal use of . after %s in if statement",inp);
04692                                   if ( type == NAMENOTFOUND )
04693                                       MesPrint("&%s is not a properly declared variable",inp);
04694                                   error = 1;
04695                                   *p++ = c;
04696                                   while ( *p && *p != ')' && *p != ',' ) p++;
04697                                   if ( *p == ',' && FG.cTable[p[1]] == 1 ) {
04698                                       p++;
04699                                       while ( *p && *p != ')' && *p != ',' ) p++;
04700                                   }
04701                                   continue;
04702                               }
04703                           }
04704                           *p = c;
04705                           switch ( type ) {
04706                               case CSYMBOL: /* To worry about extra symbols */
04707                                   *w++ = SYMBOL;
04708                                   *w++ = c1;
04709                               break;
04710                               case CINDEX:
04711                                   *w++ = INDEX;
04712                                   *w++ = c1 + AM.OffsetIndex;
04713                               break;
04714                               case CVECTOR:
04715                                   *w++ = VECTOR;
04716                                   *w++ = c1 + AM.OffsetVector;
04717                               break;
04718                               case CDOTPRODUCT:
04719                                   *w++ = DOTPRODUCT;
04720                                   *w++ = c1 + AM.OffsetVector;
04721                                   *w++ = c2 + AM.OffsetVector;
04722                               break;
04723                               case CFUNCTION:
04724                                   *w++ = FUNCTION;
04725                                   *w++ = c1+FUNCTION;
04726                               break;
04727                               default:
04728                                   MesPrint("&Illegal variable %s in occurs condition in if
      statement",inp);
04729                                   error = 1;
04730                               break;
04731                           }
04732                           inp = p;
04733                       }
04734                       else {
04735                           MesPrint("&Illegal object %s in occurs condition in if statement",inp);
04736                           error = 1;
04737                           break;
04738                       }
04739                   }
04740                   ww[1] = w-ww;
04741                   p = pp; *p = cc; *inp = '(';
04742                   gotexp = 1;
04743                   if ( ww[1] <= 2 ) {
04744                       MesPrint("&The occurs condition in the if statement needs arguments.");
04745                       error = 1;
04746                   }
04747               }
04748               else goto NoGood;
04749               inp = p;
04750           }
04751           else if ( *p == '$' ) {
04752               if ( gotexp == 1 ) { MesCerr("position for )",p); error = 1; }
04753               p++; inp = p;
04754               while ( FG.cTable[*p] == 0 || FG.cTable[*p] == 1 ) p++;
04755               c = *p; *p = 0;
04756               if ( ( i = GetDollar(inp) ) < 0 ) {
04757                   MesPrint("&undefined dollar expression %s",inp);
04758                   error = 1;
04759                   i = AddDollar(inp,DOLUNDEFINED,0,0);
04760               }
04761               *p = c;
04762               *w++ = IFDOLLAR; *w++ = 3; *w++ = i;
04763 /*
04764               And then the IFDOLLAREXTRA pieces for [1] [$y] etc
04765 */
04766               if ( *p == '[' ) {
04767                   p++;
04768                   if ( ( w = GetIfDollarFactor(&p,w) ) == 0 ) {
04769                       error = 1;
04770                       goto endofif;
04771                   }
04772                   else if ( *p != ']' ) {
```

```
04773                       error = 1;
04774                       goto endofif;
04775                   }
04776                   p++;
04777               }
04778               inp = p;
04779               gotexp = 1;
04780           }
04781 #ifdef WITHFLOAT
04782 /* UNFINISHED_FEATURE_EXCL_START */
04783           else if ( *p == '.' ) {
04784               pp = CheckFloat(p,&spec);
04785               if ( pp > p ) goto HaveFloat;
04786           }
04787 /* UNFINISHED_FEATURE_EXCL_STOP */
04788 #endif
04789           else if ( *p == '(' ) {
04790               if ( gotexp ) {
04791                   MesCerr("parenthesis",p);
04792                   error = 1;
04793                   goto endofif;
04794               }
04795               gotexp = 0;
04796               if ( ++lenlev >= AC.MaxIf ) DoubleIfBuffers();
04797               AC.IfCount[lenpp++] = w-OldWork;
04798               *w++ = SUBEXPR;
04799               w += 2;
04800               p++;
04801           }
04802           else if ( *p == ')' ) {
04803               if ( gotexp == 0 ) { MesCerr("position for )",p); error = 1; }
04804               gotexp = 1;
04805               u = AC.IfCount[--lenpp]+OldWork;
04806               lenlev--;
04807               u[1] = w - u;
04808               if ( lenlev <= 0 ) {    /* End if condition */
04809                   AT.WorkSpace = OldSpace;
04810                   AT.WorkPointer = OldWork;
04811                   AddNtoL(OldWork[1],OldWork);
04812                   p++;
04813                   if ( *p == ')' ) {
04814                       MesPrint("&unmatched parenthesis in if/while ()");
04815                       error = 1;
04816                       while ( *++p == ')' );
04817                   }
04818                   if ( *p ) {
04819                       level = CompileStatement(p);
04820                       if ( level ) error = level;
04821                       while ( *p ) p++;
04822                       if ( CoEndIf(p) && error == 0 ) error = 1;
04823                   }
04824                   return(error);
04825               }
04826               p++;
04827           }
04828           else if ( *p == '>' ) {
04829               if ( gotexp == 0 ) goto NoExp;
04830               if ( p[1] == '=' ) { *w++ = GREATEREQUAL; *w++ = 2; p += 2; }
04831               else               { *w++ = GREATER;      *w++ = 2; p++; }
04832               gotexp = 0;
04833           }
04834           else if ( *p == '<' ) {
04835               if ( gotexp == 0 ) goto NoExp;
04836               if ( p[1] == '=' ) { *w++ = LESSEQUAL; *w++ = 2; p += 2; }
04837               else               { *w++ = LESS;      *w++ = 2; p++; }
04838               gotexp = 0;
04839           }
04840           else if ( *p == '=' ) {
04841               if ( gotexp == 0 ) goto NoExp;
04842               if ( p[1] == '=' ) p++;
04843               *w++ = EQUAL; *w++ = 2; p++;
04844               gotexp = 0;
04845           }
04846           else if ( *p == '!' && p[1] == '=' ) {
04847               if ( gotexp == 0 ) { p++; goto NoExp; }
04848               *w++ = NOTEQUAL; *w++ = 2; p += 2;
04849               gotexp = 0;
04850           }
04851           else if ( *p == '|' && p[1] == '|' ) {
04852               if ( gotexp == 0 ) { p++; goto NoExp; }
04853               *w++ = ORCOND; *w++ = 2; p += 2;
04854               gotexp = 0;
04855           }
04856           else if ( *p == '&' && p[1] == '&' ) {
04857               if ( gotexp == 0 ) {
04858                   p++;
04859 NoExp:            p++;
```

```
04860                 MesCerr("sequence",p);
04861                 error = 1;
04862             }
04863             else {
04864                 *w++ = ANDCOND; *w++ = 2; p += 2;
04865                 gotexp = 0;
04866             }
04867         }
04868         else if ( *p == 0 ) {
04869             MesPrint("&Unmatched parentheses");
04870             error = 1;
04871             goto endofif;
04872         }
04873         else {
04874             if ( FG.cTable[*p] == 0 ) {
04875                 WORD ij;
04876                 inp = p;
04877                 while ( ( ij = FG.cTable[*++p] ) == 0 || ij == 1 );
04878                 c = *p; *p = 0;
04879                 goto NoGood;
04880             }
04881             MesCerr("sequence",p);
04882             error = 1;
04883             p++;
04884         }
04885     }
04886 endofif:;
04887     return(error);
04888 }
04889
04890 /*
04891     #] CoIf :
04892     #[ CoElse :
04893 */
04894
04895 int CoElse(UBYTE *p)
04896 {
04897     int error = 0;
04898     CBUF *C = cbuf+AC.cbufnum;
04899     if ( *p != 0 ) {
04900         while ( *p == ',' ) p++;
04901         if ( tolower(*p) == 'i' && tolower(p[1]) == 'f' && p[2] == '(' )
04902                                                 return(CoElseIf(p+2));
04903         MesPrint("&No extra text allowed as part of an else statement");
04904         error = 1;
04905     }
04906     if ( AC.IfLevel <= 0 ) { MesPrint("&else statement without if"); return(1); }
04907     if ( AC.IfSumCheck[AC.IfLevel-1] != NestingChecksum() - 1 ) {
04908         MesNesting();
04909         error = 1;
04910     }
04911     Add3Com(TYPEELSE,AC.IfLevel)
04912     C->Buffer[AC.IfStack[-1]] = C->numlhs;
04913     AC.IfStack[-1] = C->Pointer - C->Buffer - 1;
04914     return(error);
04915 }
04916
04917 /*
04918     #] CoElse :
04919     #[ CoElseIf :
04920 */
04921
04922 int CoElseIf(UBYTE *inp)
04923 {
04924     CBUF *C = cbuf+AC.cbufnum;
04925     if ( AC.IfLevel <= 0 ) { MesPrint("&elseif statement without if"); return(1); }
04926     Add3Com(TYPEELSE,-AC.IfLevel)
04927     AC.IfLevel--;
04928     C->Buffer[*--AC.IfStack] = C->numlhs;
04929     return(CoIf(inp));
04930 }
04931
04932 /*
04933     #] CoElseIf :
04934     #[ CoEndIf :
04935
04936         It puts a RHS-level at the position indicated in the AC.IfStack.
04937         This corresponds to the label belonging to a forward goto.
04938         It is the goto that belongs either to the failing condition
04939         of the if (no else statement), or the completion of the
04940         success path (with else statement)
04941         The code is a jump to the next statement. It is there to prevent
04942         problems with
04943         if ( .. )
04944             if ( .. )
04945             endif;
04946         elseif ( .. )
```

```
04947 */
04948
04949 int CoEndIf(UBYTE *inp)
04950 {
04951     CBUF *C = cbuf+AC.cbufnum;
04952     WORD i = C->numlhs, to, k = -AC.IfLevel;
04953     int error = 0;
04954     while ( *inp == ',' ) inp++;
04955     if ( *inp != 0 ) {
04956         error = 1;
04957         MesPrint("&No extra text allowed as part of an endif/elseif statement");
04958     }
04959     if ( AC.IfLevel <= 0 ) {
04960         MesPrint("&Endif statement without corresponding if"); return(1);
04961     }
04962     AC.IfLevel--;
04963     C->Buffer[*--AC.IfStack] = i+1;
04964     if ( AC.IfSumCheck[AC.IfLevel] != NestingChecksum() ) {
04965         MesNesting();
04966         error = 1;
04967     }
04968     Add3Com(TYPEENDIF,i+1)
04969 /*
04970     Now the search for the TYPEELSE in front of the elseif statements
04971 */
04972     to = C->numlhs;
04973     while ( i > 0 ) {
04974         if ( C->lhs[i][0] == TYPEELSE && C->lhs[i][2] == to ) to = i;
04975         if ( C->lhs[i][0] == TYPEIF ) {
04976             if ( C->lhs[i][2] == to ) {
04977                 i--;
04978                 if ( i <= 0 || C->lhs[i][0] != TYPEELSE
04979                 || C->lhs[i][2] != k ) break;
04980                 C->lhs[i][2] = C->numlhs;
04981                 to = i;
04982             }
04983         }
04984         i--;
04985     }
04986     return(error);
04987 }
04988
04989 /*
04990     #] CoEndIf :
04991     #[ CoWhile :
04992 */
04993
04994 int CoWhile(UBYTE *inp)
04995 {
04996     CBUF *C = cbuf+AC.cbufnum;
04997     WORD startnum = C->numlhs + 1;
04998     int error;
04999     AC.WhileLevel++;
05000     error = CoIf(inp);
05001     if ( C->numlhs > startnum && C->lhs[startnum][2] == C->numlhs
05002                               && C->lhs[C->numlhs][0] == TYPEENDIF ) {
05003         C->lhs[C->numlhs][2] = startnum-1;
05004         AC.WhileLevel--;
05005     }
05006     else C->lhs[startnum][2] = startnum;
05007     return(error);
05008 }
05009
05010 /*
05011     #] CoWhile :
05012     #[ CoEndWhile :
05013 */
05014
05015 int CoEndWhile(UBYTE *inp)
05016 {
05017     int error = 0;
05018     WORD i;
05019     CBUF *C = cbuf+AC.cbufnum;
05020     if ( AC.WhileLevel <= 0 ) {
05021         MesPrint("&EndWhile statement without corresponding While"); return(1);
05022     }
05023     AC.WhileLevel--;
05024     i = C->Buffer[AC.IfStack[-1]];
05025     error = CoEndIf(inp);
05026     C->lhs[C->numlhs][2] = i - 1;
05027     return(error);
05028 }
05029
05030 /*
05031     #] CoEndWhile :
05032     #[ DoFindLoop :
05033
```

```
05034     Function,arguments=number,loopsize=number,outfun=function,include=index;
05035 */
05036
05037 static char *messfind[] = {
05038     "Findloop(function,arguments=#,loopsize(=#|<#)[,include=index])"
05039    ,"Replaceloop,function,arguments=#,loopsize(=#|<#),outfun=function[,include=index]"
05040     };
05041 static WORD comfindloop[7] = { TYPEFINDLOOP,7,0,0,0,0,0 };
05042
05043 int DoFindLoop(UBYTE *inp, int mode)
05044 {
05045     UBYTE *s, c;
05046     WORD funnum, nargs = 0, nloop = 0, indexnum = 0, outfun = 0;
05047     int type, aflag, lflag, indflag, outflag, error = 0, sym;
05048     while ( *inp == ',' ) inp++;
05049     if ( ( s = SkipAName(inp) ) == 0 ) {
05050 syntax:
05051         MesPrint("&Proper syntax is:");
05052         MesPrint("%s",messfind[mode]);
05053         return(1);
05054     }
05055     c = *s; *s = 0;
05056     if ( ( ( type = GetName(AC.varnames,inp,&funnum,WITHAUTO) ) == NAMENOTFOUND )
05057         || type != CFUNCTION || ( ( sym = (functions[funnum].symmetric) & ~REVERSEORDER )
05058         != SYMMETRIC && sym != ANTISYMMETRIC ) ) {
05059         MesPrint("&%s should be a (anti)symmetric function or tensor",inp);
05060         error = 1;
05061     }
05062     funnum += FUNCTION;
05063     *s = c; inp = s;
05064     aflag = lflag = indflag = outflag = 0;
05065     while ( *inp == ',' ) {
05066         while ( *inp == ',' ) inp++;
05067         s = inp;
05068         if ( ( s = SkipAName(inp) ) == 0 ) goto syntax;
05069         c = *s; *s = 0;
05070         if ( StrICont(inp,(UBYTE *)"arguments") == 0 ) {
05071             if ( c != '=' ) goto syntax;
05072             *s++ = c;
05073             NeedNumber(nargs,s,syntax)
05074             aflag++;
05075             inp = s;
05076         }
05077         else if ( StrICont(inp,(UBYTE *)"loopsize") == 0 ) {
05078             if ( c != '=' && c != '<' ) goto syntax;
05079             *s++ = c;
05080             if ( FG.cTable[*s] == 1 ) {
05081                 NeedNumber(nloop,s,syntax)
05082                 if ( nloop < 2 ) {
05083                     MesPrint("&loopsize should be at least 2");
05084                     error = 1;
05085                 }
05086                 if ( c == '<' ) nloop = -nloop;
05087             }
05088             else if ( tolower(*s) == 'a' && tolower(s[1]) == 'l'
05089             && tolower(s[2]) == 'l' && FG.cTable[s[3]] > 1 ) {
05090                 nloop = -1; s += 3;
05091                 if ( c != '=' ) goto syntax;
05092             }
05093             inp = s;
05094             lflag++;
05095         }
05096         else if ( StrICont(inp,(UBYTE *)"include") == 0 ) {
05097             if ( c != '=' ) goto syntax;
05098             *s++ = c;
05099             if ( ( inp = SkipAName(s) ) == 0 ) goto syntax;
05100             c = *inp; *inp = 0;
05101             if ( ( type = GetName(AC.varnames,s,&indexnum,WITHAUTO) ) != CINDEX ) {
05102                 MesPrint("&%s is not a proper index",s);
05103                 error = 1;
05104             }
05105             else if ( indexnum < WILDOFFSET
05106             && indices[indexnum].dimension == 0 ) {
05107                 MesPrint("&%s should be a summable index",s);
05108                 error = 1;
05109             }
05110             indexnum += AM.OffsetIndex;
05111             *inp = c;
05112             indflag++;
05113         }
05114         else if ( StrICont(inp,(UBYTE *)"outfun") == 0 ) {
05115             if ( c != '=' ) goto syntax;
05116             *s++ = c;
05117             if ( ( inp = SkipAName(s) ) == 0 ) goto syntax;
05118             c = *inp; *inp = 0;
05119             if ( ( type = GetName(AC.varnames,s,&outfun,WITHAUTO) ) != CFUNCTION ) {
05120                 MesPrint("&%s is not a proper function or tensor",s);
```

```
05121                 error = 1;
05122             }
05123             outfun += FUNCTION;
05124             outflag++;
05125             *inp = c;
05126         }
05127         else {
05128             MesPrint("&Unrecognized option in FindLoop or ReplaceLoop: %s",inp);
05129             error = 1;
05130             *s = c; inp = s;
05131             while ( *inp && *inp != ',' ) inp++;
05132         }
05133     }
05134     if ( *inp != 0 && mode == REPLACELOOP ) goto syntax;
05135     if ( mode == FINDLOOP && outflag > 0 ) {
05136         MesPrint("&outflag option is illegal in FindLoop");
05137         error = 1;
05138     }
05139     if ( mode == REPLACELOOP && outflag == 0 ) goto syntax;
05140     if ( aflag == 0 || lflag == 0 ) goto syntax;
05141     comfindloop[3] = funnum;
05142     comfindloop[4] = nloop;
05143     comfindloop[5] = nargs;
05144     comfindloop[6] = outfun;
05145     comfindloop[1] = 7;
05146     if ( indflag ) {
05147         if ( mode == 0 ) comfindloop[2] =  indexnum + 5;
05148         else             comfindloop[2] = -indexnum - 5;
05149     }
05150     else comfindloop[2] = mode;
05151     AddNtoL(comfindloop[1],comfindloop);
05152     return(error);
05153 }
05154
05155 /*
05156     #] DoFindLoop :
05157     #[ CoFindLoop :
05158 */
05159
05160 int CoFindLoop(UBYTE *inp)
05161 { return(DoFindLoop(inp,FINDLOOP)); }
05162
05163 /*
05164     #] CoFindLoop :
05165     #[ CoReplaceLoop :
05166 */
05167
05168 int CoReplaceLoop(UBYTE *inp)
05169 {
05170     int error = DoFindLoop(inp,REPLACELOOP);
05171     if ( error ) {
05172         Terminate(-1);
05173     }
05174     return(error);
05175 }
05176
05177 /*
05178     #] CoReplaceLoop :
05179     #[ CoFunPowers :
05180 */
05181
05182 static UBYTE *FunPowOptions[] = {
05183       (UBYTE *)"nofunpowers"
05184     ,(UBYTE *)"commutingonly"
05185     ,(UBYTE *)"allfunpowers"
05186     };
05187
05188 int CoFunPowers(UBYTE *inp)
05189 {
05190     UBYTE *option, c;
05191     int i, maxoptions = sizeof(FunPowOptions)/sizeof(UBYTE *);
05192     while ( *inp == ',' ) inp++;
05193     option = inp;
05194     inp = SkipAName(inp); c = *inp; *inp = 0;
05195     for ( i = 0; i < maxoptions; i++ ) {
05196         if ( StrICont(option,FunPowOptions[i]) == 0 ) {
05197             if ( c ) {
05198                 *inp = c;
05199                 MesPrint("&Illegal FunPowers statement");
05200                 return(1);
05201             }
05202             AC.funpowers = i;
05203             return(0);
05204         }
05205     }
05206     MesPrint("&Illegal option in FunPowers statement: %s",option);
05207     return(1);
```

```
05208 }
05209
05210 /*
05211     #] CoFunPowers :
05212     #[ CoUnitTrace :
05213 */
05214
05215 int CoUnitTrace(UBYTE *s)
05216 {
05217     WORD num;
05218     if ( FG.cTable[*s] == 1 ) {
05219         ParseNumber(num,s)
05220         if ( *s != 0 ) {
05221 nogood:     MesPrint("&Value of UnitTrace should be a (positive) number or a symbol");
05222             return(1);
05223         }
05224         AC.lUniTrace[0] = SNUMBER;
05225         AC.lUniTrace[2] = num;
05226     }
05227     else {
05228         if ( GetName(AC.varnames,s,&num,WITHAUTO) == CSYMBOL ) {
05229             AC.lUniTrace[0] = SYMBOL;
05230             AC.lUniTrace[2] = num;
05231             num = -num;
05232         }
05233         else goto nogood;
05234         s = SkipAName(s);
05235         if ( *s ) goto nogood;
05236     }
05237     AC.lUnitTrace = num;
05238     return(0);
05239 }
05240
05241 /*
05242     #] CoUnitTrace :
05243     #[ CoTerm :
05244
05245     Note: termstack holds the offset of the term statement in the compiler
05246     buffer. termsortstack holds the offset of the last sort statement
05247         (or the corresponding term statement)
05248 */
05249
05250 int CoTerm(UBYTE *s)
05251 {
05252     GETIDENTITY
05253     WORD *w = AT.WorkPointer;
05254     int error = 0;
05255     while ( *s == ',' ) s++;
05256     if ( *s ) {
05257         MesPrint("&Illegal syntax for Term statement");
05258         return(1);
05259     }
05260     if ( AC.termlevel+1 >= AC.maxtermlevel ) {
05261         if ( AC.maxtermlevel <= 0 ) {
05262             AC.maxtermlevel = 20;
05263             AC.termstack = (LONG *)Malloc1(AC.maxtermlevel*sizeof(LONG),"termstack");
05264             AC.termsortstack = (LONG *)Malloc1(AC.maxtermlevel*sizeof(LONG),"termsortstack");
05265             AC.termsumcheck = (WORD *)Malloc1(AC.maxtermlevel*sizeof(WORD),"termsumcheck");
05266         }
05267         else {
05268             DoubleBuffer((void **)AC.termstack,(void **)AC.termstack+AC.maxtermlevel,
05269                 sizeof(LONG),"doubling termstack");
05270             DoubleBuffer((void **)AC.termsortstack,
05271                 (void **)AC.termsortstack+AC.maxtermlevel,
05272                 sizeof(LONG),"doubling termsortstack");
05273             DoubleBuffer((void **)AC.termsumcheck,
05274                 (void **)AC.termsumcheck+AC.maxtermlevel,
05275                 sizeof(LONG),"doubling termsumcheck");
05276             AC.maxtermlevel *= 2;
05277         }
05278     }
05279     AC.termsumcheck[AC.termlevel] = NestingChecksum();
05280     AC.termstack[AC.termlevel] = cbuf[AC.cbufnum].Pointer
05281                             - cbuf[AC.cbufnum].Buffer + 2;
05282     AC.termsortstack[AC.termlevel] = AC.termstack[AC.termlevel] + 1;
05283     AC.termlevel++;
05284     *w++ = TYPETERM;
05285     w++;
05286     *w++ = cbuf[AC.cbufnum].numlhs;
05287     *w++ = cbuf[AC.cbufnum].numlhs;
05288     AT.WorkPointer[1] = w - AT.WorkPointer;
05289     AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
05290     return(error);
05291 }
05292
05293 /*
05294     #] CoTerm :
```

```
05295     #[ CoEndTerm :
05296 */
05297
05298 int CoEndTerm(UBYTE *s)
05299 {
05300     CBUF *C = cbuf+AC.cbufnum;
05301     while ( *s == ',' ) s++;
05302     if ( *s ) {
05303         MesPrint("&Illegal syntax for EndTerm statement");
05304         return(1);
05305     }
05306     if ( AC.termlevel <= 0 ) {
05307         MesPrint("&EndTerm without corresponding Argument statement");
05308         return(1);
05309     }
05310     AC.termlevel--;
05311     cbuf[AC.cbufnum].Buffer[AC.termstack[AC.termlevel]] = C->numlhs;
05312     cbuf[AC.cbufnum].Buffer[AC.termsortstack[AC.termlevel]] = C->numlhs;
05313     if ( AC.termsumcheck[AC.termlevel] != NestingChecksum() ) {
05314         MesNesting();
05315         return(1);
05316     }
05317     return(0);
05318 }
05319
05320 /*
05321     #] CoEndTerm :
05322     #[ CoSort :
05323 */
05324
05325 int CoSort(UBYTE *s)
05326 {
05327     GETIDENTITY
05328     WORD *w = AT.WorkPointer;
05329     int error = 0;
05330     while ( *s == ',' ) s++;
05331     if ( *s ) {
05332         MesPrint("&Illegal syntax for Sort statement");
05333         error = 1;
05334     }
05335     if ( AC.termlevel <= 0 ) {
05336         MesPrint("&The Sort statement can only be used inside a term environment");
05337         error = 1;
05338     }
05339     if ( error ) return(error);
05340     *w++ = TYPESORT;
05341     w++;
05342     w++;
05343     cbuf[AC.cbufnum].Buffer[AC.termsortstack[AC.termlevel-1]] =
05344                                         *w = cbuf[AC.cbufnum].numlhs+1;
05345     w++;
05346     AC.termsortstack[AC.termlevel-1] = cbuf[AC.cbufnum].Pointer
05347                             - cbuf[AC.cbufnum].Buffer + 3;
05348     if ( AC.termsumcheck[AC.termlevel-1] != NestingChecksum() - 1 ) {
05349         MesNesting();
05350         return(1);
05351     }
05352     AT.WorkPointer[1] = w - AT.WorkPointer;
05353     AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
05354     return(error);
05355 }
05356
05357 /*
05358     #] CoSort :
05359     #[ CoPolyFun :
05360
05361     Collect,functionname
05362 */
05363
05364 int CoPolyFun(UBYTE *s)
05365 {
05366     GETIDENTITY
05367     WORD numfun;
05368     int type, error = 0;
05369     UBYTE *t;
05370     AR.PolyFun = AC.lPolyFun = 0;
05371     AR.PolyFunInv = AC.lPolyFunInv = 0;
05372     AR.PolyFunType = AC.lPolyFunType = 0;
05373     AR.PolyFunExp = AC.lPolyFunExp = 0;
05374     AR.PolyFunVar = AC.lPolyFunVar = 0;
05375     AR.PolyFunPow = AC.lPolyFunPow = 0;
05376     if ( *s == 0 ) { return(0); }
05377     t = SkipAName(s);
05378     if ( t == 0 || *t != 0 ) {
05379         MesPrint("&PolyFun statement needs a single commuting function for its argument");
05380         return(1);
05381     }
```

```
05382     if ( ( ( type = GetName(AC.varnames,s,&numfun,WITHAUTO) ) != CFUNCTION )
05383     || ( functions[numfun].spec != 0 ) || ( functions[numfun].commute != 0 ) ) {
05384         MesPrint("&%s should be a regular commuting function",s);
05385         if ( type < 0 ) {
05386             if ( GetName(AC.exprnames,s,&numfun,NOAUTO) == NAMENOTFOUND )
05387                 AddFunction(s,0,0,0,0,0,-1,-1);
05388         }
05389         error = 1;
05390     }
05391     else {
05392         AR.PolyFun = AC.lPolyFun = numfun+FUNCTION;
05393         AR.PolyFunType = AC.lPolyFunType = 1;
05394     }
05395 #ifdef WITHFLOAT
05396     if ( mpfaux_ != 0 ) {
05397         MesPrint("&Simultaneous use of PolyFun and float_ is not allowed.");
05398         error = 1;
05399     }
05400 #endif
05401     return(error);
05402 }
05403
05404 /*
05405     #] CoPolyFun :
05406     #[ CoPolyRatFun :
05407
05408     PolyRatFun [,functionname[,functionname](option)]
05409 */
05410
05411 int CoPolyRatFun(UBYTE *s)
05412 {
05413     GETIDENTITY
05414     WORD numfun;
05415     int type, error = 0;
05416     UBYTE *t, c;
05417     AR.PolyFun = AC.lPolyFun = 0;
05418     AR.PolyFunInv = AC.lPolyFunInv = 0;
05419     AR.PolyFunType = AC.lPolyFunType = 0;
05420     AR.PolyFunExp = AC.lPolyFunExp = 0;
05421     AR.PolyFunVar = AC.lPolyFunVar = 0;
05422     AR.PolyFunPow = AC.lPolyFunPow = 0;
05423     if ( *s == 0 ) return(error);
05424     t = SkipAName(s);
05425     if ( t == 0 ) goto NumErr;
05426     c = *t; *t = 0;
05427 #ifdef WITHFLOAT
05428     if ( mpfaux_ != 0 ) {
05429         MesPrint("&Simultaneous use of PolyFun and float_ is not allowed.");
05430         error = 1;
05431     }
05432 #endif
05433     if ( ( ( type = GetName(AC.varnames,s,&numfun,WITHAUTO) ) != CFUNCTION )
05434     || ( functions[numfun].spec != 0 ) || ( functions[numfun].commute != 0 ) ) {
05435         MesPrint("&%s should be a regular commuting function",s);
05436         if ( type < 0 ) {
05437             if ( GetName(AC.exprnames,s,&numfun,NOAUTO) == NAMENOTFOUND )
05438                 AddFunction(s,0,0,0,0,0,-1,-1);
05439         }
05440         return(1);
05441     }
05442     AR.PolyFun = AC.lPolyFun = numfun+FUNCTION;
05443     AR.PolyFunInv = AC.lPolyFunInv = 0;
05444     AR.PolyFunType = AC.lPolyFunType = 2;
05445     AC.PolyRatFunChanged = 1;
05446     if ( c == 0 ) return(error);
05447     *t = c;
05448     if ( *t == '-' ) { AC.PolyRatFunChanged = 0; t++; }
05449     while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
05450     if ( *t == 0 ) return(error);
05451     if ( *t != '(' ) {
05452         s = t;
05453         t = SkipAName(s);
05454         if ( t == 0 ) goto NumErr;
05455         c = *t; *t = 0;
05456         if ( ( ( type = GetName(AC.varnames,s,&numfun,WITHAUTO) ) != CFUNCTION )
05457         || ( functions[numfun].spec != 0 ) || ( functions[numfun].commute != 0 ) ) {
05458             MesPrint("&%s should be a regular commuting function",s);
05459             if ( type < 0 ) {
05460                 if ( GetName(AC.exprnames,s,&numfun,NOAUTO) == NAMENOTFOUND )
05461                     AddFunction(s,0,0,0,0,0,-1,-1);
05462             }
05463             return(1);
05464         }
05465         AR.PolyFunInv = AC.lPolyFunInv = numfun+FUNCTION;
05466         if ( c == 0 ) return(error);
05467         *t = c;
05468         if ( *t == '-' ) { AC.PolyRatFunChanged = 0; t++; }
```

```
05469         while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
05470         if ( *t == 0 ) return(error);
05471     }
05472     if ( *t == '(' ) {
05473         t++;
05474         while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
05475 /*
05476         Next we need a keyword like
05477             (divergence,ep)
05478             (expand,ep,maxpow)
05479 */
05480         s = t;
05481         t = SkipAName(s);
05482         if ( t == 0 ) goto NumErr;
05483         c = *t; *t = 0;
05484         if ( ( StrICmp(s,(UBYTE *)"divergence") == 0 )
05485         || ( StrICmp(s,(UBYTE *)"finddivergence") == 0 ) ) {
05486             if ( c != ',' ) {
05487                 MesPrint("&Illegal option field in PolyRatFun statement.");
05488                 return(1);
05489             }
05490             *t = c;
05491             while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
05492             s = t;
05493             t = SkipAName(s);
05494             if ( t == 0 ) goto NumErr;
05495             c = *t; *t = 0;
05496             if ( ( type = GetName(AC.varnames,s,&AC.lPolyFunVar,WITHAUTO) ) != CSYMBOL ) {
05497                 MesPrint("&Illegal symbol %s in option field in PolyRatFun statement.",s);
05498                 return(1);
05499             }
05500             *t = c;
05501             while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
05502             if ( *t != ')' ) {
05503                 MesPrint("&Illegal termination of option in PolyRatFun statement.");
05504                 return(1);
05505             }
05506             AR.PolyFunExp = AC.lPolyFunExp = 1;
05507             AR.PolyFunVar = AC.lPolyFunVar;
05508             symbols[AC.lPolyFunVar].minpower = -MAXPOWER;
05509             symbols[AC.lPolyFunVar].maxpower =  MAXPOWER;
05510         }
05511         else if ( StrICmp(s,(UBYTE *)"expand") == 0 ) {
05512             WORD x = 0, etype = 2;
05513             if ( c != ',' ) {
05514                 MesPrint("&Illegal option field in PolyRatFun statement.");
05515                 return(1);
05516             }
05517             *t = c;
05518             while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
05519             s = t;
05520             t = SkipAName(s);
05521             if ( t == 0 ) goto NumErr;
05522             c = *t; *t = 0;
05523             if ( ( type = GetName(AC.varnames,s,&AC.lPolyFunVar,WITHAUTO) ) != CSYMBOL ) {
05524                 MesPrint("&Illegal symbol %s in option field in PolyRatFun statement.",s);
05525                 return(1);
05526             }
05527             *t = c;
05528             while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
05529             if ( *t > '9' || *t < '0' ) {
05530                 MesPrint("&Illegal option field in PolyRatFun statement.");
05531                 return(1);
05532             }
05533             while ( *t <= '9' && *t >= '0' ) x = 10*x + *t++ - '0';
05534             while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
05535             if ( *t != ')' ) {
05536                 s = t;
05537                 t = SkipAName(s);
05538                 if ( t == 0 ) goto ParErr;
05539                 c = *t; *t = 0;
05540                 if ( StrICmp(s,(UBYTE *)"fixed") == 0 ) {
05541                     etype = 3;
05542                 }
05543                 else if ( StrICmp(s,(UBYTE *)"relative") == 0 ) {
05544                     etype = 2;
05545                 }
05546                 else {
05547                     MesPrint("&Illegal termination of option in PolyRatFun statement.");
05548                     return(1);
05549                 }
05550                 *t = c;
05551                 while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
05552                 if ( *t != ')' ) {
05553                     MesPrint("&Illegal termination of option in PolyRatFun statement.");
05554                     return(1);
05555                 }
```

```
05556             }
05557             AR.PolyFunExp = AC.lPolyFunExp = etype;
05558             AR.PolyFunVar = AC.lPolyFunVar;
05559             AR.PolyFunPow = AC.lPolyFunPow = x;
05560             symbols[AC.lPolyFunVar].minpower = -MAXPOWER;
05561             symbols[AC.lPolyFunVar].maxpower =  MAXPOWER;
05562         }
05563         else {
05564 ParErr:     MesPrint("&Illegal option %s in PolyRatFun statement.",s);
05565             return(1);
05566         }
05567         t++;
05568         while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
05569         if ( *t == 0 ) return(error);
05570     }
05571 NumErr:;
05572     MesPrint("&PolyRatFun statement needs one or two commuting function(s) for its argument(s)");
05573     return(1);
05574 }
05575
05576 /*
05577     #] CoPolyRatFun :
05578     #[ CoMerge :
05579 */
05580
05581 int CoMerge(UBYTE *inp)
05582 {
05583     UBYTE *s = inp;
05584     int type;
05585     WORD numfunc, option = 0;
05586     if ( tolower(s[0]) == 'o' && tolower(s[1]) == 'n' && tolower(s[2]) == 'c' &&
05587          tolower(s[3]) == 'e' && tolower(s[4]) == ',' ) {
05588         option = 1; s += 5;
05589     }
05590     else if ( tolower(s[0]) == 'a' && tolower(s[1]) == 'l' && tolower(s[2]) == 'l' &&
05591          tolower(s[3]) == ',' ) {
05592         option = 0; s += 4;
05593     }
05594     if ( *s == '$' ) {
05595         if ( ( type = GetName(AC.dollarnames,s+1,&numfunc,NOAUTO) ) == CDOLLAR )
05596             numfunc = -numfunc;
05597         else {
05598             MesPrint("&%s is undefined",s);
05599             numfunc = AddDollar(s+1,DOLINDEX,&one,1);
05600             return(1);
05601         }
05602 tests:  s = SkipAName(s);
05603         if ( *s != 0 ) {
05604             MesPrint("&Merge/shuffle should have a single function or $variable for its argument");
05605             return(1);
05606         }
05607     }
05608     else if ( ( type = GetName(AC.varnames,s,&numfunc,WITHAUTO) ) == CFUNCTION ) {
05609         numfunc += FUNCTION;
05610         goto tests;
05611     }
05612     else if ( type != -1 ) {
05613         if ( type != CDUBIOUS ) {
05614             NameConflict(type,s);
05615             type = MakeDubious(AC.varnames,s,&numfunc);
05616         }
05617         return(1);
05618     }
05619     else {
05620         MesPrint("&%s is not a function",s);
05621         numfunc = AddFunction(s,0,0,0,0,0,-1,-1) + FUNCTION;
05622         return(1);
05623     }
05624     Add4Com(TYPEMERGE,numfunc,option);
05625     return(0);
05626 }
05627
05628 /*
05629     #] CoMerge :
05630     #[ CoStuffle :
05631
05632     Important for future options: The bit, given by 256 (bit 8) is reserved
05633     internally for keeping track of the sign in the number of Stuffle
05634     additions.
05635 */
05636
05637 int CoStuffle(UBYTE *inp)
05638 {
05639     UBYTE *s = inp, *ss, c;
05640     int type;
05641     WORD numfunc, option = 0;
05642     if ( tolower(s[0]) == 'o' && tolower(s[1]) == 'n' && tolower(s[2]) == 'c' &&
```

```
05643          tolower(s[3]) == 'e' && tolower(s[4]) == ',' ) {
05644         option = 1; s += 5;
05645     }
05646     else if ( tolower(s[0]) == 'a' && tolower(s[1]) == 'l' && tolower(s[2]) == 'l' &&
05647          tolower(s[3]) == ',' ) {
05648         option = 0; s += 4;
05649     }
05650     ss = SkipAName(s);
05651     c = *ss; *ss = 0;
05652     if ( *s == '$' ) {
05653         if ( ( type = GetName(AC.dollarnames,s+1,&numfunc,NOAUTO) ) == CDOLLAR )
05654             numfunc = -numfunc;
05655         else {
05656             MesPrint("&%s is undefined",s);
05657             numfunc = AddDollar(s+1,DOLINDEX,&one,1);
05658             return(1);
05659         }
05660 tests:  *ss = c;
05661         if ( *ss != '+' && *ss != '-' && ss[1] != 0 ) {
05662             MesPrint("&Stuffle should have a single function or $variable for its argument, followed
      by either + or -");
05663             return(1);
05664         }
05665         if ( *ss == '-' ) option += 2;
05666     }
05667     else if ( ( type = GetName(AC.varnames,s,&numfunc,WITHAUTO) ) == CFUNCTION ) {
05668         numfunc += FUNCTION;
05669         goto tests;
05670     }
05671     else if ( type != -1 ) {
05672         if ( type != CDUBIOUS ) {
05673             NameConflict(type,s);
05674             type = MakeDubious(AC.varnames,s,&numfunc);
05675         }
05676         return(1);
05677     }
05678     else {
05679         MesPrint("&%s is not a function",s);
05680         numfunc = AddFunction(s,0,0,0,0,0,-1,-1) + FUNCTION;
05681         return(1);
05682     }
05683     Add4Com(TYPESTUFFLE,numfunc,option);
05684     return(0);
05685 }
05686
05687 /*
05688     #] CoStuffle :
05689     #[ CoProcessBucket :
05690 */
05691
05692 int CoProcessBucket(UBYTE *s)
05693 {
05694     LONG x;
05695     while ( *s == ',' || *s == '=' ) s++;
05696     ParseNumber(x,s)
05697     if ( *s && *s != ' ' && *s != '\t' ) {
05698         MesPrint("&Numerical value expected for ProcessBucketSize");
05699         return(1);
05700     }
05701     AC.ProcessBucketSize = x;
05702     return(0);
05703 }
05704
05705 /*
05706     #] CoProcessBucket :
05707     #[ CoThreadBucket :
05708 */
05709
05710 int CoThreadBucket(UBYTE *s)
05711 {
05712     LONG x;
05713     while ( *s == ',' || *s == '=' ) s++;
05714     ParseNumber(x,s)
05715     if ( *s && *s != ' ' && *s != '\t' ) {
05716         MesPrint("&Numerical value expected for ThreadBucketSize");
05717         return(1);
05718     }
05719     if ( x <= 0 ) {
05720         Warning("Negative of zero value not allowed for ThreadBucketSize. Adjusted to 1.");
05721         x = 1;
05722     }
05723     AC.ThreadBucketSize = x;
05724 #ifdef WITHPTHREADS
05725     if ( AS.MultiThreaded ) MakeThreadBuckets(-1,1);
05726 #endif
05727     return(0);
05728 }
```

```
05729
05730 /*
05731     #] CoThreadBucket :
05732     #[ DoArgPlode :
05733
05734     Syntax: a list of functions.
05735     If the functions have an argument it must be a function.
05736     In the case f(g) we treat f(g(...)) with g any argument.
05737       (not yet implemented)
05738 */
05739
05740 int DoArgPlode(UBYTE *s, int par)
05741 {
05742     GETIDENTITY
05743     WORD numfunc, type, error = 0, *w, n;
05744     UBYTE *t,c;
05745     int i;
05746     w = AT.WorkPointer;
05747     *w++ = par;
05748     w++;
05749     while ( *s == ',' ) s++;
05750     while ( *s ) {
05751         if ( *s == '$' ) {
05752             MesPrint("&We don't do dollar variables yet in ArgImplode/ArgExplode");
05753             return(1);
05754         }
05755         t = s;
05756         if ( ( s = SkipAName(s) ) == 0 ) return(1);
05757         c = *s; *s = 0;
05758         if ( ( type = GetName(AC.varnames,t,&numfunc,WITHAUTO) ) == CFUNCTION ) {
05759             numfunc += FUNCTION;
05760         }
05761         else if ( type != -1 ) {
05762             if ( type != CDUBIOUS ) {
05763                 NameConflict(type,t);
05764                 type = MakeDubious(AC.varnames,t,&numfunc);
05765             }
05766             error = 1;
05767         }
05768         else {
05769             MesPrint("&%s is not a function",t);
05770             numfunc = AddFunction(s,0,0,0,0,0,-1,-1) + FUNCTION;
05771             return(1);
05772         }
05773         *s = c;
05774         *w++ = numfunc;
05775         *w++ = FUNHEAD;
05776 #if FUNHEAD > 2
05777         for ( i = 2; i < FUNHEAD; i++ ) *w++ = 0;
05778 #endif
05779         if ( *s && *s != ',' ) {
05780             MesPrint("&Illegal character in ArgImplode/ArgExplode statement: %s",s);
05781             return(1);
05782         }
05783         while ( *s == ',' ) s++;
05784     }
05785     n = w - AT.WorkPointer;
05786     AT.WorkPointer[1] = n;
05787     AddNtoL(n,AT.WorkPointer);
05788     return(error);
05789 }
05790
05791 /*
05792     #] DoArgPlode :
05793     #[ CoArgExplode :
05794 */
05795
05796 int CoArgExplode(UBYTE *s) { return(DoArgPlode(s,TYPEARGEXPLODE)); }
05797
05798 /*
05799     #] CoArgExplode :
05800     #[ CoArgImplode :
05801 */
05802
05803 int CoArgImplode(UBYTE *s) { return(DoArgPlode(s,TYPEARGIMPLODE)); }
05804
05805 /*
05806     #] CoArgImplode :
05807     #[ CoClearTable :
05808 */
05809
05810 int CoClearTable(UBYTE *s)
05811 {
05812     UBYTE c, *t;
05813     int j, type, error = 0;
05814     WORD numfun;
05815     TABLES T, TT;
```

```
05816     if ( *s == 0 ) {
05817         MesPrint("&The ClearTable statement needs at least one (table) argument.");
05818         return(1);
05819     }
05820     while ( *s ) {
05821         t = s;
05822         s = SkipAName(s);
05823         c = *s; *s = 0;
05824         if ( ( ( type = GetName(AC.varnames,t,&numfun,WITHAUTO) ) != CFUNCTION )
05825         && type != CDUBIOUS ) {
05826 nofunc:     MesPrint("&%s is not a table",t);
05827             error = 4;
05828             if ( type < 0 ) numfun = AddFunction(t,0,0,0,0,0,-1,-1);
05829             *s = c;
05830             if ( *s == ',' ) s++;
05831             continue;
05832         }
05833 /*
05834         else if ( ( ( T = functions[numfun].tabl ) == 0 )
05835          || ( T->sparse == 0 ) ) goto nofunc;
05836 */
05837         else if ( ( T = functions[numfun].tabl ) == 0 ) goto nofunc;
05838         numfun += FUNCTION;
05839         *s = c;
05840         if ( *s == ',' ) s++;
05841 /*
05842         Now we clear the table.
05843 */
05844         if ( T->sparse ) {
05845         if ( T->boomlijst ) M_free(T->boomlijst,"TableTree");
05846         for (j = 0; j < T->buffersfill; j++ ) { /* was <= */
05847             finishcbuf(T->buffers[j]);
05848         }
05849         if ( T->buffers ) M_free(T->buffers,"Table buffers");
05850         finishcbuf(T->bufnum);
05851
05852         T->boomlijst = 0;
05853         T->numtree = 0; T->rootnum = 0; T->MaxTreeSize = 0;
05854         T->boomlijst = 0;
05855         T->bufnum = inicbufs();
05856         T->bufferssize = 8;
05857         T->buffers = (WORD *)Malloc1(sizeof(WORD)*T->bufferssize,"Table buffers");
05858         T->buffersfill = 0;
05859         T->buffers[T->buffersfill++] = T->bufnum;
05860
05861         T->totind = 0;            /* At the moment there are this many */
05862         T->reserved = 0;
05863
05864         ClearTableTree(T);
05865
05866         if ( T->spare ) {
05867             if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
05868             T->tablepointers = 0;
05869             TT = T->spare;
05870             if ( TT->tablepointers ) M_free(TT->tablepointers,"tablepointers");
05871             for (j = 0; j < TT->buffersfill; j++ ) {
05872                 finishcbuf(TT->buffers[j]);
05873             }
05874             if ( TT->boomlijst ) M_free(TT->boomlijst,"TableTree");
05875             if ( TT->buffers )M_free(TT->buffers,"Table buffers");
05876             if ( TT->mm ) M_free(TT->mm,"tableminmax");
05877             if ( TT->flags ) M_free(TT->flags,"tableflags");
05878             M_free(TT,"table");
05879             SpareTable(T);
05880         }
05881         }
05882         else EmptyTable(T);
05883     }
05884     return(error);
05885 }
05886
05887 /*
05888     #] CoClearTable :
05889     #[ CoDenominators :
05890 */
05891
05892 int CoDenominators(UBYTE *s)
05893 {
05894     WORD numfun;
05895     int type;
05896     UBYTE *t = SkipAName(s), *t1;
05897     if ( t == 0 ) goto syntaxerror;
05898     t1 = t; while ( *t1 == ',' || *t1 == ' ' || *t1 == '\t' ) t1++;
05899     if ( *t1 ) goto syntaxerror;
05900     *t = 0;
05901     if ( ( ( type = GetName(AC.varnames,s,&numfun,WITHAUTO) ) != CFUNCTION )
05902     || ( functions[numfun].spec != 0 ) ) {
```

```
05903         if ( type < 0 ) {
05904             if ( GetName(AC.exprnames,s,&numfun,NOAUTO) == NAMENOTFOUND )
05905                 AddFunction(s,0,0,0,0,0,-1,-1);
05906         }
05907         goto syntaxerror;
05908     }
05909     Add3Com(TYPEDENOMINATORS,numfun+FUNCTION);
05910     return(0);
05911 syntaxerror:
05912     MesPrint("&Denominators statement needs one regular function for its argument");
05913     return(1);
05914 }
05915
05916 /*
05917     #] CoDenominators :
05918     #[ CoDropCoefficient :
05919 */
05920
05921 int CoDropCoefficient(UBYTE *s)
05922 {
05923     if ( *s == 0 ) {
05924         Add2Com(TYPEDROPCOEFFICIENT)
05925         return(0);
05926     }
05927     MesPrint("&Illegal argument in DropCoefficient statement: '%s'",s);
05928     return(1);
05929 }
05930 /*
05931     #] CoDropCoefficient :
05932     #[ CoDropSymbols :
05933 */
05934
05935 int CoDropSymbols(UBYTE *s)
05936 {
05937     if ( *s == 0 ) {
05938         Add2Com(TYPEDROPSYMBOLS)
05939         return(0);
05940     }
05941     MesPrint("&Illegal argument in DropSymbols statement: '%s'",s);
05942     return(1);
05943 }
05944 /*
05945     #] CoDropSymbols :
05946     #[ CoToPolynomial :
05947
05948     Converts the current term as much as possible to symbols.
05949     Keeps a list of all objects converted to symbols in AM.sbufnum.
05950     Note that this cannot be executed in parallel because we have only
05951     a single compiler buffer for this. Hence we switch on the noparallel
05952     module option.
05953
05954     Option(s):
05955         OnlyFunctions [,name1][,name2][,...,namem];
05956 */
05957
05958 int CoToPolynomial(UBYTE *inp)
05959 {
05960     int error = 0;
05961     while ( *inp == ' ' || *inp == ',' || *inp == '\t' ) inp++;
05962     if ( ( AC.topolynomialflag & ~TOPOLYNOMIALFLAG ) != 0 ) {
05963         MesPrint("&ToPolynomial statement and FactArg statement are not allowed in the same module");
05964         return(1);
05965     }
05966     if ( AO.OptimizeResult.code != NULL ) {
05967         MesPrint("&Using ToPolynomial statement when there are still optimization results active.");
05968         MesPrint("&Please use #ClearOptimize instruction first.");
05969         MesPrint("&This will loose the optimized expression.");
05970         return(1);
05971     }
05972     if ( *inp == 0 ) {
05973         Add3Com(TYPETOPOLYNOMIAL,DOALL)
05974     }
05975     else {
05976         int numargs = 0;
05977         WORD *funnums = 0, type, num;
05978         UBYTE *s, c;
05979         s = SkipAName(inp);
05980         if ( s == 0 ) return(1);
05981         c = *s; *s = 0;
05982         if ( StrICmp(inp,(UBYTE *)"onlyfunctions") ) {
05983             MesPrint("&Illegal option %s in ToPolynomial statement",inp);
05984             *s = c;
05985             return(1);
05986         }
05987         *s = c;
05988         inp = s;
05989         while ( *inp == ' ' || *inp == ',' || *inp == '\t' ) inp++;
```

```
05990         s = inp;
05991         while ( *s ) s++;
05992 /*
05993         Get definitely enough space for the numbers of the functions
05994 */
05995         funnums = (WORD *)Malloc1(((LONG)(s-inp)+3)*sizeof(WORD),"ToPlynomial");
05996         while ( *inp ) {
05997             s = SkipAName(inp);
05998             if ( s == 0 ) return(1);
05999             c = *s; *s = 0;
06000             type = GetName(AC.varnames,inp,&num,WITHAUTO);
06001             if ( type != CFUNCTION ) {
06002                 MesPrint("&%s is not a function in ToPolynomial statement",inp);
06003                 error = 1;
06004             }
06005             funnums[3+numargs++] = num+FUNCTION;
06006             *s = c;
06007             inp = s;
06008             while ( *inp == ' ' || *inp == ',' || *inp == '\t' ) inp++;
06009         }
06010         funnums[0] = TYPETOPOLYNOMIAL;
06011         funnums[1] = numargs+3;
06012         funnums[2] = ONLYFUNCTIONS;
06013
06014         AddNtoL(numargs+3,funnums);
06015         if ( funnums ) M_free(funnums,"ToPolynomial");
06016     }
06017     AC.topolynomialflag |= TOPOLYNOMIALFLAG;
06018 #ifdef WITHMPI
06019     /* In ParFORM, ToPolynomial has to be executed on the master. */
06020     AC.mparallelflag |= NOPARALLEL_CONVPOLY;
06021 #endif
06022     return(error);
06023 }
06024
06025 /*
06026     #] CoToPolynomial :
06027     #[ CoFromPolynomial :
06028
06029     Converts the current term as much as possible back from extra symbols
06030     to their original values. Does not look inside functions.
06031 */
06032
06033 int CoFromPolynomial(UBYTE *inp)
06034 {
06035     while ( *inp == ' ' || *inp == ',' || *inp == '\t' ) inp++;
06036     if ( *inp == 0 ) {
06037         if ( AO.OptimizeResult.code != NULL ) {
06038             MesPrint("&Using FromPolynomial statement when there are still optimization results
      active.");
06039             MesPrint("&Please use #ClearOptimize instruction first.");
06040             MesPrint("&This will loose the optimized expression.");
06041             return(1);
06042         }
06043         Add2Com(TYPEFROMPOLYNOMIAL)
06044         return(0);
06045     }
06046     MesPrint("&Illegal argument in FromPolynomial statement: '%s'",inp);
06047     return(1);
06048 }
06049
06050 /*
06051     #] CoFromPolynomial :
06052     #[ CoArgToExtraSymbol :
06053
06054     Converts the specified function arguments into extra symbols.
06055
06056     Syntax: ArgToExtraSymbol [ToNumber] [<argument specifications>]
06057 */
06058
06059 int CoArgToExtraSymbol(UBYTE *s)
06060 {
06061     CBUF *C = cbuf + AC.cbufnum;
06062     WORD *lhs;
06063
06064     /* TODO: resolve interference with rational arithmetic. (#138) */
06065     if ( ( AC.topolynomialflag & ~TOPOLYNOMIALFLAG ) != 0 ) {
06066         MesPrint("&ArgToExtraSymbol statement and FactArg statement are not allowed in the same
      module");
06067         return(1);
06068     }
06069     if ( AO.OptimizeResult.code != NULL ) {
06070         MesPrint("&Using ArgToExtraSymbol statement when there are still optimization results
      active.");
06071         MesPrint("&Please use #ClearOptimize instruction first.");
06072         MesPrint("&This will loose the optimized expression.");
06073         return(1);
```

```
06074     }
06075
06076     SkipSpaces(&s);
06077     int tonumber = ConsumeOption(&s, "tonumber");
06078
06079     int ret = DoArgument(s,TYPEARGTOEXTRASYMBOL);
06080     if ( ret ) return(ret);
06081
06082     /*
06083      * The "scale" parameter is unused. Instead, we put the "tonumber"
06084      * parameter.
06085      */
06086     lhs = C->lhs[C->numlhs];
06087     if ( lhs[4] != 1 ) {
06088         Warning("scale parameter (^n) is ignored in ArgToExtraSymbol");
06089     }
06090     lhs[4] = tonumber;
06091
06092     AC.topolynomialflag |= TOPOLYNOMIALFLAG;  /* This flag is also used in ParFORM. */
06093 #ifdef WITHMPI
06094     /*
06095      * In ParFORM, the conversion to extra symbols has to be performed on
06096      * the master.
06097      */
06098     AC.mparallelflag |= NOPARALLEL_CONVPOLY;
06099 #endif
06100
06101     return(0);
06102 }
06103
06104 /*
06105     #] CoArgToExtraSymbol :
06106     #[ CoExtraSymbols :
06107 */
06108
06109 int CoExtraSymbols(UBYTE *inp)
06110 {
06111     UBYTE *arg1, *arg2, c, *s;
06112     WORD i, j, type, number;
06113     while ( *inp == ' ' || *inp == ',' || *inp == '\t' ) inp++;
06114     if ( FG.cTable[*inp] != 0 ) {
06115         MesPrint("&Illegal argument in ExtraSymbols statement: '%s'",inp);
06116         return(1);
06117     }
06118     arg1 = inp;
06119     while ( FG.cTable[*inp] == 0 ) inp++;
06120     c = *inp; *inp = 0;
06121     if ( ( StrICmp(arg1,(UBYTE *)"array") == 0 )
06122             || ( StrICmp(arg1,(UBYTE *)"vector") == 0 ) ) {
06123         AC.extrasymbols = 1;
06124     }
06125     else if ( StrICmp(arg1,(UBYTE *)"underscore") == 0 ) {
06126         AC.extrasymbols = 0;
06127     }
06128 /*
06129     else if ( StrICmp(arg1,(UBYTE *)"nothing") == 0 ) {
06130         AC.extrasymbols = 2;
06131     }
06132 */
06133     else {
06134         MesPrint("&Illegal keyword in ExtraSymbols statement: '%s'",arg1);
06135         return(1);
06136     }
06137     *inp = c;
06138     while ( *inp == ' ' || *inp == ',' || *inp == '\t' ) inp++;
06139     if ( FG.cTable[*inp] != 0 ) {
06140         MesPrint("&Illegal argument in ExtraSymbols statement: '%s'",inp);
06141         return(1);
06142     }
06143     arg2 = inp;
06144     while ( FG.cTable[*inp] <= 1 ) inp++;
06145     if ( *inp != 0 ) {
06146         MesPrint("&Illegal end of ExtraSymbols statement: '%s'",inp);
06147         return(1);
06148     }
06149 /*
06150         Now check whether this object has been declared already.
06151         That would not be allowed.
06152 */
06153     if ( AC.extrasymbols == 1 ) {
06154         type = GetName(AC.varnames,arg2,&number,NOAUTO);
06155         if ( type != NAMENOTFOUND ) {
06156             MesPrint("&ExtraSymbols statement: '%s' has already been declared before",arg2);
06157             return(1);
06158         }
06159     }
06160     else if ( AC.extrasymbols == 0 ) {
```

```
06161         if ( *arg2 == 'N' ) {
06162             s = arg2+1;
06163             while ( FG.cTable[*s] == 1 ) s++;
06164             if ( *s == 0 ) {
06165                 MesPrint("&ExtraSymbols statement: '%s' creates conflicts with summed indices",arg2);
06166                 return(1);
06167             }
06168         }
06169     }
06170     if ( AC.extrasym ) { M_free(AC.extrasym,"extrasym"); AC.extrasym = 0; }
06171     i = inp - arg2 + 1;
06172     AC.extrasym = (UBYTE *)Malloc1(i*sizeof(UBYTE),"extrasym");
06173     for ( j = 0; j < i; j++ ) AC.extrasym[j] = arg2[j];
06174     return(0);
06175 }
06176
06177 /*
06178     #] CoExtraSymbols :
06179     #[ GetIfDollarFactor :
06180 */
06181
06182 WORD *GetIfDollarFactor(UBYTE **inp, WORD *w)
06183 {
06184     LONG x;
06185     WORD number;
06186     UBYTE *name, c, *s;
06187     s = *inp;
06188     if ( FG.cTable[*s] == 1 ) {
06189         x = 0;
06190         while ( FG.cTable[*s] == 1 ) {
06191             x = 10*x + *s++ - '0';
06192             if ( x >= MAXPOSITIVE ) {
06193                 MesPrint("&Value in dollar factor too large");
06194                 while ( FG.cTable[*s] == 1 ) s++;
06195                 *inp = s;
06196                 return(0);
06197             }
06198         }
06199         *w++ = IFDOLLAREXTRA;
06200         *w++ = 3;
06201         *w++ = -x-1;
06202         *inp = s;
06203         return(w);
06204     }
06205     if ( *s != '$' ) {
06206         MesPrint("&Factor indicator for $-variable should be a number or a $-variable.");
06207         return(0);
06208     }
06209     s++; name = s;
06210     while ( FG.cTable[*s] < 2 ) s++;
06211     c = *s; *s = 0;
06212     if ( GetName(AC.dollarnames,name,&number,NOAUTO) == NAMENOTFOUND ) {
06213         MesPrint("&dollar in if statement should have been defined previously");
06214         return(0);
06215     }
06216     *s = c;
06217     *w++ = IFDOLLAREXTRA;
06218     *w++ = 3;
06219     *w++ = number;
06220     if ( c == '[' ) {
06221         s++;
06222         *inp = s;
06223         if ( ( w = GetIfDollarFactor(inp,w) ) == 0 ) return(0);
06224         s = *inp;
06225         if ( *s != ']' ) {
06226             MesPrint("&unmatched [] in $ in if statement");
06227             return(0);
06228         }
06229         s++;
06230         *inp = s;
06231     }
06232     return(w);
06233 }
06234
06235 /*
06236     #] GetIfDollarFactor :
06237     #[ GetDoParam :
06238 */
06239
06240 UBYTE *GetDoParam(UBYTE *inp, WORD **wp, int par)
06241 {
06242     LONG x;
06243     WORD number;
06244     UBYTE *name, c;
06245     if ( FG.cTable[*inp] == 1 ) {
06246         x = 0;
06247         while ( *inp >= '0' && *inp <= '9' ) {
```

```
06248             x = 10*x + *inp++ - '0';
06249             if ( x > MAXPOSITIVE ) {
06250                 if ( par == -1 ) {
06251                     MesPrint("&Value in dollar factor too large");
06252                 }
06253                 else {
06254                     MesPrint("&Value in do loop boundaries too large");
06255                 }
06256                 while ( FG.cTable[*inp] == 1 ) inp++;
06257                 return(0);
06258             }
06259         }
06260         if ( par > 0 ) {
06261             *(*wp)++ = SNUMBER;
06262             *(*wp)++ = (WORD)x;
06263         }
06264         else {
06265             *(*wp)++ = DOLLAREXPR2;
06266             *(*wp)++ = -((WORD)x)-1;
06267         }
06268         return(inp);
06269     }
06270     if ( *inp != '$' ) {
06271         return(0);
06272     }
06273     inp++; name = inp;
06274     while ( FG.cTable[*inp] < 2 ) inp++;
06275     c = *inp; *inp = 0;
06276     if ( GetName(AC.dollarnames,name,&number,NOAUTO) == NAMENOTFOUND ) {
06277         if ( par == -1 ) {
06278             MesPrint("&dollar in print statement should have been defined previously");
06279         }
06280         else {
06281             MesPrint("&dollar in do loop boundaries should have been defined previously");
06282         }
06283         return(0);
06284     }
06285     *inp = c;
06286     if ( par > 0 ) {
06287         *(*wp)++ = DOLLAREXPRESSION;
06288         *(*wp)++ = number;
06289     }
06290     else {
06291         *(*wp)++ = DOLLAREXPR2;
06292         *(*wp)++ = number;
06293     }
06294     if ( c == '[' ) {
06295         inp++;
06296         inp = GetDoParam(inp,wp,0);
06297         if ( inp == 0 ) return(0);
06298         if ( *inp != ']' ) {
06299             if ( par == -1 ) {
06300                 MesPrint("&unmatched [] in $ in print statement");
06301             }
06302             else {
06303                 MesPrint("&unmatched [] in do loop boundaries");
06304             }
06305             return(0);
06306         }
06307         inp++;
06308     }
06309     return(inp);
06310 }
06311
06312 /*
06313     #] GetDoParam :
06314     #[ CoDo :
06315 */
06316
06317 int CoDo(UBYTE *inp)
06318 {
06319     GETIDENTITY
06320     CBUF *C = cbuf+AC.cbufnum;
06321     WORD *w, numparam;
06322     int error = 0, i;
06323     UBYTE *name, c;
06324     if ( AC.doloopstack == 0 ) {
06325         AC.doloopstacksize = 20;
06326         AC.doloopstack = (WORD *)Malloc1(AC.doloopstacksize*2*sizeof(WORD),"doloop stack");
06327         AC.doloopnest = AC.doloopstack + AC.doloopstacksize;
06328     }
06329     if ( AC.dolooplevel >= AC.doloopstacksize ) {
06330         WORD *newstack, *newnest, newsize;
06331         newsize = AC.doloopstacksize * 2;
06332         newstack = (WORD *)Malloc1(newsize*2*sizeof(WORD),"doloop stack");
06333         newnest = newstack + newsize;
06334         for ( i = 0; i < newsize; i++ ) {
```

```
06335             newstack[i] = AC.doloopstack[i];
06336             newnest[i] = AC.doloopnest[i];
06337         }
06338         M_free(AC.doloopstack,"doloop stack");
06339         AC.doloopstack = newstack;
06340         AC.doloopnest = newnest;
06341         AC.doloopstacksize = newsize;
06342     }
06343     AC.doloopnest[AC.doolooplevel] = NestingChecksum();
06344
06345     w = AT.WorkPointer;
06346     *w++ = TYPEDOLOOP;
06347     w++; /* Space for the length of the statement */
06348 /*
06349     Now the $loopvariable
06350 */
06351     while ( *inp == ',' ) inp++;
06352     if ( *inp != '$' ) {
06353         error = 1;
06354         MesPrint("&do loop parameter should be a dollar variable");
06355     }
06356     else {
06357         inp++;
06358         name = inp;
06359         if ( FG.cTable[*inp] != 0 ) {
06360             error = 1;
06361             MesPrint("&illegal name for do loop parameter");
06362         }
06363         while ( FG.cTable[*inp] < 2 ) inp++;
06364         c = *inp; *inp = 0;
06365         if ( GetName(AC.dollarnames,name,&numparam,NOAUTO) == NAMENOTFOUND ) {
06366             numparam = AddDollar(name,DOLUNDEFINED,0,0);
06367         }
06368         *w++ = numparam;
06369         *inp = c;
06370         AddPotModdollar(numparam);
06371     }
06372     w++;  /* space for the level of the enddo statement */
06373     while ( *inp == ',' ) inp++;
06374     if ( *inp != '=' ) goto IllSyntax;
06375     inp++;
06376     while ( *inp == ',' ) inp++;
06377 /*
06378     The start value
06379 */
06380     inp = GetDoParam(inp,&w,1);
06381     if ( inp == 0 || *inp != ',' ) goto IllSyntax;
06382     while ( *inp == ',' ) inp++;
06383 /*
06384     The end value
06385 */
06386     inp = GetDoParam(inp,&w,1);
06387     if ( inp == 0 || ( *inp != 0 && *inp != ',' ) ) goto IllSyntax;
06388 /*
06389     The increment value
06390 */
06391     if ( *inp != ',' ) {
06392         if ( *inp == 0 ) { *w++ = SNUMBER; *w++ = 1; }
06393         else goto IllSyntax;
06394     }
06395     else {
06396         while ( *inp == ',' ) inp++;
06397         inp = GetDoParam(inp,&w,1);
06398     }
06399     if ( inp == 0 || *inp != 0 ) goto IllSyntax;
06400     *w = 0;
06401     AT.WorkPointer[1] = w - AT.WorkPointer;
06402 /*
06403     Put away and set information for placing enddo information.
06404 */
06405     AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
06406     AC.doloopstack[AC.doolooplevel++] = C->numlhs;
06407
06408     return(error);
06409
06410 IllSyntax:
06411     MesPrint("&Illegal syntax for do statement");
06412     return(1);
06413 }
06414
06415 /*
06416     #] CoDo :
06417     #[ CoEndDo :
06418 */
06419
06420 int CoEndDo(UBYTE *inp)
06421 {
```

```
06422     CBUF *C = cbuf+AC.cbufnum;
06423     WORD scratch[3];
06424     while ( *inp == ',' ) inp++;
06425     if ( *inp ) {
06426         MesPrint("&Illegal syntax for EndDo statement");
06427         return(1);
06428     }
06429     if ( AC.dolooplevel <= 0 ) {
06430         MesPrint("&EndDo without corresponding Do statement");
06431         return(1);
06432     }
06433     AC.dolooplevel--;
06434     scratch[0] = TYPEENDDOLOOP;
06435     scratch[1] = 3;
06436     scratch[2] = AC.doloopstack[AC.dolooplevel];
06437     AddNtoL(3,scratch);
06438     cbuf[AC.cbufnum].lhs[AC.doloopstack[AC.dolooplevel]][3] = C->numlhs;
06439     if ( AC.doloopnest[AC.dolooplevel] != NestingChecksum() ) {
06440         MesNesting();
06441         return(1);
06442     }
06443     return(0);
06444 }
06445
06446 /*
06447     #] CoEndDo :
06448     #[ CoFactDollar :
06449 */
06450
06451 int CoFactDollar(UBYTE *inp)
06452 {
06453     WORD numdollar;
06454     if ( *inp == '$' ) {
06455         if ( GetName(AC.dollarnames,inp+1,&numdollar,NOAUTO) != CDOLLAR ) {
06456             MesPrint("&%s is undefined",inp);
06457             numdollar = AddDollar(inp+1,DOLINDEX,&one,1);
06458             return(1);
06459         }
06460         inp = SkipAName(inp+1);
06461         if ( *inp != 0 ) {
06462             MesPrint("&FactDollar should have a single $variable for its argument");
06463             return(1);
06464         }
06465         AddPotModdollar(numdollar);
06466     }
06467     else {
06468         MesPrint("&%s is not a $-variable",inp);
06469         return(1);
06470     }
06471     Add3Com(TYPEFACTOR,numdollar);
06472     return(0);
06473 }
06474
06475 /*
06476     #] CoFactDollar :
06477     #[ CoFactorize :
06478 */
06479
06480 int CoFactorize(UBYTE *s) { return(DoFactorize(s,1)); }
06481
06482 /*
06483     #] CoFactorize :
06484     #[ CoNFactorize :
06485 */
06486
06487 int CoNFactorize(UBYTE *s) { return(DoFactorize(s,0)); }
06488
06489 /*
06490     #] CoNFactorize :
06491     #[ CoUnFactorize :
06492 */
06493
06494 int CoUnFactorize(UBYTE *s) { return(DoFactorize(s,3)); }
06495
06496 /*
06497     #] CoUnFactorize :
06498     #[ CoNUnFactorize :
06499 */
06500
06501 int CoNUnFactorize(UBYTE *s) { return(DoFactorize(s,2)); }
06502
06503 /*
06504     #] CoNUnFactorize :
06505     #[ DoFactorize :
06506 */
06507
06508 int DoFactorize(UBYTE *s,int par)
```

```
06509 {
06510     EXPRESSIONS e;
06511     WORD i;
06512     WORD number;
06513     UBYTE *t, c;
06514     int error = 0, keepzeroflag = 0;
06515     if ( *s == '(' ) {
06516         s++;
06517         while ( *s != ')' && *s ) {
06518             if ( FG.cTable[*s] == 0 ) {
06519                 t = s; while ( FG.cTable[*s] == 0 ) s++;
06520                 c = *s; *s = 0;
06521                 if ( StrICmp((UBYTE *)"keepzero",t) == 0 ) {
06522                     keepzeroflag = 1;
06523                 }
06524                 else {
06525                     MesPrint("&Illegal option in [N][Un]Factorize statement: %s",t);
06526                     error = 1;
06527                 }
06528                 *s = c;
06529             }
06530             while ( *s == ',' ) s++;
06531             if ( *s && *s != ')' && FG.cTable[*s] != 0 ) {
06532                 MesPrint("&Illegal character in option field of [N][Un]Factorize statement");
06533                 error = 1;
06534                 return(error);
06535             }
06536         }
06537         if ( *s ) s++;
06538         while ( *s == ',' || *s == ' ' ) s++;
06539     }
06540     if ( *s == 0 ) {
06541         for ( i = NumExpressions-1; i >= 0; i-- ) {
06542             e = Expressions+i;
06543             if ( e->replace >= 0 ) {
06544                 e = Expressions + e->replace;
06545             }
06546             if ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION
06547             || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION
06548           || e->status == INTOHIDELEXPRESSION || e->status == INTOHIDEGEXPRESSION
06549             ) {
06550                 switch ( par ) {
06551                     case 0:
06552                         e->vflags &= ~TOBEFACTORED;
06553                         break;
06554                     case 1:
06555                         e->vflags |=  TOBEFACTORED;
06556                         e->vflags &= ~TOBEUNFACTORED;
06557                         break;
06558                     case 2:
06559                         e->vflags &= ~TOBEUNFACTORED;
06560                         break;
06561                     case 3:
06562                         e->vflags |=  TOBEUNFACTORED;
06563                         e->vflags &= ~TOBEFACTORED;
06564                         break;
06565                 }
06566             }
06567             if ( ( e->vflags & TOBEFACTORED ) != 0 ) {
06568                 if ( keepzeroflag ) e->vflags |=  KEEPZERO;
06569                 else                e->vflags &= ~KEEPZERO;
06570             }
06571             else                    e->vflags &= ~KEEPZERO;
06572         }
06573     }
06574     else {
06575         for(;;) {   /* Look for a (comma separated) list of variables */
06576             while ( *s == ',' ) s++;
06577             if ( *s == 0 ) break;
06578             if ( *s == '[' || FG.cTable[*s] == 0 ) {
06579                 t = s;
06580                 if ( ( s = SkipAName(s) ) == 0 ) {
06581                     MesPrint("&Improper name for an expression: '%s'",t);
06582                     return(1);
06583                 }
06584                 c = *s; *s = 0;
06585                 if ( GetName(AC.exprnames,t,&number,NOAUTO) == CEXPRESSION ) {
06586                     e = Expressions+number;
06587                     if ( e->replace >= 0 ) {
06588                         e = Expressions + e->replace;
06589                     }
06590                     if ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION
06591                     || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION
06592                     || e->status == INTOHIDELEXPRESSION || e->status == INTOHIDEGEXPRESSION
06593                     ) {
06594                         switch ( par ) {
06595                             case 0:
```

```
06596                                   e->vflags &= ~TOBEFACTORED;
06597                                   break;
06598                               case 1:
06599                                   e->vflags |=  TOBEFACTORED;
06600                                   e->vflags &= ~TOBEUNFACTORED;
06601                                   break;
06602                               case 2:
06603                                   e->vflags &= ~TOBEUNFACTORED;
06604                                   break;
06605                               case 3:
06606                                   e->vflags |=  TOBEUNFACTORED;
06607                                   e->vflags &= ~TOBEFACTORED;
06608                                   break;
06609                           }
06610                       }
06611                       if ( ( e->vflags & TOBEFACTORED ) != 0 ) {
06612                           if ( keepzeroflag ) e->vflags |=  KEEPZERO;
06613                           else                e->vflags &= ~KEEPZERO;
06614                       }
06615                       else                    e->vflags &= ~KEEPZERO;
06616                   }
06617                   else if ( GetName(AC.varnames,t,&number,NOAUTO) != NAMENOTFOUND ) {
06618                       MesPrint("&%s is not an expression",t);
06619                       error = 1;
06620                   }
06621                   *s = c;
06622               }
06623               else {
06624                   MesPrint("&Illegal object in (N)Factorize statement");
06625                   error = 1;
06626                   while ( *s && *s != ',' ) s++;
06627                   if ( *s == 0 ) break;
06628               }
06629           }
06630
06631       }
06632       return(error);
06633   }
06634
06635   /*
06636       #] DoFactorize :
06637       #[ CoOptimizeOption :
06638
06639   */
06640
06641   int CoOptimizeOption(UBYTE *s)
06642   {
06643       UBYTE *name, *t1, *t2, c1, c2, *value, *u;
06644       int error = 0, x;
06645       double d;
06646       while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
06647       while ( *s ) {
06648           name = s; while ( FG.cTable[*s] == 0 ) s++;
06649           t1 = s; c1 = *t1;
06650           while ( *s == ' ' || *s == '\t' ) s++;
06651           if ( *s != '=' ) {
06652   correctuse:
06653               MesPrint("&Correct use in Format,Optimize statement is Optionname=value");
06654               error = 1;
06655               while ( *s == ' ' || *s == ',' || *s == '\t' || *s == '=' ) s++;
06656               *t1 = c1;
06657               continue;
06658           }
06659           *t1 = 0;
06660           s++;
06661           while ( *s == ' ' || *s == '\t' ) s++;
06662           if ( *s == 0 ) goto correctuse;
06663           value = s;
06664           while ( FG.cTable[*s] <= 1 || *s=='.' || *s=='*' || *s == '(' || *s == ')' ) {
06665               if ( *s == '(' ) { SKIPBRA4(s) }
06666               s++;
06667           }
06668           t2 = s; c2 = *t2;
06669           while ( *s == ' ' || *s == '\t' ) s++;
06670           if ( *s && *s != ',' ) goto correctuse;
06671           if ( *s ) {
06672               s++;
06673               while ( *s == ' ' || *s == '\t' ) s++;
06674           }
06675           *t2 = 0;
06676   /*
06677           Now we have name=value with name and value zero terminated strings.
06678   */
06679           if ( StrICmp(name,(UBYTE *)"horner") == 0 ) {
06680               if ( StrICmp(value,(UBYTE *)"occurrence") == 0 ) {
06681                   AO.Optimize.horner = O_OCCURRENCE;
06682               }
```

```
06683             else if ( StrICmp(value,(UBYTE *)"mcts") == 0 ) {
06684                 AO.Optimize.horner = O_MCTS;
06685             }
06686             else if ( StrICmp(value,(UBYTE *)"sa") == 0 ) {
06687                 AO.Optimize.horner = O_SIMULATED_ANNEALING;
06688             }
06689             else {
06690                 AO.Optimize.horner = -1;
06691                 MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s",name,value);
06692                 error = 1;
06693             }
06694         }
06695         else if ( StrICmp(name,(UBYTE *)"hornerdirection") == 0 ) {
06696             if ( StrICmp(value,(UBYTE *)"forward") == 0 ) {
06697                 AO.Optimize.hornerdirection = O_FORWARD;
06698             }
06699             else if ( StrICmp(value,(UBYTE *)"backward") == 0 ) {
06700                 AO.Optimize.hornerdirection = O_BACKWARD;
06701             }
06702             else if ( StrICmp(value,(UBYTE *)"forwardorbackward") == 0 ) {
06703                 AO.Optimize.hornerdirection = O_FORWARDORBACKWARD;
06704             }
06705             else if ( StrICmp(value,(UBYTE *)"forwardandbackward") == 0 ) {
06706                 AO.Optimize.hornerdirection = O_FORWARDANDBACKWARD;
06707             }
06708             else {
06709                 AO.Optimize.method = -1;
06710                 MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s",name,value);
06711                 error = 1;
06712             }
06713         }
06714         else if ( StrICmp(name,(UBYTE *)"method") == 0 ) {
06715             if ( StrICmp(value,(UBYTE *)"none") == 0 ) {
06716                 AO.Optimize.method = O_NONE;
06717             }
06718             else if ( StrICmp(value,(UBYTE *)"cse") == 0 ) {
06719                 AO.Optimize.method = O_CSE;
06720             }
06721             else if ( StrICmp(value,(UBYTE *)"csegreedy") == 0 ) {
06722                 AO.Optimize.method = O_CSEGREEDY;
06723             }
06724             else if ( StrICmp(value,(UBYTE *)"greedy") == 0 ) {
06725                 AO.Optimize.method = O_GREEDY;
06726             }
06727             else {
06728                 AO.Optimize.method = -1;
06729                 MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s",name,value);
06730                 error = 1;
06731             }
06732         }
06733         else if ( StrICmp(name,(UBYTE *)"timelimit") == 0 ) {
06734             x = 0;
06735             u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06736             if ( *u != 0 ) {
06737                 MesPrint("&Option TimeLimit in Format,Optimize statement should be a positive number:
      %s",value);
06738                 AO.Optimize.mctstimelimit = 0;
06739                 AO.Optimize.greedytimelimit = 0;
06740                 error = 1;
06741             }
06742             else {
06743                 AO.Optimize.mctstimelimit = x/2;
06744                 AO.Optimize.greedytimelimit = x/2;
06745             }
06746         }
06747         else if ( StrICmp(name,(UBYTE *)"mctstimelimit") == 0 ) {
06748             x = 0;
06749             u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06750             if ( *u != 0 ) {
06751                 MesPrint("&Option MCTSTimeLimit in Format,Optimize statement should be a positive
      number: %s",value);
06752                 AO.Optimize.mctstimelimit = 0;
06753                 error = 1;
06754             }
06755             else {
06756                 AO.Optimize.mctstimelimit = x;
06757             }
06758         }
06759         else if ( StrICmp(name,(UBYTE *)"mctsnumexpand") == 0 ) {
06760             int y;
06761             x = 0;
06762             u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06763             if ( *u == '*' || *u == 'x' || *u == 'X' ) {
06764                 u++; y = x;
06765                 x = 0;
06766                 while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06767             }
```

```
06768             else { y = 1; }
06769             if ( *u != 0 ) {
06770                 MesPrint("&Option MCTSNumExpand in Format,Optimize statement should be a positive
      number: %s",value);
06771                 AO.Optimize.mctsnumexpand= 0;
06772                 AO.Optimize.mctsnumrepeat= 1;
06773                 error = 1;
06774             }
06775             else {
06776                 AO.Optimize.mctsnumexpand= x;
06777                 AO.Optimize.mctsnumrepeat= y;
06778             }
06779         }
06780         else if ( StrICmp(name,(UBYTE *)"mctsnumrepeat") == 0 ) {
06781             x = 0;
06782             u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06783             if ( *u != 0 ) {
06784                 MesPrint("&Option MCTSNumExpand in Format,Optimize statement should be a positive
      number: %s",value);
06785                 AO.Optimize.mctsnumrepeat= 1;
06786                 error = 1;
06787             }
06788             else {
06789                 AO.Optimize.mctsnumrepeat= x;
06790             }
06791         }
06792         else if ( StrICmp(name,(UBYTE *)"mctsnumkeep") == 0 ) {
06793             x = 0;
06794             u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06795             if ( *u != 0 ) {
06796                 MesPrint("&Option MCTSNumKeep in Format,Optimize statement should be a positive
      number: %s",value);
06797                 AO.Optimize.mctsnumkeep= 0;
06798                 error = 1;
06799             }
06800             else {
06801                 AO.Optimize.mctsnumkeep= x;
06802             }
06803         }
06804         else if ( StrICmp(name,(UBYTE *)"mctsconstant") == 0 ) {
06805             d = 0;
06806             if ( sscanf ((char*)value, "%lf", &d) != 1 ) {
06807                 MesPrint("&Option MCTSConstant in Format,Optimize statement should be a positive
      number: %s",value);
06808                 AO.Optimize.mctsconstant.fval = 0;
06809                 error = 1;
06810             }
06811             else {
06812                 AO.Optimize.mctsconstant.fval = d;
06813             }
06814         }
06815         else if ( StrICmp(name,(UBYTE *)"greedytimelimit") == 0 ) {
06816             x = 0;
06817             u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06818             if ( *u != 0 ) {
06819                 MesPrint("&Option GreedyTimeLimit in Format,Optimize statement should be a positive
      number: %s",value);
06820                 AO.Optimize.greedytimelimit = 0;
06821                 error = 1;
06822             }
06823             else {
06824                 AO.Optimize.greedytimelimit = x;
06825             }
06826         }
06827         else if ( StrICmp(name,(UBYTE *)"greedyminnum") == 0 ) {
06828             x = 0;
06829             u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06830             if ( *u != 0 ) {
06831                 MesPrint("&Option GreedyMinNum in Format,Optimize statement should be a positive
      number: %s",value);
06832                 AO.Optimize.greedyminnum= 0;
06833                 error = 1;
06834             }
06835             else {
06836                 AO.Optimize.greedyminnum= x;
06837             }
06838         }
06839         else if ( StrICmp(name,(UBYTE *)"greedymaxperc") == 0 ) {
06840             x = 0;
06841             u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06842             if ( *u != 0 ) {
06843                 MesPrint("&Option GreedyMaxPerc in Format,Optimize statement should be a positive
      number: %s",value);
06844                 AO.Optimize.greedymaxperc= 0;
06845                 error = 1;
06846             }
06847             else {
```

```
06848                 AO.Optimize.greedymaxperc= x;
06849             }
06850         }
06851         else if ( StrICmp(name,(UBYTE *)"stats") == 0 ) {
06852             if ( StrICmp(value,(UBYTE *)"on") == 0 ) {
06853                 AO.Optimize.printstats = 1;
06854             }
06855             else if ( StrICmp(value,(UBYTE *)"off") == 0 ) {
06856                 AO.Optimize.printstats = 0;
06857             }
06858             else {
06859                 AO.Optimize.printstats = 0;
06860                 MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s",name,value);
06861                 error = 1;
06862             }
06863         }
06864         else if ( StrICmp(name,(UBYTE *)"printscheme") == 0 ) {
06865             if ( StrICmp(value,(UBYTE *)"on") == 0 ) {
06866                 AO.Optimize.schemeflags |= 1;
06867             }
06868             else if ( StrICmp(value,(UBYTE *)"off") == 0 ) {
06869                 AO.Optimize.schemeflags &= ~1;
06870             }
06871             else {
06872                 AO.Optimize.schemeflags &= ~1;
06873                 MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s",name,value);
06874                 error = 1;
06875             }
06876         }
06877         else if ( StrICmp(name,(UBYTE *)"debugflag") == 0 ) {
06878 /*
06879             This option is for debugging purposes only. Not in the manual!
06880             0x1: Print statements in reverse order.
06881             0x2: Print the scheme of the variables.
06882 */
06883             x = 0;
06884             u = value;
06885             if ( FG.cTable[*u] == 1 ) {
06886                 while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06887                 if ( *u != 0 ) {
06888                     MesPrint("&Numerical value for DebugFlag in Format,Optimize statement should be a
      nonnegative number: %s",value);
06889                     AO.Optimize.debugflags = 0;
06890                     error = 1;
06891                 }
06892                 else {
06893                     AO.Optimize.debugflags = x;
06894                 }
06895             }
06896             else if ( StrICmp(value,(UBYTE *)"on") == 0 ) {
06897                 AO.Optimize.debugflags = 1;
06898             }
06899             else if ( StrICmp(value,(UBYTE *)"off") == 0 ) {
06900                 AO.Optimize.debugflags = 0;
06901             }
06902             else {
06903                 AO.Optimize.debugflags = 0;
06904                 MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s",name,value);
06905                 error = 1;
06906             }
06907         }
06908         else if ( StrICmp(name,(UBYTE *)"scheme") == 0 ) {
06909             UBYTE *ss, *s1, c;
06910             WORD type, numsym;
06911             AO.schemenum = 0;
06912             u = value;
06913             if ( *u != '(' ) {
06914 noscheme:
06915                 MesPrint("&Option Scheme in Format,Optimize statement should be an array of names or
      integers between (): %s",value);
06916                 error = 1;
06917                 break;
06918             }
06919             u++; ss = u;
06920             while ( *ss == ' ' || *ss == '\t' || *ss == ',' ) ss++;
06921             if ( FG.cTable[*ss] == 0 || *ss == '$' || *ss == '[' ) { /* Name */
06922                 s1 = u; SKIPBRA3(s1)
06923                 if ( *s1 != ')' ) goto noscheme;
06924                 while ( ss < s1 ) { if ( *ss++ == ',' ) AO.schemenum++; }
06925                 *ss++ = 0; while ( *ss == ' ' ) ss++;
06926                 if ( *ss != 0 ) goto noscheme;
06927                 ss = u;
06928                 if ( AO.schemenum < 1 ) {
06929                     MesPrint("&Option Scheme in Format,Optimize statement should have at least one
      name or number between ()");
06930                     error = 1;
06931                     break;
```

```
06932                 }
06933                 if ( AO.inscheme ) M_free(AO.inscheme,"Horner input scheme");
06934                 AO.inscheme = (WORD *)Malloc1((AO.schemenum+1)*sizeof(WORD),"Horner input scheme");
06935                 while ( *ss == ' ' || *ss == '\t' || *ss == ',' ) ss++;
06936                 AO.schemenum = 0;
06937                 for(;;) {
06938                     if ( *ss == 0 ) break;
06939                     s1 = ss; ss = SkipAName(s1); c = *ss; *ss = 0;
06940 
06941                     if ( ss[-1] == '_' ) {
06942 /*
06943                         Now AC.extrasym followed by a number and _
06944 */
06945                         UBYTE *u1, *u2;
06946                         u1 = s1; u2 = AC.extrasym;
06947                         while ( *u1 == *u2 ) { u1++; u2++; }
06948                         if ( *u2 == 0 ) { /* Good start */
06949                             numsym = 0;
06950                             while ( *u1 >= '0' && *u1 <= '9' ) numsym = 10*numsym + *u1++ - '0';
06951                             if ( u1 != ss-1 || numsym == 0 || AC.extrasymbols != 0 ) {
06952                                 MesPrint("&Improper use of extra symbol in scheme format option");
06953                                 goto noscheme;
06954                             }
06955                             numsym = MAXVARIABLES-numsym;
06956                             ss++;
06957                             goto GotTheNumber;
06958                         }
06959                     }
06960                     else if ( *s1 == '$' ) {
06961                         GETIDENTITY
06962                         int numdollar;
06963                         if ( ( numdollar = GetDollar(s1+1) ) < 0 ) {
06964                             MesPrint("&Undefined variable %s",s1);
06965                             error = 5;
06966                         }
06967                         else if ( ( numsym = DolToSymbol(BHEAD numdollar) ) < 0 ) {
06968                             MesPrint("&$%s does not evaluate to a symbol",s1);
06969                             error = 5;
06970                         }
06971                         *ss = c;
06972                         goto GotTheNumber;
06973                     }
06974                     else if ( c == '(' ) {
06975                         if ( StrCmp(s1,AC.extrasym) == 0 ) {
06976                             if ( (AC.extrasymbols&1) != 1 ) {
06977                                 MesPrint("&Improper use of extra symbol in scheme format option");
06978                                 goto noscheme;
06979                             }
06980                             *ss++ = c;
06981                             numsym = 0;
06982                             while ( *ss >= '0' && *ss <= '9' ) numsym = 10*numsym + *ss++ - '0';
06983                             if ( *ss != ')' ) {
06984                                 MesPrint("&Extra symbol should have a number for its argument.");
06985                                 goto noscheme;
06986                             }
06987                             numsym = MAXVARIABLES-numsym;
06988                             ss++;
06989                             goto GotTheNumber;
06990                         }
06991                     }
06992                     type = GetName(AC.varnames,s1,&numsym,WITHAUTO);
06993                     if ( ( type != CSYMBOL ) && type != CDUBIOUS ) {
06994                         MesPrint("&%s is not a symbol",s1);
06995                         error = 4;
06996                         if ( type < 0 ) numsym = AddSymbol(s1,-MAXPOWER,MAXPOWER,0,0);
06997                     }
06998                     *ss = c;
06999 GotTheNumber:
07000                     AO.inscheme[AO.schemenum++] = numsym;
07001                     while ( *ss == ' ' || *ss == '\t' || *ss == ',' ) ss++;
07002                 }
07003             }
07004         }
07005         else if ( StrICmp(name,(UBYTE *)"mctsdecaymode") == 0 ) {
07006             x = 0;
07007             u = value;
07008             if ( FG.cTable[*u] == 1 ) {
07009                 while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
07010                 if ( *u != 0 ) {
07011                     MesPrint("&Option MCTSDecayMode in Format,Optimize statement should be a
      nonnegative integer: %s",value);
07012                     AO.Optimize.mctsdecaymode = 0;
07013                     error = 1;
07014                 }
07015                 else {
07016                     AO.Optimize.mctsdecaymode = x;
07017                 }
```

```
07018             }
07019             else {
07020                 AO.Optimize.mctsdecaymode = 0;
07021                 MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s",name,value);
07022                 error = 1;
07023             }
07024         }
07025         else if ( StrICmp(name,(UBYTE *)"saiter") == 0 ) {
07026             x = 0;
07027             u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
07028             if ( *u != 0 ) {
07029                 MesPrint("&Option SAIter in Format,Optimize statement should be a positive integer:
      %s",value);
07030                 AO.Optimize.saIter = 0;
07031                 error = 1;
07032             }
07033             else {
07034                 AO.Optimize.saIter= x;
07035             }
07036         }
07037         else if ( StrICmp(name,(UBYTE *)"samaxt") == 0 ) {
07038             d = 0;
07039             if ( sscanf ((char*)value, "%lf", &d) != 1 ) {
07040                 MesPrint("&Option SAMaxT in Format,Optimize statement should be a positive number:
      %s",value);
07041                 AO.Optimize.saMaxT.fval = 0;
07042                 error = 1;
07043             }
07044             else {
07045                 AO.Optimize.saMaxT.fval = d;
07046             }
07047         }
07048         else if ( StrICmp(name,(UBYTE *)"samint") == 0 ) {
07049             d = 0;
07050             if ( sscanf ((char*)value, "%lf", &d) != 1 ) {
07051                 MesPrint("&Option SAMinT in Format,Optimize statement should be a positive number:
      %s",value);
07052                 AO.Optimize.saMinT.fval = 0;
07053                 error = 1;
07054             }
07055             else {
07056                 AO.Optimize.saMinT.fval = d;
07057             }
07058         }
07059         else {
07060             MesPrint("&Unrecognized option name in Format,Optimize statement: %s",name);
07061             error = 1;
07062         }
07063         *t1 = c1; *t2 = c2;
07064     }
07065     return(error);
07066 }
07067
07068 /*
07069     #] CoOptimizeOption :
07070     #[ DoPutInside :
07071
07072     Syntax:
07073         PutIn[side],functionname[,brackets]  -> par = 1
07074         AntiPutIn[side],functionname,antibrackets  -> par = -1
07075 */
07076
07077 int CoPutInside(UBYTE *inp) { return(DoPutInside(inp,1)); }
07078 int CoAntiPutInside(UBYTE *inp) { return(DoPutInside(inp,-1)); }
07079
07080 int DoPutInside(UBYTE *inp, int par)
07081 {
07082     GETIDENTITY
07083     UBYTE *p, c;
07084     WORD *to, type, c1,c2,funnum, *WorkSave;
07085     int error = 0;
07086     while ( *inp == ' ' || *inp == '\t' || *inp == ',' ) inp++;
07087 /*
07088     First we need the name of a function. (Not a tensor or table!)
07089 */
07090     p = SkipAName(inp);
07091     if ( p == 0 ) return(1);
07092     c = *p; *p = 0;
07093     type = GetName(AC.varnames,inp,&funnum,WITHAUTO);
07094     if ( type != CFUNCTION || functions[funnum].tabl != 0 || functions[funnum].spec ) {
07095         MesPrint("&PutInside/AntiPutInside expects a regular function for its first argument");
07096         MesPrint("&Argument is %s",inp);
07097         error = 1;
07098     }
07099     funnum += FUNCTION;
07100     *p = c;
07101     inp = p;
```

```
07102     while ( *inp == ' ' || *inp == '\t' || *inp == ',' ) inp++;
07103     if ( *inp == 0 ) {
07104         if ( par == 1 ) {
07105             WORD tocompiler[4];
07106             tocompiler[0] = TYPEPUTINSIDE;
07107             tocompiler[1] = 4;
07108             tocompiler[2] = 0;
07109             tocompiler[3] = funnum;
07110             AddNtoL(4,tocompiler);
07111         }
07112         else {
07113             MesPrint("&AntiPutInside needs inside information.");
07114             error = 1;
07115         }
07116         return(error);
07117     }
07118     WorkSave = to = AT.WorkPointer;
07119     *to++ = TYPEPUTINSIDE;
07120     *to++ = 4;
07121     *to++ = par;
07122     *to++ = funnum;
07123     to++;
07124     while ( *inp ) {
07125         while ( *inp == ' ' || *inp == '\t' || *inp == ',' ) inp++;
07126         if ( *inp == 0 ) break;
07127         p = SkipAName(inp);
07128         if ( p == 0 ) { error = 1; break; }
07129         c = *p; *p = 0;
07130         type = GetName(AC.varnames,inp,&c1,WITHAUTO);
07131         if ( c == '.' ) {
07132             if ( type == CVECTOR || type == CDUBIOUS ) {
07133                 *p++ = c;
07134                 inp = p;
07135                 p = SkipAName(inp);
07136                 if ( p == 0 ) return(1);
07137                 c = *p; *p = 0;
07138                 type = GetName(AC.varnames,inp,&c2,WITHAUTO);
07139                 if ( type != CVECTOR && type != CDUBIOUS ) {
07140                     MesPrint("&Not a vector in dotproduct in PutInside/AntiPutInside statement:
      %s",inp);
07141                     error = 1;
07142                 }
07143                 else type = CDOTPRODUCT;
07144             }
07145             else {
07146                 MesPrint("&Illegal use of . after %s in PutInside/AntiPutInside statement",inp);
07147                 error = 1;
07148                 *p = c; inp = p;
07149                 continue;
07150             }
07151         }
07152         switch ( type ) {
07153             case CSYMBOL :
07154                 *to++ = SYMBOL; *to++ = 4; *to++ = c1; *to++ = 1; break;
07155             case CVECTOR :
07156                 *to++ = INDEX; *to++ = 3; *to++ = AM.OffsetVector + c1; break;
07157             case CFUNCTION :
07158                 *to++ = c1+FUNCTION; *to++ = FUNHEAD; *to++ = 0;
07159                 FILLFUN3(to)
07160                 break;
07161             case CDOTPRODUCT :
07162                 *to++ = DOTPRODUCT; *to++ = 5; *to++ = c1 + AM.OffsetVector;
07163                 *to++ = c2 + AM.OffsetVector; *to++ = 1; break;
07164             case CDELTA :
07165                 *to++ = DELTA; *to++ = 4; *to++ = EMPTYINDEX; *to++ = EMPTYINDEX; break;
07166             default :
07167                 MesPrint("&Illegal variable request for %s in PutInside/AntiPutInside statement",inp);
07168                 error = 1; break;
07169         }
07170         *p = c;
07171         inp = p;
07172     }
07173     *to++ = 1; *to++ = 1; *to++ = 3;
07174     AT.WorkPointer[1] = to - AT.WorkPointer;
07175     AT.WorkPointer[4] = AT.WorkPointer[1]-4;
07176     AT.WorkPointer = to;
07177     AC.BracketNormalize = 1;
07178     if ( Normalize(BHEAD WorkSave+4) ) { error = 1; }
07179     else {
07180         WorkSave[1] = WorkSave[4]+4;
07181         to = WorkSave + WorkSave[1] - 1;
07182         c1 = ABS(*to);
07183         WorkSave[1] -= c1;
07184         WorkSave[4] -= c1;
07185         AddNtoL(WorkSave[1],WorkSave);
07186     }
07187     AC.BracketNormalize = 0;
```

```
07188     AT.WorkPointer = WorkSave;
07189     return(error);
07190 }
07191
07192 /*
07193     #] DoPutInside :
07194     #[ CoSwitch :
07195
07196     Syntax: Switch $var;
07197     Be careful with illegal nestings with repeat, if, while.
07198 */
07199
07200 int CoSwitch(UBYTE *s)
07201 {
07202     WORD numdollar;
07203     SWITCH *sw;
07204     if ( *s == '$' ) {
07205         if ( GetName(AC.dollarnames,s+1,&numdollar,NOAUTO) != CDOLLAR ) {
07206             MesPrint("&%s is undefined in switch statement",s);
07207             numdollar = AddDollar(s+1,DOLINDEX,&one,1);
07208             return(1);
07209         }
07210         s = SkipAName(s+1);
07211         if ( *s != 0 ) {
07212             MesPrint("&Switch should have a single $variable for its argument");
07213             return(1);
07214         }
07215 /*      AddPotModdollar(numdollar);  */
07216     }
07217     else {
07218         MesPrint("&%s is not a $-variable in switch statement",s);
07219         return(1);
07220     }
07221 /*
07222     Now create the switch table. We will add to it each time we run
07223     into a new case. It will all be sorted out the moment we run into
07224     the endswitch statement.
07225 */
07226     AC.SwitchLevel++;
07227     if ( AC.SwitchInArray >= AC.MaxSwitch ) DoubleSwitchBuffers();
07228     AC.SwitchHeap[AC.SwitchLevel] = AC.SwitchInArray;
07229     sw = AC.SwitchArray + AC.SwitchInArray;
07230
07231     sw->iflevel = AC.IfLevel;
07232     sw->whilelevel = AC.WhileLevel;
07233     sw->nestingsum = NestingChecksum();
07234
07235     Add4Com(TYPESWITCH,numdollar,AC.SwitchInArray);
07236
07237     AC.SwitchInArray++;
07238     return(0);
07239 }
07240
07241 /*
07242     #] CoSwitch :
07243     #[ CoCase :
07244 */
07245
07246 int CoCase(UBYTE *s)
07247 {
07248     SWITCH *sw = AC.SwitchArray + AC.SwitchHeap[AC.SwitchLevel];
07249     WORD x = 0, sign = 1;
07250     while ( *s == ',' ) s++;
07251     SKIPBLANKS(s);
07252     while ( *s == '-' || *s == '+' ) {
07253         if ( *s == '-' ) sign = -sign;
07254         s++;
07255     }
07256     while ( FG.cTable[*s] == 1 ) { x = 10*x + *s++ - '0'; }
07257     x = sign*x;
07258
07259     if ( sw->iflevel != AC.IfLevel || sw->whilelevel != AC.WhileLevel
07260         || sw->nestingsum != NestingChecksum() ) {
07261         MesPrint("&Illegal nesting of switch/case/default with
      if/while/repeat/loop/argument/term/...");
07262         return(-1);
07263     }
07264 /*
07265     Now add a case to the table with the current 'address'.
07266 */
07267     if ( sw->numcases >= sw->tablesize ) {
07268         int i;
07269         SWITCHTABLE *newtable;
07270         WORD newsize;
07271         if ( sw->tablesize == 0 ) newsize = 10;
07272         else                      newsize = 2*sw->tablesize;
07273         newtable = (SWITCHTABLE *)Malloc1(newsize*sizeof(SWITCHTABLE),"Switch table");
```

```
07274         if ( sw->table ) {
07275             for ( i = 0; i < sw->tablesize; i++ ) newtable[i] = sw->table[i];
07276             M_free(sw->table,"Switch table");
07277         }
07278         sw->table = newtable;
07279         sw->tablesize = newsize;
07280     }
07281     if ( sw->numcases == 0 ) { sw->mincase = sw->maxcase = x; }
07282     else if ( x > sw->maxcase ) sw->maxcase = x;
07283     else if ( x < sw->mincase ) sw->mincase = x;
07284     sw->table[sw->numcases].ncase = x;
07285     sw->table[sw->numcases].value = cbuf[AC.cbufnum].numlhs;
07286     sw->table[sw->numcases].compbuffer = AC.cbufnum;
07287     sw->numcases++;
07288     return(0);
07289 }
07290
07291 /*
07292     #] CoCase :
07293     #[ CoBreak :
07294 */
07295
07296 int CoBreak(UBYTE *s)
07297 {
07298 /*
07299     This involves a 'postponed' jump to the end. This can be done
07300     in a special routine during execution.
07301     That routine should also pop the switch level.
07302 */
07303     SWITCH *sw = AC.SwitchArray + AC.SwitchHeap[AC.SwitchLevel];
07304     if ( sw->iflevel != AC.IfLevel || sw->whilelevel != AC.WhileLevel
07305         || sw->nestingsum != NestingChecksum() ) {
07306         MesPrint("&Illegal nesting of switch/case/default with
    if/while/repeat/loop/argument/term/...");
07307         return(-1);
07308     }
07309     if ( *s ) {
07310         MesPrint("&No parameters allowed in Break statement");
07311         return(-1);
07312     }
07313     Add3Com(TYPEENDSWITCH,AC.SwitchHeap[AC.SwitchLevel]);
07314     return(0);
07315 }
07316
07317 /*
07318     #] CoBreak :
07319     #[ CoDefault :
07320 */
07321
07322 int CoDefault(UBYTE *s)
07323 {
07324 /*
07325     A bit like case, except that the address gets stored directly in the
07326     SWITCH struct.
07327 */
07328     SWITCH *sw = AC.SwitchArray + AC.SwitchHeap[AC.SwitchLevel];
07329     if ( sw->iflevel != AC.IfLevel || sw->whilelevel != AC.WhileLevel
07330         || sw->nestingsum != NestingChecksum() ) {
07331         MesPrint("&Illegal nesting of switch/case/default with
    if/while/repeat/loop/argument/term/...");
07332         return(-1);
07333     }
07334     if ( *s ) {
07335         MesPrint("&No parameters allowed in Default statement");
07336         return(-1);
07337     }
07338     sw->defaultcase.ncase = 0;
07339     sw->defaultcase.value = cbuf[AC.cbufnum].numlhs;
07340     sw->defaultcase.compbuffer = AC.cbufnum;
07341     return(0);
07342 }
07343
07344 /*
07345     #] CoDefault :
07346     #[ CoEndSwitch :
07347 */
07348
07349 int CoEndSwitch(UBYTE *s)
07350 {
07351 /*
07352     We store this address in the SWITCH struct.
07353     Next we sort the table by ncase.
07354     Then we decide whether the table is DENSE or SPARSE.
07355     If it is dense we change the allocation and spread the cases is necessary.
07356     Finally we pop levels.
07357 */
07358     SWITCH *sw = AC.SwitchArray + AC.SwitchHeap[AC.SwitchLevel];
```

```
07359     WORD i;
07360     WORD totcases = sw->maxcase-sw->mincase+1;
07361     while ( *s == ',' ) s++;
07362     SKIPBLANKS(s)
07363     if ( *s ) {
07364         MesPrint("&No parameters allowed in EndSwitch statement");
07365         return(-1);
07366     }
07367     if ( sw->iflevel != AC.IfLevel || sw->whilelevel != AC.WhileLevel
07368         || sw->nestingsum != NestingChecksum() ) {
07369         MesPrint("&Illegal nesting of switch/case/default with
      if/while/repeat/loop/argument/term/...");
07370         return(-1);
07371     }
07372     if ( sw->defaultcase.value == 0 ) CoDefault(s);
07373     if ( totcases > sw->numcases*AM.jumpratio ) { /* The factor is experimental */
07374         sw->caseoffset = 0;
07375         sw->typetable = SPARSETABLE;
07376 /*
07377         Now we need to sort sw->table
07378 */
07379         SwitchSplitMerge(sw->table,sw->numcases);
07380     }
07381     else {  /* DENSE */
07382         SWITCHTABLE *ntable;
07383         sw->caseoffset = sw->mincase;
07384         sw->typetable = DENSETABLE;
07385         ntable = (SWITCHTABLE *)Malloc1(totcases*sizeof(SWITCHTABLE),"Switch table");
07386         for ( i = 0; i < totcases; i++ ) {
07387             ntable[i].ncase = i+sw->caseoffset;
07388             ntable[i].value = sw->defaultcase.value;
07389             ntable[i].compbuffer = sw->defaultcase.compbuffer;
07390         }
07391         for ( i = 0; i < sw->numcases; i++ ) {
07392             ntable[sw->table[i].ncase-sw->caseoffset] = sw->table[i];
07393         }
07394         M_free(sw->table,"Switch table");
07395         sw->table = ntable;
07396         sw->numcases = totcases;
07397     }
07398     sw->endswitch.ncase = 0;
07399     sw->endswitch.value = cbuf[AC.cbufnum].numlhs;
07400     sw->endswitch.compbuffer = AC.cbufnum;
07401     if ( sw->defaultcase.value == 0 ) {
07402         sw->defaultcase = sw->endswitch;
07403     }
07404     Add3Com(TYPEENDSWITCH,AC.SwitchHeap[AC.SwitchLevel]);
07405 /*
07406     Now we need to pop.
07407 */
07408     AC.SwitchLevel--;
07409     return(0);
07410 }
07411
07412 /*
07413     #] CoEndSwitch :
07414     #[ CoSetUserFlag :
07415 */
07416 /* UNFINISHED_FEATURE_EXCL_START */
07417 int CoSetUserFlag(UBYTE *s)
07418 {
07419     int error = 0;
07420     while ( *s && ( *s == ',' || *s == ' ' || *s == '\t' ) ) s++;
07421     while ( *s && ( FG.cTable[*s] == 1 ) ) {
07422         int x = 0;
07423         while ( *s && ( FG.cTable[*s] == 1 ) ) x = 10*x+(*s++ - '0');
07424         if ( x < 1 || x > BITSINWORD ) {
07425             MesPrint("&Flag number %d outside the permitted range 1-%d.",BITSINWORD);
07426             error = 1;
07427         }
07428         else {
07429             Add3Com(TYPESETUSERFLAG,x-1);
07430         }
07431         while ( *s && ( *s == ',' || *s == ' ' || *s == '\t' ) ) s++;
07432     }
07433     if ( *s ) {
07434         MesPrint("&Illegal character in SetUserFlag statement: %s",s);
07435         error = 1;
07436     }
07437     return(error);
07438 }
07439 /* UNFINISHED_FEATURE_EXCL_STOP */
07440 /*
07441     #] CoSetUserFlag :
07442     #[ CoClearUserFlag :
07443 */
07444 /* UNFINISHED_FEATURE_EXCL_START */
```

```
07445 int CoClearUserFlag(UBYTE *s)
07446 {
07447     int error = 0;
07448     while ( *s && ( *s == ',' || *s == ' ' || *s == '\t' ) ) s++;
07449     while ( *s && ( FG.cTable[*s] == 1 ) ) {
07450         int x = 0;
07451         while ( *s && ( FG.cTable[*s] == 1 ) ) x = 10*x+(*s++ - '0');
07452         if ( x < 1 || x > BITSINWORD ) {
07453             MesPrint("&Flag number %d outside the permitted range 1-%d.",BITSINWORD);
07454             error = 1;
07455         }
07456         else {
07457             Add3Com(TYPECLEARUSERFLAG,x);
07458         }
07459         while ( *s && ( *s == ',' || *s == ' ' || *s == '\t' ) ) s++;
07460     }
07461     if ( *s ) {
07462         MesPrint("&Illegal character in SetUserFlag statement: %s",s);
07463         error = 1;
07464     }
07465     return(error);
07466 }
07467 /* UNFINISHED_FEATURE_EXCL_STOP */
07468 /*
07469     #] CoClearUserFlag :
07470     #[ CoCreateAllLoops :
07471
07472     Syntax:
07473         CoCreateAllLoops,in-function,out-function,type-argument,ifnoloop;
07474     Types allowed:
07475         vector, index, symbol, snumber
07476     ifnoloop:
07477         ifnoloop=0  or ifnoloop=1
07478     in-function can be a tensor. In that case type can be only vector or index.
07479     out-function can be a tensor. In that case type can be only vector or index.
07480 */
07481
07482 int CoCreateAllLoops(UBYTE *s)
07483 {
07484     GETIDENTITY
07485     UBYTE *inname, *outname, *stype, c;
07486     WORD infun, outfun, x, type, tensorflag, typenum;
07487     WORD *WorkSave, *to;
07488     while ( *s == ',' || *s == ' ' ) s++;
07489     inname = s; s = SkipAName(s);
07490     c = *s; *s = 0;
07491     if ( ( ( type = GetName(AC.varnames,inname,&infun,WITHAUTO) ) != CFUNCTION )
07492     || ( ( functions[infun].spec != 0 ) && ( functions[infun].spec != TENSORFUNCTION ) ) ) {
07493         MesPrint("&%s should be a regular function or a tensor.",inname);
07494         if ( type < 0 ) {
07495             if ( GetName(AC.exprnames,s,&infun,NOAUTO) == NAMENOTFOUND )
07496                 AddFunction(s,0,0,0,0,0,-1,-1);
07497         }
07498         return(1);
07499     }
07500     infun += FUNCTION;
07501     *s++ = c;
07502     while ( *s == ',' || *s == ' ' ) s++;
07503     outname = s; s = SkipAName(s);
07504     c = *s; *s = 0;
07505     if ( ( ( type = GetName(AC.varnames,outname,&outfun,WITHAUTO) ) != CFUNCTION )
07506     || ( ( functions[outfun].spec != 0 ) && ( functions[outfun].spec != TENSORFUNCTION ) ) ) {
07507         MesPrint("&%s should be a regular function or a tensor.",outname);
07508         if ( type < 0 ) {
07509             if ( GetName(AC.exprnames,s,&outfun,NOAUTO) == NAMENOTFOUND )
07510                 AddFunction(s,0,0,0,0,0,-1,-1);
07511         }
07512         return(1);
07513     }
07514     outfun += FUNCTION;
07515     *s++ = c;
07516     if ( functions[infun].spec == TENSORFUNCTION ||
07517          functions[outfun].spec == TENSORFUNCTION ) tensorflag = 1;
07518     else tensorflag = 0;
07519 /*
07520     Now the type: type=....
07521 */
07522     while ( *s == ',' || *s == ' ' ) s++;
07523     stype = s;
07524     while ( FG.cTable[*s] == 0 ) s++;
07525     c = *s; *s = 0;
07526     if ( StrICmp(stype,(UBYTE *)"type") != 0 || c != '=' ) {
07527         MesPrint("&In CreateAllLoops statement: expected type=vartype.");
07528         return(1);
07529     }
07530     *s++ = c;
07531     stype = s;
```

```
07532     while ( FG.cTable[*s] == 0 ) s++;
07533     c = *s; *s = 0;
07534     if ( StrICmp(stype,(UBYTE *)"vector") == 0 ) {
07535         typenum = -VECTOR;
07536     }
07537     else if ( StrICmp(stype,(UBYTE *)"index") == 0 ) {
07538         typenum = -INDEX;
07539     }
07540     else if ( StrICmp(stype,(UBYTE *)"symbol") == 0 ) {
07541         if ( tensorflag ) goto notintensor;
07542         typenum = -SYMBOL;
07543     }
07544     else if ( StrICmp(stype,(UBYTE *)"snumber") == 0 ) {
07545         if ( tensorflag ) goto notintensor;
07546         typenum = -SNUMBER;
07547     }
07548     else {
07549         MesPrint("&Unknown/not allowed variable type in CreateAllLoops: %s",stype);
07550         return(1);
07551     }
07552     *s = c;
07553     while ( *s == ',' || *s == ' ' ) s++;
07554
07555     stype = s;
07556     while ( FG.cTable[*s] == 0 ) s++;
07557     c = *s; *s = 0;
07558     if ( StrICmp(stype,(UBYTE *)"ifnoloop") != 0 || c != '=' ) {
07559         MesPrint("&Unrecognised option in CreateAllLoops statement: %s",stype);
07560         return(1);
07561     }
07562     *s++ = c;
07563     x = -1;
07564     if ( FG.cTable[*s] == 1 ) {
07565         x = 0;
07566         do { x = 10*x + (*s++-'0'); } while (FG.cTable[*s] == 1);
07567     }
07568     if ( x != 0 && x != 1 ) {
07569         MesPrint("&Only options allowed for ifnoloop are 0 or 1.");
07570         return(1);
07571     }
07572     WorkSave = to = AT.WorkPointer;
07573     *to++ = TYPEALLLOOPS;
07574     *to++ = 6;
07575     *to++ = infun;
07576     *to++ = outfun;
07577     *to++ = typenum;
07578     *to++ = x;
07579
07580     AddNtoL(WorkSave[1],WorkSave);
07581
07582     return(0);
07583 notintensor:
07584     MesPrint("&Variable type not allowed in tensors: %s",stype);
07585     return(1);
07586 }
07587
07588 /*
07589     #] CoCreateAllLoops :
07590     #[ CoCreateAllPaths :
07591
07592     Syntax:
07593         CreateAllPaths,end-function,intermediate-function,out-function,type-argument,ifnopath;
07594     Types allowed:
07595         vector, index, symbol, snumber
07596     ifnoloop:
07597         ifnoloop=0  or ifnoloop=1
07598     in-function can be a tensor. In that case type can be only vector or index.
07599     out-function can be a tensor. In that case type can be only vector or index.
07600 */
07601
07602 int CoCreateAllPaths(UBYTE *s)
07603 {
07604     GETIDENTITY
07605     UBYTE *endname,*inname, *outname, *stype, c;
07606     WORD endfun, infun, outfun, x, type, tensorflag, typenum;
07607     WORD *WorkSave, *to;
07608     while ( *s == ',' || *s == ' ' ) s++;
07609     endname = s; s = SkipAName(s);
07610     c = *s; *s = 0;
07611     if ( ( ( type = GetName(AC.varnames,endname,&endfun,WITHAUTO) ) != CFUNCTION )
07612     || ( ( functions[endfun].spec != 0 ) && ( functions[endfun].spec != TENSORFUNCTION ) ) ) {
07613         MesPrint("&%s should be a regular function or a tensor.",endname);
07614         if ( type < 0 ) {
07615             if ( GetName(AC.exprnames,s,&endfun,NOAUTO) == NAMENOTFOUND )
07616                 AddFunction(s,0,0,0,0,0,-1,-1);
07617         }
07618         return(1);
```

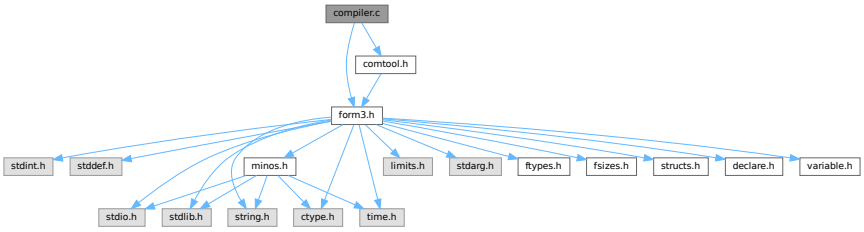
```
07619     }
07620     endfun += FUNCTION;
07621     *s++ = c;
07622     while ( *s == ',' || *s == ' ' ) s++;
07623     inname = s; s = SkipAName(s);
07624     c = *s; *s = 0;
07625     if ( ( ( type = GetName(AC.varnames,inname,&infun,WITHAUTO) ) != CFUNCTION )
07626     || ( ( functions[infun].spec != 0 ) && ( functions[infun].spec != TENSORFUNCTION ) ) ) {
07627         MesPrint("&%s should be a regular function or a tensor.",inname);
07628         if ( type < 0 ) {
07629             if ( GetName(AC.exprnames,s,&infun,NOAUTO) == NAMENOTFOUND )
07630                 AddFunction(s,0,0,0,0,0,-1,-1);
07631         }
07632         return(1);
07633     }
07634     infun += FUNCTION;
07635     *s++ = c;
07636     while ( *s == ',' || *s == ' ' ) s++;
07637     outname = s; s = SkipAName(s);
07638     c = *s; *s = 0;
07639     if ( ( ( type = GetName(AC.varnames,outname,&outfun,WITHAUTO) ) != CFUNCTION )
07640     || ( ( functions[outfun].spec != 0 ) && ( functions[outfun].spec != TENSORFUNCTION ) ) ) {
07641         MesPrint("&%s should be a regular function or a tensor.",outname);
07642         if ( type < 0 ) {
07643             if ( GetName(AC.exprnames,s,&outfun,NOAUTO) == NAMENOTFOUND )
07644                 AddFunction(s,0,0,0,0,0,-1,-1);
07645         }
07646         return(1);
07647     }
07648     outfun += FUNCTION;
07649     *s++ = c;
07650     if ( functions[infun].spec == TENSORFUNCTION ||
07651          functions[outfun].spec == TENSORFUNCTION ) tensorflag = 1;
07652     else tensorflag = 0;
07653 /*
07654     Now the type: type=....
07655 */
07656     while ( *s == ',' || *s == ' ' ) s++;
07657     stype = s;
07658     while ( FG.cTable[*s] == 0 ) s++;
07659     c = *s; *s = 0;
07660     if ( StrICmp(stype,(UBYTE *)"type") != 0 || c != '=' ) {
07661         MesPrint("&In CreateAllPaths statement: expected type=vartype.");
07662         return(1);
07663     }
07664     *s++ = c;
07665     stype = s;
07666     while ( FG.cTable[*s] == 0 ) s++;
07667     c = *s; *s = 0;
07668     if ( StrICmp(stype,(UBYTE *)"vector") == 0 ) {
07669         typenum = -VECTOR;
07670     }
07671     else if ( StrICmp(stype,(UBYTE *)"index") == 0 ) {
07672         typenum = -INDEX;
07673     }
07674     else if ( StrICmp(stype,(UBYTE *)"symbol") == 0 ) {
07675         if ( tensorflag ) goto notintensor;
07676         typenum = -SYMBOL;
07677     }
07678     else if ( StrICmp(stype,(UBYTE *)"snumber") == 0 ) {
07679         if ( tensorflag ) goto notintensor;
07680         typenum = -SNUMBER;
07681     }
07682     else {
07683         MesPrint("&Unknown/not allowed variable type in CreateAllPaths: %s",stype);
07684         return(1);
07685     }
07686     *s = c;
07687     while ( *s == ',' || *s == ' ' ) s++;
07688
07689     stype = s;
07690     while ( FG.cTable[*s] == 0 ) s++;
07691     c = *s; *s = 0;
07692     if ( StrICmp(stype,(UBYTE *)"ifnopath") != 0 || c != '=' ) {
07693         MesPrint("&Unrecognised option in CreateAllPaths statement: %s",stype);
07694         return(1);
07695     }
07696     *s++ = c;
07697     x = -1;
07698     if ( FG.cTable[*s] == 1 ) {
07699         x = 0;
07700         do { x = 10*x + (*s++-'0'); } while (FG.cTable[*s] == 1);
07701     }
07702     if ( x != 0 && x != 1 ) {
07703         MesPrint("&Only options allowed for ifnopath are 0 or 1.");
07704         return(1);
07705     }
```

```
07706     WorkSave = to = AT.WorkPointer;
07707     *to++ = TYPEALLPATHS;
07708     *to++ = 7;
07709     *to++ = endfun;
07710     *to++ = infun;
07711     *to++ = outfun;
07712     *to++ = typenum;
07713     *to++ = x;
07714
07715     AddNtoL(WorkSave[1],WorkSave);
07716
07717     return(0);
07718 notintensor:
07719     MesPrint("&CreateAllPaths: Variable type not allowed in tensors: %s",stype);
07720     return(1);
07721 }
07722
07723 /*
07724     #] CoCreateAllPaths :
07725     #[ CoCreateAll :
07726
07727     Syntax: subkey, two or three functions, type, ifnone
07728 */
07729
07730 int CoCreateAll(UBYTE *s)
07731 {
07732     UBYTE *subkey;
07733     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
07734     subkey = s;
07735     while ( FG.cTable[*s] == 0 ) s++;
07736     if ( *s != ' ' && *s != ',' && *s != '\t' ) {
07737         MesPrint("&Illegal subkey in CoCreate statement.");
07738         return(1);
07739     }
07740     /* c = *s; */ *s++ = 0;
07741     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
07742     if ( StrICmp(subkey,(UBYTE *)"loops") == 0 ) {
07743         return(CoCreateAllLoops(s));
07744     }
07745     else if ( StrICmp(subkey,(UBYTE *)"paths") == 0 ) {
07746         return(CoCreateAllPaths(s));
07747     }
07748 /*
07749     else if ( StrICmp(subkey,(UBYTE *)"motics") == 0 ) {
07750     }
07751     else if ( StrICmp(subkey,(UBYTE *)"onepi") == 0 ) {
07752     }
07753     else if ( StrICmp(subkey,(UBYTE *)"cuts") == 0 ) {
07754     }
07755 */
07756     else {
07757         MesPrint("&Illegal subkey in CoCreate statement: %s.",subkey);
07758         return(1);
07759     }
07760 }
07761
07762 /*
07763     #] CoCreateAll :
07764 */
```

4.11 compiler.c File Reference

```
#include "form3.h"
#include "comtool.h"
```

Include dependency graph for compiler.c:



Data Structures

- struct SuBbUf

Macros

- #define OPTION0 1
- #define OPTION1 2
- #define OPTION2 3
- #define REDUCESUBEXPBUFFERS

Typedefs

- typedef struct SuBbUf **SUBBUF**

Functions

- void inictable (void)
- KEYWORD * findcommand (UBYTE *in)
- int ParenthesesTest (UBYTE *sin)
- UBYTE * SkipAName (UBYTE *s)
- UBYTE * IsRHS (UBYTE *s, UBYTE c)
- int IsIdStatement (UBYTE *s)
- int CompileAlgebra (UBYTE *s, int leftright, WORD *prototype)
- int CompileStatement (UBYTE *in)
- int TestTables (void)
- int CompileSubExpressions (SBYTE *tokens)
- int CodeGenerator (SBYTE *tokens)
- int CompleteTerm (WORD *term, UWORD *numer, UWORD *denom, WORD nnum, WORD nden, int sign)
- int CodeFactors (SBYTE *tokens)
- WORD GenerateFactors (WORD n, WORD inc)

Variables

- int alfatable1 [27]
- SUBBUF * subexpbuffers = 0
- SUBBUF * topsubexpbuffers = 0
- LONG insubexpbuffers = 0

4.11.1 Detailed Description

The heart of the compiler. It contains the tables of statements. It finds the statements in the tables and calls the proper routines. For algebraic expressions it runs the compilation by first calling the tokenizer, splitting things into subexpressions and generating the code. There is a system for recognizing already existing subexpressions. This economizes on the length of the output.

Note: the compiler of FORM doesn't attempt to normalize the input. Hence x+1 and 1+x are different objects during compilation. Similarly (a+b-b) will not be simplified to (a).

Definition in file compiler.c.

4.11.2 Macro Definition Documentation

4.11.2.1 OPTION0

```
#define OPTION0 1
```

Definition at line 260 of file compiler.c.

4.11.2.2 OPTION1

```
#define OPTION1 2
```

Definition at line 261 of file compiler.c.

4.11.2.3 OPTION2

```
#define OPTION2 3
```

Definition at line 262 of file compiler.c.

4.11.2.4 REDUCESUBEXPBUFFERS

```
#define REDUCESUBEXPBUFFERS
```

Value:

```
    { if ( (topsubexpbuffers-subexpbuffers) > 256 ) {\
    M_free(subexpbuffers,"subexpbuffers");\
    subexpbuffers = (SUBBUF *)Malloc1(256*sizeof(SUBBUF),"subexpbuffers");\
    topsubexpbuffers = subexpbuffers+256; } insubexpbuffers = 0; }
```

Definition at line 273 of file compiler.c.

4.11.3 Function Documentation

4.11.3.1 inictable()

```
void inictable (
            void  )
```

Definition at line 315 of file compiler.c.

4.11.3.2 findcommand()

```
KEYWORD * findcommand (
            UBYTE * in )
```

Definition at line 341 of file compiler.c.

4.11.3.3 ParenthesesTest()

```
int ParenthesesTest (
            UBYTE * sin )
```

Definition at line 385 of file compiler.c.

4.11.3.4 SkipAName()

```
UBYTE * SkipAName (
            UBYTE * s )
```

Skips a name and gives a pointer to the character after the name. If there is not a proper name, it emits a compiler message and then returns a null pointer. This function supports formal names (e.g., `[x+a]`) and $-variables in addition to ordinary identifiers.

Note

The brackets are assumed to be matched already.

Parameters

in	*s*	Pointer to a null-terminated input string that starts with a name.

Returns

Pointer to the character after the name, or a null pointer if the name is invalid.

Definition at line 443 of file compiler.c.

4.11.3.5 IsRHS()

```
UBYTE * IsRHS (
            UBYTE * s,
            UBYTE c )
```

Definition at line 476 of file compiler.c.

4.11.3.6 IsIdStatement()

```
int IsIdStatement (
            UBYTE * s )
```

Definition at line 522 of file compiler.c.

4.11.3.7 CompileAlgebra()

```
int CompileAlgebra (
            UBYTE * s,
            int leftright,
            WORD * prototype )
```

Definition at line 536 of file compiler.c.

4.11.3.8 CompileStatement()

```
int CompileStatement (
            UBYTE * in )
```

Definition at line 572 of file compiler.c.

4.11.3.9 TestTables()

```
int TestTables (
            void  )
```

Definition at line 706 of file compiler.c.

4.11.3.10 CompileSubExpressions()

```
int CompileSubExpressions (
            SBYTE * tokens )
```

Definition at line 750 of file compiler.c.

4.11.3.11 CodeGenerator()

```
int CodeGenerator (
            SBYTE * tokens )
```

Definition at line 885 of file compiler.c.

4.11.3.12 CompleteTerm()

```
int CompleteTerm (
            WORD * term,
            UWORD * numer,
            UWORD * denom,
            WORD nnum,
            WORD nden,
            int sign )
```

Definition at line 1962 of file compiler.c.

4.11.3.13 CodeFactors()

```
int CodeFactors (
            SBYTE * tokens )
```

Definition at line 1999 of file compiler.c.

4.11.3.14 GenerateFactors()

```
WORD GenerateFactors (
            WORD n,
            WORD inc )
```

Definition at line 2300 of file compiler.c.

4.11.4 Variable Documentation

4.11.4.1 alfatable1

```
int alfatable1[27]
```

Definition at line 258 of file compiler.c.

4.11.4.2 subexpbuffers

```
SUBBUF* subexpbuffers = 0
```

Definition at line 269 of file compiler.c.

4.11.4.3 topsubexpbuffers

```
SUBBUF* topsubexpbuffers = 0
```

Definition at line 270 of file compiler.c.

4.11.4.4 insubexpbuffers

```
LONG insubexpbuffers = 0
```

Definition at line 271 of file compiler.c.

4.12 compiler.c

Go to the documentation of this file.

```
00001
00015 /* #[ License : */
00016 /*
00017  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00018  *   When using this file you are requested to refer to the publication
00019  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00020  *   This is considered a matter of courtesy as the development was paid
00021  *   for by FOM the Dutch physics granting agency and we would like to
00022  *   be able to track its scientific use to convince FOM of its value
00023  *   for the community.
00024  *
00025  *   This file is part of FORM.
00026  *
00027  *   FORM is free software: you can redistribute it and/or modify it under the
00028  *   terms of the GNU General Public License as published by the Free Software
00029  *   Foundation, either version 3 of the License, or (at your option) any later
00030  *   version.
00031  *
00032  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00033  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00034  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00035  *   details.
00036  *
00037  *   You should have received a copy of the GNU General Public License along
00038  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00039  */
00040 /* #] License : */
00041 /*
00042     #[ includes :
00043 */
00044
00045 #include "form3.h"
00046 #include "comtool.h"
00047
00048 /*
00049     com1commands are the commands of which only part of the word has to
00050     be present. The order is rather important here.
00051     com2commands are the commands that must have their whole word match.
00052     here we can do a binary search.
00053     {[(
00054 */
00055
00056 static KEYWORD com1commands[] = {
00057      {"also",            (TFUN)CoIdOld,             STATEMENT,    PARTEST}
00058     ,{"abrackets",       (TFUN)CoAntiBracket,       TOOUTPUT,     PARTEST}
00059     ,{"antisymmetrize",  (TFUN)CoAntiSymmetrize,    STATEMENT,    PARTEST}
00060     ,{"antibrackets",    (TFUN)CoAntiBracket,       TOOUTPUT,     PARTEST}
00061     ,{"brackets",        (TFUN)CoBracket,           TOOUTPUT,     PARTEST}
00062     ,{"cfunctions",      (TFUN)CoCFunction,         DECLARATION,  PARTEST|WITHAUTO}
00063     ,{"commuting",       (TFUN)CoCFunction,         DECLARATION,  PARTEST|WITHAUTO}
00064     ,{"compress",        (TFUN)CoCompress,          DECLARATION,  PARTEST}
00065     ,{"ctensors",        (TFUN)CoCTensor,           DECLARATION,  PARTEST|WITHAUTO}
00066     ,{"cyclesymmetrize",(TFUN)CoCycleSymmetrize,   STATEMENT,    PARTEST}
00067     ,{"dimension",       (TFUN)CoDimension,         DECLARATION,  PARTEST}
00068     ,{"discard",         (TFUN)CoDiscard,           STATEMENT,    PARTEST}
00069     ,{"functions",       (TFUN)CoNFunction,         DECLARATION,  PARTEST|WITHAUTO}
00070     ,{"format",          (TFUN)CoFormat,            TOOUTPUT,     PARTEST}
00071     ,{"fixindex",        (TFUN)CoFixIndex,          DECLARATION,  PARTEST}
00072     ,{"global",          (TFUN)CoGlobal,            DEFINITION,   PARTEST}
00073     ,{"gfactorized",     (TFUN)CoGlobalFactorized, DEFINITION,   PARTEST}
00074     ,{"globalfactorized",(TFUN)CoGlobalFactorized,DEFINITION,   PARTEST}
00075     ,{"goto",            (TFUN)CoGoTo,              STATEMENT,    PARTEST}
00076     ,{"indexes",         (TFUN)CoIndex,             DECLARATION,  PARTEST|WITHAUTO}
00077     ,{"indices",         (TFUN)CoIndex,             DECLARATION,  PARTEST|WITHAUTO}
00078     ,{"identify",        (TFUN)CoId,                STATEMENT,    PARTEST}
00079     ,{"idnew",           (TFUN)CoIdNew,             STATEMENT,    PARTEST}
00080     ,{"idold",           (TFUN)CoIdOld,             STATEMENT,    PARTEST}
00081     ,{"local",           (TFUN)CoLocal,             DEFINITION,   PARTEST}
00082     ,{"lfactorized",     (TFUN)CoLocalFactorized,   DEFINITION,   PARTEST}
00083     ,{"localfactorized",(TFUN)CoLocalFactorized,   DEFINITION,   PARTEST}
00084     ,{"load",            (TFUN)CoLoad,              DECLARATION,  PARTEST}
00085     ,{"label",           (TFUN)CoLabel,             STATEMENT,    PARTEST}
00086     ,{"modulus",         (TFUN)CoModulus,           DECLARATION,  PARTEST}
00087     ,{"multiply",        (TFUN)CoMultiply,          STATEMENT,    PARTEST}
00088     ,{"nfunctions",      (TFUN)CoNFunction,         DECLARATION,  PARTEST|WITHAUTO}
00089     ,{"nprint",          (TFUN)CoNPrint,            TOOUTPUT,     PARTEST}
00090     ,{"ntensors",        (TFUN)CoNTensor,           DECLARATION,  PARTEST|WITHAUTO}
00091     ,{"nwrite",          (TFUN)CoNWrite,            DECLARATION,  PARTEST}
00092     ,{"print",           (TFUN)CoPrint,             MIXED,        0}
00093     ,{"redefine",        (TFUN)CoRedefine,          STATEMENT,    0}
00094     ,{"rcyclesymmetrize",(TFUN)CoRCycleSymmetrize,STATEMENT,    PARTEST}
00095     ,{"symbols",         (TFUN)CoSymbol,            DECLARATION,  PARTEST|WITHAUTO}
```

```
00096     ,{"save",           (TFUN)CoSave,             DECLARATION,  PARTEST}
00097     ,{"symmetrize",     (TFUN)CoSymmetrize,       STATEMENT,    PARTEST}
00098     ,{"tensors",        (TFUN)CoCTensor,          DECLARATION,  PARTEST|WITHAUTO}
00099     ,{"unittrace",      (TFUN)CoUnitTrace,        DECLARATION,  PARTEST}
00100     ,{"vectors",        (TFUN)CoVector,           DECLARATION,  PARTEST|WITHAUTO}
00101     ,{"write",          (TFUN)CoWrite,            DECLARATION,  PARTEST}
00102 };
00103
00104 static KEYWORD com2commands[] = {
00105      {"antiputinside",  (TFUN)CoAntiPutInside,    STATEMENT,    PARTEST}
00106     ,{"apply",          (TFUN)CoApply,            STATEMENT,    PARTEST}
00107     ,{"aputinside",     (TFUN)CoAntiPutInside,    STATEMENT,    PARTEST}
00108     ,{"argexplode",     (TFUN)CoArgExplode,       STATEMENT,    PARTEST}
00109     ,{"argimplode",     (TFUN)CoArgImplode,       STATEMENT,    PARTEST}
00110     ,{"argtoextrasymbol",(TFUN)CoArgToExtraSymbol,STATEMENT,    PARTEST}
00111     ,{"argument",       (TFUN)CoArgument,         STATEMENT,    PARTEST}
00112     ,{"assign",         (TFUN)CoAssign,           STATEMENT,    PARTEST}
00113     ,{"auto",           (TFUN)CoAuto,             DECLARATION,  PARTEST}
00114     ,{"autodeclare",    (TFUN)CoAuto,             DECLARATION,  PARTEST}
00115     ,{"break",          (TFUN)CoBreak,            STATEMENT,    PARTEST}
00116     ,{"canonicalize",   (TFUN)CoCanonicalize,     STATEMENT,    PARTEST}
00117     ,{"case",           (TFUN)CoCase,             STATEMENT,    PARTEST}
00118     ,{"chainin",        (TFUN)CoChainin,          STATEMENT,    PARTEST}
00119     ,{"chainout",       (TFUN)CoChainout,         STATEMENT,    PARTEST}
00120     ,{"chisholm",       (TFUN)CoChisholm,         STATEMENT,    PARTEST}
00121 #ifdef WITHFLOAT
00122     ,{"chop",           (TFUN)CoChop,             STATEMENT,    PARTEST}
00123 #endif
00124     ,{"clearflag",      (TFUN)CoClearUserFlag,    STATEMENT,    PARTEST}
00125     ,{"cleartable",     (TFUN)CoClearTable,       DECLARATION,  PARTEST}
00126     ,{"collect",        (TFUN)CoCollect,          SPECIFICATION,PARTEST}
00127     ,{"commuteinset",   (TFUN)CoCommuteInSet,     DECLARATION,  PARTEST}
00128     ,{"contract",       (TFUN)CoContract,         STATEMENT,    PARTEST}
00129     ,{"copyspectator",  (TFUN)CoCopySpectator,    DEFINITION,   PARTEST}
00130     ,{"createall",      (TFUN)CoCreateAll,        STATEMENT,    PARTEST}
00131     ,{"createspectator",(TFUN)CoCreateSpectator,  DECLARATION,  PARTEST}
00132     ,{"ctable",         (TFUN)CoCTable,           DECLARATION,  PARTEST}
00133     ,{"deallocatetable",(TFUN)CoDeallocateTable,  DECLARATION,  PARTEST}
00134     ,{"default",        (TFUN)CoDefault,          STATEMENT,    PARTEST}
00135     ,{"delete",         (TFUN)CoDelete,           SPECIFICATION,PARTEST}
00136     ,{"denominators",   (TFUN)CoDenominators,     STATEMENT,    PARTEST}
00137     ,{"disorder",       (TFUN)CoDisorder,         STATEMENT,    PARTEST}
00138     ,{"do",             (TFUN)CoDo,               STATEMENT,    PARTEST}
00139     ,{"drop",           (TFUN)CoDrop,             SPECIFICATION,PARTEST}
00140     ,{"dropcoefficient",(TFUN)CoDropCoefficient,  STATEMENT,    PARTEST}
00141     ,{"dropsymbols",    (TFUN)CoDropSymbols,      STATEMENT,    PARTEST}
00142     ,{"else",           (TFUN)CoElse,             STATEMENT,    PARTEST}
00143     ,{"elseif",         (TFUN)CoElseIf,           STATEMENT,    PARTEST}
00144     ,{"emptyspectator", (TFUN)CoEmptySpectator,   SPECIFICATION,PARTEST}
00145     ,{"endargument",    (TFUN)CoEndArgument,      STATEMENT,    PARTEST}
00146     ,{"enddo",          (TFUN)CoEndDo,            STATEMENT,    PARTEST}
00147     ,{"endif",          (TFUN)CoEndIf,            STATEMENT,    PARTEST}
00148     ,{"endinexpression",(TFUN)CoEndInExpression,  STATEMENT,    PARTEST}
00149     ,{"endinside",      (TFUN)CoEndInside,        STATEMENT,    PARTEST}
00150     ,{"endmodel",       (TFUN)CoEndModel,         DECLARATION,  PARTEST}
00151     ,{"endrepeat",      (TFUN)CoEndRepeat,        STATEMENT,    PARTEST}
00152     ,{"endswitch",      (TFUN)CoEndSwitch,        STATEMENT,    PARTEST}
00153     ,{"endterm",        (TFUN)CoEndTerm,          STATEMENT,    PARTEST}
00154     ,{"endwhile",       (TFUN)CoEndWhile,         STATEMENT,    PARTEST}
00155 #ifdef WITHFLOAT
00156     ,{"evaluate",       (TFUN)CoEvaluate,         STATEMENT,    PARTEST}
00157 #endif
00158     ,{"exit",           (TFUN)CoExit,             STATEMENT,    PARTEST}
00159     ,{"extrasymbols",   (TFUN)CoExtraSymbols,     DECLARATION,  PARTEST}
00160     ,{"factarg",        (TFUN)CoFactArg,          STATEMENT,    PARTEST}
00161     ,{"factdollar",     (TFUN)CoFactDollar,       STATEMENT,    PARTEST}
00162     ,{"factorize",      (TFUN)CoFactorize,        TOOUTPUT,     PARTEST}
00163     ,{"fill",           (TFUN)CoFill,             DECLARATION,  PARTEST}
00164     ,{"fillexpression", (TFUN)CoFillExpression,   DECLARATION,  PARTEST}
00165     ,{"frompolynomial", (TFUN)CoFromPolynomial,   STATEMENT,    PARTEST}
00166     ,{"funpowers",      (TFUN)CoFunPowers,        DECLARATION,  PARTEST}
00167     ,{"hide",           (TFUN)CoHide,             SPECIFICATION,PARTEST}
00168     ,{"if",             (TFUN)CoIf,               STATEMENT,    PARTEST}
00169     ,{"ifmatch",        (TFUN)CoIfMatch,          STATEMENT,    PARTEST}
00170     ,{"ifnomatch",      (TFUN)CoIfNoMatch,        STATEMENT,    PARTEST}
00171     ,{"ifnotmatch",     (TFUN)CoIfNoMatch,        STATEMENT,    PARTEST}
00172     ,{"inexpression",   (TFUN)CoInExpression,     STATEMENT,    PARTEST}
00173     ,{"inparallel",     (TFUN)CoInParallel,       SPECIFICATION,PARTEST}
00174     ,{"inside",         (TFUN)CoInside,           STATEMENT,    PARTEST}
00175     ,{"insidefirst",    (TFUN)CoInsideFirst,      DECLARATION,  PARTEST}
00176     ,{"intohide",       (TFUN)CoIntoHide,         SPECIFICATION,PARTEST}
00177     ,{"keep",           (TFUN)CoKeep,             SPECIFICATION,PARTEST}
00178     ,{"makeinteger",    (TFUN)CoMakeInteger,      STATEMENT,    PARTEST}
00179     ,{"many",           (TFUN)CoMany,             STATEMENT,    PARTEST}
00180     ,{"merge",          (TFUN)CoMerge,            STATEMENT,    PARTEST}
00181     ,{"metric",         (TFUN)CoMetric,           DECLARATION,  PARTEST}
00182     ,{"model",          (TFUN)CoModel,            DECLARATION,  PARTEST}
```

```
00183     ,{"moduleoption",    (TFUN)CoModuleOption,     ATENDOFMODULE,PARTEST}
00184     ,{"multi",           (TFUN)CoMulti,            STATEMENT,    PARTEST}
00185     ,{"multibracket",    (TFUN)CoMultiBracket,     STATEMENT,    PARTEST}
00186     ,{"ndrop",           (TFUN)CoNoDrop,           SPECIFICATION,PARTEST}
00187     ,{"nfactorize",      (TFUN)CoNFactorize,       TOOUTPUT,     PARTEST}
00188     ,{"nhide",           (TFUN)CoNoHide,           SPECIFICATION,PARTEST}
00189     ,{"nintohide",       (TFUN)CoNoIntoHide,       SPECIFICATION,PARTEST}
00190     ,{"normalize",       (TFUN)CoNormalize,        STATEMENT,    PARTEST}
00191     ,{"notinparallel",   (TFUN)CoNotInParallel,    SPECIFICATION,PARTEST}
00192     ,{"nskip",           (TFUN)CoNoSkip,           SPECIFICATION,PARTEST}
00193     ,{"ntable",          (TFUN)CoNTable,           DECLARATION,  PARTEST}
00194     ,{"nunfactorize",    (TFUN)CoNUnFactorize,     TOOUTPUT,     PARTEST}
00195     ,{"nunhide",         (TFUN)CoNoUnHide,         SPECIFICATION,PARTEST}
00196     ,{"off",             (TFUN)CoOff,              DECLARATION,  PARTEST}
00197     ,{"on",              (TFUN)CoOn,               DECLARATION,  PARTEST}
00198     ,{"once",            (TFUN)CoOnce,             STATEMENT,    PARTEST}
00199     ,{"only",            (TFUN)CoOnly,             STATEMENT,    PARTEST}
00200     ,{"particle",        (TFUN)CoParticle,         DECLARATION,  PARTEST}
00201     ,{"polyfun",         (TFUN)CoPolyFun,          DECLARATION,  PARTEST}
00202     ,{"polyratfun",      (TFUN)CoPolyRatFun,       DECLARATION,  PARTEST}
00203     ,{"pophide",         (TFUN)CoPopHide,          SPECIFICATION,PARTEST}
00204     ,{"print[]",         (TFUN)CoPrintB,           TOOUTPUT,     PARTEST}
00205     ,{"printtable",      (TFUN)CoPrintTable,       MIXED,        PARTEST}
00206     ,{"processbucketsize",(TFUN)CoProcessBucket,   DECLARATION,  PARTEST}
00207     ,{"propercount",     (TFUN)CoProperCount,      DECLARATION,  PARTEST}
00208     ,{"pushhide",        (TFUN)CoPushHide,         SPECIFICATION,PARTEST}
00209     ,{"putinside",       (TFUN)CoPutInside,        STATEMENT,    PARTEST}
00210     ,{"ratio",           (TFUN)CoRatio,            STATEMENT,    PARTEST}
00211     ,{"removespectator",(TFUN)CoRemoveSpectator,   SPECIFICATION,PARTEST}
00212     ,{"renumber",        (TFUN)CoRenumber,         STATEMENT,    PARTEST}
00213     ,{"repeat",          (TFUN)CoRepeat,           STATEMENT,    PARTEST}
00214     ,{"replaceloop",     (TFUN)CoReplaceLoop,      STATEMENT,    PARTEST}
00215     ,{"select",          (TFUN)CoSelect,           STATEMENT,    PARTEST}
00216     ,{"set",             (TFUN)CoSet,              DECLARATION,  PARTEST}
00217     ,{"setexitflag",     (TFUN)CoSetExitFlag,      STATEMENT,    PARTEST}
00218     ,{"setflag",         (TFUN)CoSetUserFlag,      STATEMENT,    PARTEST}
00219     ,{"shuffle",         (TFUN)CoMerge,            STATEMENT,    PARTEST}
00220     ,{"skip",            (TFUN)CoSkip,             SPECIFICATION,PARTEST}
00221     ,{"sort",            (TFUN)CoSort,             STATEMENT,    PARTEST}
00222     ,{"splitarg",        (TFUN)CoSplitArg,         STATEMENT,    PARTEST}
00223     ,{"splitfirstarg",   (TFUN)CoSplitFirstArg,    STATEMENT,    PARTEST}
00224     ,{"splitlastarg",    (TFUN)CoSplitLastArg,     STATEMENT,    PARTEST}
00225 #ifdef WITHFLOAT
00226     ,{"strictrounding", (TFUN)CoStrictRounding,    STATEMENT,    PARTEST}
00227 #endif
00228     ,{"stuffle",         (TFUN)CoStuffle,          STATEMENT,    PARTEST}
00229     ,{"sum",             (TFUN)CoSum,              STATEMENT,    PARTEST}
00230     ,{"switch",          (TFUN)CoSwitch,           STATEMENT,    PARTEST}
00231     ,{"table",           (TFUN)CoTable,            DECLARATION,  PARTEST}
00232     ,{"tablebase",       (TFUN)CoTableBase,        DECLARATION,  PARTEST}
00233     ,{"tb",              (TFUN)CoTableBase,        DECLARATION,  PARTEST}
00234     ,{"term",            (TFUN)CoTerm,             STATEMENT,    PARTEST}
00235     ,{"testuse",         (TFUN)CoTestUse,          STATEMENT,    PARTEST}
00236     ,{"threadbucketsize",(TFUN)CoThreadBucket,     DECLARATION,  PARTEST}
00237 #ifdef WITHFLOAT
00238     ,{"tofloat",         (TFUN)CoToFloat,          STATEMENT,    PARTEST}
00239 #endif
00240     ,{"topolynomial",    (TFUN)CoToPolynomial,     STATEMENT,    PARTEST}
00241 #ifdef WITHFLOAT
00242     ,{"torat",           (TFUN)CoToRat,            STATEMENT,    PARTEST}
00243     ,{"torational",      (TFUN)CoToRat,            STATEMENT,    PARTEST}
00244 #endif
00245     ,{"tospectator",     (TFUN)CoToSpectator,      STATEMENT,    PARTEST}
00246     ,{"totensor",        (TFUN)CoToTensor,         STATEMENT,    PARTEST}
00247     ,{"tovector",        (TFUN)CoToVector,         STATEMENT,    PARTEST}
00248     ,{"trace4",          (TFUN)CoTrace4,           STATEMENT,    PARTEST}
00249     ,{"tracen",          (TFUN)CoTraceN,           STATEMENT,    PARTEST}
00250     ,{"transform",       (TFUN)CoTransform,        STATEMENT,    PARTEST}
00251     ,{"tryreplace",      (TFUN)CoTryReplace,       STATEMENT,    PARTEST}
00252     ,{"unfactorize",     (TFUN)CoUnFactorize,      TOOUTPUT,     PARTEST}
00253     ,{"unhide",          (TFUN)CoUnHide,           SPECIFICATION,PARTEST}
00254     ,{"vertex",          (TFUN)CoVertex,           DECLARATION,  PARTEST}
00255     ,{"while",           (TFUN)CoWhile,            STATEMENT,    PARTEST}
00256 };
00257
00258 int alfatable1[27];
00259
00260 #define OPTION0 1
00261 #define OPTION1 2
00262 #define OPTION2 3
00263
00264 typedef struct SuBbUf {
00265     WORD    subexpnum;
00266     WORD    buffernum;
00267 } SUBBUF;
00268
00269 SUBBUF *subexpbuffers = 0;
```

```
00270 SUBBUF *topsubexpbuffers = 0;
00271 LONG insubexpbuffers = 0;
00272
00273 #define REDUCESUBEXPBUFFERS { if ( (topsubexpbuffers-subexpbuffers) > 256 ) {\
00274     M_free(subexpbuffers,"subexpbuffers");\
00275     subexpbuffers = (SUBBUF *)Malloc1(256*sizeof(SUBBUF),"subexpbuffers");\
00276     topsubexpbuffers = subexpbuffers+256; } insubexpbuffers = 0; }
00277
00278 #if BITSINWORD == 32
00279     #define PUTNUMBER128(t,n) { if ( n >= 2097152 ) { \
00280                 *t++ = ((n/128)/128)/128; *t++ = ((n/128)/128)%128; *t++ = (n/128)%128; *t++ = n%128;
    } \
00281             else if ( n >= 16384 ) { \
00282                 *t++ = n/(128*128); *t++ = (n/128)%128; *t++ = n%128; } \
00283             else if ( n >= 128 ) { *t++ = n/128; *t++ = n%128; }      \
00284             else *t++ = n; }
00285     #define PUTNUMBER100(t,n) { if ( n >= 1000000 ) { \
00286                 *t++ = ((n/100)/100)/100; *t++ = ((n/100)/100)%100; *t++ = (n/100)%100; *t++ = n%100;
    } \
00287             else if ( n >= 10000 ) { \
00288                 *t++ = n/10000; *t++ = (n/100)%100; *t++ = n%100; } \
00289             else if ( n >= 100 ) { *t++ = n/100; *t++ = n%100; }   \
00290             else *t++ = n; }
00291 #elif BITSINWORD == 16
00292     #define PUTNUMBER128(t,n) { if ( n >= 16384 ) { \
00293                 *t++ = n/(128*128); *t++ = (n/128)%128; *t++ = n%128; } \
00294             else if ( n >= 128 ) { *t++ = n/128; *t++ = n%128; }      \
00295             else *t++ = n; }
00296     #define PUTNUMBER100(t,n) { if ( n >= 10000 ) { \
00297                 *t++ = n/10000; *t++ = (n/100)%100; *t++ = n%100; } \
00298             else if ( n >= 100 ) { *t++ = n/100; *t++ = n%100; }   \
00299             else *t++ = n; }
00300 #else
00301     #error Only 64-bit and 32-bit platforms are supported.
00302 #endif
00303
00304 /*
00305     )]}
00306     #] includes :
00307     #[ Compiler :
00308         #[ inictable :
00309
00310         Routine sets the table for 1-st characters that allow a faster
00311         start in the search in table 1 which should be sequential.
00312         Search in table 2 can be binary.
00313 */
00314
00315 void inictable(void)
00316 {
00317     KEYWORD *k = com1commands;
00318     int i, j, ksize;
00319     ksize = sizeof(com1commands)/sizeof(KEYWORD);
00320     j = 0;
00321     alfatable1[0] = 0;
00322     for ( i = 0; i < 26; i++ ) {
00323         while ( j < ksize && k[j].name[0] == 'a'+i ) j++;
00324         alfatable1[i+1] = j;
00325     }
00326 }
00327
00328 /*
00329         #] inictable :
00330         #[ findcommand :
00331
00332         Checks whether a command is in the command table.
00333         If so a pointer to the table element is returned.
00334         If not we return 0.
00335         Note that when a command is not in the table, we have
00336         to test whether it is an id command without id. It should
00337         then have the structure pattern = rhs. This should be done
00338         in the calling routine.
00339 */
00340
00341 KEYWORD *findcommand(UBYTE *in)
00342 {
00343     int hi, med, lo, i;
00344     UBYTE *s, c;
00345     s = in;
00346     while ( FG.cTable[*s] <= 1 ) s++;
00347     if ( s > in && *s == '[' && s[1] == ']' ) s += 2;
00348     if ( *s ) { c = *s; *s = 0; }
00349     else c = 0;
00350 /*
00351     First do a binary search in the second table
00352 */
00353     lo = 0;
00354     hi = sizeof(com2commands)/sizeof(KEYWORD)-1;
```

```
00355     do {
00356         med = ( hi + lo ) / 2;
00357         i = StrICmp(in,(UBYTE *)com2commands[med].name);
00358         if ( i == 0 ) { if ( c ) *s = c; return(com2commands+med); }
00359         if ( i < 0 ) hi = med-1;
00360         else         lo = med+1;
00361     } while ( hi >= lo );
00362 /*
00363     Now do a 'hashed' search in the first table. It is sequential.
00364 */
00365     i = tolower(*in) - 'a';
00366     med = alfatable1[i];
00367     hi  = alfatable1[i+1];
00368     while ( med < hi ) {
00369         if ( StrICont(in,(UBYTE *)com1commands[med].name) == 0 )
00370             { if ( c ) *s = c; return(com1commands+med); }
00371         med++;
00372     }
00373     if ( c ) *s = c;
00374 /*
00375     Unrecognized. Too bad!
00376 */
00377     return(0);
00378 }
00379
00380 /*
00381         #] findcommand :
00382         #[ ParenthesesTest :
00383 */
00384
00385 int ParenthesesTest(UBYTE *sin)
00386 {
00387     WORD L1 = 0, L2 = 0, L3 = 0;
00388     UBYTE *s = sin;
00389     while ( *s ) {
00390         if ( *s == '[' ) L1++;
00391         else if ( *s == ']' ) {
00392             L1--;
00393             if ( L1 < 0 ) { MesPrint("&Unmatched []"); return(1); }
00394         }
00395         s++;
00396     }
00397     if ( L1 > 0 ) { MesPrint("&Unmatched []"); return(1); }
00398     s = sin;
00399     while ( *s ) {
00400         if ( *s == '[' ) SKIPBRA1(s)
00401         else if ( *s == '(' ) { L2++; s++; }
00402         else if ( *s == ')' ) {
00403             L2--; s++;
00404             if ( L2 < 0 ) { MesPrint("&Unmatched ()"); return(1); }
00405         }
00406         else s++;
00407     }
00408     if ( L2 > 0 ) { MesPrint("&Unmatched ()"); return(1); }
00409     s = sin;
00410     while ( *s ) {
00411         if ( *s == '[' ) SKIPBRA1(s)
00412         else if ( *s == '[' ) SKIPBRA4(s)
00413         else if ( *s == '{' ) { L3++; s++; }
00414         else if ( *s == '}' ) {
00415             L3--; s++;
00416             if ( L3 < 0 ) { MesPrint("&Unmatched {}"); return(1); }
00417         }
00418         else s++;
00419     }
00420     if ( L3 > 0 ) { MesPrint("&Unmatched {}"); return(1); }
00421     return(0);
00422 }
00423
00424 /*
00425         #] ParenthesesTest :
00426         #[ SkipAName :
00427 */
00428
00443 UBYTE *SkipAName(UBYTE *s)
00444 {
00445     UBYTE *t = s;
00446     if ( *s == '[' ) {
00447         SKIPBRA1(s)
00448 /*
00449         In principle the brackets match already, so the `if ( *s == 0 )'
00450         code is not really needed, but you never know how the program
00451         is extended later.
00452 */
00453         if ( *s == 0 ) {
00454             MesPrint("&Illegal name: '%s'",t);
00455             return(0);
```

```
00456         }
00457         s++;
00458     }
00459     else if ( FG.cTable[*s] == 0 || *s == '_' || *s == '$' ) {
00460         if ( *s == '$' ) s++;
00461         while ( FG.cTable[*s] <= 1 ) s++;
00462         if ( *s == '_' ) s++;
00463     }
00464     else {
00465         MesPrint("&Illegal name: '%s'",t);
00466         return(0);
00467     }
00468     return(s);
00469 }
00470
00471 /*
00472         #] SkipAName :
00473         #[ IsRHS :
00474 */
00475
00476 UBYTE *IsRHS(UBYTE *s, UBYTE c)
00477 {
00478     while ( *s && *s != c ) {
00479         if ( *s == '[' ) {
00480             SKIPBRA1(s);
00481             if ( *s != ']' ) {
00482                 MesPrint("&Unmatched []");
00483                 return(0);
00484             }
00485         }
00486         else if ( *s == '{' ) {
00487             SKIPBRA2(s);
00488             if ( *s != '}' ) {
00489                 MesPrint("&Unmatched {}");
00490                 return(0);
00491             }
00492         }
00493         else if ( *s == '(' ) {
00494             SKIPBRA3(s);
00495             if ( *s != ')' ) {
00496                 MesPrint("&Unmatched ()");
00497                 return(0);
00498             }
00499         }
00500         else if ( *s == ')' ) {
00501             MesPrint("&Unmatched ()");
00502             return(0);
00503         }
00504         else if ( *s == '}' ) {
00505             MesPrint("&Unmatched {}");
00506             return(0);
00507         }
00508         else if ( *s == ']' ) {
00509             MesPrint("&Unmatched []");
00510             return(0);
00511         }
00512         s++;
00513     }
00514     return(s);
00515 }
00516
00517 /*
00518         #] IsRHS :
00519         #[ IsIdStatement :
00520 */
00521
00522 int IsIdStatement(UBYTE *s)
00523 {
00524     DUMMYUSE(s);
00525     return(0);
00526 }
00527
00528 /*
00529         #] IsIdStatement :
00530         #[ CompileAlgebra :
00531
00532         Returns either the number of the main level RHS (>= 0)
00533         or an error code (< 0)
00534 */
00535
00536 int CompileAlgebra(UBYTE *s, int leftright, WORD *prototype)
00537 {
00538     GETIDENTITY
00539     int error;
00540     WORD *oldproto = AC.ProtoType;
00541     AC.ProtoType = prototype;
00542     if ( AC.TokensWriteFlag ) {
```

```
00543         MesPrint("To tokenize: %s",s);
00544         error = tokenize(s,leftright);
00545         MesPrint("  The contents of the token buffer are:");
00546         WriteTokens(AC.tokens);
00547     }
00548     else error = tokenize(s,leftright);
00549     if ( error == 0 ) {
00550         AR.Eside = leftright;
00551         AC.CompileLevel = 0;
00552         if ( leftright == LHSIDE ) { AC.DumNum = AR.CurDum = 0; }
00553         error = CompileSubExpressions(AC.tokens);
00554         REDUCESUBEXPBUFFERS
00555     }
00556     else {
00557         AC.ProtoType = oldproto;
00558         return(-1);
00559     }
00560     AC.ProtoType = oldproto;
00561     if ( error < 0 ) return(-1);
00562     else if ( leftright == LHSIDE ) return(cbuf[AC.cbufnum].numlhs);
00563     else                            return(cbuf[AC.cbufnum].numrhs);
00564 }
00565
00566 /*
00567         #] CompileAlgebra :
00568         #[ CompileStatement :
00569
00570 */
00571
00572 int CompileStatement(UBYTE *in)
00573 {
00574     KEYWORD *k;
00575     UBYTE *s;
00576     int error1 = 0, error2;
00577     /* A.iStatement = */ s = in;
00578     if ( *s == 0 ) return(0);
00579     if ( *s == '$' ) {
00580         k = findcommand((UBYTE *)"assign");
00581     }
00582     else {
00583         if ( ( k = findcommand(s) ) == 0 && IsIdStatement(s) == 0 ) {
00584             MesPrint("&Unrecognized statement %s",s);
00585             return(1);
00586         }
00587         if ( k == 0 ) { /* Id statement without id. Note: id must be in table */
00588             k = com1commands + alfatable1['i'-'a'];
00589             while ( k->name[1] != 'd' || k->name[2] ) k++;
00590         }
00591         else {
00592             while ( FG.cTable[*s] <= 1 ) s++;
00593             if ( s > in && *s == '[' && s[1] == ']' ) s += 2;
00594 /*
00595             The next statement is rather mysterious
00596             It is undone in DoPrint and CoMultiply, but it also causes effects
00597             in other (wrong) statements like dimension -4; or Trace4 -1;
00598             The code in pre.c (LoadStatement) has been changed 8-sep-2009
00599             to force a comma after the keyword. This means that the
00600             'mysterious' line is automatically inactive. Hence it is taken out.
00601
00602             if ( *s == '+' || *s == '-' ) s++;
00603 */
00604             if ( *s == ',' ) s++;
00605         }
00606     }
00607 /*
00608     First the test on the order of the statements.
00609     This is relatively new (2.2c) and may cause some problems with old
00610     programs. Hence the first error message should explain!
00611 */
00612     if ( AP.PreAssignFlag == 0 && AM.OldOrderFlag == 0 ) {
00613      if ( AP.PreInsideLevel ) {
00614       if ( k->type != STATEMENT && k->type != MIXED ) {
00615         MesPrint("&Only executable and print statements are allowed in an %#inside/%#endinside
    construction");
00616         return(-1);
00617       }
00618      }
00619      else {
00620       if ( ( AC.compiletype == DECLARATION || AC.compiletype == SPECIFICATION )
00621       && ( k->type == STATEMENT || k->type == DEFINITION || k->type == TOOUTPUT ) ) {
00622         if ( AC.tablecheck == 0 ) {
00623             AC.tablecheck = 1;
00624             if ( TestTables() ) error1 = 1;
00625         }
00626       }
00627       if ( k->type == MIXED ) {
00628         if ( AC.compiletype <= DEFINITION ) {
```

```
00629             AC.compiletype = STATEMENT;
00630           }
00631         }
00632         else if ( k->type > AC.compiletype ) {
00633           /*
00634            * We intentionally do NOT update "compiletype" for:
00635            * - Format statements (type = TOOUTPUT)
00636            * - ModuleOption statements (type = ATENDOFMODULE)
00637            *   with sum/maximum/minimum/local (i.e., $-variable-related options)
00638            *
00639            * This relaxes the ordering constraint, allowing statements with
00640            * type >= the current "compiletype" to follow.
00641            */
00642           if ( StrCmp((UBYTE *)(k->name),(UBYTE *)"format") == 0 )
00643               goto NoUpdateCompileType;
00644           if ( StrCmp((UBYTE *)(k->name),(UBYTE *)"moduleoption") == 0 ) {
00645               UBYTE *ss = s;
00646               SkipSpaces(&ss);
00647               if ( ConsumeOption(&ss,"sum")
00648               || ConsumeOption(&ss,"maximum")
00649               || ConsumeOption(&ss,"minimum")
00650               || ConsumeOption(&ss,"local") ) goto NoUpdateCompileType;
00651           }
00652           AC.compiletype = k->type;
00653 NoUpdateCompileType:
00654           ;
00655         }
00656         else if ( k->type < AC.compiletype ) {
00657           switch ( k->type ) {
00658               case DECLARATION:
00659                   MesPrint("&Declaration out of order");
00660                   MesPrint("&  %s",in);
00661                   break;
00662               case DEFINITION:
00663                   MesPrint("&Definition out of order");
00664                   MesPrint("&  %s",in);
00665                   break;
00666               case SPECIFICATION:
00667                   MesPrint("&Specification out of order");
00668                   MesPrint("&  %s",in);
00669                   break;
00670               case STATEMENT:
00671                   MesPrint("&Statement out of order");
00672                   break;
00673               case TOOUTPUT:
00674                   MesPrint("&Output control statement out of order");
00675                   MesPrint("&  %s",in);
00676                   break;
00677           }
00678           AC.compiletype = k->type;
00679           if ( AC.firstctypemessage == 0 ) {
00680               MesPrint("&Proper order inside a module is:");
00681               MesPrint("Declarations, specifications, definitions, statements, output control
      statements");
00682               AC.firstctypemessage = 1;
00683           }
00684           error1 = 1;
00685         }
00686       }
00687     }
00688 /*
00689     Now we execute the tests that are prescribed by the flags.
00690 */
00691     if ( AC.AutoDeclareFlag && ( ( k->flags & WITHAUTO ) == 0 ) ) {
00692         MesPrint("&Illegal type of auto-declaration");
00693         return(1);
00694     }
00695     if ( ( ( k->flags & PARTEST ) != 0 ) && ParenthesesTest(s) ) return(1);
00696     error2 = (*k->func)(s);
00697     if ( error2 == 0 ) return(error1);
00698     return(error2);
00699 }
00700
00701 /*
00702         #] CompileStatement :
00703         #[ TestTables :
00704 */
00705
00706 int TestTables(void)
00707 {
00708     FUNCTIONS f = functions;
00709     TABLES t;
00710     WORD j;
00711     int error = 0, i;
00712     LONG x;
00713     i = NumFunctions + FUNCTION - MAXBUILTINFUNCTION - 1;
00714     f = f + MAXBUILTINFUNCTION - FUNCTION + 1;
```

```
00715     if ( AC.MustTestTable > 0 ) {
00716       while ( i > 0 ) {
00717         if ( ( t = f->tabl ) != 0 && t->strict > 0 && !t->sparse ) {
00718             for ( x = 0, j = 0; x < t->totind; x++ ) {
00719                 if ( t->tablepointers[TABLEEXTENSION*x] < 0 ) j++;
00720             }
00721             if ( j > 0 ) {
00722                 if ( j > 1 ) {
00723                     MesPrint("&In table %s there are %d unfilled elements",
00724                     AC.varnames->namebuffer+f->name,j);
00725                 }
00726                 else {
00727                     MesPrint("&In table %s there is one unfilled element",
00728                     AC.varnames->namebuffer+f->name);
00729                 }
00730                 error = 1;
00731             }
00732         }
00733         i--; f++;
00734       }
00735       AC.MustTestTable--;
00736     }
00737     return(error);
00738 }
00739
00740 /*
00741         #] TestTables :
00742         #[ CompileSubExpressions :
00743
00744         Now we attack the subexpressions from inside out.
00745         We try to see whether we had any of them already.
00746         We have to worry about adding the wildcard sum parameter
00747         to the prototype.
00748 */
00749
00750 int CompileSubExpressions(SBYTE *tokens)
00751 {
00752     GETIDENTITY
00753     SBYTE *fill = tokens, *s = tokens, *t;
00754     WORD number[MAXNUMSIZE], *oldwork, *w1, *w2;
00755     int level, num, i, sumlevel = 0, sumtype = SYMTOSYM;
00756     int retval, error = 0;
00757 /*
00758     Eliminate all subexpressions. They are marked by LPARENTHESIS,RPARENTHESIS
00759 */
00760     AC.CompileLevel++;
00761     while ( *s != TENDOFIT ) {
00762         if ( *s == TFUNOPEN ) {
00763             if ( fill < s ) *fill = TENDOFIT;
00764             t = fill - 1;
00765             while ( t >= tokens && t[0] >= 0 ) t--;
00766             if ( t >= tokens && *t == TFUNCTION ) {
00767                 t++; i = 0; while ( *t >= 0 ) i = 128*i + *t++;
00768                 if ( i == AM.sumnum || i == AM.sumpnum ) {
00769                     t = s + 1;
00770                     if ( *t == TSYMBOL || *t == TINDEX ) {
00771                         t++; i = 0; while ( *t >= 0 ) i = 128*i + *t++;
00772                         if ( s[1] == TINDEX ) {
00773                             i += AM.OffsetIndex;
00774                             sumtype = INDTOIND;
00775                         }
00776                         else sumtype = SYMTOSYM;
00777                         sumlevel = i;
00778                     }
00779                 }
00780             }
00781             *fill++ = *s++;
00782         }
00783         else if ( *s == TFUNCLOSE ) { sumlevel = 0; *fill++ = *s++; }
00784         else if ( *s == LPARENTHESIS ) {
00785 /*
00786             We must make an exception here.
00787             If the subexpression is just an integer, whatever its length,
00788             we should try to keep it.
00789             This is important when we have a function with an integer
00790             argument. In particular this is relevant for the MZV program.
00791 */
00792             t = s; level = 0;
00793             while ( level >= 0 ) {
00794                 s++;
00795                 if ( *s == LPARENTHESIS ) level++;
00796                 else if ( *s == RPARENTHESIS ) level--;
00797                 else if ( *s == TENDOFIT ) {
00798                     MesPrint("&Unbalanced subexpression parentheses");
00799                     return(-1);
00800                 }
00801             }
```

```
00802             t++; *s = TENDOFIT;
00803             if ( sumlevel > 0 ) { /* Inside sum. Add wildcard to prototype */
00804                 oldwork = w1 = AT.WorkPointer;
00805                 w2 = AC.ProtoType;
00806                 i = w2[1];
00807                 while ( --i >= 0 ) *w1++ = *w2++;
00808                 oldwork[1] += 4;
00809                 *w1++ = sumtype; *w1++ = 4; *w1++ = sumlevel; *w1++ = sumlevel;
00810                 w2 = AC.ProtoType; AT.WorkPointer = w1;
00811                 AC.ProtoType = oldwork;
00812                 num = CompileSubExpressions(t);
00813                 AC.ProtoType = w2; AT.WorkPointer = oldwork;
00814             }
00815             else num = CompileSubExpressions(t);
00816             if ( num < 0 ) return(-1);
00817 /*
00818             Note that the subexpression code should always fit.
00819             We had two parentheses and at least two bytes contents.
00820             There cannot be more than 2^21 subexpressions or we get outside
00821             this minimum. Ignoring this might lead to really rare and
00822             hard to find errors, years from now.
00823 */
00824             if ( insubexpbuffers >= MAXSUBEXPRESSIONS ) {
00825                 MesPrint("&More than %d subexpressions inside one
      expression",(WORD)MAXSUBEXPRESSIONS);
00826                 Terminate(-1);
00827             }
00828             if ( subexpbuffers+insubexpbuffers >= topsubexpbuffers ) {
00829                 DoubleBuffer((void **)((void *)(&subexpbuffers))
00830                 ,(void **)((void *)(&topsubexpbuffers)),sizeof(SUBBUF),"subexpbuffers");
00831             }
00832             subexpbuffers[insubexpbuffers].subexpnum = num;
00833             subexpbuffers[insubexpbuffers].buffernum = AC.cbufnum;
00834             num = insubexpbuffers++;
00835             *fill++ = TSUBEXP;
00836             i = 0;
00837             do { number[i++] = num & 0x7F; num >>= 7; } while ( num );
00838             while ( --i >= 0 ) *fill++ = (SBYTE)(number[i]);
00839             s++;
00840         }
00841         else if ( *s == TEMPTY ) s++;
00842         else *fill++ = *s++;
00843     }
00844     *fill = TENDOFIT;
00845 /*
00846     At this stage there are no more subexpressions.
00847     Hence we can do the basic compilation.
00848 */
00849     if ( AC.CompileLevel == 1 && AC.ToBeInFactors ) {
00850         error = CodeFactors(tokens);
00851     }
00852     AC.CompileLevel--;
00853     retval = CodeGenerator(tokens);
00854     if ( error < 0 ) return(error);
00855     return(retval);
00856 }
00857
00858 /*
00859         #] CompileSubExpressions :
00860         #[ CodeGenerator :
00861
00862         This routine does the real code generation.
00863         It returns the number of the rhs subexpression.
00864         At this point we do not have to worry about subexpressions,
00865         sets, setelements, simple vs complicated function arguments
00866         simple vs complicated powers etc.
00867
00868         The variable 'first' indicates whether we are starting a new term
00869
00870         The major complication are the set elements of type set[n].
00871         We have marked them as TSETNUM,n,Ttype,setnum
00872         They go into
00873         SETSET,size,subterm,relocation list
00874         in which the subterm should be ready to become a regular
00875         subterm in which the sets have been replaced by their element
00876         The relocation list consists of pairs of numbers:
00877         1: offset in the subterm, 2: the symbol n.
00878         Note that such a subterm can be a whole function with its arguments.
00879         We use the variable inset to indicate that we have something going.
00880         The relocation list is collected in the top of the WorkSpace.
00881 */
00882
00883 static UWORD *CGscrat7 = 0;
00884
00885 int CodeGenerator(SBYTE *tokens)
00886 {
00887     GETIDENTITY
```

```
00888     SBYTE *s = tokens, c;
00889     int i, sign = 1, first = 1, deno = 1, error = 0, minus, n, needarg, numexp, cc;
00890     int base, sumlevel = 0, sumtype = SYMTOSYM, firstsumarg, inset = 0;
00891     int funflag = 0, settype, x1, x2, mulflag = 0;
00892     WORD *t, *v, *r, *term, nnumerator, ndenominator, *oldwork, x3, y, nin;
00893     WORD *w1, *w2, *tsize = 0, *relo = 0;
00894     UWORD *numerator, *denominator, *innum;
00895     CBUF *C;
00896     POSITION position;
00897     WORD TMproto[SUBEXPSIZE];
00898 /*
00899 #ifdef WITHPTHREADS
00900     RENUMBER renumber;
00901 #endif
00902 */
00903     RENUMBER renumber;
00904     if ( AC.TokensWriteFlag ) WriteTokens(tokens);
00905     if ( CGscrat7 == 0 )
00906         CGscrat7 = (UWORD *)Malloc1((AM.MaxTal+2)*sizeof(WORD),"CodeGenerator");
00907     AddRHS(AC.cbufnum,0);
00908     C = cbuf + AC.cbufnum;
00909     numexp = C->numrhs;
00910     C->NumTerms[numexp] = 0;
00911     C->numdum[numexp] = 0;
00912     oldwork = AT.WorkPointer;
00913     numerator = (UWORD *)(AT.WorkPointer);
00914     denominator = numerator + 2*AM.MaxTal;
00915     innum = denominator + 2*AM.MaxTal;
00916     term = (WORD *)(innum + 2*AM.MaxTal);
00917     AT.WorkPointer = term + AM.MaxTer/sizeof(WORD);
00918     if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
00919     cc = 0;
00920     t = term+1;
00921     numerator[0] = denominator[0] = 1;
00922     nnumerator = ndenominator = 1;
00923     while ( *s != TENDOFIT ) {
00924         if ( *s == TPLUS || *s == TMINUS ) {
00925             if ( first || mulflag ) { if ( *s == TMINUS ) sign = -sign; }
00926             else {
00927                 *term = t-term;
00928                 C->NumTerms[numexp]++;
00929                 if ( cc && sign ) C->CanCommu[numexp]++;
00930                 CompleteTerm(term,numerator,denominator,nnumerator,ndenominator,sign);
00931                 first = 1; cc = 0; t = term + 1; deno = 1;
00932                 numerator[0] = denominator[0] = 1;
00933                 nnumerator = ndenominator = 1;
00934                 if ( *s == TMINUS ) sign = -1;
00935                 else sign = 1;
00936             }
00937             s++;
00938         }
00939         else {
00940             mulflag = first = 0; c = *s++;
00941             switch ( c ) {
00942             case TSYMBOL:
00943                 x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
00944                 if ( *s == TWILDCARD ) { s++; x1 += 2*MAXPOWER; }
00945                 *t++ = SYMBOL; *t++ = 4; *t++ = x1;
00946                 if ( inset ) *relo = 2;
00947 TryPower:       if ( *s == TPOWER ) {
00948                     s++;
00949                     if ( *s == TMINUS ) { s++; deno = -deno; }
00950                     c = *s++;
00951                     base = ( c == TNUMBER ) ? 100: 128;
00952                     x2 = 0; while ( *s >= 0 ) { x2 = base*x2 + *s++; }
00953                     if ( c == TSYMBOL ) {
00954                         if ( *s == TWILDCARD ) s++;
00955                         x2 += 2*MAXPOWER;
00956                     }
00957                     *t++ = deno*x2;
00958                 }
00959                 else *t++ = deno;
00960 fin:            deno = 1;
00961                 if ( inset ) {
00962                     while ( relo < AT.WorkTop ) *t++ = *relo++;
00963                     inset = 0; tsize[1] = t - tsize;
00964                 }
00965                 break;
00966             case TINDEX:
00967                 x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
00968                 *t++ = INDEX; *t++ = 3;
00969                 if ( *s == TWILDCARD ) { s++; x1 += WILDOFFSET; }
00970                 if ( inset ) { *t++ = x1; *relo = 2; }
00971                 else         *t++ = x1 + AM.OffsetIndex;
00972                 if ( t[-1] > AM.IndDum ) {
00973                     x1 = t[-1] - AM.IndDum;
00974                     if ( x1 > C->numdum[numexp] ) C->numdum[numexp] = x1;
```

```
00975                 }
00976                 goto fin;
00977             case TGENINDEX:
00978                 *t++ = INDEX; *t++ = 3; *t++ = AC.DumNum+WILDOFFSET;
00979                 deno = 1;
00980                 break;
00981             case TVECTOR:
00982                 x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
00983 dovector:       if ( inset == 0 ) x1 += AM.OffsetVector;
00984                 if ( *s == TWILDCARD ) { s++; x1 += WILDOFFSET; }
00985                 if ( inset ) *relo = 2;
00986                 if ( *s == TDOT ) {     /* DotProduct ? */
00987                     s++;
00988                     if ( *s == TSETNUM || *s == TSETDOL ) {
00989                         settype = ( *s == TSETDOL );
00990                         s++; x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
00991                         if ( settype ) x2 = -x2;
00992                         if ( inset == 0 ) {
00993                             tsize = t; *t++ = SETSET; *t++ = 0;
00994                             relo = AT.WorkTop;
00995                         }
00996                         inset += 2;
00997                         *--relo = x2; *--relo = 3;
00998                     }
00999                     if ( *s != TVECTOR && *s != TDUBIOUS ) {
01000                         MesPrint("&Illegally formed dotproduct");
01001                         error = 1;
01002                     }
01003                     s++; x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
01004                     if ( inset < 2 ) x2 += AM.OffsetVector;
01005                     if ( *s == TWILDCARD ) { s++; x2 += WILDOFFSET; }
01006                     *t++ = DOTPRODUCT; *t++ = 5; *t++ = x1; *t++ = x2;
01007                     goto TryPower;
01008                 }
01009                 else if ( *s == TFUNOPEN ) {
01010                     s++;
01011                     if ( *s == TSETNUM || *s == TSETDOL ) {
01012                         settype = ( *s == TSETDOL );
01013                         s++; x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
01014                         if ( settype ) x2 = -x2;
01015                         if ( inset == 0 ) {
01016                             tsize = t; *t++ = SETSET; *t++ = 0;
01017                             relo = AT.WorkTop;
01018                         }
01019                         inset += 2;
01020                         *--relo = x2; *--relo = 3;
01021                     }
01022                     if ( *s == TINDEX || *s == TDUBIOUS ) {
01023                         s++;
01024                         x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
01025                         if ( inset < 2 ) x2 += AM.OffsetIndex;
01026                         if ( *s == TWILDCARD ) { s++; x2 += WILDOFFSET; }
01027                         *t++ = VECTOR; *t++ = 4; *t++ = x1; *t++ = x2;
01028                         if ( t[-1] > AM.IndDum ) {
01029                             x2 = t[-1] - AM.IndDum;
01030                             if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
01031                         }
01032                     }
01033                     else if ( *s == TGENINDEX ) {
01034                         *t++ = VECTOR; *t++ = 4; *t++ = x1;
01035                         *t++ = AC.DumNum + WILDOFFSET;
01036                     }
01037                     else if ( *s == TNUMBER || *s == TNUMBER1 ) {
01038                         base = ( *s == TNUMBER ) ? 100: 128;
01039                         s++;
01040                         x2 = 0; while ( *s >= 0 ) { x2 = x2*base + *s++; }
01041                         if ( x2 >= AM.OffsetIndex && inset < 2 ) {
01042                             MesPrint("&Fixed index in vector greater than %d",
01043                             AM.OffsetIndex);
01044                             return(-1);
01045                         }
01046                         *t++ = VECTOR; *t++ = 4; *t++ = x1; *t++ = x2;
01047                     }
01048                     else if ( *s == TVECTOR || ( *s == TMINUS && s[1] == TVECTOR ) ) {
01049                         if ( *s == TMINUS ) { s++; sign = -sign; }
01050                         s++;
01051                         x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
01052                         if ( inset < 2 ) x2 += AM.OffsetVector;
01053                         if ( *s == TWILDCARD ) { s++; x2 += WILDOFFSET; }
01054                         *t++ = DOTPRODUCT; *t++ = 5; *t++ = x1; *t++ = x2; *t++ = deno;
01055                     }
01056                     else {
01057                         MesPrint("&Illegal argument for vector");
01058                         return(-1);
01059                     }
01060                     if ( *s != TFUNCLOSE ) {
01061                         MesPrint("&Illegal argument for vector");
```

```
01062                       return(-1);
01063                   }
01064                   s++;
01065               }
01066               else {
01067                   if ( AC.DumNum ) {
01068                       *t++ = VECTOR; *t++ = 4; *t++ = x1;
01069                       *t++ = AC.DumNum + WILDOFFSET;
01070                   }
01071                   else {
01072                       *t++ = INDEX; *t++ = 3; *t++ = x1;
01073                   }
01074               }
01075               goto fin;
01076           case TDELTA:
01077               if ( *s != TFUNOPEN ) {
01078                   MesPrint("&d_ needs two arguments");
01079                   error = -1;
01080               }
01081               v = t; *t++ = DELTA; *t++ = 4;
01082               needarg = 2; x3 = x1 = -1;
01083               goto dotensor;
01084           case TFUNCTION:
01085               x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
01086               if ( x1 == AM.sumnum || x1 == AM.sumpnum ) sumlevel = x1;
01087               x1 += FUNCTION;
01088               if ( x1 == FIRSTBRACKET ) {
01089                   if ( s[0] == TFUNOPEN && s[1] == TEXPRESSION ) {
01090 doexpr:               s += 2;
01091                       *t++ = x1; *t++ = FUNHEAD+2; *t++ = 0;
01092                       if ( x1 == AR.PolyFun && AR.PolyFunType == 2 && AR.Eside != LHSIDE )
01093                           t[-1] |= MUSTCLEANPRF;
01094                       FILLFUN3(t)
01095                       x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
01096                       *t++ = -EXPRESSION; *t++ = x2;
01097 /*
01098                       The next code is added to facilitate parallel processing
01099                       We need to call GetTable here to make sure all processors
01100                       have the same numbering of all variables.
01101 */
01102                       if ( Expressions[x2].status == STOREDEXPRESSION ) {
01103                           TMproto[0] = EXPRESSION;
01104                           TMproto[1] = SUBEXPSIZE;
01105                           TMproto[2] = x2;
01106                           TMproto[3] = 1;
01107                           { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
01108                           AT.TMaddr = TMproto;
01109                           PUTZERO(position);
01110 /*
01111                           if ( (
01112 #ifdef WITHPTHREADS
01113                                   renumber =
01114 #endif
01115                                   GetTable(x2,&position,0) ) == 0 ) {
01116                               error = 1;
01117                               MesPrint("&Problems getting information about stored expression %s(1)"
01118                               ,EXPRNAME(x2));
01119                           }
01120 #ifdef WITHPTHREADS
01121                           M_free(renumber->symb.lo,"VarSpace");
01122                           M_free(renumber,"Renumber");
01123 #endif
01124 */
01125                           if ( ( renumber = GetTable(x2,&position,0) ) == 0 ) {
01126 /* INTERNAL_ERROR_EXCL_START */
01127                               error = 1;
01128                               MesPrint("!>Problems getting information about stored expression
      %s(1)"
01129                               ,EXPRNAME(x2));
01130 /* INTERNAL_ERROR_EXCL_STOP */
01131                           }
01132                           if ( renumber->symb.lo != AN.dummyrenumlist )
01133                               M_free(renumber->symb.lo,"VarSpace");
01134                           M_free(renumber,"Renumber");
01135                           AR.StoreData.dirtyflag = 1;
01136                       }
01137                       if ( *s != TFUNCLOSE ) {
01138                           if ( x1 == FIRSTBRACKET )
01139                               MesPrint("&Problems with argument of FirstBracket_");
01140                           else if ( x1 == FIRSTTERM )
01141                               MesPrint("&Problems with argument of FirstTerm_");
01142                           else if ( x1 == CONTENTTERM )
01143                               MesPrint("&Problems with argument of FirstTerm_");
01144                           else if ( x1 == TERMSINEXPR )
01145                               MesPrint("&Problems with argument of TermsIn_");
01146                           else if ( x1 == SIZEOFFUNCTION )
01147                               MesPrint("&Problems with argument of SizeOf_");
```

```
01148                           else if ( x1 == NUMFACTORS )
01149                               MesPrint("&Problems with argument of NumFactors_");
01150                           else
01151                               MesPrint("&Problems with argument of FactorIn_");
01152                           error = 1;
01153                           while ( *s != TENDOFIT && *s != TFUNCLOSE ) s++;
01154                       }
01155                       if ( *s == TFUNCLOSE ) s++;
01156                       goto fin;
01157                   }
01158               }
01159               else if ( x1 == TERMSINEXPR || x1 == SIZEOFFUNCTION || x1 == FACTORIN
01160                || x1 == NUMFACTORS || x1 == FIRSTTERM || x1 == CONTENTTERM ) {
01161                   if ( s[0] == TFUNOPEN && s[1] == TEXPRESSION ) goto doexpr;
01162                   if ( s[0] == TFUNOPEN && s[1] == TDOLLAR ) {
01163                       s += 2;
01164                       *t++ = x1; *t++ = FUNHEAD+2; *t++ = 0;
01165                       FILLFUN3(t)
01166                       x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
01167                       *t++ = -DOLLAREXPRESSION; *t++ = x2;
01168                       if ( *s != TFUNCLOSE ) {
01169                           if ( x1 == TERMSINEXPR )
01170                               MesPrint("&Problems with argument of TermsIn_");
01171                           else if ( x1 == SIZEOFFUNCTION )
01172                               MesPrint("&Problems with argument of SizeOf_");
01173                           else if ( x1 == NUMFACTORS )
01174                               MesPrint("&Problems with argument of NumFactors_");
01175                           else
01176                               MesPrint("&Problems with argument of FactorIn_");
01177                           error = 1;
01178                           while ( *s != TENDOFIT && *s != TFUNCLOSE ) s++;
01179                       }
01180                       if ( *s == TFUNCLOSE ) s++;
01181                       goto fin;
01182                   }
01183               }
01184               x3 = x1;
01185               if ( inset && ( t-tsize == 2 ) ) x1 -= FUNCTION;
01186               if ( *s == TWILDCARD ) { x1 += WILDOFFSET; s++; }
01187               if ( functions[x3-FUNCTION].commute ) cc = 1;
01188               if ( *s != TFUNOPEN ) {
01189                   *t++ = x1; *t++ = FUNHEAD; *t++ = 0;
01190                   if ( x1 == AR.PolyFun && AR.PolyFunType == 2 && AR.Eside != LHSIDE )
01191                           t[-1] |= MUSTCLEANPRF;
01192                   FILLFUN3(t) sumlevel = 0; goto fin;
01193               }
01194               v = t; *t++ = x1; *t++ = FUNHEAD; *t++ = DIRTYFLAG;
01195               if ( x1 == AR.PolyFun && AR.PolyFunType == 2 && AR.Eside != LHSIDE )
01196                   t[-1] |= MUSTCLEANPRF;
01197               FILLFUN3(t)
01198               needarg = -1;
01199               if ( !inset && functions[x3-FUNCTION].spec >= TENSORFUNCTION ) {
01200 dotensor:
01201                   do {
01202                       if ( needarg == 0 ) {
01203                           if ( x1 >= 0 ) {
01204                               x3 = x1;
01205                               if ( x3 >= FUNCTION+WILDOFFSET ) x3 -= WILDOFFSET;
01206                               MesPrint("&Too many arguments in function %s",
01207                                   VARNAME(functions,(x3-FUNCTION)) );
01208                           }
01209                           else
01210                               MesPrint("&d_ needs exactly two arguments");
01211                           error = -1;
01212                           needarg--;
01213                       }
01214                       else if ( needarg > 0 ) needarg--;
01215                       s++;
01216                       c = *s++;
01217                       if ( c == TMINUS && *s == TVECTOR ) { sign = -sign; c = *s++; }
01218                       base = ( c == TNUMBER ) ? 100: 128;
01219                       x2 = 0; while ( *s >= 0 ) { x2 = base*x2 + *s++; }
01220                       if ( *s == TWILDCARD && c != TNUMBER ) { x2 += WILDOFFSET; s++; }
01221                       if ( c == TSETNUM || c == TSETDOL ) {
01222                           if ( c == TSETDOL ) x2 = -x2;
01223                           if ( inset == 0 ) {
01224                               w1 = t; t += 2; w2 = t;
01225                               while ( w1 > v ) *--w2 = *--w1;
01226                               tsize = v; relo = AT.WorkTop;
01227                               *v++ = SETSET; *v++ = 0;
01228                           }
01229                           inset = 2; *--relo = x2; *--relo = t - v;
01230                           c = *s++;
01231                           x2 = 0; while ( *s >= 0 ) x2 = 128*x2 + *s++;
01232                           switch ( c ) {
01233                               case TINDEX:
01234                                   *t++ = x2;
```

```
01235                                   if ( t[-1]+AM.OffsetIndex > AM.IndDum ) {
01236                                       x2 = t[-1]+AM.OffsetIndex - AM.IndDum;
01237                                       if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
01238                                   }
01239                                   break;
01240                               case TVECTOR:
01241                                   *t++ = x2; break;
01242                               case TNUMBER1:
01243                                   if ( x2 >= 0 && x2 < AM.OffsetIndex ) {
01244                                       *t++ = x2; break;
01245                                   }
01246                                   /* fall through */
01247                               default:
01248                                   MesPrint("&Illegal type of set inside tensor");
01249                                   error = 1;
01250                                   *t++ = x2;
01251                                   break;
01252                           }
01253                       }
01254                       else { switch ( c ) {
01255                           case TINDEX:
01256                               if ( inset < 2 ) *t++ = x2 + AM.OffsetIndex;
01257                               else *t++ = x2;
01258                               if ( x2+AM.OffsetIndex > AM.IndDum ) {
01259                                   x2 = x2+AM.OffsetIndex - AM.IndDum;
01260                                   if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
01261                               }
01262                               break;
01263                           case TGENINDEX:
01264                               *t++ = AC.DumNum + WILDOFFSET;
01265                               break;
01266                           case TVECTOR:
01267                               if ( inset < 2 ) *t++ = x2 + AM.OffsetVector;
01268                               else *t++ = x2;
01269                               break;
01270                           case TWILDARG:
01271                               *t++ = FUNNYWILD; *t++ = x2;
01272 /*                            v[2] = 0; */
01273                               break;
01274                           case TDOLLAR:
01275                               *t++ = FUNNYDOLLAR; *t++ = x2;
01276                               break;
01277                           case TDUBIOUS:
01278                               if ( inset < 2 ) *t++ = x2 + AM.OffsetVector;
01279                               else *t++ = x2;
01280                               break;
01281                           case TSGAMMA:   /* Special gamma's */
01282                               if ( x3 != GAMMA ) {
01283                                   MesPrint("&5_,6_,7_ can only be used inside g_");
01284                                   error = -1;
01285                               }
01286                               *t++ = -x2;
01287                               break;
01288                           case TNUMBER:
01289                           case TNUMBER1:
01290                               if ( x2 >= AM.OffsetIndex && inset < 2 ) {
01291                                   MesPrint("&Value of constant index in tensor too large");
01292                                   error = -1;
01293                               }
01294                               *t++ = x2;
01295                               break;
01296                           default:
01297                               MesPrint("&Illegal object in tensor");
01298                               error = -1;
01299                               break;
01300                       }}
01301                       if ( inset >= 2 ) inset = 1;
01302                   } while ( *s == TCOMMA );
01303               }
01304               else {
01305 dofunction:       firstsumarg = 1;
01306                   do {
01307                       unsigned int ux2;
01308                       s++;
01309                       c = *s++;
01310                       if ( c == TMINUS && ( *s == TVECTOR || *s == TNUMBER
01311                       || *s == TNUMBER1 || *s == TSUBEXP ) ) {
01312                           minus = 1; c = *s++;
01313                       }
01314                       else minus = 0;
01315                       base = ( c == TNUMBER ) ? 100: 128;
01316                       ux2 = 0; while ( *s >= 0 ) { ux2 = base*ux2 + *s++; }
01317                       x2 = ux2;  /* may cause an implementation-defined behaviour */
01318 /*
01319         !!!!!!!!  What if it does not fit?
01320 */
01321                       if ( firstsumarg ) {
```

```
01322                               firstsumarg = 0;
01323                               if ( sumlevel > 0 ) {
01324                                   if ( c == TSYMBOL ) {
01325                                       sumlevel = x2; sumtype = SYMTOSYM;
01326                                   }
01327                                   else if ( c == TINDEX ) {
01328                                       sumlevel = x2+AM.OffsetIndex; sumtype = INDTOIND;
01329                                       if ( sumlevel > AM.IndDum ) {
01330                                           x2 = sumlevel - AM.IndDum;
01331                                           if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
01332                                       }
01333                                   }
01334                               }
01335                           }
01336                           if ( *s == TWILDCARD ) {
01337                               if ( c == TSYMBOL ) x2 += 2*MAXPOWER;
01338                               else if ( c != TNUMBER ) x2 += WILDOFFSET;
01339                               s++;
01340                           }
01341                           switch ( c ) {
01342                               case TSYMBOL:
01343                                   *t++ = -SYMBOL; *t++ = x2; break;
01344                               case TDOLLAR:
01345                                   *t++ = -DOLLAREXPRESSION; *t++ = x2; break;
01346                               case TEXPRESSION:
01347                                   *t++ = -EXPRESSION; *t++ = x2;
01348 /*
01349                                   The next code is added to facilitate parallel processing
01350                                   We need to call GetTable here to make sure all processors
01351                                   have the same numbering of all variables.
01352 */
01353                                   if ( Expressions[x2].status == STOREDEXPRESSION ) {
01354                                       TMproto[0] = EXPRESSION;
01355                                       TMproto[1] = SUBEXPSIZE;
01356                                       TMproto[2] = x2;
01357                                       TMproto[3] = 1;
01358                                       { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
01359                                       AT.TMaddr = TMproto;
01360                                       PUTZERO(position);
01361 /*
01362                                       if ( (
01363 #ifdef WITHPTHREADS
01364                                           renumber =
01365 #endif
01366                                               GetTable(x2,&position,0) ) == 0 ) {
01367                                           error = 1;
01368                                           MesPrint("&Problems getting information about stored
      expression %s(2)"
01369                                           ,EXPRNAME(x2));
01370                                       }
01371 #ifdef WITHPTHREADS
01372                                       M_free(renumber->symb.lo,"VarSpace");
01373                                       M_free(renumber,"Renumber");
01374 #endif
01375 */
01376                                       if ( ( renumber = GetTable(x2,&position,0) ) == 0 ) {
01377 /* INTERNAL_ERROR_EXCL_START */
01378                                           error = 1;
01379                                           MesPrint("!>Problems getting information about stored
      expression %s(2)"
01380                                           ,EXPRNAME(x2));
01381 /* INTERNAL_ERROR_EXCL_STOP */
01382                                       }
01383                                       if ( renumber->symb.lo != AN.dummyrenumlist )
01384                                           M_free(renumber->symb.lo,"VarSpace");
01385                                       M_free(renumber,"Renumber");
01386                                       AR.StoreData.dirtyflag = 1;
01387                                   }
01388                                   break;
01389                               case TINDEX:
01390                                   *t++ = -INDEX; *t++ = x2 + AM.OffsetIndex;
01391                                   if ( t[-1] > AM.IndDum ) {
01392                                       x2 = t[-1] - AM.IndDum;
01393                                       if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
01394                                   }
01395                                   break;
01396                               case TGENINDEX:
01397                                   *t++ = -INDEX; *t++ = AC.DumNum + WILDOFFSET;
01398                                   break;
01399                               case TVECTOR:
01400                                   if ( minus ) *t++ = -MINVECTOR;
01401                                   else *t++ = -VECTOR;
01402                                   *t++ = x2 + AM.OffsetVector;
01403                                   break;
01404                               case TSGAMMA:   /* Special gamma's */
01405                                   MesPrint("&5_,6_,7_ can only be used inside g_");
01406                                   error = -1;
```

```
01407                                   *t++ = -INDEX;
01408                                   *t++ = -x2;
01409                                   break;
01410                               case TDUBIOUS:
01411                                   *t++ = -SYMBOL; *t++ = x2; break;
01412                               case TFUNCTION:
01413                                   *t++ = -x2-FUNCTION;
01414                                   break;
01415                               case TSET:
01416                                   *t++ = -SETSET;
01417                                   *t++ = x2;
01418                                   break;
01419                               case TWILDARG:
01420                                   *t++ = -ARGWILD; *t++ = x2; break;
01421                               case TSETDOL:
01422                                   x2 = -x2;
01423                                   /* fall through */
01424                               case TSETNUM:
01425                                   if ( inset == 0 ) {
01426                                       w1 = t; t += 2; w2 = t;
01427                                       while ( w1 > v ) *--w2 = *--w1;
01428                                       tsize = v; relo = AT.WorkTop;
01429                                       *v++ = SETSET; *v++ = 0;
01430                                       inset = 1;
01431                                   }
01432                                   *--relo = x2; *--relo = t-v+1;
01433                                   c = *s++;
01434                                   x2 = 0; while ( *s >= 0 ) x2 = 128*x2 + *s++;
01435                                   switch ( c ) {
01436                                       case TFUNCTION:
01437                                           (*relo)--; *t++ = -x2-1; break;
01438                                       case TSYMBOL:
01439                                           *t++ = -SYMBOL; *t++ = x2; break;
01440                                       case TINDEX:
01441                                           *t++ = -INDEX; *t++ = x2;
01442                                           if ( x2+AM.OffsetIndex > AM.IndDum ) {
01443                                               x2 = x2+AM.OffsetIndex - AM.IndDum;
01444                                               if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
01445                                           }
01446                                           break;
01447                                       case TVECTOR:
01448                                           *t++ = -VECTOR; *t++ = x2; break;
01449                                       case TNUMBER1:
01450                                           *t++ = -SNUMBER; *t++ = x2; break;
01451                                       default:
01452 /* INTERNAL_ERROR_EXCL_START */
01453                                           MesPrint("!>Internal error 435");
01454                                           error = 1;
01455                                           *t++ = -SYMBOL; *t++ = x2; break;
01456 /* INTERNAL_ERROR_EXCL_STOP */
01457                                   }
01458                                   break;
01459                               case TSUBEXP:
01460                                   w2 = AC.ProtoType; i = w2[1];
01461                                   w1 = t;
01462                                   *t++ = i+ARGHEAD+4;
01463                                   *t++ = 1;
01464                                   FILLARG(t);
01465                                   *t++ = i + 4;
01466                                   while ( --i >= 0 ) *t++ = *w2++;
01467                                   w1[ARGHEAD+3] = subexpbuffers[x2].subexpnum;
01468                                   w1[ARGHEAD+5] = subexpbuffers[x2].buffernum;
01469                                   if ( sumlevel > 0 ) {
01470                                       w1[0] += 4;
01471                                       w1[ARGHEAD] += 4;
01472                                       w1[ARGHEAD+2] += 4;
01473                                       *t++ = sumtype; *t++ = 4;
01474                                       *t++ = sumlevel; *t++ = sumlevel;
01475                                   }
01476                                   *t++ = 1; *t++ = 1;
01477                                   if ( minus ) *t++ = -3;
01478                                   else         *t++ =  3;
01479                                   break;
01480                               case TNUMBER:
01481                               case TNUMBER1:
01482                                   if ( minus ) x2 = UnsignedToInt(-IntAbs(x2));
01483                                   *t++ = -SNUMBER;
01484                                   *t++ = x2;
01485                                   break;
01486                               default:
01487                                   MesPrint("&Illegal object in function");
01488                                   error = -1;
01489                                   break;
01490                           }
01491                       } while ( *s == TCOMMA );
01492                   }
01493                   if ( *s != TFUNCLOSE ) {
```

```
01494                   MesPrint("&Illegal argument field for function. Expected )");
01495                   return(-1);
01496               }
01497               s++; sumlevel = 0;
01498               v[1] = t-v;
01499 /*
01500               if ( *v == AM.termfunnum && ( v[1] != FUNHEAD+2 ||
01501               v[FUNHEAD] != -DOLLAREXPRESSION ) ) {
01502                   MesPrint("&The function term_ can only have one argument with a single
      $-expression");
01503                   error = 1;
01504               }
01505 */
01506               goto fin;
01507           case TDUBIOUS:
01508               x1 = 0; while ( *s >= 0 ) x1 = 128*x1 + *s++;
01509               if ( *s == TWILDCARD ) s++;
01510               if ( *s == TDOT ) goto dovector;
01511               if ( *s == TFUNOPEN ) {
01512                   x1 += FUNCTION;
01513                   cc = 1;
01514                   v = t; *t++ = x1; *t++ = FUNHEAD; *t++ = DIRTYFLAG;
01515                   if ( x1 == AR.PolyFun && AR.PolyFunType == 2 && AR.Eside != LHSIDE )
01516                           t[-1] |= MUSTCLEANPRF;
01517                   FILLFUN3(t)
01518                   needarg = -1; goto dofunction;
01519               }
01520               *t++ = SYMBOL; *t++ = 4; *t++ = 0;
01521               if ( inset ) *relo = 2;
01522               goto TryPower;
01523           case TSUBEXP:
01524               x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
01525               if ( *s == TPOWER ) {
01526                   s++; c = *s++;
01527                   base = ( c == TNUMBER ) ? 100: 128;
01528                   x2 = 0; while ( *s >= 0 ) { x2 = base*x2 + *s++; }
01529                   if ( *s == TWILDCARD ) { x2 += 2*MAXPOWER; s++; }
01530                   else if ( c == TSYMBOL ) x2 += 2*MAXPOWER;
01531               }
01532               else x2 = 1;
01533               r = AC.ProtoType; n = r[1] - 5; r += 5;
01534               *t++ = SUBEXPRESSION; *t++ = r[-4];
01535               *t++ = subexpbuffers[x1].subexpnum;
01536               *t++ = x2*deno;
01537               *t++ = subexpbuffers[x1].buffernum;
01538               NCOPY(t,r,n);
01539               if ( cbuf[subexpbuffers[x1].buffernum].CanCommu[subexpbuffers[x1].subexpnum] ) cc = 1;
01540               deno = 1;
01541               break;
01542           case TMULTIPLY:
01543               mulflag = 1;
01544               break;
01545           case TDIVIDE:
01546               mulflag = 1;
01547               deno = -deno;
01548               break;
01549           case TEXPRESSION:
01550               cc = 1;
01551               x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
01552               v = t;
01553               *t++ = EXPRESSION; *t++ = SUBEXPSIZE; *t++ = x1; *t++ = deno;
01554               *t++ = 0; FILLSUB(t)
01555 /*
01556               Here we had some erroneous code before. It should be after
01557               the reading of the parameters as it is now (after 15-jan-2007).
01558               Thomas Hahn noticed this and reported it.
01559 */
01560               if ( *s == TFUNOPEN ) {
01561                   do {
01562                       s++; c = *s++;
01563                       base = ( c == TNUMBER ) ? 100: 128;
01564                       x2 = 0; while ( *s >= 0 ) { x2 = base*x2 + *s++; }
01565                       switch ( c ) {
01566                           case TSYMBOL:
01567                               *t++ = SYMBOL; *t++ = 4; *t++ = x2; *t++ = 1;
01568                               break;
01569                           case TINDEX:
01570                               *t++ = INDEX; *t++ = 3; *t++ = x2+AM.OffsetIndex;
01571                               if ( t[-1] > AM.IndDum ) {
01572                                   x2 = t[-1] - AM.IndDum;
01573                                   if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
01574                               }
01575                               break;
01576                           case TVECTOR:
01577                               *t++ = INDEX; *t++ = 3; *t++ = x2+AM.OffsetVector;
01578                               break;
01579                           case TFUNCTION:
```

```
01580                                 *t++ = x2+FUNCTION; *t++ = 2; break;
01581                             case TNUMBER:
01582                             case TNUMBER1:
01583                                 if ( x2 >= AM.OffsetIndex || x2 < 0 ) {
01584                                     MesPrint("&Index as argument of expression has illegal value");
01585                                     error = -1;
01586                                 }
01587                                 *t++ = INDEX; *t++ = 3; *t++ = x2; break;
01588                             case TSETDOL:
01589                                 x2 = -x2;
01590                                 /* fall through */
01591                             case TSETNUM:
01592                                 if ( inset == 0 ) {
01593                                     w1 = t; t += 2; w2 = t;
01594                                     while ( w1 > v ) *--w2 = *--w1;
01595                                     tsize = v; relo = AT.WorkTop;
01596                                     *v++ = SETSET; *v++ = 0;
01597                                     inset = 1;
01598                                 }
01599                                 *--relo = x2; *--relo = t-v+2;
01600                                 c = *s++;
01601                                 x2 = 0; while ( *s >= 0 ) x2 = 128*x2 + *s++;
01602                                 switch ( c ) {
01603                                     case TFUNCTION:
01604                                         *relo -= 2; *t++ = -x2-1; break;
01605                                     case TSYMBOL:
01606                                         *t++ = SYMBOL; *t++ = 4; *t++ = x2; *t++ = 1; break;
01607                                     case TINDEX:
01608                                         *t++ = INDEX; *t++ = 3; *t++ = x2;
01609                                         if ( x2+AM.OffsetIndex > AM.IndDum ) {
01610                                             x2 = x2+AM.OffsetIndex - AM.IndDum;
01611                                             if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
01612                                         }
01613                                         break;
01614                                     case TVECTOR:
01615                                         *t++ = VECTOR; *t++ = 3; *t++ = x2; break;
01616                                     case TNUMBER1:
01617                                         *t++ = SNUMBER; *t++ = 4; *t++ = x2; *t++ = 1; break;
01618                                     default:
01619 /* INTERNAL_ERROR_EXCL_START */
01620                                         MesPrint("!>Internal error 435");
01621                                         error = 1;
01622                                         *t++ = SYMBOL; *t++ = 4; *t++ = x2; *t++ = 1; break;
01623 /* INTERNAL_ERROR_EXCL_STOP */
01624                                 }
01625                                 break;
01626                             default:
01627                                 MesPrint("&Argument of expression can only be symbol, index, vector or
      function");
01628                                 error = -1;
01629                                 break;
01630                         }
01631                     } while ( *s == TCOMMA );
01632                     if ( *s != TFUNCLOSE ) {
01633                         MesPrint("&Illegal object in argument field for expression");
01634                         error = -1;
01635                         while ( *s != TFUNCLOSE ) s++;
01636                     }
01637                     s++;
01638                 }
01639                 r = AC.ProtoType; n = r[1];
01640                 if ( n > SUBEXPSIZE ) {
01641                     *t++ = WILDCARDS; *t++ = n+2;
01642                     NCOPY(t,r,n);
01643                 }
01644 /*
01645                 Code added for parallel processing.
01646                 This is different from the other occurrences to test immediately
01647                 for renumbering. Here we have to read the parameters first.
01648 */
01649                 if ( Expressions[x1].status == STOREDEXPRESSION ) {
01650                     v[1] = t-v;
01651                     AT.TMaddr = v;
01652                     PUTZERO(position);
01653 /*
01654                     if ( (
01655 #ifdef WITHPTHREADS
01656                         renumber =
01657 #endif
01658                             GetTable(x1,&position,0) ) == 0 ) {
01659                         error = 1;
01660                         MesPrint("&Problems getting information about stored expression %s(3)"
01661                         ,EXPRNAME(x1));
01662                     }
01663 #ifdef WITHPTHREADS
01664                     M_free(renumber->symb.lo,"VarSpace");
01665                     M_free(renumber,"Renumber");
```

```
01666 #endif
01667 */
01668                     if ( ( renumber = GetTable(x1,&position,0) ) == 0 ) {
01669 /* INTERNAL_ERROR_EXCL_START */
01670                         error = 1;
01671                         MesPrint("!>Problems getting information about stored expression %s(3)"
01672                         ,EXPRNAME(x1));
01673 /* INTERNAL_ERROR_EXCL_STOP */
01674                     }
01675                     if ( renumber->symb.lo != AN.dummyrenumlist )
01676                         M_free(renumber->symb.lo,"VarSpace");
01677                     M_free(renumber,"Renumber");
01678                     AR.StoreData.dirtyflag = 1;
01679                 }
01680                 if ( *s == LBRACE ) {
01681 /*
01682                     This should be one term that should be inserted
01683                     FROMBRAC size+2 ( term )
01684                     Because this term should have been translated
01685                     already we can copy it from the 'subexpression'
01686 */
01687                     s++;
01688                     if ( *s != TSUBEXP ) {
01689 /* INTERNAL_ERROR_EXCL_START */
01690                         MesPrint("!>Internal error 23");
01691                         Terminate(-1);
01692 /* INTERNAL_ERROR_EXCL_STOP */
01693                     }
01694                     s++; x2 = 0; while ( *s >= 0 ) { x2 = 128*x2 + *s++; }
01695                     r = cbuf[subexpbuffers[x2].buffernum].rhs[subexpbuffers[x2].subexpnum];
01696                     *t++ = FROMBRAC; *t++ = *r+2;
01697                     n = *r;
01698                     NCOPY(t,r,n);
01699                     if ( *r != 0 ) {
01700                         MesPrint("&Object between [] in expression should be a single term");
01701                         error = -1;
01702                     }
01703                     if ( *s != RBRACE ) {
01704 /* INTERNAL_ERROR_EXCL_START */
01705                         MesPrint("!>Internal error 23b");
01706                         Terminate(-1);
01707 /* INTERNAL_ERROR_EXCL_STOP */
01708                     }
01709                     s++;
01710                 }
01711                 if ( *s == TPOWER ) {
01712                     s++; c = *s++;
01713                     base = ( c == TNUMBER ) ? 100: 128;
01714                     x2 = 0; while ( *s >= 0 ) { x2 = base*x2 + *s++; }
01715                     if ( *s == TWILDCARD || c == TSYMBOL ) { x2 += 2*MAXPOWER; s++; }
01716                     v[3] = x2;
01717                 }
01718                 v[1] = t - v;
01719                 deno = 1;
01720                 break;
01721             case TNUMBER:
01722                 if ( *s == 0 ) {
01723                     s++;
01724                     if ( *s == TPOWER ) {
01725                         s++; if ( *s == TMINUS ) { s++; deno = -deno; }
01726                         c = *s++; base = ( c == TNUMBER ) ? 100: 128;
01727                         x2 = 0; while ( *s >= 0 ) { x2 = x2*base + *s++; }
01728                         if ( x2 == 0 ) {
01729                             error = -1;
01730                             MesPrint("&Encountered 0^0 during compilation");
01731                         }
01732                         if ( deno < 0 ) {
01733                             error = -1;
01734                             MesPrint("&Division by zero during compilation (0 to the power negative
      number)");
01735                         }
01736                     }
01737                     else if ( deno < 0 ) {
01738                         error = -1;
01739                         MesPrint("&Division by zero during compilation");
01740                     }
01741                     sign = 0; break; /* term is zero */
01742                 }
01743                 y = *s++;
01744                 if ( *s >= 0 ) { y = 100*y + *s++; }
01745                 innum[0] = y; nin = 1;
01746                 while ( *s >= 0 ) {
01747                     y = *s++; x2 = 100;
01748                     if ( *s >= 0 ) { y = 100*y + *s++; x2 = 10000; }
01749                     Product(innum,&nin,(WORD)x2);
01750                     if ( y ) AddLong(innum,nin,(UWORD *)(&y),(WORD)1,innum,&nin);
01751                 }
```

```
01752 docoef:
01753                 if ( *s == TPOWER ) {
01754                     s++; if ( *s == TMINUS ) { s++; deno = -deno; }
01755                     c = *s++; base = ( c == TNUMBER ) ? 100: 128;
01756                     x2 = 0; while ( *s >= 0 ) { x2 = x2*base + *s++; }
01757                     if ( x2 == 0 ) {
01758                         innum[0] = 1; nin = 1;
01759                     }
01760                     else if ( RaisPow(BHEAD innum,&nin,x2) ) {
01761                         error = -1; innum[0] = 1; nin = 1;
01762                     }
01763                 }
01764                 if ( deno > 0 ) {
01765                     Simplify(BHEAD innum,&nin,denominator,&ndenominator);
01766                     for ( i = 0; i < nnumerator; i++ ) CGscrat7[i] = numerator[i];
01767                     MulLong(innum,nin,CGscrat7,nnumerator,numerator,&nnumerator);
01768                 }
01769                 else if ( deno < 0 ) {
01770                     Simplify(BHEAD innum,&nin,numerator,&nnumerator);
01771                     for ( i = 0; i < ndenominator; i++ ) CGscrat7[i] = denominator[i];
01772                     MulLong(innum,nin,CGscrat7,ndenominator,denominator,&ndenominator);
01773                 }
01774                 deno = 1;
01775                 break;
01776             case TNUMBER1:
01777                 if ( *s == 0 ) { s++; sign = 0; break; /* term is zero */ }
01778                 y = *s++;
01779                 if ( *s >= 0 ) { y = 128*y + *s++; }
01780                 if ( inset == 0 ) {
01781                     innum[0] = y; nin = 1;
01782                     while ( *s >= 0 ) {
01783                         y = *s++; x2 = 128;
01784                         if ( *s >= 0 ) { y = 128*y + *s++; x2 = 16384; }
01785                         Product(innum,&nin,(WORD)x2);
01786                         if ( y ) AddLong(innum,nin,(UWORD *)&y,(WORD)1,innum,&nin);
01787                     }
01788                     goto docoef;
01789                 }
01790                 *relo = 2; *t++ = SNUMBER; *t++ = 4; *t++ = y;
01791                 goto TryPower;
01792 #ifdef WITHFLOAT
01793             case TFLOAT:
01794                 { WORD *w;
01795                 s = ReadFloat(s);
01796                 i = AT.WorkPointer[1];
01797                 w = AT.WorkPointer;
01798                 NCOPY(t,w,i);
01799 /*
01800 Power?
01801 */
01802                 }
01803                 break;
01804 #endif
01805             case TDOLLAR:
01806             {
01807                 WORD *powplace;
01808                 x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
01809                 if ( AR.Eside != LHSIDE ) {
01810                     *t++ = SUBEXPRESSION; *t++ = SUBEXPSIZE; *t++ = x1;
01811                 }
01812                 else {
01813                     *t++ = DOLLAREXPRESSION; *t++ = SUBEXPSIZE; *t++ = x1;
01814                 }
01815                 powplace = t; t++;
01816                 *t++ = AM.dbufnum; FILLSUB(t)
01817 /*
01818                 Now we have to test for factors of dollars with [ ] and [ [ ]]
01819 */
01820                 if ( *s == LBRACE ) {
01821                     int bracelevel = 1;
01822                     s++;
01823                     while ( bracelevel > 0 ) {
01824                         if ( *s == RBRACE ) {
01825                             bracelevel--; s++;
01826                         }
01827                         else if ( *s == TNUMBER ) {
01828                             s++;
01829                             x2 = 0; while ( *s >= 0 ) { x2 = 100*x2 + *s++; }
01830                             *t++ = DOLLAREXPR2; *t++ = 3; *t++ = -x2-1;
01831 CloseBraces:
01832                             while ( bracelevel > 0 ) {
01833                                 if ( *s != RBRACE ) {
01834 ErrorBraces:
01835                                     error = -1;
01836                                     MesPrint("&Improper use of [] in $-variable.");
01837                                     return(error);
01838                                 }
```

```
01839                                   else {
01840                                       s++; bracelevel--;
01841                                   }
01842                               }
01843                           }
01844                           else if ( *s == TDOLLAR ) {
01845                               s++;
01846                               x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
01847                               *t++ = DOLLAREXPR2; *t++ = 3; *t++ = x1;
01848                               if ( *s == RBRACE ) goto CloseBraces;
01849                               else if ( *s == LBRACE ) {
01850                                   s++; bracelevel++;
01851                               }
01852                           }
01853                           else goto ErrorBraces;
01854                       }
01855                   }
01856 /*
01857                   Finally we can continue with the power
01858 */
01859                   if ( *s == TPOWER ) {
01860                       s++;
01861                       if ( *s == TMINUS ) { s++; deno = -deno; }
01862                       c = *s++;
01863                       base = ( c == TNUMBER ) ? 100: 128;
01864                       x2 = 0; while ( *s >= 0 ) { x2 = base*x2 + *s++; }
01865                       if ( c == TSYMBOL ) {
01866                           if ( *s == TWILDCARD ) s++;
01867                           x2 += 2*MAXPOWER;
01868                       }
01869                       *powplace = deno*x2;
01870                   }
01871                   else *powplace = deno;
01872                   deno = 1;
01873 /*
01874                   if ( inset ) {
01875                       while ( relo < AT.WorkTop ) *t++ = *relo++;
01876                       inset = 0; tsize[1] = t - tsize;
01877                   }
01878 */
01879               }
01880                   break;
01881               case TSETNUM:
01882                   inset = 1; tsize = t; relo = AT.WorkTop;
01883                   *t++ = SETSET; *t++ = 0;
01884                   x1 = 0; while ( *s >= 0 ) x1 = x1*128 + *s++;
01885                   *--relo = x1; *--relo = 0;
01886                   break;
01887               case TSETDOL:
01888                   inset = 1; tsize = t; relo = AT.WorkTop;
01889                   *t++ = SETSET; *t++ = 0;
01890                   x1 = 0; while ( *s >= 0 ) x1 = x1*128 + *s++;
01891                   *--relo = -x1; *--relo = 0;
01892                   break;
01893               case TFUNOPEN:
01894                   MesPrint("&Illegal use of function arguments");
01895                   error = -1;
01896                   funflag = 1;
01897                   deno = 1;
01898                   break;
01899               case TFUNCLOSE:
01900                   if ( funflag == 0 )
01901                       MesPrint("&Illegal use of function arguments");
01902                   error = -1;
01903                   funflag = 0;
01904                   deno = 1;
01905                   break;
01906               case TSGAMMA:
01907                   MesPrint("&Illegal use special gamma symbols 5_, 6_, 7_");
01908                   error = -1;
01909                   funflag = 0;
01910                   deno = 1;
01911                   break;
01912               case TCONJUGATE:
01913                   MesPrint("&Complex conjugate operator (%#) is not implemented");
01914                   error = -1;
01915                   deno = 1;
01916                   break;
01917               default:
01918                   MesPrint("&Internal error in code generator. Unknown object: %d",c);
01919                   error = -1;
01920                   deno = 1;
01921                   break;
01922               }
01923           }
01924       }
01925       if ( mulflag ) {
```

```
01926         MesPrint("&Irregular end of statement.");
01927         error = 1;
01928     }
01929     if ( !first && error == 0 ) {
01930         *term = t-term;
01931         C->NumTerms[numexp]++;
01932         if ( cc && sign ) C->CanCommu[numexp]++;
01933         error = CompleteTerm(term,numerator,denominator,nnumerator,ndenominator,sign);
01934     }
01935     AT.WorkPointer = oldwork;
01936     if ( error ) return(-1);
01937     AddToCB(C,0)
01938     if ( AC.CompileLevel > 0 && AR.Eside != LHSIDE ) {
01939         /* See whether we have this one already */
01940         error = InsTree(AC.cbufnum,C->numrhs);
01941         if ( error < (C->numrhs) ) {
01942             C->Pointer = C->rhs[C->numrhs--];
01943             return(error);
01944         }
01945     }
01946     return(C->numrhs);
01947 OverWork:
01948     MLOCK(ErrorMessageLock);
01949     MesWork();
01950     MUNLOCK(ErrorMessageLock);
01951     return(-1);
01952 }
01953
01954 /*
01955         #] CodeGenerator :
01956         #[ CompleteTerm :
01957
01958         Completes the term
01959         Puts it in the buffer
01960 */
01961
01962 int CompleteTerm(WORD *term, UWORD *numer, UWORD *denom, WORD nnum, WORD nden, int sign)
01963 {
01964     int nsize, i;
01965     WORD *t;
01966     if ( sign == 0 ) return(0);     /* Term is zero */
01967     if ( nnum >= nden ) nsize = nnum;
01968     else                nsize = nden;
01969     t = term + *term;
01970     for ( i = 0; i < nnum; i++ ) *t++ = numer[i];
01971     for ( ; i < nsize; i++ ) *t++ = 0;
01972     for ( i = 0; i < nden; i++ ) *t++ = denom[i];
01973     for ( ; i < nsize; i++ ) *t++ = 0;
01974     *t++ = (2*nsize+1)*sign;
01975     *term = t - term;
01976     AddNtoC(AC.cbufnum,*term,term,7);
01977     return(0);
01978 }
01979
01980 /*
01981         #] CompleteTerm :
01982         #[ CodeFactors :
01983
01984         This routine does the part of reading in in terms of factors.
01985         If there is more than one term at this level we have only one
01986         factor. In that case any expression should first be unfactorized.
01987         Then the whole expression gets read as a new subexpression and finally
01988         we generate factor_*subexpression.
01989         If the whole has only multiplications we have factors. Then the
01990         nasty thing is powers of objects and in particular powers of
01991         factorized expressions or dollars.
01992         For a power we generate a new subexpression of the type
01993           1+factor_+...+factor_^(power-1)
01994         with which we multiply.
01995
01996         WE HAVE NOT YET WORRIED ABOUT SETS
01997 */
01998
01999 int CodeFactors(SBYTE *tokens)
02000 {
02001     GETIDENTITY
02002     EXPRESSIONS e = Expressions + AR.CurExpr;
02003     int nfactor = 1, nparenthesis, i, last = 0, error = 0;
02004     SBYTE *t, *startobject, *tt, *s1, *out, *outtokens;
02005     WORD nexp, subexp = 0, power, pow, x2, powfactor, first;
02006 /*
02007     First scan the number of factors
02008 */
02009     t = tokens;
02010     while ( *t != TENDOFIT ) {
02011         if ( *t >= 0 ) { while ( *t >= 0 ) t++; continue; }
02012         if ( *t == LPARENTHESIS || *t == LBRACE || *t == TSETOPEN || *t == TFUNOPEN ) {
```

```
02013             nparenthesis = 0; t++;
02014             while ( nparenthesis >= 0 ) {
02015                 if ( *t == LPARENTHESIS || *t == LBRACE || *t == TSETOPEN || *t == TFUNOPEN )
      nparenthesis++;
02016                 else if ( *t == RPARENTHESIS || *t == RBRACE || *t == TSETCLOSE || *t == TFUNCLOSE )
      nparenthesis--;
02017                 t++;
02018             }
02019             continue;
02020         }
02021         else if ( ( *t == TPLUS || *t == TMINUS ) && ( t > tokens )
02022         && ( t[-1] != TPLUS && t[-1] != TMINUS ) ) {
02023             if ( t[-1] >= 0 || t[-1] == RPARENTHESIS || t[-1] == RBRACE
02024             || t[-1] == TSETCLOSE || t[-1] == TFUNCLOSE ) {
02025                 subexp = CodeGenerator(tokens);
02026                 if ( subexp < 0 ) error = -1;
02027                 if ( insubexpbuffers >= MAXSUBEXPRESSIONS ) {
02028                     MesPrint("&More than %d subexpressions inside one
      expression",(WORD)MAXSUBEXPRESSIONS);
02029                     Terminate(-1);
02030                 }
02031                 if ( subexpbuffers+insubexpbuffers >= topsubexpbuffers ) {
02032                     DoubleBuffer((void **)((void *)(&subexpbuffers))
02033                     ,(void **)((void *)(&topsubexpbuffers)),sizeof(SUBBUF),"subexpbuffers");
02034                 }
02035                 subexpbuffers[insubexpbuffers].subexpnum = subexp;
02036                 subexpbuffers[insubexpbuffers].buffernum = AC.cbufnum;
02037                 subexp = insubexpbuffers++;
02038                 t = tokens;
02039                 *t++ = TSYMBOL; *t++ = FACTORSYMBOL;
02040                 *t++ = TMULTIPLY; *t++ = TSUBEXP;
02041                 PUTNUMBER128(t,subexp)
02042                 *t++ = TENDOFIT;
02043                 e->numfactors = 1;
02044                 e->vflags |= ISFACTORIZED;
02045                 return(subexp);
02046             }
02047         }
02048         else if ( ( *t == TMULTIPLY || *t == TDIVIDE ) && t > tokens ) {
02049             nfactor++;
02050         }
02051         else if ( *t == TEXPRESSION ) {
02052             t++;
02053             nexp = 0; while ( *t >= 0 ) { nexp = nexp*128 + *t++; }
02054             if ( *t == LBRACE ) continue;
02055             if ( ( AS.Oldvflags[nexp] & ISFACTORIZED ) != 0 ) {
02056                 nfactor += AS.OldNumFactors[nexp];
02057             }
02058             else { nfactor++; }
02059             continue;
02060         }
02061         else if ( *t == TDOLLAR ) {
02062             t++;
02063             nexp = 0; while ( *t >= 0 ) { nexp = nexp*128 + *t++; }
02064             if ( *t == LBRACE ) continue;
02065             if ( Dollars[nexp].nfactors > 0 ) {
02066                 nfactor += Dollars[nexp].nfactors;
02067             }
02068             else { nfactor++; }
02069             continue;
02070         }
02071         t++;
02072     }
02073 /*
02074     Now the real pass.
02075     nfactor is a not so reliable measure for the space we need.
02076 */
02077     outtokens = (SBYTE *)Malloc1(((t-tokens)+(nfactor+2)*25)*sizeof(SBYTE),"CodeFactors");
02078     out = outtokens;
02079     t = tokens; first = 1; powfactor = 1;
02080     while ( *t == TPLUS || *t == TMINUS ) { if ( *t == TMINUS ) first = -first; t++; }
02081     if ( first < 0 ) {
02082         *out++ = TMINUS; *out++ = TSYMBOL; *out++ = FACTORSYMBOL;
02083         *out++ = TPOWER; *out++ = TNUMBER; PUTNUMBER100(out,powfactor)
02084         powfactor++;
02085     }
02086     startobject = t; power = 1;
02087     while ( *t != TENDOFIT ) {
02088         if ( *t >= 0 ) { while ( *t >= 0 ) t++; continue; }
02089         if ( *t == LPARENTHESIS || *t == LBRACE || *t == TSETOPEN || *t == TFUNOPEN ) {
02090             nparenthesis = 0; t++;
02091             while ( nparenthesis >= 0 ) {
02092                 if ( *t == LPARENTHESIS || *t == LBRACE || *t == TSETOPEN || *t == TFUNOPEN )
      nparenthesis++;
02093                 else if ( *t == RPARENTHESIS || *t == RBRACE || *t == TSETCLOSE || *t == TFUNCLOSE )
      nparenthesis--;
02094                 t++;
```

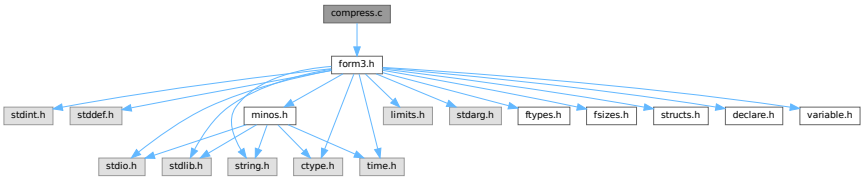
```
02095             }
02096             continue;
02097         }
02098         else if ( ( *t == TMULTIPLY || *t == TDIVIDE ) && ( t > tokens ) ) {
02099             if ( t[-1] >= 0 || t[-1] == RPARENTHESIS || t[-1] == RBRACE
02100             || t[-1] == TSETCLOSE || t[-1] == TFUNCLOSE ) {
02101 dolast:
02102                 if ( startobject ) {    /* apparently power is 1 or -1 */
02103                     *out++ = TPLUS;
02104                     if ( power < 0 ) { *out++ = TNUMBER; *out++ = 1; *out++ = TDIVIDE; }
02105                     s1 = startobject;
02106                     while ( s1 < t ) *out++ = *s1++;
02107                     *out++ = TMULTIPLY; *out++ = TSYMBOL; *out++ = FACTORSYMBOL;
02108                     *out++ = TPOWER; *out++ = TNUMBER; PUTNUMBER100(out,powfactor)
02109                     powfactor++;
02110                 }
02111                 if ( last ) { startobject = 0; break; }
02112                 startobject = t+1;
02113                 if ( *t == TDIVIDE ) power = -1;
02114                 if ( *t == TMULTIPLY ) power = 1;
02115             }
02116         }
02117         else if ( *t == TPOWER ) {
02118             pow = 1;
02119             tt = t+1;
02120             while ( ( *tt == TMINUS ) || ( *tt == TPLUS ) ) {
02121                 if ( *tt == TMINUS ) pow = -pow;
02122                 tt++;
02123             }
02124             if ( *tt == TSYMBOL ) {
02125                 tt++; while ( *tt >= 0 ) tt++;
02126                 t = tt; continue;
02127             }
02128             tt++; x2 = 0; while ( *tt >= 0 ) { x2 = 100*x2 + *tt++; }
02129 /*
02130             We have an object in startobject till t. The power is
02131             power*pow*x2
02132 */
02133             power = power*pow*x2;
02134             if ( power < 0 ) { pow = -power; power = -1; }
02135             else if ( power == 0 ) { t = tt; startobject = tt; continue; }
02136             else { pow = power; power = 1; }
02137             *out++ = TPLUS;
02138             if ( pow > 1 ) {
02139                 subexp = GenerateFactors(pow,1);
02140                 if ( subexp < 0 ) { error = -1; subexp = 0; }
02141                 *out++ = TSUBEXP; PUTNUMBER128(out,subexp);
02142             }
02143             *out++ = TSYMBOL; *out++ = FACTORSYMBOL;
02144             *out++ = TPOWER; *out++ = TNUMBER; PUTNUMBER100(out,powfactor)
02145             powfactor += pow;
02146             if ( power > 0 ) *out++ = TMULTIPLY;
02147             else *out++ = TDIVIDE;
02148             s1 = startobject; while ( s1 < t ) *out++ = *s1++;
02149             startobject = 0; t = tt; continue;
02150         }
02151         else if ( *t == TEXPRESSION ) {
02152             startobject = t;
02153             t++;
02154             nexp = 0; while ( *t >= 0 ) { nexp = nexp*128 + *t++; }
02155             if ( *t == LBRACE ) continue;
02156             if ( *t == LPARENTHESIS ) {
02157                 nparenthesis = 0; t++;
02158                 while ( nparenthesis >= 0 ) {
02159                     if ( *t == LPARENTHESIS ) nparenthesis++;
02160                     else if ( *t == RPARENTHESIS ) nparenthesis--;
02161                     t++;
02162                 }
02163             }
02164             if ( ( AS.Oldvflags[nexp] & ISFACTORIZED ) == 0 ) continue;
02165             if ( *t == TPOWER ) {
02166                 pow = 1;
02167                 tt = t+1;
02168                 while ( ( *tt == TMINUS ) || ( *tt == TPLUS ) ) {
02169                     if ( *tt == TMINUS ) pow = -pow;
02170                     tt++;
02171                 }
02172                 if ( *tt != TNUMBER ) {
02173 /* INTERNAL_ERROR_EXCL_START */
02174                     MesPrint("!>Internal problems(1) in CodeFactors");
02175                     return(-1);
02176 /* INTERNAL_ERROR_EXCL_STOP */
02177                 }
02178                 tt++; x2 = 0; while ( *tt >= 0 ) { x2 = 100*x2 + *tt++; }
02179 /*
02180                 We have an object in startobject till t. The power is
02181                 power*pow*x2
```

```
02182 */
02183 dopower:
02184                 power = power*pow*x2;
02185                 if ( power < 0 ) { pow = -power; power = -1; }
02186                 else if ( power == 0 ) { t = tt; startobject = tt; continue; }
02187                 else { pow = power; power = 1; }
02188                 *out++ = TPLUS;
02189                 if ( pow > 1 ) {
02190                     subexp = GenerateFactors(pow,AS.OldNumFactors[nexp]);
02191                     if ( subexp < 0 ) { error = -1; subexp = 0; }
02192                     *out++ = TSUBEXP; PUTNUMBER128(out,subexp)
02193                     *out++ = TMULTIPLY;
02194                 }
02195                 i = powfactor-1;
02196                 if ( i > 0 ) {
02197                     *out++ = TSYMBOL; *out++ = FACTORSYMBOL;
02198                     if ( i > 1 ) {
02199                         *out++ = TPOWER; *out++ = TNUMBER; PUTNUMBER100(out,i)
02200                     }
02201                     *out++ = TMULTIPLY;
02202                 }
02203                 powfactor += AS.OldNumFactors[nexp]*pow;
02204                 s1 = startobject;
02205                 while ( s1 < t ) *out++ = *s1++;
02206                 startobject = 0; t = tt; continue;
02207             }
02208             else {
02209                 tt = t; pow = 1; x2 = 1; goto dopower;
02210             }
02211         }
02212         else if ( *t == TDOLLAR ) {
02213             startobject = t;
02214             t++;
02215             nexp = 0; while ( *t >= 0 ) { nexp = nexp*128 + *t++; }
02216             if ( *t == LBRACE ) continue;
02217             if ( Dollars[nexp].nfactors == 0 ) continue;
02218             if ( *t == TPOWER ) {
02219                 pow = 1;
02220                 tt = t+1;
02221                 while ( ( *tt == TMINUS ) || ( *tt == TPLUS ) ) {
02222                     if ( *tt == TMINUS ) pow = -pow;
02223                     tt++;
02224                 }
02225                 if ( *tt != TNUMBER ) {
02226 /* INTERNAL_ERROR_EXCL_START */
02227                     MesPrint("!>Internal problems(2) in CodeFactors");
02228                     return(-1);
02229 /* INTERNAL_ERROR_EXCL_STOP */
02230                 }
02231                 tt++; x2 = 0; while ( *tt >= 0 ) { x2 = 100*x2 + *tt++; }
02232 /*
02233                 We have an object in startobject till t. The power is
02234                 power*pow*x2
02235 */
02236 dopowerd:
02237                 power = power*pow*x2;
02238                 if ( power < 0 ) { pow = -power; power = -1; }
02239                 else if ( power == 0 ) { t = tt; startobject = tt; continue; }
02240                 else { pow = power; power = 1; }
02241                 if ( pow > 1 ) {
02242                     subexp = GenerateFactors(pow,1);
02243                     if ( subexp < 0 ) { error = -1; subexp = 0; }
02244                 }
02245                 for ( i = 1; i <= Dollars[nexp].nfactors; i++ ) {
02246                     s1 = startobject; *out++ = TPLUS;
02247                     while ( s1 < t ) *out++ = *s1++;
02248                     *out++ = LBRACE; *out++ = TNUMBER; PUTNUMBER128(out,i)
02249                     *out++ = RBRACE;
02250                     *out++ = TMULTIPLY;
02251                     *out++ = TSYMBOL; *out++ = FACTORSYMBOL;
02252                     *out++ = TPOWER; *out++ = TNUMBER; PUTNUMBER100(out,powfactor)
02253                     powfactor += pow;
02254                     if ( pow > 1 ) {
02255                         *out++ = TSUBEXP; PUTNUMBER128(out,subexp)
02256                     }
02257                 }
02258                 startobject = 0; t = tt; continue;
02259             }
02260             else {
02261                 tt = t; pow = 1; x2 = 1; goto dopowerd;
02262             }
02263         }
02264         t++;
02265     }
02266     if ( last == 0 ) { last = 1; goto dolast; }
02267     *out = TENDOFIT;
02268     e->numfactors = powfactor-1;
```

```
02269     e->vflags |= ISFACTORIZED;
02270     subexp = CodeGenerator(outtokens);
02271     if ( subexp < 0 ) error = -1;
02272     if ( insubexpbuffers >= MAXSUBEXPRESSIONS ) {
02273         MesPrint("&More than %d subexpressions inside one expression",(WORD)MAXSUBEXPRESSIONS);
02274         Terminate(-1);
02275     }
02276     if ( subexpbuffers+insubexpbuffers >= topsubexpbuffers ) {
02277         DoubleBuffer((void **)((void *)(&subexpbuffers))
02278         ,(void **)((void *)(&topsubexpbuffers)),sizeof(SUBBUF),"subexpbuffers");
02279     }
02280     subexpbuffers[insubexpbuffers].subexpnum = subexp;
02281     subexpbuffers[insubexpbuffers].buffernum = AC.cbufnum;
02282     subexp = insubexpbuffers++;
02283     M_free(outtokens,"CodeFactors");
02284     s1 = tokens;
02285     *s1++ = TSUBEXP; PUTNUMBER128(s1,subexp); *s1++ = TENDOFIT;
02286     if ( error < 0 ) return(-1);
02287     else return(subexp);
02288 }
02289
02290 /*
02291         #] CodeFactors :
02292         #[ GenerateFactors :
02293
02294     Generates an expression of the type
02295       1+factor_+factor_^2+...+factor_^(n-1)
02296     (this is if inc=1)
02297     Returns the subexpression pointer of it.
02298 */
02299
02300 WORD GenerateFactors(WORD n,WORD inc)
02301 {
02302     int subexp;
02303     int i, error = 0;
02304     SBYTE *s;
02305     SBYTE *tokenbuffer = (SBYTE *)Malloc1(8*n*sizeof(SBYTE),"GenerateFactors");
02306     s = tokenbuffer;
02307     *s++ = TNUMBER; *s++ = 1;
02308     for ( i = inc; i < n*inc; i += inc ) {
02309         *s++ = TPLUS; *s++ = TSYMBOL; *s++ = FACTORSYMBOL;
02310         if ( i > 1 ) {
02311             *s++ = TPOWER; *s++ = TNUMBER;
02312             PUTNUMBER100(s,i)
02313         }
02314     }
02315     *s++ = TENDOFIT;
02316     subexp = CodeGenerator(tokenbuffer);
02317     if ( subexp < 0 ) error = -1;
02318     M_free(tokenbuffer,"GenerateFactors");
02319     if ( insubexpbuffers >= MAXSUBEXPRESSIONS ) {
02320         MesPrint("&More than %d subexpressions inside one expression",(WORD)MAXSUBEXPRESSIONS);
02321         Terminate(-1);
02322     }
02323     if ( subexpbuffers+insubexpbuffers >= topsubexpbuffers ) {
02324         DoubleBuffer((void **)((void *)(&subexpbuffers))
02325         ,(void **)((void *)(&topsubexpbuffers)),sizeof(SUBBUF),"subexpbuffers");
02326     }
02327     subexpbuffers[insubexpbuffers].subexpnum = subexp;
02328     subexpbuffers[insubexpbuffers].buffernum = AC.cbufnum;
02329     subexp = insubexpbuffers++;
02330     if ( error < 0 ) return(error);
02331     return(subexp);
02332 }
02333
02334 /*
02335         #] GenerateFactors :
02336     #] Compiler :
02337 */
```

4.13 compress.c File Reference

`#include "form3.h"`
Include dependency graph for compress.c:



4.13.1 Detailed Description

The routines for the use of gzip (de)compression of the information in the sort file.

Definition in file compress.c.

4.14 compress.c

Go to the documentation of this file.

```
00001
00007 /* #[ License : */
00008 /*
00009  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00010  *   When using this file you are requested to refer to the publication
00011  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00012  *   This is considered a matter of courtesy as the development was paid
00013  *   for by FOM the Dutch physics granting agency and we would like to
00014  *   be able to track its scientific use to convince FOM of its value
00015  *   for the community.
00016  *
00017  *   This file is part of FORM.
00018  *
00019  *   FORM is free software: you can redistribute it and/or modify it under the
00020  *   terms of the GNU General Public License as published by the Free Software
00021  *   Foundation, either version 3 of the License, or (at your option) any later
00022  *   version.
00023  *
00024  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00025  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00026  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00027  *   details.
00028  *
00029  *   You should have received a copy of the GNU General Public License along
00030  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00031  */
00032 /* #] License : */
00033
00034 #include "form3.h"
00035
00036 #ifdef WITHZLIB
00037 /*
00038 #define GZIPDEBUG
00039
00040     Low level routines for dealing with zlib during sorting and handling
00041     the scratch files. Work started 5-sep-2005.
00042     The .sor file handling was more or less completed on 8-sep-2005
00043     The handling of the scratch files still needs some thinking.
00044     Complications are:
00045         gzip compression should be per expression, not per buffer.
00046         No gzip compression for expressions with a bracket index.
00047         Separate decompression buffers for expressions in the rhs.
00048         This last one will involve more buffer work and organization.
00049         Information about compression should be stored for each expr.
```

```
00050             (including what method/program was used)
00051     Note: Be careful with compression. By far the most compact method
00052     is the original problem....
00053
00054     #[ Variables :
00055
00056     The following variables are to contain the intermediate buffers
00057     for the inflation of the various patches in the sort file.
00058     There can be up to MaxFpatches (FilePatches in the setup) and hence
00059     we can have that many streams simultaneously. We set this up once
00060     and only when needed.
00061         (in struct A.N or AB[threadnum].N)
00062     Bytef **AN.ziobufnum;
00063     Bytef *AN.ziobuffers;
00064 */
00065
00066 /*
00067     #] Variables :
00068     #[ SetupOutputGZIP :
00069
00070     Routine prepares a gzip output stream for the given file.
00071 */
00072
00073 int SetupOutputGZIP(FILEHANDLE *f)
00074 {
00075     GETIDENTITY
00076
00077     if ( AT.SS != AT.S0 ) return(0);
00078     if ( AR.NoCompress == 1 ) return(0);
00079     if ( AR.gzipCompress <= 0 ) return(0);
00080
00081     if ( f->ziobuffer == 0 ) {
00082 /*
00083         1: Allocate a struct for the gzip stream:
00084 */
00085         f->zsp = Malloc1(sizeof(z_stream),"output zstream");
00086 /*
00087         2: Allocate the output buffer.
00088 */
00089         f->ziobuffer =
00090             (Bytef *)Malloc1(f->ziosize*sizeof(char),"output zbuffer");
00091         if ( f->zsp == 0 || f->ziobuffer == 0 ) {
00092             MLOCK(ErrorMessageLock);
00093             MesCall("SetupOutputGZIP");
00094             MUNLOCK(ErrorMessageLock);
00095             Terminate(-1);
00096         }
00097     }
00098 /*
00099     3: Set the default fields:
00100 */
00101     f->zsp->zalloc = Z_NULL;
00102     f->zsp->zfree  = Z_NULL;
00103     f->zsp->opaque = Z_NULL;
00104 /*
00105     4: Set the output space:
00106 */
00107     f->zsp->next_out  = f->ziobuffer;
00108     f->zsp->avail_out = f->ziosize;
00109     f->zsp->total_out = 0;
00110 /*
00111     5: Set the input space:
00112 */
00113     f->zsp->next_in  = (Bytef *)(f->POobuffer);
00114     f->zsp->avail_in = (Bytef *)(f->POfill) - (Bytef *)(f->POobuffer);
00115     f->zsp->total_in = 0;
00116 /*
00117     6: Initiate the deflation
00118 */
00119     if ( deflateInit(f->zsp,AR.gzipCompress) != Z_OK ) {
00120 /* INTERNAL_ERROR_EXCL_START */
00121         MLOCK(ErrorMessageLock);
00122         MesPrint("!>Error from zlib: %s",f->zsp->msg);
00123         MesCall("SetupOutputGZIP");
00124         MUNLOCK(ErrorMessageLock);
00125         Terminate(-1);
00126 /* INTERNAL_ERROR_EXCL_STOP */
00127     }
00128
00129     return(0);
00130 }
00131
00132 /*
00133     #] SetupOutputGZIP :
00134     #[ PutOutputGZIP :
00135
00136     Routine is called when the POobuffer of f is full.
```

```
00137     The contents of it will be compressed and whenever the output buffer
00138     f->ziobuffer is full it will be written and the output buffer
00139     will be reset.
00140     Upon exit the input buffer will be cleared.
00141 */
00142
00143 int PutOutputGZIP(FILEHANDLE *f)
00144 {
00145     GETIDENTITY
00146     int zerror;
00147 /*
00148     First set the number of bytes in the input
00149 */
00150     f->zsp->next_in  = (Bytef *)(f->PObuffer);
00151     f->zsp->avail_in = (Bytef *)(f->POfill) - (Bytef *)(f->PObuffer);
00152     f->zsp->total_in = 0;
00153
00154     while ( ( zerror = deflate(f->zsp,Z_NO_FLUSH) ) == Z_OK ) {
00155         if ( f->zsp->avail_out == 0 ) {
00156 /*
00157             ziobuffer is full. Write the output.
00158 */
00159 #ifdef GZIPDEBUG
00160             {
00161                 char *s = (char *)((UBYTE *)(f->ziobuffer)+f->ziosize);
00162                 MLOCK(ErrorMessageLock);
00163                 MesPrint("%wWriting %l bytes at %10p: %d %d %d %d %d"
00164                 ,f->ziosize,&(f->POposition),s[-5],s[-4],s[-3],s[-2],s[-1]);
00165                 MUNLOCK(ErrorMessageLock);
00166             }
00167 #endif
00168 #ifdef ALLLOCK
00169             LOCK(f->pthreadslock);
00170 #endif
00171             if ( f == AR.hidefile ) {
00172                 LOCK(AS.inputslock);
00173             }
00174             SeekFile(f->handle,&(f->POposition),SEEK_SET);
00175             if ( WriteFile(f->handle,(UBYTE *)(f->ziobuffer),f->ziosize)
00176                     != f->ziosize ) {
00177                 if ( f == AR.hidefile ) {
00178                     UNLOCK(AS.inputslock);
00179                 }
00180 #ifdef ALLLOCK
00181                 UNLOCK(f->pthreadslock);
00182 #endif
00183                 MLOCK(ErrorMessageLock);
00184                 MesPrint("%wWrite error during compressed sort. Disk full?");
00185                 MUNLOCK(ErrorMessageLock);
00186                 return(-1);
00187             }
00188             if ( f == AR.hidefile ) {
00189                 UNLOCK(AS.inputslock);
00190             }
00191 #ifdef ALLLOCK
00192             UNLOCK(f->pthreadslock);
00193 #endif
00194             ADDPOS(f->filesize,f->ziosize);
00195             ADDPOS(f->POposition,f->ziosize);
00196 #ifdef WITHPTHREADS
00197             if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
00198                 if ( f->handle >= 0 ) SynchFile(f->handle);
00199             }
00200 #endif
00201 /*
00202             Reset the output
00203 */
00204             f->zsp->next_out  = f->ziobuffer;
00205             f->zsp->avail_out = f->ziosize;
00206             f->zsp->total_out = 0;
00207         }
00208         else if ( f->zsp->avail_in == 0 ) {
00209 /*
00210             We compressed everything and it sits in ziobuffer. Finish
00211 */
00212             return(0);
00213         }
00214         else {
00215 /* INTERNAL_ERROR_EXCL_START */
00216             MLOCK(ErrorMessageLock);
00217             MesPrint("!>%w avail_in = %d, avail_out = %d.",f->zsp->avail_in,f->zsp->avail_out);
00218             MUNLOCK(ErrorMessageLock);
00219             break;
00220 /* INTERNAL_ERROR_EXCL_STOP */
00221         }
00222     }
00223 /* INTERNAL_ERROR_EXCL_START */
```

```
00224     MLOCK(ErrorMessageLock);
00225     MesPrint("!>%wError in gzip handling of output. zerror = %d",zerror);
00226     MUNLOCK(ErrorMessageLock);
00227     return(-1);
00228 /* INTERNAL_ERROR_EXCL_STOP */
00229 }
00230
00231 /*
00232     #] PutOutputGZIP :
00233     #[ FlushOutputGZIP :
00234
00235     Routine is called to flush a stream. The compression of the input buffer
00236     will be completed and the contents of f->ziobuffer will be written.
00237     Both buffers will be cleared.
00238 */
00239
00240 int FlushOutputGZIP(FILEHANDLE *f)
00241 {
00242     GETIDENTITY
00243     int zerror;
00244 /*
00245     Set the proper parameters
00246 */
00247     f->zsp->next_in  = (Bytef *)(f->PObuffer);
00248     f->zsp->avail_in = (Bytef *)(f->POfill) - (Bytef *)(f->PObuffer);
00249     f->zsp->total_in = 0;
00250
00251     while ( ( zerror = deflate(f->zsp,Z_FINISH) ) == Z_OK ) {
00252         if ( f->zsp->avail_out == 0 ) {
00253 /*
00254             Write the output
00255 */
00256 #ifdef GZIPDEBUG
00257             MLOCK(ErrorMessageLock);
00258             MesPrint("%wWriting %l bytes at %10p",f->ziosize,&(f->POposition));
00259             MUNLOCK(ErrorMessageLock);
00260 #endif
00261 #ifdef ALLLOCK
00262             LOCK(f->pthreadslock);
00263 #endif
00264             if ( f == AR.hidefile ) {
00265                 UNLOCK(AS.inputslock);
00266             }
00267             SeekFile(f->handle,&(f->POposition),SEEK_SET);
00268             if ( WriteFile(f->handle,(UBYTE *)(f->ziobuffer),f->ziosize)
00269                     != f->ziosize ) {
00270                 if ( f == AR.hidefile ) {
00271                     UNLOCK(AS.inputslock);
00272                 }
00273 #ifdef ALLLOCK
00274                 UNLOCK(f->pthreadslock);
00275 #endif
00276                 MLOCK(ErrorMessageLock);
00277                 MesPrint("%wWrite error during compressed sort. Disk full?");
00278                 MUNLOCK(ErrorMessageLock);
00279                 return(-1);
00280             }
00281             if ( f == AR.hidefile ) {
00282                 UNLOCK(AS.inputslock);
00283             }
00284 #ifdef ALLLOCK
00285             UNLOCK(f->pthreadslock);
00286 #endif
00287             ADDPOS(f->filesize,f->ziosize);
00288             ADDPOS(f->POposition,f->ziosize);
00289 #ifdef WITHPTHREADS
00290             if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
00291                 if ( f->handle >= 0 ) SynchFile(f->handle);
00292             }
00293 #endif
00294 /*
00295             Reset the output
00296 */
00297             f->zsp->next_out  = f->ziobuffer;
00298             f->zsp->avail_out = f->ziosize;
00299             f->zsp->total_out = 0;
00300         }
00301     }
00302     if ( zerror == Z_STREAM_END ) {
00303 /*
00304         Write the output
00305 */
00306 #ifdef GZIPDEBUG
00307         MLOCK(ErrorMessageLock);
00308         MesPrint("%wWriting %l bytes at %10p",(LONG)(f->zsp->avail_out),&(f->POposition));
00309         MUNLOCK(ErrorMessageLock);
00310 #endif
```

```
00311 #ifdef ALLLOCK
00312         LOCK(f->pthreadslock);
00313 #endif
00314         if ( f == AR.hidefile ) {
00315             LOCK(AS.inputslock);
00316         }
00317         SeekFile(f->handle,&(f->POposition),SEEK_SET);
00318         if ( WriteFile(f->handle,(UBYTE *)(f->ziobuffer),f->zsp->total_out)
00319                         != (LONG)(f->zsp->total_out) ) {
00320             if ( f == AR.hidefile ) {
00321                 UNLOCK(AS.inputslock);
00322             }
00323 #ifdef ALLLOCK
00324             UNLOCK(f->pthreadslock);
00325 #endif
00326             MLOCK(ErrorMessageLock);
00327             MesPrint("%wWrite error during compressed sort. Disk full?");
00328             MUNLOCK(ErrorMessageLock);
00329             return(-1);
00330         }
00331         if ( f == AR.hidefile ) {
00332             LOCK(AS.inputslock);
00333         }
00334 #ifdef ALLLOCK
00335         UNLOCK(f->pthreadslock);
00336 #endif
00337         ADDPOS(f->filesize,f->zsp->total_out);
00338         ADDPOS(f->POposition,f->zsp->total_out);
00339 #ifdef WITHPTHREADS
00340         if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
00341             if ( f->handle >= 0 ) SynchFile(f->handle);
00342         }
00343 #endif
00344 #ifdef GZIPDEBUG
00345         MLOCK(ErrorMessageLock);
00346         {  char *s = f->ziobuffer+f->zsp->total_out;
00347             MesPrint("%w   Last bytes written: %d %d %d %d %d",s[-5],s[-4],s[-3],s[-2],s[-1]);
00348         }
00349         MesPrint("%w     Perceived position in FlushOutputGZIP is %10p",&(f->POposition));
00350         MUNLOCK(ErrorMessageLock);
00351 #endif
00352 /*
00353             Reset the output
00354 */
00355         f->zsp->next_out  = f->ziobuffer;
00356         f->zsp->avail_out = f->ziosize;
00357         f->zsp->total_out = 0;
00358         if ( ( zerror = deflateEnd(f->zsp) ) == Z_OK ) return(0);
00359 /* INTERNAL_ERROR_EXCL_START */
00360         MLOCK(ErrorMessageLock);
00361         if ( f->zsp->msg ) {
00362             MesPrint("!>%wError in finishing gzip handling of output: %s",f->zsp->msg);
00363         }
00364         else {
00365             MesPrint("!>%wError in finishing gzip handling of output.");
00366         }
00367         MUNLOCK(ErrorMessageLock);
00368     }
00369     else {
00370         MLOCK(ErrorMessageLock);
00371         MesPrint("!>%wError in gzip handling of output.");
00372         MUNLOCK(ErrorMessageLock);
00373     }
00374     return(-1);
00375 /* INTERNAL_ERROR_EXCL_STOP */
00376 }
00377
00378 /*
00379     #] FlushOutputGZIP :
00380     #[ SetupAllInputGZIP :
00381
00382     Routine prepares all gzip input streams for a merge.
00383
00384     Problem (29-may-2008): If we never use GZIP compression, this routine
00385     will still allocate the array space. This is an enormous amount!
00386     It places an effective restriction on the value of SortIOsize
00387 */
00388
00389 int SetupAllInputGZIP(SORTING *S)
00390 {
00391     GETIDENTITY
00392     int i, NumberOpened = 0;
00393     z_streamp zsp;
00394 /*
00395     This code was added 29-may-2008 by JV to prevent further processing if
00396     there is no compression at all (usually).
00397 */
```

```
00398     for ( i = 0; i < S->inNum; i++ ) {
00399         if ( S->fpincompressed[i] ) break;
00400     }
00401     if ( i >= S->inNum ) return(0);
00402 
00403     if ( S->zsparray == 0 ) {
00404         S->zsparray = (z_streamp)Malloc1(sizeof(z_stream)*S->MaxFpatches,"input zstreams");
00405         if ( S->zsparray == 0 ) {
00406             MLOCK(ErrorMessageLock);
00407             MesCall("SetupAllInputGZIP");
00408             MUNLOCK(ErrorMessageLock);
00409             Terminate(-1);
00410         }
00411 /*
00412         We add 128 bytes in the hope that if it can happen that it goes
00413         outside the buffer during decompression, it does not do damage.
00414 */
00415         AN.ziobuffers = (Bytef *)Malloc1(S->MaxFpatches*(S->file.ziosize+128)*sizeof(Bytef),"input raw
      buffers");
00416 /*
00417         This seems to be one of the really stupid errors:
00418         We allocate way too much space. Way way way too much.
00419         AN.ziobufnum  = (Bytef **)Malloc1(S->MaxFpatches*S->file.ziosize*sizeof(Bytef *),"input raw
      pointers");
00420 */
00421         AN.ziobufnum  = (Bytef **)Malloc1(S->MaxFpatches*sizeof(Bytef *),"input raw pointers");
00422         if ( AN.ziobuffers == 0 || AN.ziobufnum == 0 ) {
00423             MLOCK(ErrorMessageLock);
00424             MesCall("SetupAllInputGZIP");
00425             MUNLOCK(ErrorMessageLock);
00426             Terminate(-1);
00427         }
00428         for ( i  = 0 ; i < S->MaxFpatches; i++ ) {
00429             AN.ziobufnum[i] = AN.ziobuffers + i * (S->file.ziosize+128);
00430         }
00431     }
00432     for ( i = 0; i < S->inNum; i++ ) {
00433 #ifdef GZIPDEBUG
00434         MLOCK(ErrorMessageLock);
00435         MesPrint("%wPreparing z-stream %d with compression %d",i,S->fpincompressed[i]);
00436         MUNLOCK(ErrorMessageLock);
00437 #endif
00438         if ( S->fpincompressed[i] ) {
00439             zsp = &(S->zsparray[i]);
00440 /*
00441             1: Set the default fields:
00442 */
00443             zsp->zalloc = Z_NULL;
00444             zsp->zfree  = Z_NULL;
00445             zsp->opaque = Z_NULL;
00446 /*
00447             2: Set the output space:
00448 */
00449             zsp->next_out  = Z_NULL;
00450             zsp->avail_out = 0;
00451             zsp->total_out = 0;
00452 /*
00453             3: Set the input space temporarily:
00454 */
00455             zsp->next_in  = Z_NULL;
00456             zsp->avail_in = 0;
00457             zsp->total_in = 0;
00458 /*
00459             4: Initiate the inflation
00460 */
00461             if ( inflateInit(zsp) != Z_OK ) {
00462 /* INTERNAL_ERROR_EXCL_START */
00463                 MLOCK(ErrorMessageLock);
00464                 if ( zsp->msg ) MesPrint("!>%wError from inflateInit: %s",zsp->msg);
00465                 else            MesPrint("!>%wError from inflateInit");
00466                 MesCall("SetupAllInputGZIP");
00467                 MUNLOCK(ErrorMessageLock);
00468                 Terminate(-1);
00469 /* INTERNAL_ERROR_EXCL_STOP */
00470             }
00471             NumberOpened++;
00472         }
00473     }
00474     return(NumberOpened);
00475 }
00476 
00477 /*
00478     #] SetupAllInputGZIP :
00479     #[ FillInputGZIP :
00480 
00481     Routine is called when we need new input in the specified buffer.
00482     This buffer is used for the output and we keep reading and uncompressing
```

```
00483     input till either this buffer is full or the input stream is finished.
00484     The return value is the number of bytes in the buffer.
00485 */
00486
00487 LONG FillInputGZIP(FILEHANDLE *f, POSITION *position, UBYTE *buffer, LONG buffersize, int numstream)
00488 {
00489     GETIDENTITY
00490     int zerror;
00491     LONG readsize, toread = 0;
00492     SORTING *S = AT.SS;
00493     z_streamp zsp;
00494     POSITION pos;
00495     if ( S->fpincompressed[numstream] ) {
00496         zsp = &(S->zsparray[numstream]);
00497         zsp->next_out = (Bytef *)buffer;
00498         zsp->avail_out = buffersize;
00499         zsp->total_out = 0;
00500         if ( zsp->avail_in == 0 ) {
00501 /*
00502             First loading of the input
00503 */
00504             if ( ISGEPOSINC(S->fPatchesStop[numstream],*position,f->ziosize) ) {
00505                 toread = f->ziosize;
00506             }
00507             else {
00508                 DIFPOS(pos,S->fPatchesStop[numstream],*position);
00509                 toread = (LONG)(BASEPOSITION(pos));
00510             }
00511             if ( toread > 0 ) {
00512 #ifdef GZIPDEBUG
00513                 MLOCK(ErrorMessageLock);
00514                 MesPrint("%w-+Reading %l bytes in stream %d at position %10p; stop at
    %10p",toread,numstream,position,&(S->fPatchesStop[numstream]));
00515                 MUNLOCK(ErrorMessageLock);
00516 #endif
00517 #ifdef ALLLOCK
00518                 LOCK(f->pthreadslock);
00519 #endif
00520                 SeekFile(f->handle,position,SEEK_SET);
00521                 readsize = ReadFile(f->handle,(UBYTE *)(AN.ziobufnum[numstream]),toread);
00522                 SeekFile(f->handle,position,SEEK_CUR);
00523 #ifdef ALLLOCK
00524                 UNLOCK(f->pthreadslock);
00525 #endif
00526 #ifdef GZIPDEBUG
00527                 MLOCK(ErrorMessageLock);
00528                 {  char *s = AN.ziobufnum[numstream]+readsize;
00529                     MesPrint("%w read: %l +Last bytes read: %d %d %d %d %d in %s, newpos =
    %10p",readsize,s[-5],s[-4],s[-3],s[-2],s[-1],f->name,position);
00530                 }
00531                 MUNLOCK(ErrorMessageLock);
00532 #endif
00533                 if ( readsize == 0 ) {
00534                     zsp->next_in  = AN.ziobufnum[numstream];
00535                     zsp->avail_in = f->ziosize;
00536                     zsp->total_in = 0;
00537                     return(zsp->total_out);
00538                 }
00539                 if ( readsize < 0 ) {
00540 /* INTERNAL_ERROR_EXCL_START */
00541                     MLOCK(ErrorMessageLock);
00542                     MesPrint("!>%wFillInputGZIP: Read error during compressed sort.");
00543                     MUNLOCK(ErrorMessageLock);
00544                     return(-1);
00545 /* INTERNAL_ERROR_EXCL_STOP */
00546                 }
00547                 ADDPOS(f->filesize,readsize);
00548                 ADDPOS(f->POposition,readsize);
00549 /*
00550                 Set the input
00551 */
00552                 zsp->next_in  = AN.ziobufnum[numstream];
00553                 zsp->avail_in = readsize;
00554                 zsp->total_in = 0;
00555             }
00556         }
00557         if ( toread > 0 || zsp->avail_in ) {
00558           while ( ( zerror = inflate(zsp,Z_NO_FLUSH) ) == Z_OK ) {
00559             if ( zsp->avail_out == 0 ) {
00560 /*
00561                 Finish
00562 */
00563                 return((LONG)(zsp->total_out));
00564             }
00565             if ( zsp->avail_in == 0 ) {
00566
00567                 if ( ISEQUALPOS(S->fPatchesStop[numstream],*position) ) {
```

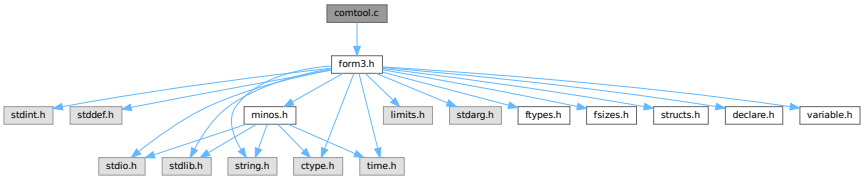
```
00568 /*
00569                     We finished this stream. Try to terminate.
00570 */
00571                     zerror = Z_STREAM_END;
00572                     break;
00573 /*
00574                     if ( ( zerror = inflate(zsp,Z_SYNC_FLUSH) ) == Z_OK ) {
00575                         return((LONG)(zsp->total_out));
00576                     }
00577                     else
00578                         break;
00579 */
00580 /*
00581 #ifdef GZIPDEBUG
00582                     MLOCK(ErrorMessageLock);
00583                     MesPrint("%wClosing stream %d",numstream);
00584 #endif
00585                     readsize = zsp->total_out;
00586 #ifdef GZIPDEBUG
00587                     if ( readsize > 0 ) {
00588                         WORD *s = (WORD *)(buffer+zsp->total_out);
00589                         MesPrint("%w   Last words: %d %d %d %d %d",s[-5],s[-4],s[-3],s[-2],s[-1]);
00590                     }
00591                     else {
00592                         MesPrint("%w No words");
00593                     }
00594                     MUNLOCK(ErrorMessageLock);
00595 #endif
00596                     if ( ( zerror = inflateEnd(zsp) ) == Z_OK ) return(readsize);
00597                     break;
00598 */
00599                 }
00600 /*
00601                 Read more input
00602 */
00603 #ifdef GZIPDEBUG
00604                 if ( numstream == 0 ) {
00605                     MLOCK(ErrorMessageLock);
00606                     MesPrint("%wWant to read in stream 0 at position %10p",position);
00607                     MUNLOCK(ErrorMessageLock);
00608                 }
00609 #endif
00610                 if ( ISGEPOSINC(S->fPatchesStop[numstream],*position,f->ziosize) ) {
00611                     toread = f->ziosize;
00612                 }
00613                 else {
00614                     DIFPOS(pos,S->fPatchesStop[numstream],*position);
00615                     toread = (LONG)(BASEPOSITION(pos));
00616                 }
00617 #ifdef GZIPDEBUG
00618                 MLOCK(ErrorMessageLock);
00619                 MesPrint("%w--Reading %l bytes in stream %d at position
    %10p",toread,numstream,position);
00620                 MUNLOCK(ErrorMessageLock);
00621 #endif
00622 #ifdef ALLLOCK
00623                 LOCK(f->pthreadslock);
00624 #endif
00625                 SeekFile(f->handle,position,SEEK_SET);
00626                 readsize = ReadFile(f->handle,(UBYTE *)(AN.ziobufnum[numstream]),toread);
00627                 SeekFile(f->handle,position,SEEK_CUR);
00628 #ifdef ALLLOCK
00629                 UNLOCK(f->pthreadslock);
00630 #endif
00631 #ifdef GZIPDEBUG
00632                 MLOCK(ErrorMessageLock);
00633                 {  char *s = AN.ziobufnum[numstream]+readsize;
00634                     MesPrint("%w   Last bytes read: %d %d %d %d %d",s[-5],s[-4],s[-3],s[-2],s[-1]);
00635                 }
00636                 MUNLOCK(ErrorMessageLock);
00637 #endif
00638                 if ( readsize == 0 ) {
00639                     zsp->next_in  = AN.ziobufnum[numstream];
00640                     zsp->avail_in = f->ziosize;
00641                     zsp->total_in = 0;
00642                     return(zsp->total_out);
00643                 }
00644                 if ( readsize < 0 ) {
00645 /* INTERNAL_ERROR_EXCL_START */
00646                     MLOCK(ErrorMessageLock);
00647                     MesPrint("!>%wFillInputGZIP: Read error during compressed sort.");
00648                     MUNLOCK(ErrorMessageLock);
00649                     return(-1);
00650 /* INTERNAL_ERROR_EXCL_STOP */
00651                 }
00652                 ADDPOS(f->filesize,readsize);
00653                 ADDPOS(f->POposition,readsize);
```

```
00654 /*
00655                 Reset the input
00656 */
00657                 zsp->next_in  = AN.ziobufnum[numstream];
00658                 zsp->avail_in = readsize;
00659                 zsp->total_in = 0;
00660             }
00661             else {
00662                 break;
00663             }
00664           }
00665         }
00666         else {
00667             zerror = Z_STREAM_END;
00668             zsp->total_out = 0;
00669         }
00670 #ifdef GZIPDEBUG
00671             MLOCK(ErrorMessageLock);
00672             MesPrint("%w zerror = %d in stream %d. At position %10p",zerror,numstream,position);
00673             MUNLOCK(ErrorMessageLock);
00674 #endif
00675         if ( zerror == Z_STREAM_END ) {
00676 /*
00677             Reset the input
00678 */
00679             zsp->next_in  = Z_NULL;
00680             zsp->avail_in = 0;
00681             zsp->total_in = 0;
00682 /*
00683             Make the final call and finish
00684 */
00685 #ifdef GZIPDEBUG
00686             MLOCK(ErrorMessageLock);
00687             MesPrint("%wClosing stream %d",numstream);
00688 #endif
00689             readsize = zsp->total_out;
00690 #ifdef GZIPDEBUG
00691             if ( readsize > 0 ) {
00692                 WORD *s = (WORD *)(buffer+zsp->total_out);
00693                 MesPrint("%w  -Last words: %d %d %d %d %d",s[-5],s[-4],s[-3],s[-2],s[-1]);
00694             }
00695             else {
00696                 MesPrint("%w No words");
00697             }
00698             MUNLOCK(ErrorMessageLock);
00699 #endif
00700             if ( zsp->zalloc != Z_NULL ) {
00701                 zerror = inflateEnd(zsp);
00702                 zsp->zalloc = Z_NULL;
00703             }
00704             if ( zerror == Z_OK || zerror == Z_STREAM_END ) return(readsize);
00705         }
00706
00707 /* INTERNAL_ERROR_EXCL_START */
00708         MLOCK(ErrorMessageLock);
00709         MesPrint("!>%wFillInputGZIP: Error in gzip handling of input. zerror = %d",zerror);
00710         MUNLOCK(ErrorMessageLock);
00711         return(-1);
00712 /* INTERNAL_ERROR_EXCL_STOP */
00713     }
00714     else {
00715 #ifdef GZIPDEBUG
00716         MLOCK(ErrorMessageLock);
00717         MesPrint("%w++Reading %l bytes at position %10p",buffersize,position);
00718         MUNLOCK(ErrorMessageLock);
00719 #endif
00720 #ifdef ALLLOCK
00721         LOCK(f->pthreadslock);
00722 #endif
00723         SeekFile(f->handle,position,SEEK_SET);
00724         readsize = ReadFile(f->handle,buffer,buffersize);
00725         SeekFile(f->handle,position,SEEK_CUR);
00726 #ifdef ALLLOCK
00727         UNLOCK(f->pthreadslock);
00728 #endif
00729         if ( readsize < 0 ) {
00730 /* INTERNAL_ERROR_EXCL_START */
00731             MLOCK(ErrorMessageLock);
00732             MesPrint("!>%wFillInputGZIP: Read error during uncompressed sort.");
00733             MesPrint("%w++Reading %l bytes at position %10p",buffersize,position);
00734             MUNLOCK(ErrorMessageLock);
00735 /* INTERNAL_ERROR_EXCL_STOP */
00736         }
00737         return(readsize);
00738     }
00739 }
00740
```

```
00741 /*
00742     #] FillInputGZIP :
00743     #[ ClearSortGZIP :
00744 */
00745
00746 void ClearSortGZIP(FILEHANDLE *f)
00747 {
00748     if ( f->ziobuffer ) {
00749         M_free(f->ziobuffer,"output zbuffer");
00750         M_free(f->zsp,"output zstream");
00751         f->ziobuffer = 0;
00752     }
00753 }
00754
00755 /*
00756     #] ClearSortGZIP :
00757 */
00758 #endif
```

4.15 comtool.c File Reference

`#include "form3.h"`
Include dependency graph for comtool.c:



Functions

- int inicbufs (void)
- void finishcbuf (WORD num)
- void clearcbuf (WORD num)
- WORD * DoubleCbuffer (int num, WORD *w, int par)
- WORD * AddLHS (int num)
- WORD * AddRHS (int num, int type)
- int AddNtoL (int n, WORD *array)
- int AddNtoC (int bufnum, int n, WORD *array, int par)
- int InsTree (int bufnum, int h)
- int FindTree (int bufnum, WORD *subexpr)
- void RedoTree (CBUF *C, int size)
- void ClearTree (int i)
- int IniFbuffer (WORD bufnum)
- LONG numcommute (WORD *terms, LONG *numterms)

4.15.1 Detailed Description

Utility routines for the compiler.

Definition in file comtool.c.

4.15.2 Function Documentation

4.15.2.1 inicbufs()

```
int inicbufs (
            void  )
```

Creates a new compiler buffer and returns its ID number.

Returns

The ID number for the new compiler buffer.

Definition at line 47 of file comtool.c.

References CbUf::boomlijst, CbUf::Buffer, CbUf::BufferSize, CbUf::CanCommu, CbUf::dimension, CbUf::lhs, CbUf::numdum, CbUf::NumTerms, CbUf::Pointer, CbUf::rhs, and CbUf::Top.

Referenced by AddRHS(), and StartVariables().

4.15.2.2 finishcbuf()

```
void finishcbuf (
            WORD num )
```

Frees a compiler buffer.

Parameters

num	The ID number for the buffer to be freed.

Definition at line 89 of file comtool.c.

References CbUf::boomlijst, CbUf::Buffer, CbUf::BufferSize, CbUf::CanCommu, CbUf::lhs, CbUf::NumTerms, CbUf::Pointer, CbUf::rhs, and CbUf::Top.

Referenced by PF_BroadcastCBuf().

4.15.2.3 clearcbuf()

```
void clearcbuf (
            WORD num )
```

Clears contents in a compiler buffer.

Parameters

num	The ID number for the buffer to be cleared.

Definition at line 116 of file comtool.c.

References CbUf::boomlijst, CbUf::Buffer, CbUf::Pointer, and CbUf::rhs.

4.15.2.4 DoubleCbuffer()

```
WORD * DoubleCbuffer (
            int num,
            WORD * w,
            int par )
```

Doubles a compiler buffer.

Parameters

num	The ID number for the buffer to be doubled.
w	The pointer to the end (exclusive) of the current buffer. The contents in the range of [cbuf[num].Buffer,w) will be kept.

Definition at line 143 of file comtool.c.

References CbUf::Buffer, CbUf::BufferSize, CbUf::lhs, CbUf::Pointer, CbUf::rhs, and CbUf::Top.

Referenced by AddNtoC(), and AddNtoL().

4.15.2.5 AddLHS()

```
WORD * AddLHS (
            int num )
```

Adds an LHS to a compiler buffer and returns the pointer to a buffer for the new LHS.

Parameters

num	The ID number for the buffer to get another LHS.

Definition at line 184 of file comtool.c.

References CbUf::lhs, and CbUf::Pointer.

Referenced by AddNtoL().

4.15.2.6 AddRHS()

```
WORD * AddRHS (
            int num,
            int type )
```

Adds an RHS to a compiler buffer and returns the pointer to a buffer for the new RHS.

Parameters

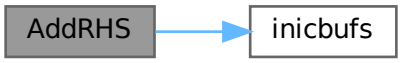
num	The ID number for the buffer to get another RHS.
type	If 0, the subexpression tree will be reallocated.

Definition at line 210 of file comtool.c.

References TaBIEs::buffers, TaBIEs::buffersfill, TaBIEs::bufferssize, TaBIEs::bufnum, CbUf::CanCommu, CbUf::dimension, inicbufs(), CbUf::numdum, CbUf::NumTerms, CbUf::Pointer, and CbUf::rhs.

Referenced by InsertArg(), StartVariables(), and TestMatch().

Here is the call graph for this function:



4.15.2.7 AddNtoL()

```
int AddNtoL (
             int n,
             WORD * array )
```

Adds an LHS with the given data to the current compiler buffer.

Parameters

n	The length of the data.
array	The data to be added.

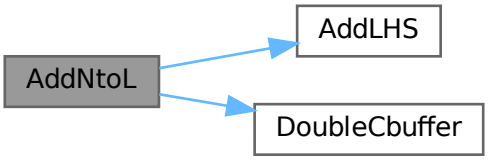
Returns

0 if succeeds.

Definition at line 284 of file comtool.c.

References AddLHS(), DoubleCbuffer(), CbUf::Pointer, and CbUf::Top.

Here is the call graph for this function:



4.15.2.8 AddNtoC()

```
int AddNtoC (
              int bufnum,
              int n,
              WORD * array,
              int par )
```

Adds the given data to the last LHS/RHS in a compiler buffer.

Parameters

bufnum	The ID number for the buffer where the data will be added.
n	The length of the data.
array	The data to be added.

Returns

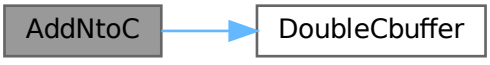
0 if succeeds.

Definition at line 313 of file comtool.c.

References DoubleCbuffer(), CbUf::Pointer, and CbUf::Top.

Referenced by InsertArg(), and StartVariables().

Here is the call graph for this function:



4.15.2.9 InsTree()

```
int InsTree (
            int bufnum,
            int h )
```

Definition at line 351 of file comtool.c.

4.15.2.10 FindTree()

```
int FindTree (
            int bufnum,
            WORD * subexpr )
```

Definition at line 531 of file comtool.c.

4.15.2.11 RedoTree()

```
void RedoTree (
            CBUF * C,
            int size )
```

Definition at line 567 of file comtool.c.

4.15.2.12 ClearTree()

```
void ClearTree (
            int i )
```

Definition at line 586 of file comtool.c.

4.15.2.13 IniFbuffer()

```
int IniFbuffer (
            WORD bufnum )
```

Initialize a factorization cache buffer. We set the size of the rhs and boomlijst buffers immediately to their final values.

Definition at line 612 of file comtool.c.

References tree::blnce, CbUf::boomlijst, CbUf::CanCommu, CbUf::dimension, tree::left, CbUf::numdum, CbUf::NumTerms, tree::parent, CbUf::rhs, tree::right, tree::usage, and tree::value.

4.15.2.14 numcommute()

```
LONG numcommute (
            WORD * terms,
            LONG * numterms )
```

Definition at line 657 of file comtool.c.

4.16 comtool.c

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
00026  *
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
00031 /*
00032     #[ Includes :
00033 */
00034
00035 #include "form3.h"
00036
00037 /*
00038     #] Includes :
00039     #[ inicbufs :
00040 */
00041
00047 int inicbufs(void)
00048 {
00049     int i, num = AC.cbufList.num;
00050     CBUF *C = cbuf;
00051     for ( i = 0; i < num; i++, C++ ) {
00052         if ( C->Buffer == 0 ) break;
00053     }
00054     if ( i >= num ) C = (CBUF *)FromList(&AC.cbufList);
00055     else num = i;
00056     C->BufferSize = 2000;
00057     C->Buffer = (WORD *)Malloc1(C->BufferSize*sizeof(WORD),"compiler buffer-1");
00058     C->Pointer = C->Buffer;
00059     C->Top = C->Buffer + C->BufferSize;
00060     C->maxlhs = 10;
00061     C->lhs = (WORD **)Malloc1(C->maxlhs*sizeof(WORD *),"compiler buffer-2");
00062     C->numlhs = 0;
00063     C->mnumlhs = 0;
00064     C->maxrhs = 25;
00065     C->rhs = (WORD **)Malloc1(C->maxrhs*(sizeof(WORD *)+2*sizeof(LONG)+2*sizeof(WORD)),"compiler
      buffer-3");
00066     C->CanCommu = (LONG *)(C->rhs+C->maxrhs);
00067     C->NumTerms = C->CanCommu+C->maxrhs;
00068     C->numdum = (WORD *)(C->NumTerms+C->maxrhs);
00069     C->dimension = C->numdum + C->maxrhs;
00070     C->numrhs = 0;
00071     C->mnumrhs = 0;
00072     C->rhs[0] = C->rhs[1] = C->Pointer;
00073     C->boomlijst = 0;
00074     RedoTree(C,C->maxrhs);
00075     ClearTree(num);
00076     return(num);
00077 }
00078
00079 /*
00080     #] inicbufs :
00081     #[ finishcbuf :
00082 */
00083
00089 void finishcbuf(WORD num)
00090 {
00091     CBUF *C = cbuf+num;
00092     if ( C->Buffer ) M_free(C->Buffer,"compiler buffer-1");
00093     if ( C->rhs ) M_free(C->rhs,"compiler buffer-3");
00094     if ( C->lhs ) M_free(C->lhs,"compiler buffer-2");
```

```
00095     if ( C->boomlijst ) M_free(C->boomlijst,"boomlijst");
00096     C->Top = C->Pointer = C->Buffer = 0;
00097     C->rhs = C->lhs = 0;
00098     C->CanCommu = 0;
00099     C->NumTerms = 0;
00100     C->BufferSize = 0;
00101     C->boomlijst = 0;
00102     C->numlhs = C->numrhs = C->maxlhs = C->maxrhs = C->mnumlhs =
00103     C->mnumrhs = C->numtree = C->rootnum = C->MaxTreeSize = 0;
00104 }
00105
00106 /*
00107     #] finishcbuf :
00108     #[ clearcbuf :
00109 */
00110
00116 void clearcbuf(WORD num)
00117 {
00118     CBUF *C = cbuf+num;
00119     if ( C->boomlijst ) M_free(C->boomlijst,"boomlijst");
00120     C->Pointer = C->Buffer;
00121     C->numrhs = C->numlhs = 0;
00122     C->mnumlhs = 0;
00123     C->boomlijst = 0;
00124     C->mnumrhs = 0;
00125     C->rhs[0] = C->rhs[1] = C->Pointer;
00126     C->numtree = C->rootnum = C->MaxTreeSize = 0;
00127     RedoTree(C,C->maxrhs);
00128     ClearTree(num);
00129 }
00130
00131 /*
00132     #] clearcbuf :
00133     #[ DoubleCbuffer :
00134 */
00135
00143 WORD *DoubleCbuffer(int num, WORD *w,int par)
00144 {
00145     CBUF *C = cbuf + num;
00146     LONG newsize = C->BufferSize*2;
00147     WORD *newbuffer = (WORD *)Malloc1(newsize*sizeof(WORD),"compiler buffer-4");
00148     WORD *w1, *w2;
00149     LONG offset, j, i;
00150     DUMMYUSE(par)
00151
00152     w1 = C->Buffer; w2 = newbuffer;
00153     i = w - w1;
00154     j = i & 7;
00155     while ( --j >= 0 ) *w2++ = *w1++;
00156     i >>= 3;
00157     while ( --i >= 0 ) {
00158         *w2++ = *w1++; *w2++ = *w1++; *w2++ = *w1++; *w2++ = *w1++;
00159         *w2++ = *w1++; *w2++ = *w1++; *w2++ = *w1++; *w2++ = *w1++;
00160     }
00161     offset = newbuffer - C->Buffer;
00162     for ( i = 0; i <= C->numlhs; i++ ) C->lhs[i] += offset;
00163     for ( i = 1; i <= C->numrhs; i++ ) C->rhs[i] += offset;
00164     w1 = C->Buffer;
00165     C->Pointer += offset;
00166     C->Top = newbuffer + newsize;
00167     C->BufferSize = newsize;
00168     C->Buffer = newbuffer;
00169     M_free(w1,"DoubleCbuffer");
00170     return(w2);
00171 }
00172
00173 /*
00174     #] DoubleCbuffer :
00175     #[ AddLHS :
00176 */
00177
00184 WORD *AddLHS(int num)
00185 {
00186     CBUF *C = cbuf + num;
00187     C->numlhs++;
00188     if ( C->numlhs >= (C->maxlhs-2) ) {
00189         WORD ***ppp = &(C->lhs); /* to avoid compiler warning */
00190         if ( DoubleList((void ***)ppp,&(C->maxlhs),sizeof(WORD *),
00191         "statement lists") ) Terminate(-1);
00192     }
00193     C->lhs[C->numlhs] = C->Pointer;
00194     C->lhs[C->numlhs+1] = 0;
00195     return(C->Pointer);
00196 }
00197
00198 /*
00199     #] AddLHS :
```

```
00200     #[ AddRHS :
00201 */
00202
00210 WORD *AddRHS(int num, int type)
00211 {
00212     LONG fullsize, *lold, newsize;
00213     int i;
00214     WORD **old, *wold;
00215     CBUF *C;
00216 restart:;
00217     C = cbuf + num;
00218     if ( C->numrhs >= (C->maxrhs-2) ) {
00219         if ( C->maxrhs == 0 ) newsize = 100;
00220         else                  newsize = C->maxrhs * 2;
00221         if ( newsize > MAXCOMBUFRHS ) newsize = MAXCOMBUFRHS;
00222         if ( newsize == C->maxrhs ) {
00223             if ( AC.tablefilling ) {
00224                 TABLES T = functions[AC.tablefilling].tabl;
00225 /*
00226                 We add a compiler buffer, change a few settings and continue.
00227 */
00228                 if ( T->buffersfill >= T->bufferssize ) {
00229                     int new1 = 2*T->bufferssize;
00230                     WORD *nbufs = (WORD *)Malloc1(new1*sizeof(WORD),"Table compile buffers");
00231                     for ( i = 0; i < T->buffersfill; i++ )
00232                             nbufs[i] = T->buffers[i];
00233                     for ( ; i < new1; i++ ) nbufs[i] = 0;
00234                     M_free(T->buffers,"Table compile buffers");
00235                     T->buffers = nbufs;
00236                     T->bufferssize = new1;
00237                 }
00238                 T->buffers[T->buffersfill++] = T->bufnum = inicbufs();
00239                 AC.cbufnum = num = T->bufnum;
00240                 goto restart;
00241             }
00242             else {
00243                 MesPrint("@Compiler buffer overflow. Try to make modules smaller");
00244                 Terminate(-1);
00245             }
00246         }
00247         old = C->rhs;
00248         fullsize = newsize * (sizeof(WORD *) + 2*sizeof(LONG) + 2*sizeof(WORD));
00249         C->rhs = (WORD **)Malloc1(fullsize,"subexpression lists");
00250         for ( i = 0; i < C->maxrhs; i++ ) C->rhs[i] = old[i];
00251         lold = C->CanCommu; C->CanCommu = (LONG *)(C->rhs+newsize);
00252         for ( i = 0; i < C->maxrhs; i++ ) C->CanCommu[i] = lold[i];
00253         lold = C->NumTerms; C->NumTerms = (LONG *)(C->rhs+2*newsize);
00254         for ( i = 0; i < C->maxrhs; i++ ) C->NumTerms[i] = lold[i];
00255         wold = C->numdum; C->numdum = (WORD *)(C->NumTerms+newsize);
00256         for ( i = 0; i < C->maxrhs; i++ ) C->numdum[i] = wold[i];
00257         wold = C->dimension; C->dimension = (WORD *)(C->numdum+newsize);
00258         for ( i = 0; i < C->maxrhs; i++ ) C->dimension[i] = wold[i];
00259         if ( old ) M_free(old,"subexpression lists");
00260         C->maxrhs = newsize;
00261         if ( type == 0 ) RedoTree(C,C->maxrhs);
00262     }
00263     C->numrhs++;
00264     C->CanCommu[C->numrhs] = 0;
00265     C->NumTerms[C->numrhs] = 0;
00266     C->numdum[C->numrhs] = 0;
00267     C->dimension[C->numrhs] = 0;
00268     C->rhs[C->numrhs] = C->Pointer;
00269     return(C->Pointer);
00270 }
00271
00272 /*
00273     #] AddRHS :
00274     #[ AddNtoL :
00275 */
00276
00284 int AddNtoL(int n, WORD *array)
00285 {
00286     int i;
00287     CBUF *C = cbuf+AC.cbufnum;
00288 #ifdef COMPBUFDEBUG
00289     MesPrint("LH: %a",n,array);
00290 #endif
00291     AddLHS(AC.cbufnum);
00292     while ( C->Pointer+n >= C->Top ) DoubleCbuffer(AC.cbufnum,C->Pointer,1);
00293     for ( i = 0; i < n; i++ ) *(C->Pointer)++ = *array++;
00294     return(0);
00295 }
00296
00297 /*
00298     #] AddNtoL :
00299     #[ AddNtoC :
00300
```

```
00301     Commentary: added the bufnum on 14-sep-2010 to make the whole a bit
00302     more flexible (JV). Still to do with AddNtoL.
00303 */
00304
00313 int AddNtoC(int bufnum, int n, WORD *array,int par)
00314 {
00315     int i;
00316     WORD *w;
00317     CBUF *C = cbuf+bufnum;
00318 #ifdef COMPBUFDEBUG
00319     MesPrint("RH: %a",n,array);
00320 #endif
00321     while ( C->Pointer+n+1 >= C->Top ) DoubleCbuffer(bufnum,C->Pointer,50+par);
00322     w = C->Pointer;
00323     for ( i = 0; i < n; i++ ) *w++ = *array++;
00324     C->Pointer = w;
00325     return(0);
00326 }
00327
00328 /*
00329     #] AddNtoC :
00330     #[ InsTree :
00331
00332     Routines for balanced tree searching and insertion.
00333     Compared to Knuth we have a parent link. This minimizes the
00334     number of compares. That is better for anything that is more
00335     complicated than just single numbers.
00336     There are no provisions for removing elements from the tree.
00337     The routines are:
00338     void RedoTree(size) Re-allocates the tree space. There will
00339                         be MaxTreeSize = size elements.
00340     void ClearTree()    Prunes the tree down to the root element.
00341     int InsTree(int,int)Searches for the requested element. If not found it
00342                         will allocate a new element, balance the tree if
00343                         necessary and return the called number.
00344                         If it was in the tree, it returns the tree 'value'.
00345
00346     Commentary: added the bufnum on 14-sep-2010 to make the whole a bit
00347     more flexible (JV).
00348 */
00349 static COMPTREE comptreezero = {0,0,0,0,0,0};
00350
00351 int InsTree(int bufnum, int h)
00352 {
00353     CBUF *C = cbuf + bufnum;
00354     COMPTREE *boomlijst = C->boomlijst, *q = boomlijst + C->rootnum, *p, *s;
00355     WORD *v1, *v2, *v3;
00356     int ip, iq, is;
00357
00358     if ( C->numtree + 1 >= C->MaxTreeSize ) {
00359         if ( C->MaxTreeSize == 0 ) {
00360             COMPTREE *root;
00361             C->MaxTreeSize = 125;
00362             C->boomlijst = (COMPTREE *)Malloc1((C->MaxTreeSize+1)*sizeof(COMPTREE),
00363                 "ClearInsTree");
00364             root = C->boomlijst;
00365             C->numtree = 0;
00366             C->rootnum = 0;
00367             root->left = -1;
00368             root->right = -1;
00369             root->parent = -1;
00370             root->blnce = 0;
00371             root->value = -1;
00372             root->usage = 0;
00373             for ( ip = 1; ip < C->MaxTreeSize; ip++ ) { C->boomlijst[ip] = comptreezero; }
00374         }
00375         else {
00376             is = C->MaxTreeSize * 2;
00377             s  = (COMPTREE *)Malloc1((is+1)*sizeof(COMPTREE),"InsTree");
00378             for ( ip = 0; ip < C->MaxTreeSize; ip++ ) { s[ip] = C->boomlijst[ip]; }
00379             for ( ip = C->MaxTreeSize; ip <= is; ip++ ) { s[ip] = comptreezero; }
00380             if ( C->boomlijst ) M_free(C->boomlijst,"InsTree");
00381             C->boomlijst = s;
00382             C->MaxTreeSize = is;
00383         }
00384         boomlijst = C->boomlijst;
00385         q = boomlijst + C->rootnum;
00386     }
00387
00388     if ( q->right == -1 ) { /* First element */
00389         C->numtree++;
00390         s = boomlijst+C->numtree;
00391         q->right = C->numtree;
00392         s->parent = C->rootnum;
00393         s->left = s->right = -1;
00394         s->blnce = 0;
00395         s->value = h;
```

```
00396         s->usage = 1;
00397         return(h);
00398     }
00399     ip = q->right;
00400     while ( ip >= 0 ) {
00401         p = boomlijst + ip;
00402         v1 = C->rhs[p->value]; v2 = v3 = C->rhs[h];
00403         while ( *v3 ) v3 += *v3;  /* find the 0 that indicates end-of-expr */
00404         while ( *v1 == *v2 && v2 < v3 ) { v1++; v2++; }
00405         if ( *v1 > *v2 ) {
00406             iq = p->right;
00407             if ( iq >= 0 ) { ip = iq; }
00408             else {
00409                 C->numtree++;
00410                 is = C->numtree;
00411                 p->right = is;
00412                 s = boomlijst + is;
00413                 s->parent = ip; s->left = s->right = -1;
00414                 s->blnce = 0;   s->value = h; s->usage = 1;
00415                 p->blnce++;
00416                 if ( p->blnce == 0 ) return(h);
00417                 goto balance;
00418             }
00419         }
00420         else if ( *v1 < *v2 ) {
00421             iq = p->left;
00422             if ( iq >= 0 ) { ip = iq; }
00423             else {
00424                 C->numtree++;
00425                 is = C->numtree;
00426                 s = boomlijst+is;
00427                 p->left = is;
00428                 s->parent = ip; s->left = s->right = -1;
00429                 s->blnce = 0;   s->value = h; s->usage = 1;
00430                 p->blnce--;
00431                 if ( p->blnce == 0 ) return(h);
00432                 goto balance;
00433             }
00434         }
00435         else {
00436             p->usage++;
00437             return(p->value);
00438         }
00439     }
00440 /* INTERNAL_ERROR_EXCL_START */
00441     MesPrint("!>We vallen uit de boom!");
00442     Terminate(-1);
00443 /* INTERNAL_ERROR_EXCL_STOP */
00444     return(h);
00445 balance:;
00446     for (;;) {
00447         p = boomlijst + ip;
00448         iq = p->parent;
00449         if ( iq == C->rootnum ) break;
00450         q = boomlijst + iq;
00451         if ( ip == q->left ) q->blnce--;
00452         else                 q->blnce++;
00453         if ( q->blnce == 0 ) break;
00454         if ( q->blnce == -2 ) {
00455             if ( p->blnce == -1 ) { /* single rotation */
00456                 q->left = p->right;
00457                 p->right = iq;
00458                 p->parent = q->parent;
00459                 q->parent = ip;
00460                 if ( boomlijst[p->parent].left == iq ) boomlijst[p->parent].left = ip;
00461                 else                                   boomlijst[p->parent].right = ip;
00462                 if ( q->left >= 0 ) boomlijst[q->left].parent = iq;
00463                 q->blnce = p->blnce = 0;
00464             }
00465             else {  /* double rotation */
00466                 s = boomlijst + is;
00467                 q->left = s->right;
00468                 p->right = s->left;
00469                 s->right = iq;
00470                 s->left = ip;
00471                 if ( p->right >= 0 ) boomlijst[p->right].parent = ip;
00472                 if ( q->left >= 0 ) boomlijst[q->left].parent = iq;
00473                 s->parent = q->parent;
00474                 q->parent = is;
00475                 p->parent = is;
00476                 if ( boomlijst[s->parent].left == iq )
00477                     boomlijst[s->parent].left = is;
00478                 else   boomlijst[s->parent].right = is;
00479                 if ( s->blnce > 0 ) { q->blnce = s->blnce = 0; p->blnce = -1; }
00480                 else if ( s->blnce < 0 ) { p->blnce = s->blnce = 0; q->blnce = 1; }
00481                 else { p->blnce = s->blnce = q->blnce = 0; }
00482             }
```

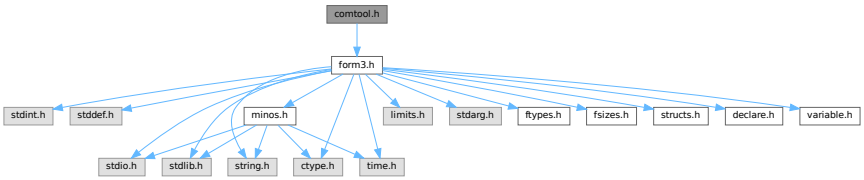
```
00483             break;
00484         }
00485         else if ( q->blnce == 2 ) {
00486             if ( p->blnce == 1 ) {  /* single rotation */
00487                 q->right = p->left;
00488                 p->left = iq;
00489                 p->parent = q->parent;
00490                 q->parent = ip;
00491                 if ( boomlijst[p->parent].left == iq ) boomlijst[p->parent].left = ip;
00492                 else                                   boomlijst[p->parent].right = ip;
00493                 if ( q->right >= 0 ) boomlijst[q->right].parent = iq;
00494                 q->blnce = p->blnce = 0;
00495             }
00496             else {  /* double rotation */
00497                 s = boomlijst + is;
00498                 q->right = s->left;
00499                 p->left = s->right;
00500                 s->left = iq;
00501                 s->right = ip;
00502                 if ( p->left >= 0 ) boomlijst[p->left].parent = ip;
00503                 if ( q->right >= 0 ) boomlijst[q->right].parent = iq;
00504                 s->parent = q->parent;
00505                 q->parent = is;
00506                 p->parent = is;
00507                 if ( boomlijst[s->parent].left == iq ) boomlijst[s->parent].left = is;
00508                 else                                   boomlijst[s->parent].right = is;
00509                 if ( s->blnce < 0 ) { q->blnce = s->blnce = 0; p->blnce = 1; }
00510                 else if ( s->blnce > 0 ) { p->blnce = s->blnce = 0; q->blnce = -1; }
00511                 else { p->blnce = s->blnce = q->blnce = 0; }
00512             }
00513             break;
00514         }
00515         is = ip; ip = iq;
00516     }
00517     return(h);
00518 }
00519
00520 /*
00521     #] InsTree :
00522     #[ FindTree :
00523
00524     Routines for balanced tree searching.
00525     Is like InsTree but without the insertions.
00526     Returns -1 if the element is not in the tree.
00527     The advantage of this routine over InsTree is that this routine
00528     can be run in parallel.
00529 */
00530
00531 int FindTree(int bufnum, WORD *subexpr)
00532 {
00533     CBUF *C = cbuf + bufnum;
00534     COMPTREE *boomlijst = C->boomlijst, *q = boomlijst + C->rootnum, *p;
00535     WORD *v1, *v2, *v3;
00536     int ip, iq;
00537
00538     ip = q->right;
00539     while ( ip >= 0 ) {
00540         p = boomlijst + ip;
00541         v1 = C->rhs[p->value]; v2 = v3 = subexpr;
00542         while ( *v3 ) v3 += *v3;  /* find the 0 that indicates end-of-expr */
00543         while ( *v1 == *v2 && v2 < v3 ) { v1++; v2++; }
00544         if ( *v1 > *v2 ) {
00545             iq = p->right;
00546             if ( iq >= 0 ) { ip = iq; }
00547             else { return(-1); }
00548         }
00549         else if ( *v1 < *v2 ) {
00550             iq = p->left;
00551             if ( iq >= 0 ) { ip = iq; }
00552             else { return(-1); }
00553         }
00554         else {
00555             p->usage++;
00556             return(p->value);
00557         }
00558     }
00559     return(-1);
00560 }
00561
00562 /*
00563     #] FindTree :
00564     #[ RedoTree :
00565 */
00566
00567 void RedoTree(CBUF *C, int size)
00568 {
00569     COMPTREE *newboomlijst;
```

```
00570     int i;
00571     newboomlijst = (COMPTREE *)Malloc1((size+1)*sizeof(COMPTREE),"newboomlijst");
00572     if ( C->boomlijst ) {
00573         if ( C->MaxTreeSize > size ) C->MaxTreeSize = size;
00574         for ( i = 0; i < C->MaxTreeSize; i++ ) newboomlijst[i] = C->boomlijst[i];
00575         M_free(C->boomlijst,"boomlijst");
00576     }
00577     C->boomlijst = newboomlijst;
00578     C->MaxTreeSize = size;
00579 }
00580
00581 /*
00582     #] RedoTree :
00583     #[ ClearTree :
00584 */
00585
00586 void ClearTree(int i)
00587 {
00588     CBUF *C = cbuf + i;
00589     COMPTREE *root = C->boomlijst;
00590     if ( root ) {
00591         C->numtree = 0;
00592         C->rootnum = 0;
00593         root->left = -1;
00594         root->right = -1;
00595         root->parent = -1;
00596         root->blnce = 0;
00597         root->value = -1;
00598         root->usage = 0;
00599     }
00600 }
00601
00602 /*
00603     #] ClearTree :
00604     #[ IniFbuffer :
00605 */
00612 int IniFbuffer(WORD bufnum)
00613 {
00614     CBUF *C = cbuf + bufnum;
00615     COMPTREE *root;
00616     int i;
00617     LONG fullsize;
00618     C->maxrhs = AM.fbuffersize;
00619     C->MaxTreeSize = AM.fbuffersize;
00620
00621     /*
00622      * Note that bufnum is a return value of inicbufs(). So C has been already
00623      * initialized. (TU 20 Dec 2011)
00624      */
00625     if ( C->boomlijst ) M_free(C->boomlijst, "IniFbuffer-tree");
00626     if ( C->rhs ) M_free(C->rhs, "IniFbuffer-rhs");
00627
00628     C->boomlijst = (COMPTREE *)Malloc1((C->MaxTreeSize+1)*sizeof(COMPTREE),"IniFbuffer-tree");
00629     root = C->boomlijst;
00630     C->numtree = 0;
00631     C->rootnum = 0;
00632     root->left = -1;
00633     root->right = -1;
00634     root->parent = -1;
00635     root->blnce = 0;
00636     root->value = -1;
00637     root->usage = 0;
00638     for ( i = 1; i < C->MaxTreeSize; i++ ) { C->boomlijst[i] = comptreezero; }
00639
00640     fullsize = (C->maxrhs+1) * (sizeof(WORD *) + 2*sizeof(LONG) + 2*sizeof(WORD));
00641     C->rhs = (WORD **)Malloc1(fullsize,"IniFbuffer-rhs");
00642     C->CanCommu = (LONG *)(C->rhs+C->maxrhs);
00643     C->NumTerms = (LONG *)(C->rhs+2*C->maxrhs);
00644     C->numdum = (WORD *)(C->NumTerms+C->maxrhs);
00645     C->dimension = (WORD *)(C->numdum+C->maxrhs);
00646
00647     return(0);
00648 }
00649
00650 /*
00651     #] IniFbuffer :
00652     #[ numcommute :
00653
00654     Returns the number of non-commuting terms in the expression
00655 */
00656
00657 LONG numcommute(WORD *terms, LONG *numterms)
00658 {
00659     LONG num = 0;
00660     WORD *t, *m;
00661     *numterms = 0;
00662     while ( *terms ) {
```

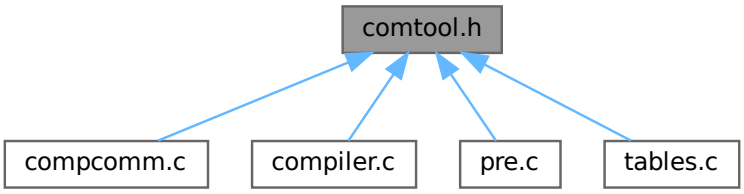
```
00663         *numterms += 1;
00664         t = terms + 1;
00665         GETSTOP(terms,m);
00666         while ( t < m ) {
00667             if ( *t >= FUNCTION ) {
00668                 if ( functions[*t-FUNCTION].commute ) { num++; break; }
00669             }
00670             t += t[1];
00671         }
00672         terms = terms + *terms;
00673     }
00674     return(num);
00675 }
00676
00677 /*
00678     #] numcommute :
00679 */
```

4.17 comtool.h File Reference

`#include "form3.h"`
Include dependency graph for comtool.h:



This graph shows which files directly or indirectly include this file:



4.17.1 Detailed Description

Utility routines for the compiler.

Definition in file comtool.h.

4.18 comtool.h

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
00026  *
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
00031 #ifndef FORM_COMTOOL_H_
00032 #define FORM_COMTOOL_H_
00033 /*
00034     #[ Includes :
00035 */
00036
00037 #include "form3.h"
00038
00039 /*
00040     #] Includes :
00041     #[ Inline functions :
00042 */
00043
00055 static inline void SkipSpaces(UBYTE **s)
00056 {
00057     const char *p = (const char *)*s;
00058     while ( *p == ' ' || *p == ',' || *p == '\t' ) p++;
00059     *s = (UBYTE *)p;
00060 }
00061
00073 static inline int ConsumeOption(UBYTE **s, const char *opt)
00074 {
00075     const char *p = (const char *)*s;
00076     while ( *p && *opt && tolower(*p) == tolower(*opt) ) {
00077         p++;
00078         opt++;
00079     }
00080     /* Check if `opt' ended. */
00081     if ( !*opt ) {
00082         /* Check if `*p' is a word boundary. */
00083         UINT c = FG.cTable[(unsigned char)*p];
00084         if ( c != 0 && c != 1 && *p != '_' && *p != '$' ) {
00085             /* Consume the option. Skip the trailing whitespace. */
00086             *s = (UBYTE *)p;
00087             SkipSpaces(s);
00088             return(1);
00089         }
00090     }
00091     return(0);
00092 }
00093
00094 /*
00095     #] Inline functions :
00096 */
00097 #endif  /* FORM_COMTOOL_H_ */
```

4.19 declare.h File Reference

This graph shows which files directly or indirectly include this file:

Macros

- #define MaX(x, y) ((x) > (y) ? (x): (y))
- #define MiN(x, y) ((x) < (y) ? (x): (y))
- #define ABS(x) ((x) < 0 ? -(x): (x))
- #define SGN(x) ((x) > 0 ? 1 : (x) < 0 ? -1 : 0)
- #define REDLENG(x) ((((x)<0)?((x)+1):((x)-1))/2)
- #define INCLENG(x) (((x)<0)?(((x)*2)-1):(((x)*2)+1))
- #define GETCOEF(x, y) x += *x;y = x[-1];x -= ABS(y);y=REDLENG(y)
- #define GETSTOP(x, y) y=x+(*x)-1;y -= ABS(*y)-1
- #define StuffAdd(x, y) (((x)<0?-1:1)*(y)+((y)<0?-1:1)*(x))
- #define EXCHN(t1, t2, n) { WORD a,i; for(i=0;i<n;i++){a=t1[i];t1[i]=t2[i];t2[i]=a;} }
- #define EXCH(x, y) { WORD a = (x); (x) = (y); (y) = a; }
- #define TOKENTOLINE(x, y)
- #define UngetFromStream(stream, c) ((stream)->nextchar[(stream)->isnextchar++]=c)
- #define AddLineFeed(s, n) { (s)[(n)++] = LINEFEED; }
- #define TryRecover(x) Terminate(-1)
- #define UngetChar(c) { pushbackchar = c; }
- #define ParseNumber(x, s) {(x)=0;while(*(s)>='0'&&*(s)<='9')(x)=10*(x)+*(s)++ -'0';}
- #define ParseSign(sgn, s)
- #define ParseSignedNumber(x, s)
- #define NCOPY(s, t, n) { while ((n)-- > 0) { *s++ = *t++; } }
- #define NCOPYI(s, t, n) { while ((n)-- > 0) { *s++ = *t++; } }
- #define NCOPYB(s, t, n) { while ((n)-- > 0) { *s++ = *t++; } }
- #define NCOPYI32(s, t, n) { while ((n)-- > 0) { *s++ = *t++; } }
- #define WCOPY(s, t, n) { int nn=n; WORD *ss=(WORD *)s, *tt=(WORD *)t; while ((nn)-- > 0) { *ss++ = *tt++; } }
- #define NeedNumber(x, s, err)
- #define SKIPBLANKS(s) { while (*(s) == ' ' || *(s) == '\t') (s)++; }
- #define FLUSHCONSOLE if (AP.InOutBuf > 0) CharOut(LINEFEED)
- #define SKIPBRA1(s)
- #define SKIPBRA2(s)
- #define SKIPBRA3(s)
- #define SKIPBRA4(s)
- #define SKIPBRA5(s)
- #define CYCLE1(t, a, i) {t iX=*a; WORD jX; for(jX=1;jX<i;jX++)a[jX-1]=a[jX]; a[i-1]=iX;}
- #define AddToCB(c, wx)
- #define EXCHINOUT
- #define BACKINOUT
- #define CopyArg(to, from)
- #define FILLARG(w)
- #define COPYARG(w, t)
- #define ZEROARG(w)
- #define FILLFUN(w)
- #define COPYFUN(w, t)

- #define COPYFUN3(w, t)
- #define FILLFUN3(w)
- #define FILLSUB(w)
- #define COPYSUB(w, ww)
- #define FILLEXPR(w)
- #define NEXTARG(x) if(*x>0) x += *x; else if(*x <= -FUNCTION)x++; else x += 2;
- #define COPY1ARG(s1, t1)
- #define ZeroFillRange(w, begin, end)
- #define TABLESIZE(a, b) (((WORD)sizeof(a))/((WORD)sizeof(b)))
- #define WORDDIF(x, y) (WORD)(x-y)
- #define wsizeof(a) ((WORD)sizeof(a))
- #define VARNAME(type, num) (AC.varnames->namebuffer+type[num].name)
- #define DOLLARNAME(type, num) (AC.dollarnames->namebuffer+type[num].name)
- #define EXPRNAME(num) (AC.exprnames->namebuffer+Expressions[num].name)
- #define PREV(x) prevorder?prevorder:x
- #define Terminate(x) do { TerminateImpl(x, __FILE__, __LINE__, __FUNCTION__); } while(0)
- #define SETERROR(x) { Terminate(-1); return(-1); }
- #define DUMMYUSE(x) (void)(x);
- #define ADDPOS(pp, x) (pp).p1 = ((pp).p1+(LONG)(x))
- #define SETBASELENGTH(ss, x) (ss).p1 = (LONG)(x)
- #define SETBASEPOSITION(pp, x) (pp).p1 = (LONG)(x)
- #define ISEQUALPOSINC(pp1, pp2, x) ((pp1).p1 == ((pp2).p1+(LONG)(x)))
- #define ISGEPOSINC(pp1, pp2, x) ((pp1).p1 >= ((pp2).p1+(LONG)(x)))
- #define DIVPOS(pp, n) ((pp).p1/(LONG)(n))
- #define MULPOS(pp, n) (pp).p1 *= (LONG)(n)
- #define DIFPOS(ss, pp1, pp2) (ss).p1 = ((pp1).p1-(pp2).p1)
- #define DIFBASE(pp1, pp2) ((pp1).p1-(pp2).p1)
- #define ADD2POS(pp1, pp2) (pp1).p1 += (pp2).p1
- #define PUTZERO(pp) (pp).p1 = 0
- #define BASEPOSITION(pp) ((pp).p1)
- #define SETSTARTPOS(pp) (pp).p1 = -2
- #define NOTSTARTPOS(pp) ((pp).p1 > -2)
- #define ISMINPOS(pp) ((pp).p1 == -1)
- #define ISEQUALPOS(pp1, pp2) ((pp1).p1 == (pp2).p1)
- #define ISNOTEQUALPOS(pp1, pp2) ((pp1).p1 != (pp2).p1)
- #define ISLESSPOS(pp1, pp2) ((pp1).p1 < (pp2).p1)
- #define ISGEPOS(pp1, pp2) ((pp1).p1 >= (pp2).p1)
- #define ISNOTZEROPOS(pp) ((pp).p1 != 0)
- #define ISZEROPOS(pp) ((pp).p1 == 0)
- #define ISPOSPOS(pp) ((pp).p1 > 0)
- #define ISNEGPOS(pp) ((pp).p1 < 0)
- #define TOLONG(x) ((LONG)(x))
- #define Add2Com(x) { WORD cod[2]; cod[0] = x; cod[1] = 2; AddNtoL(2,cod); }
- #define Add3Com(x1, x2) { WORD cod[3]; cod[0] = x1; cod[1] = 3; cod[2] = x2; AddNtoL(3,cod); }
- #define Add4Com(x1, x2, x3)
- #define Add5Com(x1, x2, x3, x4)
- #define WantAddPointers(x)
- #define WantAddLongs(x)
- #define WantAddPositions(x)
- #define FORM_INLINE inline
- #define MEMORYMACROS
- #define TermMalloc(x) ((AT.TermMemTop <= 0) ? TermMallocAddMemory(BHEAD0), AT.TermMemHeap[--AT.TermMemTop]: AT.TermMemHeap[--AT.TermMemTop])
- #define NumberMalloc(x) ((AT.NumberMemTop <= 0) ? NumberMallocAddMemory(BHEAD0), AT.↩ NumberMemHeap[--AT.NumberMemTop]: AT.NumberMemHeap[--AT.NumberMemTop])

- #define CacheNumberMalloc(x) ((AT.CacheNumberMemTop <= 0) ? CacheNumberMallocAdd↩ Memory(BHEAD0), AT.CacheNumberMemHeap[--AT.CacheNumberMemTop]: AT.CacheNumberMemHeap[--AT.CacheNumberMemTop])
- #define TermFree(TermMem, x) AT.TermMemHeap[AT.TermMemTop++] = (WORD *)(TermMem)
- #define NumberFree(NumberMem, x) AT.NumberMemHeap[AT.NumberMemTop++] = (UWORD *)(Number↩ Mem)
- #define CacheNumberFree(NumberMem, x) AT.CacheNumberMemHeap[AT.CacheNumberMemTop++] = (UWORD *)(NumberMem)
- #define NestingChecksum() (AC.IfLevel + AC.RepLevel + AC.arglevel + AC.insidelevel + AC.termlevel + AC.↩ inexprlevel + AC.dolooplevel +AC.SwitchLevel)
- #define MesNesting() MesPrint("&Illegal nesting of if, repeat, argument, inside, term, inexpression and do")
- #define MarkPolyRatFunDirty(T)
- #define PUSHPREASSIGNLEVEL
- #define POPPREASSIGNLEVEL
- #define EXTERNLOCK(x)
- #define INILOCK(x)
- #define LOCK(x)
- #define UNLOCK(x)
- #define EXTERNRWLOCK(x)
- #define INIRWLOCK(x)
- #define RWLOCKR(x)
- #define RWLOCKW(x)
- #define UNRWLOCK(x)
- #define MLOCK(x)
- #define MUNLOCK(x)
- #define GETIDENTITY
- #define GETBIDENTITY
- #define M_alloc(x) malloc((size_t)(x))
- #define CompareTerms ((COMPARE)AR.CompareRoutine)
- #define FiniShuffle AN.SHvar.finishuf
- #define DoShtuffle ((DO_UFFLE)AN.SHvar.do_uffle)

Typedefs

- typedef int(* WRITEBUFTOEXTCHANNEL) (char *, size_t)
- typedef int(* GETCFROMEXTCHANNEL) (void)
- typedef int(* SETTERMINATORFOREXTERNALCHANNEL) (char *)
- typedef int(* SETKILLMODEFOREXTERNALCHANNEL) (int, int)
- typedef LONG(* WRITEFILE) (int, UBYTE *, LONG)
- typedef WORD(* GETTERM) (PHEAD WORD *)

Functions

- void TELLFILE (int, POSITION *)
- void StartVariables (void)
- void **setSignalHandlers** (void)
- UBYTE * CodeToLine (WORD, UBYTE *)
- UBYTE * AddArrayIndex (WORD, UBYTE *)
- INDEXENTRY * FindInIndex (WORD, FILEDATA *, WORD, WORD)
- INDEXENTRY * NextFileIndex (POSITION *)
- WORD * PasteTerm (PHEAD WORD, WORD *, WORD *, WORD, WORD)
- UBYTE * StrCopy (UBYTE *, UBYTE *)
- UBYTE * WrtPower (UBYTE *, WORD)

- int AccumGCD (PHEAD UWORD *, WORD *, UWORD *, WORD)
- void AddArgs (PHEAD WORD *, WORD *, WORD *)
- int AddCoef (PHEAD WORD **, WORD **)
- int AddLong (UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- int AddPLon (UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- int AddPoly (PHEAD WORD **, WORD **)
- int AddRat (PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- void AddToLine (UBYTE *)
- int AddWild (PHEAD WORD, WORD, WORD)
- int BigLong (UWORD *, WORD, UWORD *, WORD)
- int BinomGen (PHEAD WORD *, WORD, WORD **, WORD, WORD, WORD, WORD, WORD, UWORD *, WORD)
- int CheckWild (PHEAD WORD, WORD, WORD, WORD *)
- int Chisholm (PHEAD WORD *, WORD)
- int CleanExpr (WORD)
- void CleanUp (WORD)
- void ClearWild (PHEAD0)
- int CompareFunctions (WORD *, WORD *)
- int Commute (WORD *, WORD *)
- WORD DetCommu (WORD *)
- WORD DoesCommu (WORD *)
- int CompArg (WORD *, WORD *)
- WORD CompCoef (WORD *, WORD *)
- int CompGroup (PHEAD WORD, WORD **, WORD *, WORD *, WORD)
- WORD Compare1 (PHEAD WORD *, WORD *, WORD)
- WORD CountDo (WORD *, WORD *)
- WORD CountFun (WORD *, WORD *)
- WORD DimensionSubterm (WORD *)
- WORD DimensionTerm (WORD *)
- WORD DimensionExpression (PHEAD WORD *)
- int Deferred (PHEAD WORD *, WORD)
- int DeleteStore (WORD)
- WORD DetCurDum (PHEAD WORD *)
- void DetVars (WORD *, WORD)
- int Distribute (DISTRIBUTE *, WORD)
- int DivLong (UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *, UWORD *, WORD *)
- int DivRat (PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- int Divvy (PHEAD UWORD *, WORD *, UWORD *, WORD)
- int DoDelta (WORD *)
- int DoDelta3 (PHEAD WORD *, WORD)
- int TestPartitions (WORD *, PARTI *)
- int DoPartitions (PHEAD WORD *, WORD)
- int CoCanonicalize (UBYTE *)
- int DoCanonicalize (PHEAD WORD *, WORD *)
- int GenDiagrams (PHEAD WORD *, WORD)
- int DoTopologyCanonicalize (PHEAD WORD *, WORD, WORD, WORD *)
- int DoShattering (PHEAD WORD *, WORD *, WORD *, WORD)
- int DoTableExpansion (WORD *, WORD)
- int DoDistrib (PHEAD WORD *, WORD)
- int DoShuffle (WORD *, WORD, WORD, WORD)
- int DoPermutations (PHEAD WORD *, WORD)
- int Shuffle (WORD *, WORD *, WORD *)
- int FinishShuffle (WORD *)
- int DoStuffle (WORD *, WORD, WORD, WORD)
- int Stuffle (WORD *, WORD *, WORD *)

- int FinishStuffle (WORD *)
- WORD * StuffRootAdd (WORD *, WORD *, WORD *)
- int TestUse (WORD *, WORD)
- DBASE * FindTB (UBYTE *)
- int CheckTableDeclarations (DBASE *)
- void Apply (WORD *, WORD)
- int ApplyExec (WORD *, int, WORD)
- void ApplyReset (WORD)
- void TableReset (void)
- void ReWorkT (WORD *, WORD *, WORD)
- WORD GetIfDollarNum (WORD *, WORD *)
- int FindVar (WORD *, WORD *)
- int DoIfStatement (PHEAD WORD *, WORD *)
- int DoOnePow (PHEAD WORD *, WORD, WORD, WORD *, WORD *, WORD, WORD *)
- void DoRevert (WORD *, WORD *)
- int DoSumF1 (PHEAD WORD *, WORD *, WORD, WORD)
- int DoSumF2 (PHEAD WORD *, WORD *, WORD, WORD)
- int DoTheta (PHEAD WORD *)
- LONG EndSort (PHEAD WORD *, int)
- int EntVar (WORD, UBYTE *, WORD, WORD, WORD, WORD)
- int EpfCon (PHEAD WORD *, WORD *, WORD, WORD)
- int EpfFind (PHEAD WORD *, WORD *)
- WORD EpfGen (WORD, WORD *, WORD *, WORD *, WORD)
- int EqualArg (WORD *, WORD, WORD)
- int Factorial (PHEAD WORD, UWORD *, WORD *)
- int Bernoulli (WORD, UWORD *, WORD *)
- int FactorIn (PHEAD WORD *, WORD)
- int FactorInExpr (PHEAD WORD *, WORD)
- int FindAll (PHEAD WORD *, WORD *, WORD, WORD *)
- WORD FindMulti (PHEAD WORD *, WORD *)
- int FindOnce (PHEAD WORD *, WORD *)
- int FindOnly (PHEAD WORD *, WORD *)
- int FindRest (PHEAD WORD *, WORD *)
- void **FindSpecial** (WORD *)
- WORD FindrNumber (WORD, VARRENUM *)
- void FiniLine (void)
- int FiniTerm (PHEAD WORD *, WORD *, WORD *, WORD, WORD)
- int FlushOut (POSITION *, FILEHANDLE *, int)
- void FunLevel (PHEAD WORD *)
- void AdjustRenumScratch (PHEAD0)
- void GarbHand (void)
- int GcdLong (PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- int LcmLong (PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- void **GCD** (UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- ULONG **GCD2** (ULONG, ULONG)
- int Generator (PHEAD WORD *, WORD)
- int GetBinom (UWORD *, WORD *, WORD, WORD)
- WORD GetFromStore (WORD *, POSITION *, RENUMBER, WORD *, WORD)
- int GetLong (UBYTE *, UWORD *, WORD *)
- WORD GetMoreTerms (WORD *)
- int GetMoreFromMem (WORD *, WORD **)
- WORD GetOneTerm (PHEAD WORD *, FILEHANDLE *, POSITION *, int)
- RENUMBER GetTable (WORD, POSITION *, WORD)
- WORD GetTerm (PHEAD WORD *)
- int Glue (PHEAD WORD *, WORD *, WORD *, WORD)

- int InFunction (PHEAD WORD *, WORD *)
- void IniLine (WORD)
- void IniVars (void)
- int InsertTerm (PHEAD WORD *, WORD, WORD, WORD *, WORD *, WORD)
- void LongToLine (UWORD *, WORD)
- int MakeDirty (WORD *, WORD *, WORD)
- void MarkDirty (WORD *, WORD)
- void PolyFunDirty (PHEAD WORD *)
- void PolyFunClean (PHEAD WORD *)
- int MakeModTable (void)
- int MatchE (PHEAD WORD *, WORD *, WORD *, WORD)
- int MatchCy (PHEAD WORD *, WORD *, WORD *, WORD)
- int FunMatchCy (PHEAD WORD *, WORD *, WORD *, WORD)
- int FunMatchSy (PHEAD WORD *, WORD *, WORD *, WORD)
- int MatchArgument (PHEAD WORD *, WORD *)
- int MatchFunction (PHEAD WORD *, WORD *, WORD *)
- int MergePatches (WORD)
- int MesCerr (char *, UBYTE *)
- int MesComp (char *, UBYTE *, UBYTE *)
- int Modulus (WORD *)
- void MoveDummies (PHEAD WORD *, WORD)
- int MulLong (UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- int MulRat (PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- int Mully (PHEAD UWORD *, WORD *, UWORD *, WORD)
- int MultDo (PHEAD WORD *, WORD *)
- int NewSort (PHEAD0)
- int ExtraSymbol (WORD, WORD, WORD, WORD *, WORD *)
- int Normalize (PHEAD WORD *)
- int BracketNormalize (PHEAD WORD *)
- void DropCoefficient (PHEAD WORD *)
- void DropSymbols (PHEAD WORD *)
- int PutInside (PHEAD WORD *, WORD *)
- void OpenTemp (void)
- void Pack (UWORD *, WORD *, UWORD *, WORD)
- LONG PasteFile (PHEAD WORD, WORD *, POSITION *, WORD **, RENUMBER, WORD *, WORD)
- int Permute (PERM *, WORD)
- int PermuteP (PERMP *, WORD)
- int PolyFunMul (PHEAD WORD *)
- int PopVariables (void)
- int PrepPoly (PHEAD WORD *, WORD)
- int Processor (void)
- int Product (UWORD *, WORD *, WORD)
- void PrtLong (UWORD *, WORD, UBYTE *)
- void PrtTerms (void)
- void PrintDeprecation (const char *, const char *)
- void PrintFeatureList (void)
- void PrintRunningTime (void)
- LONG GetRunningTime (void)
- int PutBracket (PHEAD WORD *)
- LONG PutIn (FILEHANDLE *, POSITION *, WORD *, WORD **, int)
- int PutInStore (INDEXENTRY *, WORD)
- WORD PutOut (PHEAD WORD *, POSITION *, FILEHANDLE *, WORD)
- UWORD Quotient (UWORD *, WORD *, WORD)
- int RaisPow (PHEAD UWORD *, WORD *, UWORD)
- void RaisPowCached (PHEAD WORD, WORD, UWORD **, WORD *)

- WORD RaisPowMod (WORD, WORD, WORD)
- int NormalModulus (UWORD *, WORD *)
- int MakeInverses (void)
- int GetModInverses (WORD, WORD, WORD *, WORD *)
- int GetLongModInverses (PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *, UWORD *, WORD *)
- void RatToLine (UWORD *, WORD)
- int RatioFind (PHEAD WORD *, WORD *)
- int RatioGen (PHEAD WORD *, WORD *, WORD, WORD)
- WORD ReNumber (PHEAD WORD *)
- WORD ReadSnum (UBYTE **)
- WORD Remain10 (UWORD *, WORD *)
- WORD Remain4 (UWORD *, WORD *)
- int ResetScratch (void)
- int ResolveSet (PHEAD WORD *, WORD *, WORD *)
- int RevertScratch (void)
- int ScanFunctions (PHEAD WORD *, WORD *, WORD)
- void SeekScratch (FILEHANDLE *, POSITION *)
- void SetEndScratch (FILEHANDLE *, POSITION *)
- void SetEndHScratch (FILEHANDLE *, POSITION *)
- int SetFileIndex (void)
- int Sflush (FILEHANDLE *)
- int Simplify (PHEAD UWORD *, WORD *, UWORD *, WORD *)
- int SortWild (WORD *, WORD)
- FILE * LocateBase (char **, char **, char *)
- LONG SplitMerge (PHEAD WORD **, LONG)
- int StoreTerm (PHEAD WORD *)
- void SubPLon (UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- void Substitute (PHEAD WORD *, WORD *, WORD)
- int SymFind (PHEAD WORD *, WORD *)
- int SymGen (PHEAD WORD *, WORD *, WORD, WORD)
- WORD Symmetrize (PHEAD WORD *, WORD *, WORD, WORD, WORD)
- int FullSymmetrize (PHEAD WORD *, int)
- int TakeModulus (UWORD *, WORD *, UWORD *, WORD, WORD)
- int TakeNormalModulus (UWORD *, WORD *, UWORD *, WORD, WORD)
- void TalToLine (UWORD)
- int TenVec (PHEAD WORD *, WORD *, WORD, WORD)
- int TenVecFind (PHEAD WORD *, WORD *)
- int TermRenumber (WORD *, RENUMBER, WORD)
- void TestDrop (void)
- void PutInVflags (WORD)
- int TestMatch (PHEAD WORD *, WORD *)
- WORD TestSub (PHEAD WORD *, WORD)
- LONG TimeCPU (WORD)
- LONG TimeChildren (WORD)
- LONG TimeWallClock (WORD)
- LONG **Timer** (int)
- int **GetTimerInfo** (LONG **, LONG **)
- void **WriteTimerInfo** (LONG *, LONG *)
- LONG **GetWorkerTimes** (void)
- int ToStorage (EXPRESSIONS, POSITION *)
- void TokenToLine (UBYTE *)
- int Trace4 (PHEAD WORD *, WORD *, WORD, WORD)
- int Trace4Gen (PHEAD TRACES *, WORD)
- int Trace4no (WORD, WORD *, TRACES *)

- int TraceFind (PHEAD WORD *, WORD *)
- int TraceN (PHEAD WORD *, WORD *, WORD, WORD)
- int TraceNgen (PHEAD TRACES *, WORD)
- WORD TraceNno (WORD, WORD *, TRACES *)
- int Traces (PHEAD WORD *, WORD *, WORD, WORD)
- WORD Trick (WORD *, TRACES *)
- int TryDo (PHEAD WORD *, WORD *, WORD)
- void UnPack (UWORD *, WORD, WORD *, WORD *)
- int VarStore (UBYTE *, WORD, WORD, WORD)
- WORD WildFill (PHEAD WORD *, WORD *, WORD *)
- int WriteAll (void)
- int WriteOne (UBYTE *, int, int, WORD)
- void WriteArgument (WORD *)
- int WriteExpression (WORD *, LONG)
- int WriteInnerTerm (WORD *, WORD)
- void WriteLists (void)
- void WriteSetup (void)
- void WriteStats (POSITION *, WORD, WORD)
- int WriteSubTerm (WORD *, WORD)
- int WriteTerm (WORD *, WORD *, WORD, WORD, WORD)
- int execarg (PHEAD WORD *, WORD)
- int execterm (PHEAD WORD *, WORD)
- void SpecialCleanup (PHEAD0)
- void SetMods (void)
- void UnSetMods (void)
- int DoExecute (WORD, WORD)
- void SetScratch (FILEHANDLE *, POSITION *)
- void Warning (char *)
- void HighWarning (char *)
- int SpareTable (TABLES)
- UBYTE * strDup1 (UBYTE *, char *)
- void * Malloc1 (LONG, const char *)
- int DoTail (int, UBYTE **)
- int OpenInput (void)
- int PutPreVar (UBYTE *, UBYTE *, UBYTE *, int)
- void Error0 (char *)
- void Error1 (char *, UBYTE *)
- void Error2 (char *, char *, UBYTE *)
- UBYTE ReadFromStream (STREAM *)
- UBYTE GetFromStream (STREAM *)
- UBYTE LookInStream (STREAM *)
- STREAM * OpenStream (UBYTE *, int, int, int)
- int LocateFile (UBYTE **, int)
- STREAM * CloseStream (STREAM *)
- void PositionStream (STREAM *, LONG)
- int ReverseStatements (STREAM *)
- int ProcessOption (UBYTE *, UBYTE *, int)
- int DoSetups (void)
- void TerminateImpl (int, const char *, int, const char *) NORETURN
- NAMENODE * GetNode (NAMETREE *, UBYTE *)
- int AddName (NAMETREE *, UBYTE *, WORD, WORD, int *)
- int GetName (NAMETREE *, UBYTE *, WORD *, int)
- UBYTE * GetFunction (UBYTE *, WORD *)
- UBYTE * GetNumber (UBYTE *, WORD *)
- int GetLastExprName (UBYTE *, WORD *)

- int GetAutoName (UBYTE *, WORD *)
- int GetVar (UBYTE *, WORD *, WORD *, int, int)
- int MakeDubious (NAMETREE *, UBYTE *, WORD *)
- int GetOName (NAMETREE *, UBYTE *, WORD *, int)
- void DumpTree (NAMETREE *)
- void DumpNode (NAMETREE *, WORD, WORD)
- void LinkTree (NAMETREE *, WORD, WORD)
- void CopyTree (NAMETREE *, NAMETREE *, WORD, WORD)
- int CompactifyTree (NAMETREE *, WORD)
- NAMETREE * MakeNameTree (void)
- void FreeNameTree (NAMETREE *)
- int AddExpression (UBYTE *, int, int)
- int AddSymbol (UBYTE *, int, int, int, int)
- int AddDollar (UBYTE *, WORD, WORD *, LONG)
- int ReplaceDollar (WORD, WORD, WORD *, LONG)
- int DollarRaiseLow (UBYTE *, LONG)
- int AddVector (UBYTE *, int, int)
- int AddDubious (UBYTE *)
- int AddIndex (UBYTE *, int, int)
- UBYTE * DoDimension (UBYTE *, int *, int *)
- int AddFunction (UBYTE *, int, int, int, int, int, int, int)
- int CoCommuteInSet (UBYTE *)
- int CoFunction (UBYTE *, int, int)
- int TestName (UBYTE *)
- int AddSet (UBYTE *, WORD)
- int DoElements (UBYTE *, SETS, UBYTE *)
- int DoTempSet (UBYTE *, UBYTE *)
- int NameConflict (int, UBYTE *)
- int OpenFile (char *)
- int OpenAddFile (char *)
- int ReOpenFile (char *)
- int CreateFile (char *)
- int CreateLogFile (char *)
- void CloseFile (int)
- int CopyFile (char *, char *)
- int CreateHandle (void)
- LONG ReadFile (int, UBYTE *, LONG)
- LONG ReadPosFile (PHEAD FILEHANDLE *, UBYTE *, LONG, POSITION *)
- LONG WriteFileToFile (int, UBYTE *, LONG)
- void SeekFile (int, POSITION *, int)
- LONG TellFile (int)
- void FlushFile (int)
- int GetPosFile (int, fpos_t *)
- int SetPosFile (int, fpos_t *)
- void SynchFile (int)
- void TruncateFile (int)
- int GetChannel (char *, int)
- int GetAppendChannel (char *)
- int CloseChannel (char *)
- void inictable (void)
- KEYWORD * findcommand (UBYTE *)
- int inicbufs (void)
- void StartFiles (void)
- UBYTE * MakeDate (void)
- void PreProcessor (void)

- void * FromList (LIST *)
- void * From0List (LIST *)
- void * FromVarList (LIST *)
- int DoubleList (void ***, int *, int, char *)
- int DoubleLList (void ***, LONG *, int, char *)
- void DoubleBuffer (void **, void **, int, char *)
- void ExpandBuffer (void **, LONG *, int)
- LONG iexp (LONG, int)
- int IsLikeVector (WORD *)
- int AreArgsEqual (WORD *, WORD *)
- int CompareArgs (WORD *, WORD *)
- UBYTE * SkipField (UBYTE *, int)
- int StrCmp (UBYTE *, UBYTE *)
- int StrICmp (UBYTE *, UBYTE *)
- int StrHICmp (UBYTE *, UBYTE *)
- int StrICont (UBYTE *, UBYTE *)
- int CmpArray (WORD *, WORD *, WORD)
- int ConWord (UBYTE *, UBYTE *)
- int StrLen (UBYTE *)
- UBYTE * GetPreVar (UBYTE *, int)
- void ToGeneral (WORD *, WORD *, WORD)
- WORD ToPolyFunGeneral (PHEAD WORD *)
- int ToFast (WORD *, WORD *)
- SETUPPARAMETERS * GetSetupPar (UBYTE *)
- int RecalcSetups (void)
- int AllocSetups (void)
- SORTING * AllocSort (LONG, LONG, LONG, LONG, int, int, LONG, int)
- void AllocSortFileName (SORTING *)
- UBYTE * LoadInputFile (UBYTE *, int)
- UBYTE GetInput (void)
- void ClearPushback (void)
- UBYTE GetChar (int)
- void CharOut (UBYTE)
- void UnsetAllowDelay (void)
- void PopPreVars (int)
- void IniModule (int)
- void IniSpecialModule (int)
- int ModuleInstruction (int *, int *)
- int PreProInstruction (void)
- int LoadInstruction (int)
- int LoadStatement (int)
- KEYWORD * FindKeyWord (UBYTE *, KEYWORD *, int)
- KEYWORD * FindInKeyWord (UBYTE *, KEYWORD *, int)
- int DoDefine (UBYTE *)
- int DoRedefine (UBYTE *)
- int TheDefine (UBYTE *, int)
- int TheUndefine (UBYTE *)
- int ClearMacro (UBYTE *)
- int DoUndefine (UBYTE *)
- int DoInclude (UBYTE *)
- int DoReverseInclude (UBYTE *)
- int Include (UBYTE *, int)
- int DoExternal (UBYTE *)
- int DoToExternal (UBYTE *)
- int DoFromExternal (UBYTE *)

- int DoPrompt (UBYTE *)
- int DoSetExternal (UBYTE *)
- int DoSetExternalAttr (UBYTE *)
- int DoRmExternal (UBYTE *)
- int DoFactDollar (UBYTE *)
- WORD GetDollarNumber (UBYTE **, DOLLARS)
- int DoSetRandom (UBYTE *)
- int DoOptimize (UBYTE *)
- int DoClearOptimize (UBYTE *)
- int DoSkipExtraSymbols (UBYTE *)
- int DoTimeOutAfter (UBYTE *)
- int DoMessage (UBYTE *)
- int DoPreOut (UBYTE *)
- int DoPreAppend (UBYTE *)
- int DoPreCreate (UBYTE *)
- int DoPreAssign (UBYTE *)
- int DoPreBreak (UBYTE *)
- int DoPreDefault (UBYTE *)
- int DoPreSwitch (UBYTE *)
- int DoPreEndSwitch (UBYTE *)
- int DoPreCase (UBYTE *)
- int DoPreShow (UBYTE *)
- int DoPreExchange (UBYTE *)
- int DoSystem (UBYTE *)
- int DoPipe (UBYTE *)
- void StartPrepro (void)
- int DoIfdef (UBYTE *, int)
- int DoIfydef (UBYTE *)
- int DoIfndef (UBYTE *)
- int DoElse (UBYTE *)
- int DoElseif (UBYTE *)
- int DoEndif (UBYTE *)
- int DoTerminate (UBYTE *)
- int DoIf (UBYTE *)
- int DoCall (UBYTE *)
- int DoDebug (UBYTE *)
- int DoContinueDo (UBYTE *)
- int DoDo (UBYTE *)
- int DoBreakDo (UBYTE *)
- int DoEnddo (UBYTE *)
- int DoEndprocedure (UBYTE *)
- int DoInside (UBYTE *)
- int DoEndInside (UBYTE *)
- int DoProcedure (UBYTE *)
- int DoPrePrintTimes (UBYTE *)
- int DoPreWrite (UBYTE *)
- int DoPreClose (UBYTE *)
- int DoPreRemove (UBYTE *)
- int DoPreSortReallocate (UBYTE *)
- int DoCommentChar (UBYTE *)
- int DoPrcExtension (UBYTE *)
- int DoPreReset (UBYTE *)
- void WriteString (int, UBYTE *, int)
- void WriteUnfinString (int, UBYTE *, int)
- UBYTE * AddToString (UBYTE *, UBYTE *, int)

- UBYTE * PreCalc (void)
- UBYTE * PreEval (UBYTE *, LONG *)
- void NumToStr (UBYTE *, LONG)
- int PreCmp (int, int, UBYTE *, int, int, UBYTE *, int)
- int PreEq (int, int, UBYTE *, int, int, UBYTE *, int)
- UBYTE * pParseObject (UBYTE *, int *, LONG *)
- UBYTE * PreIfEval (UBYTE *, int *)
- int EvalPreIf (UBYTE *)
- int PreLoad (PRELOAD *, UBYTE *, UBYTE *, int, char *)
- int PreSkip (UBYTE *, UBYTE *, int)
- UBYTE * EndOfToken (UBYTE *)
- void SetSpecialMode (int, int)
- void MakeGlobal (void)
- int ExecModule (int)
- int ExecStore (void)
- void FullCleanUp (void)
- int DoExecStatement (void)
- int DoPipeStatement (void)
- int DoPolyfun (UBYTE *)
- int DoPolyratfun (UBYTE *)
- int CompileStatement (UBYTE *)
- UBYTE * ToToken (UBYTE *)
- int GetDollar (UBYTE *)
- void MesWork (void) NORETURN
- int MesPrint (const char *,...)
- int MesCall (char *)
- UBYTE * NumCopy (WORD, UBYTE *)
- char * LongCopy (LONG, char *)
- char * LongLongCopy (off_t *, char *)
- void ReserveTempFiles (int)
- void PrintTerm (WORD *, char *)
- void PrintTermC (WORD *, char *)
- void PrintSubTerm (WORD *, char *)
- void PrintWords (WORD *, LONG)
- void PrintSeq (WORD *, char *)
- int ExpandTripleDots (int)
- LONG ComPress (WORD **, LONG *)
- void StageSort (FILEHANDLE *)
- void M_free (void *, const char *)
- void ClearWildcardNames (void)
- int AddWildcardName (UBYTE *)
- int GetWildcardName (UBYTE *)
- void Globalize (int)
- void ResetVariables (int)
- void AddToPreTypes (int)
- void MessPreNesting (int)
- LONG GetStreamPosition (STREAM *)
- WORD * DoubleCbuffer (int, WORD *, int)
- WORD * AddLHS (int)
- WORD * AddRHS (int, int)
- int AddNtoL (int, WORD *)
- int AddNtoC (int, int, WORD *, int)
- void DoubleIfBuffers (void)
- STREAM * CreateStream (UBYTE *)
- int setonoff (UBYTE *, int *, int, int)

- int DoPrint (UBYTE *, int)
- int SetExpr (UBYTE *, int, int)
- void AddToCom (int, WORD *)
- int Add2ComStrings (int, WORD *, UBYTE *, UBYTE *)
- int DoSymmetrize (UBYTE *, int)
- int DoArgument (UBYTE *, int)
- int ArgFactorize (PHEAD WORD *, WORD *)
- WORD * TakeArgContent (PHEAD WORD *, WORD *)
- WORD * MakeInteger (PHEAD WORD *, WORD *, WORD *)
- WORD * MakeMod (PHEAD WORD *, WORD *, WORD *)
- WORD FindArg (PHEAD WORD *)
- int InsertArg (PHEAD WORD *, WORD *, int)
- int CleanupArgCache (PHEAD WORD)
- int ArgSymbolMerge (WORD *, WORD *)
- int ArgDotproductMerge (WORD *, WORD *)
- void SortWeights (LONG *, LONG *, WORD)
- int DoBrackets (UBYTE *, int)
- int DoPutInside (UBYTE *, int)
- WORD * CountComp (UBYTE *, WORD *)
- int CoAntiBracket (UBYTE *)
- int CoAntiSymmetrize (UBYTE *)
- int DoArgPlode (UBYTE *, int)
- int CoArgExplode (UBYTE *)
- int CoArgImplode (UBYTE *)
- int CoArgument (UBYTE *)
- int CoInside (UBYTE *)
- int ExecInside (UBYTE *)
- int CoInExpression (UBYTE *)
- int CoInParallel (UBYTE *)
- int CoNotInParallel (UBYTE *)
- int DoInParallel (UBYTE *, int)
- int CoEndInExpression (UBYTE *)
- int CoBracket (UBYTE *)
- int CoPutInside (UBYTE *)
- int CoAntiPutInside (UBYTE *)
- int CoMultiBracket (UBYTE *)
- int CoCFunction (UBYTE *)
- int CoCTensor (UBYTE *)
- int CoCollect (UBYTE *)
- int CoCompress (UBYTE *)
- int CoContract (UBYTE *)
- int CoCycleSymmetrize (UBYTE *)
- int CoDelete (UBYTE *)
- int CoTableBase (UBYTE *)
- int CoApply (UBYTE *)
- int CoDenominators (UBYTE *)
- int CoDimension (UBYTE *)
- int CoDiscard (UBYTE *)
- int CoDisorder (UBYTE *)
- int CoDrop (UBYTE *)
- int CoDropCoefficient (UBYTE *)
- int CoDropSymbols (UBYTE *)
- int CoElse (UBYTE *)
- int CoElseIf (UBYTE *)
- int CoEndArgument (UBYTE *)

- int CoEndInside (UBYTE *)
- int CoEndIf (UBYTE *)
- int CoEndRepeat (UBYTE *)
- int CoEndTerm (UBYTE *)
- int CoEndWhile (UBYTE *)
- int CoExit (UBYTE *)
- int CoFactArg (UBYTE *)
- int CoFactDollar (UBYTE *)
- int CoFactorize (UBYTE *)
- int CoNFactorize (UBYTE *)
- int CoUnFactorize (UBYTE *)
- int CoNUnFactorize (UBYTE *)
- int DoFactorize (UBYTE *, int)
- int CoFill (UBYTE *)
- int CoFillExpression (UBYTE *)
- int CoFixIndex (UBYTE *)
- int CoFormat (UBYTE *)
- int CoGlobal (UBYTE *)
- int CoGlobalFactorized (UBYTE *)
- int CoGoTo (UBYTE *)
- int CoId (UBYTE *)
- int CoIdNew (UBYTE *)
- int CoIdOld (UBYTE *)
- int CoIf (UBYTE *)
- int CoIfMatch (UBYTE *)
- int CoIfNoMatch (UBYTE *)
- int CoIndex (UBYTE *)
- int CoInsideFirst (UBYTE *)
- int CoKeep (UBYTE *)
- int CoLabel (UBYTE *)
- int CoLoad (UBYTE *)
- int CoLocal (UBYTE *)
- int CoLocalFactorized (UBYTE *)
- int CoMany (UBYTE *)
- int CoMerge (UBYTE *)
- int CoStuffle (UBYTE *)
- int CoMetric (UBYTE *)
- int CoModOption (UBYTE *)
- int CoModuleOption (UBYTE *)
- int CoModulus (UBYTE *)
- int CoMulti (UBYTE *)
- int CoMultiply (UBYTE *)
- int CoNFunction (UBYTE *)
- int CoNPrint (UBYTE *)
- int CoNTensor (UBYTE *)
- int CoNWrite (UBYTE *)
- int CoNoDrop (UBYTE *)
- int CoNoSkip (UBYTE *)
- int CoNormalize (UBYTE *)
- int CoMakeInteger (UBYTE *)
- int CoFlags (UBYTE *, int)
- int CoOff (UBYTE *)
- int CoOn (UBYTE *)
- int CoOnce (UBYTE *)
- int CoOnly (UBYTE *)

- int CoOptimizeOption (UBYTE *)
- int CoPolyFun (UBYTE *)
- int CoPolyRatFun (UBYTE *)
- int CoPrint (UBYTE *)
- int CoPrintB (UBYTE *)
- int CoProperCount (UBYTE *)
- int CoUnitTrace (UBYTE *)
- int CoRCycleSymmetrize (UBYTE *)
- int CoRatio (UBYTE *)
- int CoRedefine (UBYTE *)
- int CoRenumber (UBYTE *)
- int CoRepeat (UBYTE *)
- int CoSave (UBYTE *)
- int CoSelect (UBYTE *)
- int CoSet (UBYTE *)
- int CoSetExitFlag (UBYTE *)
- int CoSkip (UBYTE *)
- int CoProcessBucket (UBYTE *)
- int CoPushHide (UBYTE *)
- int CoPopHide (UBYTE *)
- int CoHide (UBYTE *)
- int CoIntoHide (UBYTE *)
- int CoNoIntoHide (UBYTE *)
- int CoNoHide (UBYTE *)
- int CoUnHide (UBYTE *)
- int CoNoUnHide (UBYTE *)
- int CoSort (UBYTE *)
- int CoSplitArg (UBYTE *)
- int CoSplitFirstArg (UBYTE *)
- int CoSplitLastArg (UBYTE *)
- int CoSum (UBYTE *)
- int CoSymbol (UBYTE *)
- int CoSymmetrize (UBYTE *)
- int DoTable (UBYTE *, int)
- int CoTable (UBYTE *)
- int CoTerm (UBYTE *)
- int CoNTable (UBYTE *)
- int CoCTable (UBYTE *)
- void EmptyTable (TABLES)
- int CoToTensor (UBYTE *)
- int CoToVector (UBYTE *)
- int CoTrace4 (UBYTE *)
- int CoTraceN (UBYTE *)
- int CoChisholm (UBYTE *)
- int CoTransform (UBYTE *)
- int CoClearTable (UBYTE *)
- int DoChain (UBYTE *, int)
- int CoChainin (UBYTE *)
- int CoChainout (UBYTE *)
- int CoTryReplace (UBYTE *)
- int CoVector (UBYTE *)
- int CoWhile (UBYTE *)
- int CoWrite (UBYTE *)
- int CoAuto (UBYTE *)
- int CoSwitch (UBYTE *)

- int CoCase (UBYTE *)
- int CoBreak (UBYTE *)
- int CoDefault (UBYTE *)
- int CoEndSwitch (UBYTE *)
- int CoTBaddto (UBYTE *)
- int CoTBaudit (UBYTE *)
- int CoTBcleanup (UBYTE *)
- int CoTBcreate (UBYTE *)
- int CoTBenter (UBYTE *)
- int CoTBhelp (UBYTE *)
- int CoTBload (UBYTE *)
- int CoTBoff (UBYTE *)
- int CoTBon (UBYTE *)
- int CoTBopen (UBYTE *)
- int CoTBreplace (UBYTE *)
- int CoTBuse (UBYTE *)
- int CoTestUse (UBYTE *)
- int CoThreadBucket (UBYTE *)
- int AddComString (int, WORD *, UBYTE *, int)
- int CompileAlgebra (UBYTE *, int, WORD *)
- int IsIdStatement (UBYTE *)
- UBYTE * IsRHS (UBYTE *, UBYTE)
- int ParenthesesTest (UBYTE *)
- int tokenize (UBYTE *, WORD)
- void WriteTokens (SBYTE *)
- int simp1token (SBYTE *)
- int simpwtoken (SBYTE *)
- int simp2token (SBYTE *)
- int simp3atoken (SBYTE *, int)
- int simp3btoken (SBYTE *, int)
- int simp4token (SBYTE *)
- int simp5token (SBYTE *, int)
- int simp6token (SBYTE *, int)
- UBYTE * SkipAName (UBYTE *)
- int TestTables (void)
- int GetLabel (UBYTE *)
- int CoIdExpression (UBYTE *, int)
- int CoAssign (UBYTE *)
- int DoExpr (UBYTE *, int, int)
- int CompileSubExpressions (SBYTE *)
- int CodeGenerator (SBYTE *)
- int CompleteTerm (WORD *, UWORD *, UWORD *, WORD, WORD, int)
- int CodeFactors (SBYTE *s)
- WORD GenerateFactors (WORD, WORD)
- int InsTree (int, int)
- int FindTree (int, WORD *)
- void RedoTree (CBUF *, int)
- void ClearTree (int)
- int CatchDollar (int)
- int AssignDollar (PHEAD WORD *, WORD)
- UBYTE * WriteDollarToBuffer (WORD, WORD)
- UBYTE * WriteDollarFactorToBuffer (WORD, WORD, WORD)
- void AddToDollarBuffer (UBYTE *)
- int PutTermInDollar (WORD *, WORD)
- void TermAssign (WORD *)

- void WildDollars (PHEAD WORD *)
- LONG numcommute (WORD *, LONG *)
- int FullRenumber (PHEAD WORD *, WORD)
- int Lus (WORD *, WORD, WORD, WORD, WORD, WORD)
- int FindLus (int, int, int)
- int CoReplaceLoop (UBYTE *)
- int CoFindLoop (UBYTE *)
- int DoFindLoop (UBYTE *, int)
- int CoFunPowers (UBYTE *)
- int SortTheList (int *, int)
- int MatchIsPossible (WORD *, WORD *)
- int StudyPattern (WORD *)
- WORD DolToTensor (PHEAD WORD)
- WORD DolToFunction (PHEAD WORD)
- WORD DolToVector (PHEAD WORD)
- WORD DolToNumber (PHEAD WORD)
- WORD DolToSymbol (PHEAD WORD)
- WORD DolToIndex (PHEAD WORD)
- LONG DolToLong (PHEAD WORD)
- int DollarFactorize (PHEAD WORD)
- int CoPrintTable (UBYTE *)
- int CoDeallocateTable (UBYTE *)
- void CleanDollarFactors (DOLLARS)
- WORD * TakeDollarContent (PHEAD WORD *, WORD **)
- WORD * MakeDollarInteger (PHEAD WORD *, WORD **)
- WORD * MakeDollarMod (PHEAD WORD *, WORD **)
- int GetDolNum (PHEAD WORD *, WORD *)
- void AddPotModdollar (WORD)
- int Optimize (WORD, int)
- int ClearOptimize (void)
- int TakeLongRoot (UWORD *, WORD *, WORD)
- int TakeRatRoot (UWORD *, WORD *, WORD)
- int MakeRational (WORD, WORD, WORD *, WORD *)
- int MakeLongRational (PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- void ClearTableTree (TABLES)
- int InsTableTree (TABLES, WORD *)
- void RedoTableTree (TABLES, int)
- int FindTableTree (TABLES, WORD *, int)
- void finishcbuf (WORD)
- void clearcbuf (WORD)
- void CleanUpSort (int)
- FILEHANDLE * AllocFileHandle (WORD, char *)
- void DeAllocFileHandle (FILEHANDLE *)
- void LowerSortLevel (void)
- WORD * PolyRatFunSpecial (PHEAD WORD *, WORD *)
- void SimpleSplitMergeRec (WORD *, WORD, WORD *)
- void SimpleSplitMerge (WORD *, WORD)
- WORD BinarySearch (WORD *, WORD, WORD)
- int InsideDollar (PHEAD WORD *, WORD)
- DOLLARS DolToTerms (PHEAD WORD)
- WORD EvalDoLoopArg (PHEAD WORD *, WORD)
- int SetExprCases (int, int, int)
- int TestSelect (WORD *, WORD *)
- void SubsInAll (PHEAD0)
- void TransferBuffer (int, int, int)

- int TakeIDfunction (PHEAD WORD *)
- int MakeSetupAllocs (void)
- NORMDATA * AllocNormData (void)
- int TryFileSetups (void)
- void ExchangeExpressions (int, int)
- void ExchangeDollars (int, int)
- int GetFirstBracket (WORD *, int)
- int GetFirstTerm (WORD *, int, int)
- int GetContent (WORD *, int)
- int CleanupTerm (WORD *)
- WORD ContentMerge (PHEAD WORD *, WORD *)
- UBYTE * PreIfDollarEval (UBYTE *, int *)
- LONG TermsInDollar (WORD)
- LONG SizeOfDollar (WORD)
- LONG TermsInExpression (WORD)
- LONG SizeOfExpression (WORD)
- WORD * TranslateExpression (UBYTE *)
- int IsSetMember (WORD *, WORD)
- int IsMultipleOf (WORD *, WORD *)
- int TwoExprCompare (WORD *, WORD *, int)
- void UpdatePositions (void)
- void **M_check** (void)
- void M_print (void)
- void M_check1 (void)
- void PrintTime (UBYTE *)
- POSITION * FindBracket (WORD, WORD *)
- void PutBracketInIndex (PHEAD WORD *, POSITION *)
- void ClearBracketIndex (WORD)
- void OpenBracketIndex (WORD)
- int DoNoParallel (UBYTE *)
- int DoParallel (UBYTE *)
- int DoModSum (UBYTE *)
- int DoModMax (UBYTE *)
- int DoModMin (UBYTE *)
- int DoModLocal (UBYTE *)
- UBYTE * DoModDollar (UBYTE *, int)
- int DoProcessBucket (UBYTE *)
- int DoinParallel (UBYTE *)
- int DonotinParallel (UBYTE *)
- int FlipTable (FUNCTIONS, int)
- int ChainIn (PHEAD WORD *, WORD)
- int ChainOut (PHEAD WORD *, WORD)
- int ArgumentImplode (PHEAD WORD *, WORD *)
- int ArgumentExplode (PHEAD WORD *, WORD *)
- int DenToFunction (WORD *, WORD)
- WORD HowMany (PHEAD WORD *, WORD *)
- void RemoveDollars (void)
- LONG CountTerms1 (PHEAD0)
- LONG TermsInBracket (PHEAD WORD *, WORD)
- int Crash (void)
- char * str_dup (char *)
- void convertblock (INDEXBLOCK *, INDEXBLOCK *, int)
- void convertnamesblock (NAMESBLOCK *, NAMESBLOCK *, int)
- void convertiniinfo (INIINFO *, INIINFO *, int)
- int ReadIndex (DBASE *)

- int WriteIndexBlock (DBASE *, MLONG)
- int WriteNamesBlock (DBASE *, MLONG)
- int WriteIndex (DBASE *)
- int WriteIniInfo (DBASE *)
- int ReadIniInfo (DBASE *)
- int AddToIndex (DBASE *, MLONG)
- DBASE * GetDbase (char *, MLONG)
- DBASE * OpenDbase (char *)
- char * ReadObject (DBASE *, MLONG, char *)
- char * ReadijObject (DBASE *, MLONG, MLONG, char *)
- int ExistsObject (DBASE *, MLONG, char *)
- int DeleteObject (DBASE *, MLONG, char *)
- int WriteObject (DBASE *, MLONG, char *, char *, MLONG)
- MLONG AddObject (DBASE *, MLONG, char *, char *)
- DBASE * NewDbase (char *, MLONG)
- void FreeTableBase (DBASE *)
- int ComposeTableNames (DBASE *)
- int PutTableNames (DBASE *)
- MLONG AddTableName (DBASE *, char *, TABLES)
- MLONG GetTableName (DBASE *, char *)
- MLONG FindTableNumber (DBASE *, char *)
- int TryEnvironment (void)
- int CopyExpression (FILEHANDLE *, FILEHANDLE *)
- int set_in (UBYTE, set_of_char)
- one_byte set_set (UBYTE, set_of_char)
- one_byte set_del (UBYTE, set_of_char)
- one_byte set_sub (set_of_char, set_of_char, set_of_char)
- int DoPreAddSeparator (UBYTE *)
- int DoPreRmSeparator (UBYTE *)
- int openExternalChannel (UBYTE *, int, UBYTE *, UBYTE *)
- int initPresetExternalChannels (UBYTE *, int)
- int closeExternalChannel (int)
- int selectExternalChannel (int)
- int getCurrentExternalChannel (void)
- void closeAllExternalChannels (void)
- UBYTE * defineChannel (UBYTE *, HANDLERS *)
- int writeToChannel (int, UBYTE *, HANDLERS *)
- int **writeBufToExtChannelOk** (char *, size_t)
- int **getcFromExtChannelOk** (void)
- int **setKillModeForExternalChannelOk** (int, int)
- int **setTerminatorForExternalChannelOk** (char *)
- int **getcFromExtChannelFailure** (void)
- int **setKillModeForExternalChannelFailure** (int, int)
- int **setTerminatorForExternalChannelFailure** (char *)
- int **writeBufToExtChannelFailure** (char *, size_t)
- int ReleaseTB (void)
- int SymbolNormalize (WORD *)
- int TestFunFlag (PHEAD WORD *)
- WORD CompareSymbols (PHEAD WORD *, WORD *, WORD)
- WORD CompareHSymbols (PHEAD WORD *, WORD *, WORD)
- WORD NextPrime (PHEAD WORD)
- WORD Moebius (PHEAD WORD)
- UWORD wranf (PHEAD0)
- UWORD iranf (PHEAD UWORD)
- void iniwranf (PHEAD0)

- UBYTE * PreRandom (UBYTE *)
- WORD * **PolyNormPoly** (PHEAD WORD)
- WORD * **EvaluateGcd** (PHEAD WORD *)
- int TreatPolyRatFun (PHEAD WORD *)
- int ReadSaveHeader (void)
- LONG ReadSaveIndex (FILEINDEX *)
- LONG ReadSaveExpression (UBYTE *, UBYTE *, LONG *, LONG *)
- UBYTE * ReadSaveTerm32 (UBYTE *, UBYTE *, UBYTE **, UBYTE *, UBYTE *, int)
- LONG ReadSaveVariables (UBYTE *, UBYTE *, LONG *, LONG *, INDEXENTRY *, LONG *)
- LONG WriteStoreHeader (WORD)
- void InitRecovery (void)
- int CheckRecoveryFile (void)
- void DeleteRecoveryFile (void)
- char * RecoveryFilename (void)
- int DoRecovery (int *)
- void DoCheckpoint (int)
- void NumberMallocAddMemory (PHEAD0)
- void CacheNumberMallocAddMemory (PHEAD0)
- void TermMallocAddMemory (PHEAD0)
- void ExprStatus (EXPRESSIONS)
- void iniTools (void)
- int TestTerm (WORD *)
- int RunTransform (PHEAD WORD *term, WORD *params)
- int RunEncode (PHEAD WORD *fun, WORD *args, WORD *info)
- int RunDecode (PHEAD WORD *fun, WORD *args, WORD *info)
- int RunReplace (PHEAD WORD *fun, WORD *args, WORD *info)
- int RunImplode (WORD *fun, WORD *args)
- int RunExplode (PHEAD WORD *fun, WORD *args)
- int TestArgNum (int n, int totarg, WORD *args)
- WORD PutArgInScratch (WORD *arg, UWORD *scrat)
- UBYTE * ReadRange (UBYTE *s, WORD *out, int par)
- int FindRange (PHEAD WORD *, WORD *, WORD *, WORD)
- int RunPermute (PHEAD WORD *fun, WORD *args, WORD *info)
- int RunReverse (PHEAD WORD *fun, WORD *args)
- int RunCycle (PHEAD WORD *fun, WORD *args, WORD *info)
- int RunAddArg (PHEAD WORD *fun, WORD *args)
- int RunMulArg (PHEAD WORD *fun, WORD *args)
- int RunIsLyndon (PHEAD WORD *fun, WORD *args, int par)
- WORD RunToLyndon (PHEAD WORD *fun, WORD *args, int par)
- int RunDropArg (PHEAD WORD *fun, WORD *args)
- int RunSelectArg (PHEAD WORD *fun, WORD *args)
- int RunDedup (PHEAD WORD *fun, WORD *args)
- int RunZtoHArg (PHEAD WORD *fun, WORD *args)
- int RunHtoZArg (PHEAD WORD *fun, WORD *args)
- int NormPolyTerm (PHEAD WORD *)
- WORD **ComparePoly** (WORD *, WORD *, WORD)
- int ConvertToPoly (PHEAD WORD *, WORD *, WORD *, WORD)
- int LocalConvertToPoly (PHEAD WORD *, WORD *, WORD, WORD)
- int ConvertFromPoly (PHEAD WORD *, WORD *, WORD, WORD, WORD, WORD)
- int FindSubterm (WORD *)
- int FindLocalSubterm (PHEAD WORD *, WORD)
- void PrintSubtermList (int, int)
- void PrintExtraSymbol (int, WORD *, int)
- int FindSubexpression (WORD *)
- void UpdateMaxSize (void)

- int CoToPolynomial (UBYTE *)
- int CoFromPolynomial (UBYTE *)
- int CoArgToExtraSymbol (UBYTE *)
- int CoExtraSymbols (UBYTE *)
- UBYTE * GetDoParam (UBYTE *, WORD **, int)
- WORD * GetIfDollarFactor (UBYTE **, WORD *)
- int CoDo (UBYTE *)
- int CoEndDo (UBYTE *)
- int ExtraSymFun (PHEAD WORD *, WORD)
- int PruneExtraSymbols (WORD)
- int IniFbuffer (WORD)
- void **IniFbufs** (void)
- int GCDfunction (PHEAD WORD *, WORD)
- WORD * GCDfunction3 (PHEAD WORD *, WORD *)
- int ReadPolyRatFun (PHEAD WORD *)
- int FromPolyRatFun (PHEAD WORD *, WORD **, WORD **)
- WORD * PolyDiv (PHEAD WORD *, WORD *, char *)
- void GCDclean (PHEAD WORD *, WORD *)
- WORD * TakeSymbolContent (PHEAD WORD *, WORD *)
- int GCDterms (PHEAD WORD *, WORD *, WORD *)
- WORD * PutExtraSymbols (PHEAD WORD *, WORD, int *)
- WORD * TakeExtraSymbols (PHEAD WORD *, WORD)
- WORD * MultiplyWithTerm (PHEAD WORD *, WORD *, WORD)
- WORD * TakeContent (PHEAD WORD *, WORD *)
- int MergeSymbolLists (PHEAD WORD *, WORD *, int)
- int MergeDotproductLists (PHEAD WORD *, WORD *, int)
- WORD * CreateExpression (PHEAD WORD)
- int DIVfunction (PHEAD WORD *, WORD, int)
- WORD * MULfunc (PHEAD WORD *, WORD *)
- WORD * ConvertArgument (PHEAD WORD *, int *)
- int ExpandRat (PHEAD WORD *)
- int InvPoly (PHEAD WORD *, WORD, WORD)
- WORD TestDoLoop (PHEAD WORD *, WORD)
- WORD TestEndDoLoop (PHEAD WORD *, WORD)
- WORD * poly_gcd (PHEAD WORD *, WORD *, WORD)
- WORD * poly_div (PHEAD WORD *, WORD *, WORD)
- WORD * poly_rem (PHEAD WORD *, WORD *, WORD)
- WORD * poly_inverse (PHEAD WORD *, WORD *)
- WORD * poly_mul (PHEAD WORD *, WORD *)
- WORD * poly_ratfun_add (PHEAD WORD *, WORD *)
- int poly_ratfun_normalize (PHEAD WORD *)
- int poly_factorize_argument (PHEAD WORD *, WORD *)
- WORD * poly_factorize_dollar (PHEAD WORD *)
- int poly_factorize_expression (EXPRESSIONS)
- int poly_unfactorize_expression (EXPRESSIONS)
- void poly_free_poly_vars (PHEAD const char *)
- void optimize_print_code (int)
- int DoPreAdd (UBYTE *s)
- int DoPreUseDictionary (UBYTE *s)
- int DoPreCloseDictionary (UBYTE *s)
- int DoPreOpenDictionary (UBYTE *s)
- void RemoveDictionary (DICTIONARY *dict)
- void UnSetDictionary (void)
- int SetDictionaryOptions (UBYTE *options)
- int AddToDictionary (DICTIONARY *dict, UBYTE *left, UBYTE *right)

- int AddDictionary (UBYTE *name)
- int FindDictionary (UBYTE *name)
- UBYTE * IsExponentSign (void)
- UBYTE * IsMultiplySign (void)
- void TransformRational (UWORD *a, WORD na)
- void WriteDictionary (DICTIONARY *)
- void ShrinkDictionary (DICTIONARY *)
- void MultiplyToLine (void)
- UBYTE * FindSymbol (WORD num)
- UBYTE * FindVector (WORD num)
- UBYTE * FindIndex (WORD num)
- UBYTE * FindFunction (WORD num)
- UBYTE * FindFunWithArgs (WORD *t)
- UBYTE * FindExtraSymbol (WORD num)
- LONG DictToBytes (DICTIONARY *dict, UBYTE *buf)
- DICTIONARY * DictFromBytes (UBYTE *buf)
- int CoCreateSpectator (UBYTE *inp)
- int CoToSpectator (UBYTE *inp)
- int CoRemoveSpectator (UBYTE *inp)
- int CoEmptySpectator (UBYTE *inp)
- int CoCopySpectator (UBYTE *inp)
- int PutInSpectator (WORD *, WORD)
- void ClearSpectators (WORD)
- WORD GetFromSpectator (WORD *, WORD)
- void FlushSpectators (void)
- WORD * **PreGCD** (PHEAD WORD *, WORD *, int)
- int ReadFromScratch (FILEHANDLE *, POSITION *, UBYTE *, POSITION *)
- int AddToScratch (FILEHANDLE *, POSITION *, UBYTE *, POSITION *, int)
- int DoPreAppendPath (UBYTE *)
- int DoPrePrependPath (UBYTE *)
- int DoSwitch (PHEAD WORD *, WORD *)
- int DoEndSwitch (PHEAD WORD *, WORD *)
- SWITCHTABLE * FindCase (WORD, WORD)
- void SwitchSplitMergeRec (SWITCHTABLE *, WORD, SWITCHTABLE *)
- void SwitchSplitMerge (SWITCHTABLE *, WORD)
- int DoubleSwitchBuffers (void)
- int DistrN (int, int *, int, int *)
- int DoNamespace (UBYTE *)
- int DoEndNamespace (UBYTE *)
- int DoUse (UBYTE *)
- UBYTE * SkipName (UBYTE *)
- UBYTE * ConstructName (UBYTE *, UBYTE)
- int DoSetUserFlag (UBYTE *)
- int DoClearUserFlag (UBYTE *)
- VERTEX * CreateVertex (MODEL *)
- UBYTE * ReadParticle (UBYTE *, VERTEX *, MODEL *, int)
- int CoModel (UBYTE *)
- int CoParticle (UBYTE *)
- int CoVertex (UBYTE *)
- int CoEndModel (UBYTE *)
- int LoadModel (MODEL *)
- int CoSetUserFlag (UBYTE *)
- int CoClearUserFlag (UBYTE *)
- int CoCreateAllLoops (UBYTE *)
- int CoCreateAllPaths (UBYTE *)

- int CoCreateAll (UBYTE *)
- int AllLoops (PHEAD WORD *, WORD)
- LONG StartLoops (PHEAD WORD *, WORD, LONG, WORD, WORD *, WORD, WORD *, WORD)
- LONG GenLoops (PHEAD WORD *, WORD, LONG, WORD, WORD *, WORD, WORD *, WORD)
- void LoopOutput (PHEAD WORD *, WORD, WORD *, WORD)
- int AllPaths (PHEAD WORD *, WORD)
- LONG GenPaths (PHEAD WORD *, WORD, LONG, WORD, WORD *, WORD, WORD *, WORD)
- void PathOutput (PHEAD WORD *, WORD, WORD *, WORD)

4.19.1 Detailed Description

Contains macros and function declarations.

Definition in file declare.h.

4.19.2 Macro Definition Documentation

4.19.2.1 MaX

```
#define MaX(
              x,
              y ) ((x) > (y) ?  (x):  (y))
```

Definition at line 40 of file declare.h.

4.19.2.2 MiN

```
#define MiN(
              x,
              y ) ((x) < (y) ?  (x):  (y))
```

Definition at line 41 of file declare.h.

4.19.2.3 ABS

```
#define ABS(
              x ) ( (x) < 0 ?  -(x):  (x) )
```

Definition at line 42 of file declare.h.

4.19.2.4 SGN

```
#define SGN(
              x ) ( (x) > 0 ?  1 :  (x) < 0 ?  -1 :  0 )
```

Definition at line 43 of file declare.h.

4.19.2.5 REDLENG

```
#define REDLENG(
              x ) ((((x)<0)?((x)+1):((x)-1))/2)
```

Definition at line 44 of file declare.h.

4.19.2.6 INCLENG

```
#define INCLENG(
              x ) (((x)<0)?(((x)*2)-1):(((x)*2)+1))
```

Definition at line 45 of file declare.h.

4.19.2.7 GETCOEF

```
#define GETCOEF(
              x,
              y ) x += *x;y = x[-1];x -= ABS(y);y=REDLENG(y)
```

Definition at line 46 of file declare.h.

4.19.2.8 GETSTOP

```
#define GETSTOP(
              x,
              y ) y=x+(*x)-1;y -= ABS(*y)-1
```

Definition at line 47 of file declare.h.

4.19.2.9 StuffAdd

```
#define StuffAdd(
              x,
              y ) (((x)<0?-1:1)*(y)+((y)<0?-1:1)*(x))
```

Definition at line 48 of file declare.h.

4.19.2.10 EXCHN

```
#define EXCHN(
              t1,
              t2,
              n ) { WORD a,i; for(i=0;i<n;i++){a=t1[i];t1[i]=t2[i];t2[i]=a;} }
```

Definition at line 50 of file declare.h.

4.19.2.11 EXCH

```
#define EXCH(
              x,
              y ) { WORD a = (x); (x) = (y); (y) = a; }
```

Definition at line 51 of file declare.h.

4.19.2.12 TOKENTOLINE

```
#define TOKENTOLINE(
              x,
              y )
```

Value:

```
if ( AC.OutputSpaces == NOSPACEFORMAT ) { \
TokenToLine((UBYTE *)(y)); } else { TokenToLine((UBYTE *)(x)); }
```

Definition at line 53 of file declare.h.

4.19.2.13 UngetFromStream

```
#define UngetFromStream(
              stream,
              c ) ((stream)->nextchar[(stream)->isnextchar++]=c)
```

Definition at line 56 of file declare.h.

4.19.2.14 AddLineFeed

```
#define AddLineFeed(
              s,
              n ) { (s)[(n)++] = LINEFEED; }
```

Definition at line 60 of file declare.h.

4.19.2.15 TryRecover

```
#define TryRecover(
              x ) Terminate(-1)
```

Definition at line 62 of file declare.h.

4.19.2.16 UngetChar

```
#define UngetChar(
              c ) { pushbackchar = c; }
```

Definition at line 63 of file declare.h.

4.19.2.17 ParseNumber

```
#define ParseNumber(
              x,
              s ) {(x)=0;while(*(s)>='0'&&*(s)<='9')(x)=10*(x)+*(s)++ -'0';}
```

Definition at line 64 of file declare.h.

4.19.2.18 ParseSign

```
#define ParseSign(
              sgn,
              s )
```

Value:

```
{(sgn)=0;while(*(s)=='-'||*(s)=='+'){\
if ( *(s)++ == '-' ) sgn ^= 1;}}
```

Definition at line 65 of file declare.h.

4.19.2.19 ParseSignedNumber

```
#define ParseSignedNumber(
              x,
              s )
```

Value:

```
{ int sgn; ParseSign(sgn,s)\
ParseNumber(x,s) if ( sgn ) x = -x; }
```

Definition at line 67 of file declare.h.

4.19.2.20 NCOPY

```
#define NCOPY(
              s,
              t,
              n ) { while ( (n)-- > 0 ) { *s++ = *t++; } }
```

Definition at line 71 of file declare.h.

4.19.2.21 NCOPYI

```
#define NCOPYI(
              s,
              t,
              n ) { while ( (n)-- > 0 ) { *s++ = *t++; } }
```

Definition at line 72 of file declare.h.

4.19.2.22 NCOPYB

```
#define NCOPYB(
              s,
              t,
              n ) { while ( (n)-- > 0 ) { *s++ = *t++; } }
```

Definition at line 73 of file declare.h.

4.19.2.23 NCOPYI32

```
#define NCOPYI32(
              s,
              t,
              n ) { while ( (n)-- > 0 ) { *s++ = *t++; } }
```

Definition at line 74 of file declare.h.

4.19.2.24 WCOPY

```
#define WCOPY(
              s,
              t,
              n ) { int nn=n; WORD *ss=(WORD *)s, *tt=(WORD *)t; while ( (nn)-- > 0 ) { *ss++ =
*tt++; } }
```

Definition at line 75 of file declare.h.

4.19.2.25 NeedNumber

```
#define NeedNumber(
              x,
              s,
              err )
```

Value:

```
       { int sgn = 1;                                  \
       while ( *s == ' ' || *s == '\t' || *s == '-' || *s == '+' ) {       \
           if ( *s == '-' ) {sgn = -sgn;} s++; }                            \
       if ( chartype[*s] != 1 ) goto err;                                   \
       ParseNumber(x,s)                                                     \
       if ( sgn < 0 ) {(x) = -(x);} while ( *s == ' ' || *s == '\t' ) s++;\
   }
```

Definition at line 85 of file declare.h.

4.19.2.26 SKIPBLANKS

```
#define SKIPBLANKS(
              s ) { while ( *(s) == ' ' || *(s) == '\t' ) (s)++; }
```

Definition at line 92 of file declare.h.

4.19.2.27 FLUSHCONSOLE

```
#define FLUSHCONSOLE if ( AP.InOutBuf > 0 ) CharOut(LINEFEED)
```

Definition at line 93 of file declare.h.

4.19.2.28 SKIPBRA1

```
#define SKIPBRA1(
              s )
```

Value:

```
{ int lev1=0; s++; while(*s) { if(*s=='['){lev1++;} \
else {if(*s==']'&&--lev1<0){break;}} s++;} }
```

Definition at line 95 of file declare.h.

4.19.2.29 SKIPBRA2

```
#define SKIPBRA2(
              s )
```

Value:

```
{ int lev2=0; s++; while(*s) { if(*s=='{'){lev2++;} \
else {if(*s=='}'&&--lev2<0){break;} \
else {if(*s=='['){SKIPBRA1(s)}}} s++;} }
```

Definition at line 97 of file declare.h.

4.19.2.30 SKIPBRA3

```
#define SKIPBRA3(
              s )
```

Value:

```
{ int lev3=0; s++; while(*s) { if(*s=='('){lev3++;} \
else {if(*s==')'&&--lev3<0){break;} \
else {if(*s=='{'){SKIPBRA2(s)} \
else {if(*s=='['){SKIPBRA1(s)}}}} s++;} }
```

Definition at line 100 of file declare.h.

4.19.2.31 SKIPBRA4

```
#define SKIPBRA4(
              s )
```

Value:

```
{ int lev4=0; s++; while(*s) { if(*s=='('){lev4++;} \
else {if(*s==')'&&--lev4<0){break;} \
else {if(*s=='['){SKIPBRA1(s)}}} s++;} }
```

Definition at line 104 of file declare.h.

4.19.2.32 SKIPBRA5

```
#define SKIPBRA5(
                s )
```

Value:

```
{ int lev5=0; s++; while(*s) { if(*s=='{'){lev5++;} \
else {if(*s=='}'&&--lev5<0){break;} \
else {if(*s=='('){SKIPBRA4(s)} \
else {if(*s=='['){SKIPBRA1(s)}}}} s++;} }
```

Definition at line 107 of file declare.h.

4.19.2.33 CYCLE1

```
#define CYCLE1(
                t,
                a,
                i ) {t iX=*a; WORD jX; for(jX=1;jX<i;jX++)a[jX-1]=a[jX]; a[i-1]=iX;}
```

Definition at line 115 of file declare.h.

4.19.2.34 AddToCB

```
#define AddToCB(
                c,
                wx )
```

Value:

```
if(c->Pointer>=c->Top) \
{DoubleCbuffer(c->cbuf,c->Pointer,21);} \
*(c->Pointer)++ = wx;
```

Definition at line 117 of file declare.h.

4.19.2.35 EXCHINOUT

```
#define EXCHINOUT
```

Value:

```
{ FILEHANDLE *ffFi = AR.outfile; \
AR.outfile = AR.infile; AR.infile = ffFi; }
```

Definition at line 121 of file declare.h.

4.19.2.36 BACKINOUT

```
#define BACKINOUT
```

Value:

```
{ FILEHANDLE *ffFi = AR.outfile; POSITION posi; \
AR.outfile = AR.infile; AR.infile = ffFi; \
SetEndScratch(AR.infile,&posi); }
```

Definition at line 123 of file declare.h.

4.19.2.37 CopyArg

```
#define CopyArg(
                to,
                from )
```

Value:

```
{ if ( *from > 0 ) { int ica = *from; NCOPY(to,from,ica) } \
else if ( *from <= -FUNCTION ) *to++ = *from++;  \
else { *to++ = *from++; *to++ = *from++; } }
```

Definition at line 127 of file declare.h.

4.19.2.38 FILLARG

```
#define FILLARG(
                w )
```

Definition at line 136 of file declare.h.

4.19.2.39 COPYARG

```
#define COPYARG(
                w,
                t )
```

Definition at line 137 of file declare.h.

4.19.2.40 ZEROARG

```
#define ZEROARG(
                w )
```

Definition at line 138 of file declare.h.

4.19.2.41 FILLFUN

```
#define FILLFUN(
                w )
```

Definition at line 145 of file declare.h.

4.19.2.42 COPYFUN

```
#define COPYFUN(
                w,
                t )
```

Definition at line 146 of file declare.h.

4.19.2.43 COPYFUN3

```
#define COPYFUN3(
                w,
                t )
```

Definition at line 153 of file declare.h.

4.19.2.44 FILLFUN3

```
#define FILLFUN3(
                w )
```

Definition at line 154 of file declare.h.

4.19.2.45 FILLSUB

```
#define FILLSUB(
                w )
```

Definition at line 161 of file declare.h.

4.19.2.46 COPYSUB

```
#define COPYSUB(
                w,
                ww )
```

Definition at line 162 of file declare.h.

4.19.2.47 FILLEXPR

```
#define FILLEXPR(
                w )
```

Definition at line 168 of file declare.h.

4.19.2.48 NEXTARG

```
#define NEXTARG(
                x ) if(*x>0) x += *x; else if(*x <= -FUNCTION)x++; else x += 2;
```

Definition at line 171 of file declare.h.

4.19.2.49 COPY1ARG

```
#define COPY1ARG(
                s1,
                t1 )
```

Value:

```
        { int ica; if ( (ica=*t1) > 0 ) { NCOPY(s1,t1,ica) } \
        else if(*t1<=-FUNCTION){*s1++=*t1++;} else{*s1++=*t1++;*s1++=*t1++;} }
```

Definition at line 172 of file declare.h.

4.19.2.50 ZeroFillRange

```
#define ZeroFillRange(
                w,
                begin,
                end )
```

Value:

```
    do { \
    int tmp_i; \
    for ( tmp_i = begin; tmp_i < end; tmp_i++ ) { (w)[tmp_i] = 0; } \
} while (0)
```

Fills a buffer by zero in the range [begin,end).

Parameters

w	The buffer.
begin	The index for the beginning of the range.
end	The index for the end of the range (exclusive).

Definition at line 182 of file declare.h.

4.19.2.51 TABLESIZE

```
#define TABLESIZE(
                a,
                b ) (((WORD)sizeof(a))/((WORD)sizeof(b)))
```

Definition at line 187 of file declare.h.

4.19.2.52 WORDDIF

```
#define WORDDIF(
                x,
                y ) (WORD)(x-y)
```

Definition at line 188 of file declare.h.

4.19.2.53 wsizeof

```
#define wsizeof(
              a ) ((WORD)sizeof(a))
```

Definition at line 189 of file declare.h.

4.19.2.54 VARNAME

```
#define VARNAME(
              type,
              num ) (AC.varnames->namebuffer+type[num].name)
```

Definition at line 190 of file declare.h.

4.19.2.55 DOLLARNAME

```
#define DOLLARNAME(
              type,
              num ) (AC.dollarnames->namebuffer+type[num].name)
```

Definition at line 191 of file declare.h.

4.19.2.56 EXPRNAME

```
#define EXPRNAME(
              num ) (AC.exprnames->namebuffer+Expressions[num].name)
```

Definition at line 192 of file declare.h.

4.19.2.57 PREV

```
#define PREV(
              x ) prevorder?prevorder:x
```

Definition at line 194 of file declare.h.

4.19.2.58 Terminate

```
#define Terminate(
              x ) do { TerminateImpl(x, __FILE__, __LINE__, __FUNCTION__); } while(0)
```

Definition at line 196 of file declare.h.

4.19.2.59 SETERROR

```
#define SETERROR(
              x ) { Terminate(-1); return(-1); }
```

Definition at line 197 of file declare.h.

4.19.2.60 DUMMYUSE

```
#define DUMMYUSE(
              x ) (void)(x);
```

Definition at line 200 of file declare.h.

4.19.2.61 ADDPOS

```
#define ADDPOS(
              pp,
              x ) (pp).p1 = ((pp).p1+(LONG)(x))
```

Definition at line 226 of file declare.h.

4.19.2.62 SETBASELENGTH

```
#define SETBASELENGTH(
              ss,
              x ) (ss).p1 = (LONG)(x)
```

Definition at line 227 of file declare.h.

4.19.2.63 SETBASEPOSITION

```
#define SETBASEPOSITION(
              pp,
              x ) (pp).p1 = (LONG)(x)
```

Definition at line 228 of file declare.h.

4.19.2.64 ISEQUALPOSINC

```
#define ISEQUALPOSINC(
              pp1,
              pp2,
              x ) ( (pp1).p1 == ((pp2).p1+(LONG)(x)) )
```

Definition at line 229 of file declare.h.

4.19.2.65 ISGEPOSINC

```
#define ISGEPOSINC(
              pp1,
              pp2,
              x ) ( (pp1).p1 >= ((pp2).p1+(LONG)(x)) )
```

Definition at line 230 of file declare.h.

4.19.2.66 DIVPOS

```
#define DIVPOS(
              pp,
              n ) ( (pp).p1/(LONG)(n) )
```

Definition at line 231 of file declare.h.

4.19.2.67 MULPOS

```
#define MULPOS(
              pp,
              n ) (pp).p1 *= (LONG)(n)
```

Definition at line 232 of file declare.h.

4.19.2.68 DIFPOS

```
#define DIFPOS(
              ss,
              pp1,
              pp2 ) (ss).p1 = ((pp1).p1-(pp2).p1)
```

Definition at line 235 of file declare.h.

4.19.2.69 DIFBASE

```
#define DIFBASE(
              pp1,
              pp2 ) ((pp1).p1-(pp2).p1)
```

Definition at line 236 of file declare.h.

4.19.2.70 ADD2POS

```
#define ADD2POS(
              pp1,
              pp2 ) (pp1).p1 += (pp2).p1
```

Definition at line 237 of file declare.h.

4.19.2.71 PUTZERO

```
#define PUTZERO(
              pp ) (pp).p1 = 0
```

Definition at line 238 of file declare.h.

4.19.2.72 BASEPOSITION

```
#define BASEPOSITION(
              pp ) ((pp).p1)
```

Definition at line 239 of file declare.h.

4.19.2.73 SETSTARTPOS

```
#define SETSTARTPOS(
              pp ) (pp).p1 = -2
```

Definition at line 240 of file declare.h.

4.19.2.74 NOTSTARTPOS

```
#define NOTSTARTPOS(
              pp ) ( (pp).p1 > -2 )
```

Definition at line 241 of file declare.h.

4.19.2.75 ISMINPOS

```
#define ISMINPOS(
              pp ) ( (pp).p1 == -1 )
```

Definition at line 242 of file declare.h.

4.19.2.76 ISEQUALPOS

```
#define ISEQUALPOS(
              pp1,
              pp2 ) ( (pp1).p1 == (pp2).p1 )
```

Definition at line 243 of file declare.h.

4.19.2.77 ISNOTEQUALPOS

```
#define ISNOTEQUALPOS(
              pp1,
              pp2 ) ( (pp1).p1 != (pp2).p1 )
```

Definition at line 244 of file declare.h.

4.19.2.78 ISLESSPOS

```
#define ISLESSPOS(
              pp1,
              pp2 ) ( (pp1).p1 < (pp2).p1 )
```

Definition at line 245 of file declare.h.

4.19.2.79 ISGEPOS

```
#define ISGEPOS(
              pp1,
              pp2 ) ( (pp1).p1 >= (pp2).p1 )
```

Definition at line 246 of file declare.h.

4.19.2.80 ISNOTZEROPOS

```
#define ISNOTZEROPOS(
              pp ) ( (pp).p1 != 0 )
```

Definition at line 247 of file declare.h.

4.19.2.81 ISZEROPOS

```
#define ISZEROPOS(
              pp ) ( (pp).p1 == 0 )
```

Definition at line 248 of file declare.h.

4.19.2.82 ISPOSPOS

```
#define ISPOSPOS(
              pp ) ( (pp).p1 > 0 )
```

Definition at line 249 of file declare.h.

4.19.2.83 ISNEGPOS

```
#define ISNEGPOS(
              pp ) ( (pp).p1 < 0 )
```

Definition at line 250 of file declare.h.

4.19.2.84 TOLONG

```
#define TOLONG(
              x ) ((LONG)(x))
```

Definition at line 253 of file declare.h.

4.19.2.85 Add2Com

```
#define Add2Com(
              x ) { WORD cod[2]; cod[0] = x; cod[1] = 2; AddNtoL(2,cod); }
```

Definition at line 255 of file declare.h.

4.19.2.86 Add3Com

```
#define Add3Com(
              x1,
              x2 ) { WORD cod[3]; cod[0] = x1; cod[1] = 3; cod[2] = x2; AddNtoL(3,cod); }
```

Definition at line 256 of file declare.h.

4.19.2.87 Add4Com

```
#define Add4Com(
              x1,
              x2,
              x3 )
```

Value:

```
{ WORD cod[4]; cod[0] = x1; cod[1] = 4; \
cod[2] = x2; cod[3] = x3; AddNtoL(4,cod); }
```

Definition at line 257 of file declare.h.

4.19.2.88 Add5Com

```
#define Add5Com(
              x1,
              x2,
              x3,
              x4 )
```

Value:

```
{ WORD cod[5]; cod[0] = x1; cod[1] = 5; \
cod[2] = x2; cod[3] = x3; cod[4] = x4; AddNtoL(5,cod); }
```

Definition at line 259 of file declare.h.

4.19.2.89 WantAddPointers

```
#define WantAddPointers(
              x )
```

Value:

```
while((AT.pWorkPointer+(x))>AR.pWorkSize){WORD ***ppp=&AT.pWorkSpace;\
ExpandBuffer((void **)ppp,&AR.pWorkSize,(int)(sizeof(WORD *)));}
```

Definition at line 265 of file declare.h.

4.19.2.90 WantAddLongs

```
#define WantAddLongs(
              x )
```

Value:

```
    while((AT.lWorkPointer+(x))>AR.lWorkSize){LONG **ppp=&AT.lWorkSpace;\
    ExpandBuffer((void **)ppp,&AR.lWorkSize,sizeof(LONG));}
```

Definition at line 267 of file declare.h.

4.19.2.91 WantAddPositions

```
#define WantAddPositions(
              x )
```

Value:

```
    while((AT.posWorkPointer+(x))>AR.posWorkSize){POSITION **ppp=&AT.posWorkSpace;\
    ExpandBuffer((void **)ppp,&AR.posWorkSize,sizeof(POSITION));}
```

Definition at line 269 of file declare.h.

4.19.2.92 FORM_INLINE

```
#define FORM_INLINE inline
```

Definition at line 273 of file declare.h.

4.19.2.93 MEMORYMACROS

```
#define MEMORYMACROS
```

Definition at line 281 of file declare.h.

4.19.2.94 TermMalloc

```
#define TermMalloc(
              x ) ( (AT.TermMemTop <= 0 ) ?  TermMallocAddMemory(BHEAD0), AT.TermMemHeap[--AT.↩
TermMemTop]:  AT.TermMemHeap[--AT.TermMemTop] )
```

Definition at line 285 of file declare.h.

4.19.2.95 NumberMalloc

```
#define NumberMalloc(
              x ) ( (AT.NumberMemTop <= 0 ) ?  NumberMallocAddMemory(BHEAD0), AT.NumberMem↩
Heap[--AT.NumberMemTop]:  AT.NumberMemHeap[--AT.NumberMemTop] )
```

Definition at line 286 of file declare.h.

4.19.2.96 CacheNumberMalloc

```
#define CacheNumberMalloc(
              x ) ( (AT.CacheNumberMemTop <= 0 ) ?  CacheNumberMallocAddMemory(BHEAD0), AT.↩
CacheNumberMemHeap[--AT.CacheNumberMemTop]:  AT.CacheNumberMemHeap[--AT.CacheNumberMemTop] )
```

Definition at line 287 of file declare.h.

4.19.2.97 TermFree

```
#define TermFree(
              TermMem,
              x ) AT.TermMemHeap[AT.TermMemTop++] = (WORD *)(TermMem)
```

Definition at line 288 of file declare.h.

4.19.2.98 NumberFree

```
#define NumberFree(
              NumberMem,
              x ) AT.NumberMemHeap[AT.NumberMemTop++] = (UWORD *)(NumberMem)
```

Definition at line 289 of file declare.h.

4.19.2.99 CacheNumberFree

```
#define CacheNumberFree(
              NumberMem,
              x ) AT.CacheNumberMemHeap[AT.CacheNumberMemTop++] = (UWORD *)(NumberMem)
```

Definition at line 290 of file declare.h.

4.19.2.100 NestingChecksum

```
#define NestingChecksum( ) (AC.IfLevel + AC.RepLevel + AC.arglevel + AC.insidelevel + AC.↩
termlevel + AC.inexprlevel + AC.dolooplevel +AC.SwitchLevel)
```

Definition at line 318 of file declare.h.

4.19.2.101 MesNesting

```
#define MesNesting( ) MesPrint("&Illegal nesting of if, repeat, argument, inside, term, inexpression
and do")
```

Definition at line 319 of file declare.h.

4.19.2.102 MarkPolyRatFunDirty

```
#define MarkPolyRatFunDirty(
              T )
```

Value:

```
{if(*T&&AR.PolyFunType==2){WORD *TP,*TT;TT=T++*T;TT-=ABS(TT[-1]);\
TP=T+1;while(TP<TT){if(*TP==AR.PolyFun){TP[2]|=(DIRTYFLAG|MUSTCLEANPRF);}TP+=TP[1];}}}
```

Definition at line 321 of file declare.h.

4.19.2.103 PUSHPREASSIGNLEVEL

```
#define PUSHPREASSIGNLEVEL
```

Value:

```
    AP.PreAssignLevel++; { GETIDENTITY \
    if ( AP.PreAssignLevel >= AP.MaxPreAssignLevel ) { int i; \
        LONG *ap = (LONG *)Malloc1(2*AP.MaxPreAssignLevel*sizeof(LONG *),"PreAssignStack"); \
        for ( i = 0; i < AP.MaxPreAssignLevel; i++ ) ap[i] = AP.PreAssignStack[i]; \
        M_free(AP.PreAssignStack,"PreAssignStack"); \
        AP.MaxPreAssignLevel *= 2; AP.PreAssignStack = ap; \
    } \
    *AT.WorkPointer++ = AP.PreContinuation; AP.PreContinuation = 0; \
    AP.PreAssignStack[AP.PreAssignLevel] = AC.iPointer - AC.iBuffer; }
```

Definition at line 328 of file declare.h.

4.19.2.104 POPPREASSIGNLEVEL

```
#define POPPREASSIGNLEVEL
```

Value:

```
    if ( AP.PreAssignLevel > 0 ) { GETIDENTITY \
    AC.iPointer = AC.iBuffer + AP.PreAssignStack[AP.PreAssignLevel--]; \
    AP.PreContinuation = *--AT.WorkPointer; \
    *AC.iPointer = 0; }
```

Definition at line 338 of file declare.h.

4.19.2.105 EXTERNLOCK

```
#define EXTERNLOCK(
              x )
```

NOTE: We have replaced LOCK(ErrorMessageLock) and UNLOCK(ErrorMessageLock) by MLOCK(Error↩ MessageLock) and MUNLOCK(ErrorMessageLock). They are used for the synchronised output in ParFORM. (TU 28 May 2011)

Definition at line 490 of file declare.h.

4.19.2.106 INILOCK

```
#define INILOCK(
              x )
```

Definition at line 491 of file declare.h.

4.19.2.107 LOCK

```
#define LOCK(
              x )
```

Definition at line 492 of file declare.h.

4.19.2.108 UNLOCK

```
#define UNLOCK(
              x )
```

Definition at line 493 of file declare.h.

4.19.2.109 EXTERNRWLOCK

```
#define EXTERNRWLOCK(
              x )
```

Definition at line 494 of file declare.h.

4.19.2.110 INIRWLOCK

```
#define INIRWLOCK(
              x )
```

Definition at line 495 of file declare.h.

4.19.2.111 RWLOCKR

```
#define RWLOCKR(
              x )
```

Definition at line 496 of file declare.h.

4.19.2.112 RWLOCKW

```
#define RWLOCKW(
              x )
```

Definition at line 497 of file declare.h.

4.19.2.113 UNRWLOCK

```
#define UNRWLOCK(
              x )
```

Definition at line 498 of file declare.h.

4.19.2.114 MLOCK

```
#define MLOCK(
              x )
```

Definition at line 503 of file declare.h.

4.19.2.115 MUNLOCK

```
#define MUNLOCK(
              x )
```

Definition at line 504 of file declare.h.

4.19.2.116 GETIDENTITY

```
#define GETIDENTITY
```

Definition at line 506 of file declare.h.

4.19.2.117 GETBIDENTITY

```
#define GETBIDENTITY
```

Definition at line 507 of file declare.h.

4.19.2.118 M_alloc

```
#define M_alloc(
              x ) malloc((size_t)(x))
```

Definition at line 1018 of file declare.h.

4.19.2.119 CompareTerms

```
#define CompareTerms ((COMPARE)AR.CompareRoutine)
```

Definition at line 1488 of file declare.h.

4.19.2.120 FiniShuffle

```
#define FiniShuffle AN.SHvar.finishuf
```

Definition at line 1489 of file declare.h.

4.19.2.121 DoShtuffle

```
#define DoShtuffle ((DO_UFFLE)AN.SHvar.do_uffle)
```

Definition at line 1490 of file declare.h.

4.19.3 Typedef Documentation

4.19.3.1 WRITEBUFTOEXTCHANNEL

```
typedef int(* WRITEBUFTOEXTCHANNEL) (char *, size_t)
```

Definition at line 1481 of file declare.h.

4.19.3.2 GETCFROMEXTCHANNEL

```
typedef int(* GETCFROMEXTCHANNEL) (void)
```

Definition at line 1482 of file declare.h.

4.19.3.3 SETTERMINATORFOREXTERNALCHANNEL

```
typedef int(* SETTERMINATORFOREXTERNALCHANNEL) (char *)
```

Definition at line 1483 of file declare.h.

4.19.3.4 SETKILLMODEFOREXTERNALCHANNEL

```
typedef int(* SETKILLMODEFOREXTERNALCHANNEL) (int, int)
```

Definition at line 1484 of file declare.h.

4.19.3.5 WRITEFILE

```
typedef LONG(* WRITEFILE) (int, UBYTE *, LONG)
```

Definition at line 1485 of file declare.h.

4.19.3.6 GETTERM

```
typedef WORD(* GETTERM) (PHEAD WORD *)
```

Definition at line 1486 of file declare.h.

4.19.4 Function Documentation

4.19.4.1 TELLFILE()

```
void TELLFILE (
            int handle,
            POSITION * position )  [extern]
```

Definition at line 1410 of file tools.c.

4.19.4.2 StartVariables()

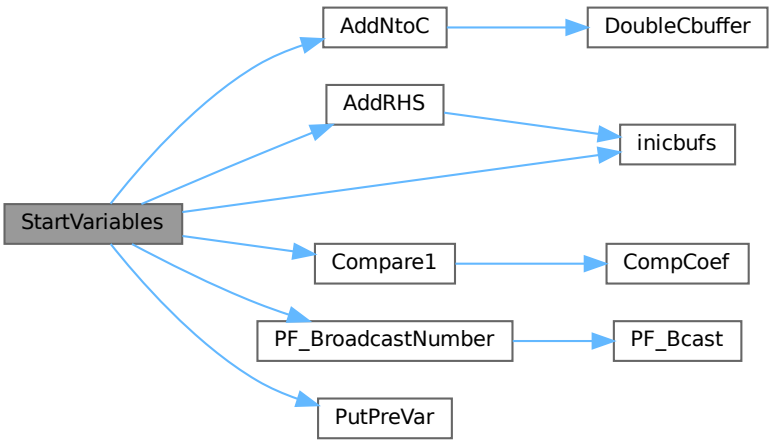
```
void StartVariables (
            void  )  [extern]
```

All functions (well, nearly all) are declared here.

Definition at line 952 of file startup.c.

References AddNtoC(), AddRHS(), Compare1(), inicbufs(), FuNcTiOn::name, PF_BroadcastNumber(), PutPreVar(), and FuNcTiOn::symmetric.

Here is the call graph for this function:



4.19.4.3 CodeToLine()

```
UBYTE * CodeToLine (
            WORD number,
            UBYTE * Out )  [extern]
```

Definition at line 640 of file sch.c.

4.19.4.4 AddArrayIndex()

```
UBYTE * AddArrayIndex (
            WORD num,
            UBYTE * out )  [extern]
```

Definition at line 678 of file sch.c.

4.19.4.5 FindInIndex()

```
INDEXENTRY * FindInIndex (
            WORD expr,
            FILEDATA * f,
            WORD par,
            WORD mode )  [extern]
```

Definition at line 2740 of file store.c.

4.19.4.6 NextFileIndex()

```
INDEXENTRY * NextFileIndex (
            POSITION * indexpos )  [extern]
```

Definition at line 2315 of file store.c.

4.19.4.7 PasteTerm()

```
WORD * PasteTerm (
            PHEAD WORD number,
            WORD * accum,
            WORD * position,
            WORD times,
            WORD divby )  [extern]
```

Puts the term at position in the accumulator accum at position 'number+1'. if times $>$ 0 the coefficient of this term is multiplied by times/divby.

Parameters

number	The number of term fragments in accum that should be skipped
accum	The accumulator of term fragments
position	A position in (typically) a compiler buffer from where a (piece of a) term comes.
times	Multiply the result by this
divby	Divide the result by this.

This routine is typically used when we have to replace a (sub)expression pointer by a power of a (sub)expression. This uses mostly a binomial expansion and the new term is the old term multiplied one by one by terms of the new expression. The factors times and divby keep track of the binomial coefficient. Once this is complete, the routine FiniTerm will make the contents of the accumulator into a proper term that still needs to be normalized.

Definition at line 3009 of file proces.c.

Referenced by Generator().

4.19.4.8 StrCopy()

```
UBYTE * StrCopy (
            UBYTE * from,
            UBYTE * to )  [extern]
```

Definition at line 49 of file sch.c.

4.19.4.9 WrtPower()

```
UBYTE * WrtPower (
            UBYTE * Out,
            WORD Power )  [extern]
```

Definition at line 720 of file sch.c.

4.19.4.10 AccumGCD()

```
int AccumGCD (
            PHEAD UWORD * a,
            WORD * na,
            UWORD * b,
            WORD nb )  [extern]
```

Definition at line 656 of file reken.c.

4.19.4.11 AddArgs()

```
void AddArgs (
            PHEAD WORD * s1,
            WORD * s2,
            WORD * m )  [extern]
```

Adds the arguments of two occurrences of the PolyFun.

Parameters

s1	Pointer to the first occurrence.
s2	Pointer to the second occurrence.
m	Pointer to where the answer should be.

Definition at line 2098 of file sort.c.

Referenced by AddPoly(), and MergePatches().

4.19.4.12 AddCoef()

```
int AddCoef (
            PHEAD WORD ** ps1,
            WORD ** ps2 )  [extern]
```

Adds the coefficients of the terms ∗ps1 and ∗ps2. The problem comes when there is not enough space for a new longer coefficient. First a local solution is tried. If this is not successful we need to move terms around. The possibility of a garbage collection should not be ignored, as avoiding this costs very much extra space which is nearly wasted otherwise.

If the return value is zero the terms cancelled.

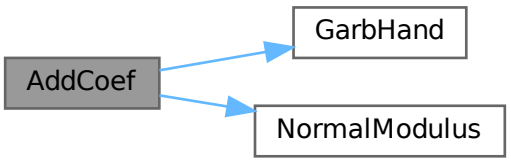
The resulting term is left in ∗ps1.

Definition at line 1795 of file sort.c.

References GarbHand(), and NormalModulus().

Referenced by SplitMerge().

Here is the call graph for this function:



4.19.4.13 AddLong()

```
int AddLong (
            UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )  [extern]
```

Definition at line 714 of file reken.c.

4.19.4.14 AddPLon()

```
int AddPLon (
            UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )  [extern]
```

Definition at line 759 of file reken.c.

4.19.4.15 AddPoly()

```
int AddPoly (
            PHEAD WORD ** ps1,
            WORD ** ps2 )  [extern]
```

Routine should be called when S->PolyWise != 0. It points then to the position of AR.PolyFun in both terms.

We add the contents of the arguments of the two polynomials. Special attention has to be given to special arguments. We have to reserve a space equal to the size of one term + the size of the argument of the other. The addition has to be done in this routine because not all objects are reentrant.

Newer addition (12-nov-2007). The PolyFun can have two arguments. In that case S->PolyFlag is 2 and we have to call the routine for adding rational polynomials. We have to be rather careful what happens with: The location of the output The order of the terms in the arguments At first we allow only univariate polynomials in the PolyFun. This restriction will be lifted a.s.a.p.

Parameters

ps1	A pointer to the position of the first term
ps2	A pointer to the position of the second term

Returns

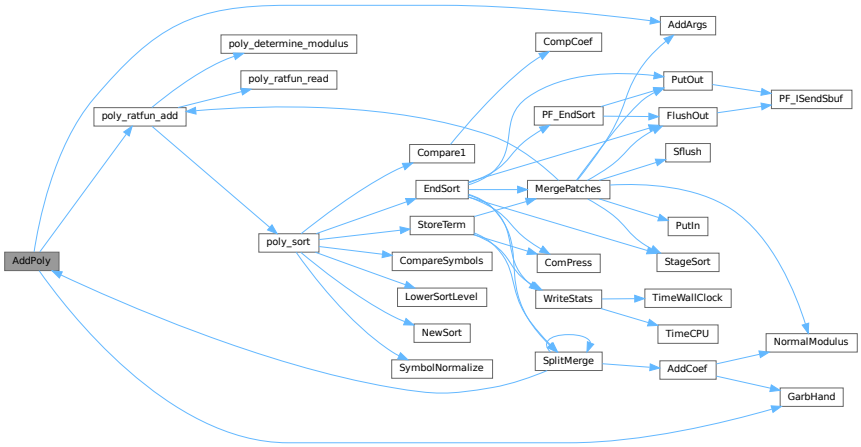
If zero the terms cancel. Otherwise the new term is in *ps1.

Definition at line 1925 of file sort.c.

References AddArgs(), GarbHand(), and poly_ratfun_add().

Referenced by SplitMerge().

Here is the call graph for this function:



4.19.4.16 AddRat()

```
int AddRat (
              PHEAD UWORD * a,
              WORD na,
              UWORD * b,
              WORD nb,
              UWORD * c,
              WORD * nc )  [extern]
```

Definition at line 216 of file reken.c.

4.19.4.17 AddToLine()

```
void AddToLine (
              UBYTE * s )  [extern]
```

Definition at line 64 of file sch.c.

4.19.4.18 AddWild()

```
int AddWild (
              PHEAD WORD,
              WORD type,
              WORD newnumber )  [extern]
```

Definition at line 1544 of file wildcard.c.

4.19.4.19 BigLong()

```
int BigLong (
              UWORD * a,
              WORD na,
              UWORD * b,
              WORD nb )  [extern]
```

Definition at line 953 of file reken.c.

4.19.4.20 BinomGen()

```
int BinomGen (
              PHEAD WORD * term,
              WORD level,
              WORD ** tstops,
              WORD x1,
              WORD x2,
              WORD pow1,
              WORD pow2,
              WORD sign,
              UWORD * coef,
              WORD ncoef )  [extern]
```

Definition at line 320 of file ratio.c.

4.19.4.21 CheckWild()

```
int CheckWild (
            PHEAD WORD,
            WORD type,
            WORD newnumber,
            WORD * newval )  [extern]
```

Definition at line 1797 of file wildcard.c.

4.19.4.22 Chisholm()

```
int Chisholm (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 1968 of file opera.c.

4.19.4.23 CleanExpr()

```
int CleanExpr (
            WORD par )  [extern]
```

Definition at line 47 of file execute.c.

4.19.4.24 CleanUp()

```
void CleanUp (
            WORD par )  [extern]
```

Definition at line 1834 of file startup.c.

4.19.4.25 ClearWild()

```
void ClearWild (
            PHEAD0  )  [extern]
```

Definition at line 1518 of file wildcard.c.

4.19.4.26 CompareFunctions()

```
int CompareFunctions (
            WORD * fleft,
            WORD * fright )  [extern]
```

Definition at line 55 of file normal.c.

4.19.4.27 Commute()

```
int Commute (
             WORD * fleft,
             WORD * fright )  [extern]
```

Definition at line 119 of file normal.c.

4.19.4.28 DetCommu()

```
WORD DetCommu (
             WORD * terms )  [extern]
```

Definition at line 4576 of file normal.c.

4.19.4.29 DoesCommu()

```
WORD DoesCommu (
             WORD * term )  [extern]
```

Definition at line 4635 of file normal.c.

4.19.4.30 CompArg()

```
int CompArg (
             WORD * s1,
             WORD * s2 )  [extern]
```

Definition at line 3253 of file tools.c.

4.19.4.31 CompCoef()

```
WORD CompCoef (
             WORD * term1,
             WORD * term2 )  [extern]
```

```
Routine takes a1 mod m1 and a2 mod m2 and returns a mod m1*m2 with
a mod m1 = a1 and a mod m2 = a2
```

Chinese remainder: a%(m1∗m2) = q1∗m1+a1 a%(m1∗m2) = q2∗m2+a2 Compute n1 such that (n1∗m1)m2 is one Compute n2 such that (n2∗m2)m1 is one Then (a1∗n2∗m2+a2∗n1∗m1)%(m1∗m2) is a%(m1∗m2)

Definition at line 3066 of file reken.c.

Referenced by Compare1().

4.19.4.32 CompGroup()

```
int CompGroup (
              PHEAD WORD,
              WORD ** args,
              WORD * a1,
              WORD * a2,
              WORD num )  [extern]
```

Definition at line 373 of file function.c.

4.19.4.33 Compare1()

```
WORD Compare1 (
              PHEAD WORD * term1,
              WORD * term2,
              WORD level )  [extern]
```

Compares two terms. The answer is: 0 equal (with exception of the coefficient if level == 0.) >0 term1 comes first. <0 term2 comes first. Some special precautions may be needed to keep the CompCoef routine from generating overflows, although this is very unlikely in subterms. This routine should not return an error condition.

Originally this routine was called Compare. With the treatment of special polynomials with terms that contain only symbols and the need for extreme speed for the polynomial routines we made a special compare routine and now we store the address of the current compare routine in AR.CompareRoutine and have a macro Compare which makes all existing code work properly and we can just replace the routine on a thread by thread basis (each thread has its own AR struct).

Parameters

term1	First input term
term2	Second input term
level	The sorting level (may influence on the result)

Returns

0 equal (with exception of the coefficient if level == 0.) >0 term1 comes first. <0 term2 comes first.

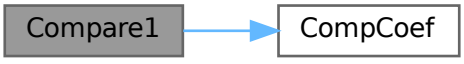
When there are floating point numbers active (float_ = FLOATFUN) the presence of one or more float_ functions is returned in AT.SortFloatMode: 0: no float_ 1: float_ in term1 only 2: float_ in term2 only 3: float_ in both terms

Definition at line 2393 of file sort.c.

References CompCoef().

Referenced by InFunction(), poly_sort(), and StartVariables().

Here is the call graph for this function:



4.19.4.34 CountDo()

```
WORD CountDo (
            WORD * term,
            WORD * instruct )  [extern]
```

Definition at line 554 of file reshuf.c.

4.19.4.35 CountFun()

```
WORD CountFun (
            WORD * term,
            WORD * countfun )  [extern]
```

Definition at line 708 of file reshuf.c.

4.19.4.36 DimensionSubterm()

```
WORD DimensionSubterm (
            WORD * subterm )  [extern]
```

Definition at line 869 of file reshuf.c.

4.19.4.37 DimensionTerm()

```
WORD DimensionTerm (
            WORD * term )  [extern]
```

Definition at line 970 of file reshuf.c.

4.19.4.38 DimensionExpression()

```
WORD DimensionExpression (
            PHEAD WORD * expr )  [extern]
```

Definition at line 1002 of file reshuf.c.

4.19.4.39 Deferred()

```
int Deferred (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Picks up the deferred brackets. These are the bracket contents of which we postpone the reading when we use the 'Keep Brackets' statement. These contents are multiplying the terms just before they are sent to the sorting system. Special attention goes to having it thread-safe We have to lock positioning the file and reading it in a thread specific buffer.

Parameters

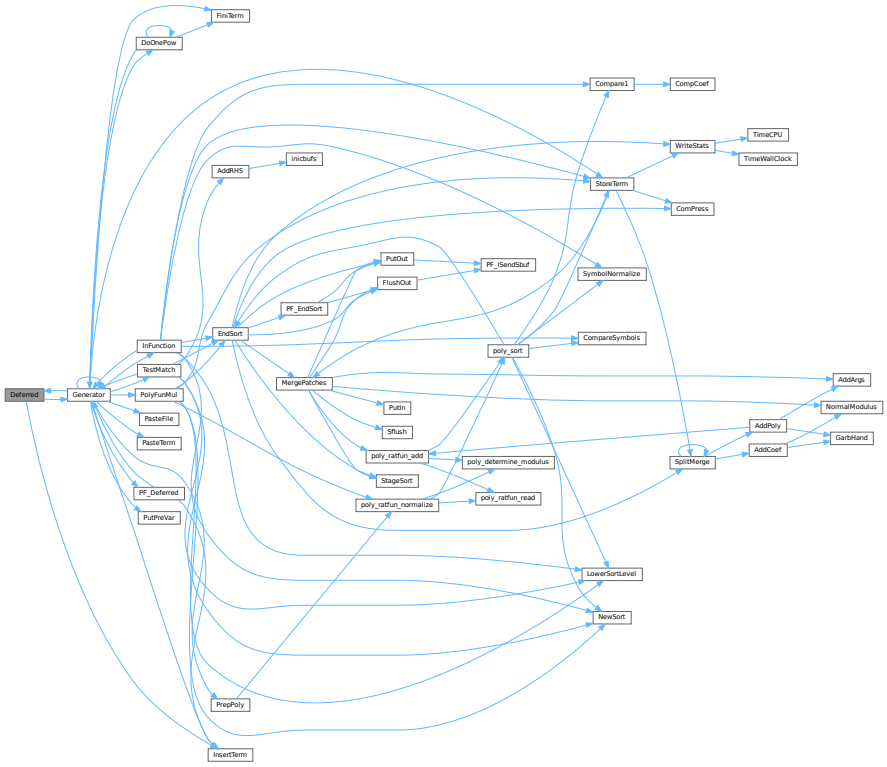
term	The term that must be multiplied by the contents of the current bracket
level	The compiler level. This is needed because after multiplying term by term we call Generator again.

Definition at line 4876 of file proces.c.

References Generator(), and InsertTerm().

Referenced by Generator().

Here is the call graph for this function:



4.19.4.40 DeleteStore()

```
int DeleteStore (
            WORD par )  [extern]
```

Definition at line 798 of file store.c.

4.19.4.41 DetCurDum()

```
WORD DetCurDum (
            PHEAD WORD * t )  [extern]
```

Definition at line 254 of file reshuf.c.

4.19.4.42 DetVars()

```
void DetVars (
            WORD * term,
            WORD par )  [extern]
```

Definition at line 1869 of file store.c.

4.19.4.43 Distribute()

```
int Distribute (
            DISTRIBUTE * d,
            WORD first )  [extern]
```

Definition at line 510 of file symmetr.c.

4.19.4.44 DivLong()

```
int DivLong (
            UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc,
            UWORD * d,
            WORD * nd )  [extern]
```

Definition at line 981 of file reken.c.

4.19.4.45 DivRat()

```
int DivRat (
            PHEAD UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )  [extern]
```

Definition at line 506 of file reken.c.

4.19.4.46 Divvy()

```
int Divvy (
            PHEAD UWORD * a,
            WORD * na,
            UWORD * b,
            WORD nb )  [extern]
```

Definition at line 176 of file reken.c.

4.19.4.47 DoDelta()

```
int DoDelta (
            WORD * t )  [extern]
```

Definition at line 4421 of file normal.c.

4.19.4.48 DoDelta3()

```
int DoDelta3 (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 1407 of file reshuf.c.

4.19.4.49 TestPartitions()

```
int TestPartitions (
            WORD * tfun,
            PARTI * parti )  [extern]
```

Definition at line 1611 of file reshuf.c.

4.19.4.50 DoPartitions()

```
int DoPartitions (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 1726 of file reshuf.c.

4.19.4.51 CoCanonicalize()

```
int CoCanonicalize (
            UBYTE * s )  [extern]
```

Definition at line 64 of file diagrams.c.

4.19.4.52 DoCanonicalize()

```
int DoCanonicalize (
            PHEAD WORD * term,
            WORD * params )  [extern]
```

Definition at line 185 of file diagrams.c.

4.19.4.53 GenDiagrams()

```
int GenDiagrams (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 806 of file diawrap.cc.

4.19.4.54 DoTopologyCanonicalize()

```
int DoTopologyCanonicalize (
            PHEAD WORD * term,
            WORD vert,
            WORD edge,
            WORD * args )  [extern]
```

Definition at line 346 of file diagrams.c.

4.19.4.55 DoShattering()

```
int DoShattering (
            PHEAD WORD * connect,
            WORD * environ,
            WORD * partitions,
            WORD nvert )  [extern]
```

Definition at line 555 of file diagrams.c.

4.19.4.56 DoTableExpansion()

```
int DoTableExpansion (
            WORD * term,
            WORD level )  [extern]
```

Definition at line 338 of file tables.c.

4.19.4.57 DoDistrib()

```
int DoDistrib (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 1121 of file reshuf.c.

4.19.4.58 DoShuffle()

```
int DoShuffle (
            WORD * term,
            WORD level,
            WORD fun,
            WORD option )  [extern]
```

Definition at line 2099 of file reshuf.c.

4.19.4.59 DoPermutations()

```
int DoPermutations (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 2011 of file reshuf.c.

4.19.4.60 Shuffle()

```
int Shuffle (
            WORD * from1,
            WORD * from2,
            WORD * to )  [extern]
```

Definition at line 2233 of file reshuf.c.

4.19.4.61 FinishShuffle()

```
int FinishShuffle (
            WORD * fini )  [extern]
```

Definition at line 2425 of file reshuf.c.

4.19.4.62 DoStuffle()

```
int DoStuffle (
            WORD * term,
            WORD level,
            WORD fun,
            WORD option )  [extern]
```

Definition at line 2491 of file reshuf.c.

4.19.4.63 Stuffle()

```
int Stuffle (
            WORD * from1,
            WORD * from2,
            WORD * to )  [extern]
```

Definition at line 2706 of file reshuf.c.

4.19.4.64 FinishStuffle()

```
int FinishStuffle (
            WORD * fini )  [extern]
```

Definition at line 2833 of file reshuf.c.

4.19.4.65 StuffRootAdd()

```
WORD * StuffRootAdd (
            WORD * t1,
            WORD * t2,
            WORD * to )  [extern]
```

Definition at line 2880 of file reshuf.c.

4.19.4.66 TestUse()

```
int TestUse (
            WORD * term,
            WORD level )  [extern]
```

Definition at line 1420 of file tables.c.

4.19.4.67 FindTB()

```
DBASE * FindTB (
            UBYTE * name )  [extern]
```

Definition at line 740 of file tables.c.

4.19.4.68 CheckTableDeclarations()

```
int CheckTableDeclarations (
            DBASE * d )  [extern]
```

Definition at line 1184 of file tables.c.

4.19.4.69 Apply()

```
void Apply (
            WORD * term,
            WORD level )  [extern]
```

Definition at line 1993 of file tables.c.

4.19.4.70 ApplyExec()

```
int ApplyExec (
            WORD * term,
            int maxtogo,
            WORD level )  [extern]
```

Definition at line 2051 of file tables.c.

4.19.4.71 ApplyReset()

```
void ApplyReset (
            WORD level )  [extern]
```

Definition at line 2268 of file tables.c.

4.19.4.72 TableReset()

```
void TableReset (
            void  )  [extern]
```

Definition at line 2303 of file tables.c.

4.19.4.73 ReWorkT()

```
void ReWorkT (
            WORD * term,
            WORD * funs,
            WORD numfuns )  [extern]
```

Definition at line 1912 of file tables.c.

4.19.4.74 GetIfDollarNum()

```
WORD GetIfDollarNum (
            WORD * ifp,
            WORD * ifstop )  [extern]
```

Definition at line 106 of file if.c.

4.19.4.75 FindVar()

```
int FindVar (
            WORD * v,
            WORD * term )  [extern]
```

Definition at line 173 of file if.c.

4.19.4.76 DoIfStatement()

```
int DoIfStatement (
            PHEAD WORD * ifcode,
            WORD * term )  [extern]
```

Definition at line 280 of file if.c.

4.19.4.77 DoOnePow()

```
int DoOnePow (
              PHEAD WORD * term,
              WORD power,
              WORD nexp,
              WORD * accum,
              WORD * aa,
              WORD level,
              WORD * freeze )  [extern]
```

Routine gets one power of an expression in the scratch system. If there are more powers needed there will be a recursion.

No attempt is made to use binomials because we have no information about commuting properties.

There is a searching for the contents of brackets if needed. This searching may be rather slow because of the single links.

Parameters

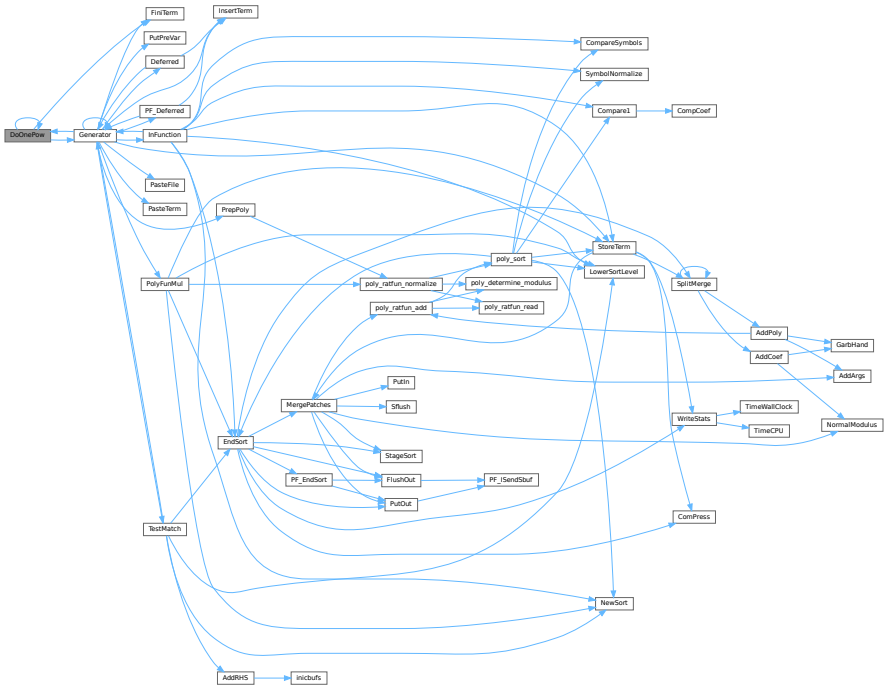
term	is the term we are adding to.
power	is the power of the expression that we need.
nexp	is the number of the expression.
accum	is the accumulator of terms. It accepts the termfragments that are made into a proper term in FiniTerm
aa	points to the start of the entire accumulator. In ∗aa we store the number of term fragments that are in the accumulator.
level	is the current depth in the tree of statements. It is needed to continue to the next operation/substitution with each generated term
freeze	is the pointer to the bracket information that should be matched.

Definition at line 4653 of file proces.c.

References DoOnePow(), FiniTerm(), Generator(), and FiLe::handle.

Referenced by DoOnePow(), and Generator().

Here is the call graph for this function:



4.19.4.78 DoRevert()

```
void DoRevert (
            WORD * fun,
            WORD * tmp )  [extern]
```

Definition at line 4488 of file normal.c.

4.19.4.79 DoSumF1()

```
int DoSumF1 (
            PHEAD WORD * term,
            WORD * params,
            WORD replac,
            WORD level )  [extern]
```

Definition at line 395 of file ratio.c.

4.19.4.80 DoSumF2()

```
int DoSumF2 (
            PHEAD WORD * term,
            WORD * params,
            WORD replac,
            WORD level )  [extern]
```

Definition at line 520 of file ratio.c.

4.19.4.81 DoTheta()

```
int DoTheta (
            PHEAD WORD * t )  [extern]
```

Definition at line 4324 of file normal.c.

4.19.4.82 EndSort()

```
LONG EndSort (
            PHEAD WORD * buffer,
            int par )  [extern]
```

Finishes a sort. At AR.sLevel == 0 the output is to the regular output stream. When AR.sLevel $>$ 0, the parameter par determines the actual output. The AR.sLevel will be popped. All ongoing stages are finished and if the sortfile is open it is closed. The statistics are printed when AR.sLevel == 0 par == 0 Output to the buffer. par == 1 Sort for function arguments. The output will be copied into the buffer. It is assumed that this is in the WorkSpace. par == 2 Sort for $-variable. We return the address of the buffer that contains the output in buffer (treated like WORD $**$). We first catch the output in a file (unless we can intercept things after the small buffer has been sorted) Then we read from the file into a buffer. Only when par == 0 data compression can be attempted at AT.SS==AT.S0.

Parameters

buffer	buffer for output when needed
par	See above

Returns

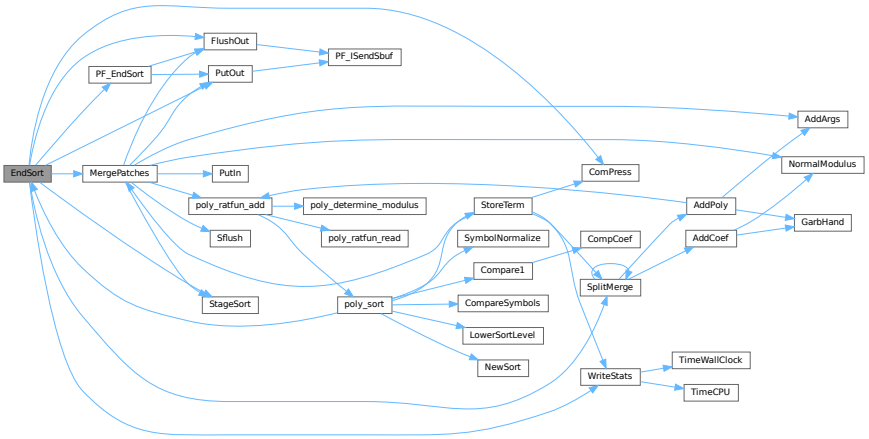
If negative: error. If positive: number of words in output.

Definition at line 488 of file sort.c.

References ComPress(), FlushOut(), MergePatches(), PF_EndSort(), PutOut(), SplitMerge(), StageSort(), and WriteStats().

Referenced by generate_expression(), get_expression(), InFunction(), MakeDollarInteger(), MakeDollarMod(), PF_CollectModifiedDollars(), PF_Processor(), poly_factorize_expression(), poly_sort(), poly_unfactorize_expression(), PolyFunMul(), Processor(), TakeArgContent(), and TestMatch().

Here is the call graph for this function:



4.19.4.83 EntVar()

```
int EntVar (
            WORD type,
            UBYTE * name,
            WORD x,
            WORD y,
            WORD z,
            WORD d )  [extern]
```

Definition at line 559 of file names.c.

4.19.4.84 EpfCon()

```
int EpfCon (
            PHEAD WORD * term,
            WORD * params,
            WORD num,
            WORD level )  [extern]
```

Definition at line 212 of file opera.c.

4.19.4.85 EpfFind()

```
int EpfFind (
            PHEAD WORD * term,
            WORD * params )  [extern]
```

Definition at line 58 of file opera.c.

4.19.4.86 EpfGen()

```
WORD EpfGen (
            WORD number,
            WORD * inlist,
            WORD * kron,
            WORD * perm,
            WORD sgn )  [extern]
```

Definition at line 284 of file opera.c.

4.19.4.87 EqualArg()

```
int EqualArg (
            WORD * parms,
            WORD num1,
            WORD num2 )  [extern]
```

Definition at line 1381 of file reshuf.c.

4.19.4.88 Factorial()

```
int Factorial (
            PHEAD WORD,
            UWORD * a,
            WORD * na )  [extern]
```

Definition at line 3470 of file reken.c.

4.19.4.89 Bernoulli()

```
int Bernoulli (
            WORD n,
            UWORD * a,
            WORD * na )  [extern]
```

Definition at line 3550 of file reken.c.

4.19.4.90 FactorIn()

```
int FactorIn (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 46 of file factor.c.

4.19.4.91 FactorInExpr()

```
int FactorInExpr (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 423 of file factor.c.

4.19.4.92 FindAll()

```
int FindAll (
            PHEAD WORD * term,
            WORD * pattern,
            WORD level,
            WORD * par )  [extern]
```

Definition at line 1199 of file pattern.c.

4.19.4.93 FindMulti()

```
WORD FindMulti (
            PHEAD WORD * term,
            WORD * pattern )  [extern]
```

Definition at line 954 of file findpat.c.

4.19.4.94 FindOnce()

```
int FindOnce (
            PHEAD WORD * term,
            WORD * pattern )  [extern]
```

Definition at line 419 of file findpat.c.

4.19.4.95 FindOnly()

```
int FindOnly (
            PHEAD WORD * term,
            WORD * pattern )  [extern]
```

Definition at line 61 of file findpat.c.

4.19.4.96 FindRest()

```
int FindRest (
            PHEAD WORD * term,
            WORD * pattern )  [extern]
```

Definition at line 1070 of file findpat.c.

4.19.4.97 FindrNumber()

```
WORD FindrNumber (
            WORD n,
            VARRENUM * v )  [extern]
```

Definition at line 2659 of file store.c.

4.19.4.98 FiniLine()

```
void FiniLine (
            void  )  [extern]
```

Definition at line 164 of file sch.c.

4.19.4.99 FiniTerm()

```
int FiniTerm (
            PHEAD WORD * term,
            WORD * accum,
            WORD * termout,
            WORD number,
            WORD tepos )  [extern]
```

Concatenates the contents of the accumulator into a single legal term, which replaces the subexpression pointer

Parameters

term	the input term with the (sub)expression subterm
accum	the accumulator with the term fragments
termout	the location where the output should be written
number	the number of term fragments in the accumulator
tepos	the position of the subterm in term to be replaced

Definition at line 3076 of file proces.c.

Referenced by DoOnePow(), and Generator().

4.19.4.100 FlushOut()

```
int FlushOut (
              POSITION * position,
              FILEHANDLE * fi,
              int compr )  [extern]
```

Completes output to an output file and writes the trailing zero.

Parameters

position	The position in the file after writing
fi	The file (or its cache)
compr	Indicates whether there should be compression with gzip.

Returns

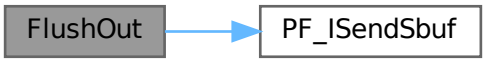
Regular conventions (OK -> 0).

Definition at line 1581 of file sort.c.

References FiLe::handle, BrAcKeTiNfO::indexbuffer, and PF_ISendSbuf().

Referenced by EndSort(), MergePatches(), PF_EndSort(), and Processor().

Here is the call graph for this function:



4.19.4.101 FunLevel()

```
void FunLevel (
            PHEAD WORD * term )  [extern]
```

Definition at line 130 of file reshuf.c.

4.19.4.102 AdjustRenumScratch()

```
void AdjustRenumScratch (
            PHEAD0  )  [extern]
```

Definition at line 508 of file reshuf.c.

4.19.4.103 GarbHand()

```
void GarbHand (
            void  )  [extern]
```

Garbage collection that takes place when the small extension is full and we need to place more terms there. When this is the case there are many holes in the small buffer and the whole can be compactified. The major complication is the buffer for SplitMerge. There are to options for temporary memory: 1: find some buffer that has enough space (maybe in the large buffer). 2: allocate a buffer. Give it back afterwards of course. If the small extension is properly dimensioned this routine should be called very rarely. Most of the time it will be called when the polyfun or polyratfun is active.

Definition at line 3405 of file sort.c.

Referenced by AddCoef(), and AddPoly().

4.19.4.104 GcdLong()

```
int GcdLong (
            PHEAD UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )  [extern]
```

Definition at line 2293 of file reken.c.

4.19.4.105 LcmLong()

```
int LcmLong (
            PHEAD UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )  [extern]
```

Definition at line 2720 of file reken.c.

4.19.4.106 Generator()

```
int Generator (
             PHEAD WORD * term,
             WORD level )  [extern]
```

The heart of the program. Here the expansion tree is set up in one giant recursion

Parameters

term	the input term. may be overwritten
level	the level in the compiler buffer (number of statement)

Returns

Normal conventions (OK = 0).

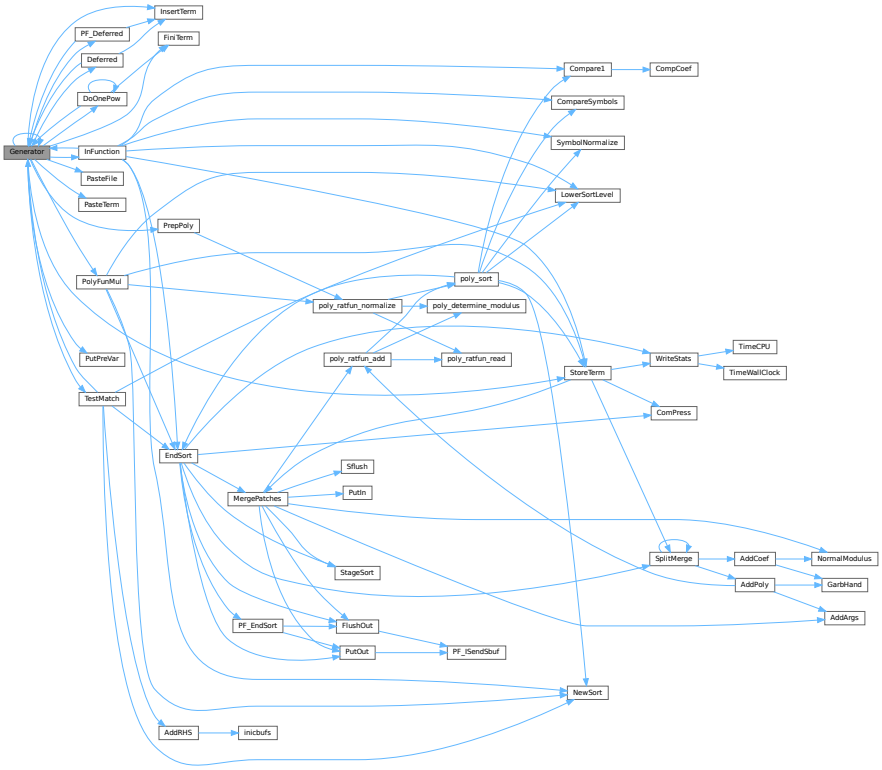
The routine looks first whether there are unsubstituted (sub)expressions. If so, one of them gets inserted term by term and the new term is used in a renewed call to Generator. If there are no (sub)expressions, the term is normalized, the compiler level is raised (next statement) and the program looks what type of statement this is. If this is a special statement it is either treated on the spot or the appropriate routine is called. If it is a substitution, the pattern matcher is called (TestMatch) which tells whether there was a match. If so we need to call TestSub again to test for (sub)expressions. If we run out of levels, the term receives a final treatment for modulus calculus and/or brackets and is then sent off to the sorting routines.

Definition at line 3275 of file proces.c.

References Deferred(), DoOnePow(), FiniTerm(), Generator(), InFunction(), InsertTerm(), CbUf::lhs, VaRrEnUm::lo, PasteFile(), PasteTerm(), PF_Deferred(), CbUf::Pointer, PolyFunMul(), PrepPoly(), PutPreVar(), CbUf::rhs, StoreTerm(), ReNuMbEr::symb, and TestMatch().

Referenced by Deferred(), DoOnePow(), generate_expression(), Generator(), InFunction(), MakeDollarInteger(), MakeDollarMod(), PF_CollectModifiedDollars(), PF_Deferred(), PF_Processor(), poly_factorize_expression(), poly_unfactorize_expression(), Processor(), and TestMatch().

Here is the call graph for this function:



4.19.4.107 GetBinom()

```
int GetBinom (
              UWORD * a,
              WORD * na,
              WORD i1,
              WORD i2 )  [extern]
```

Definition at line 2686 of file reken.c.

4.19.4.108 GetFromStore()

```
WORD GetFromStore (
              WORD * to,
              POSITION * position,
              RENUMBER renumber,
              WORD * InCompState,
              WORD nexpr )  [extern]
```

Definition at line 1660 of file store.c.

4.19.4.109 GetLong()

```
int GetLong (
              UBYTE * s,
              UWORD * a,
              WORD * na )  [extern]
```

Definition at line 1789 of file reken.c.

4.19.4.110 GetMoreTerms()

```
WORD GetMoreTerms (
              WORD * term )  [extern]
```

Definition at line 1457 of file store.c.

4.19.4.111 GetMoreFromMem()

```
int GetMoreFromMem (
              WORD * term,
              WORD ** tpoin )  [extern]
```

Definition at line 1553 of file store.c.

4.19.4.112 GetOneTerm()

```
WORD GetOneTerm (
            PHEAD WORD * term,
            FILEHANDLE * fi,
            POSITION * pos,
            int par )  [extern]
```

Definition at line 1297 of file store.c.

4.19.4.113 GetTable()

```
RENUMBER GetTable (
            WORD expr,
            POSITION * position,
            WORD mode )  [extern]
```

Definition at line 2893 of file store.c.

4.19.4.114 GetTerm()

```
WORD GetTerm (
            PHEAD WORD * term )  [extern]
```

Definition at line 1020 of file store.c.

4.19.4.115 Glue()

```
int Glue (
            PHEAD WORD * term1,
            WORD * term2,
            WORD * sub,
            WORD insert )  [extern]
```

Definition at line 461 of file ratio.c.

4.19.4.116 InFunction()

```
int InFunction (
            PHEAD WORD * term,
            WORD * termout )  [extern]
```

Makes the replacement of the subexpression with the number 'replac' in a function argument. Additional information is passed in some of the AR, AN, AT variables.

Parameters

term	The input term
termout	The output term

Returns

0: everything is fine, Negative: fatal, Positive: error.

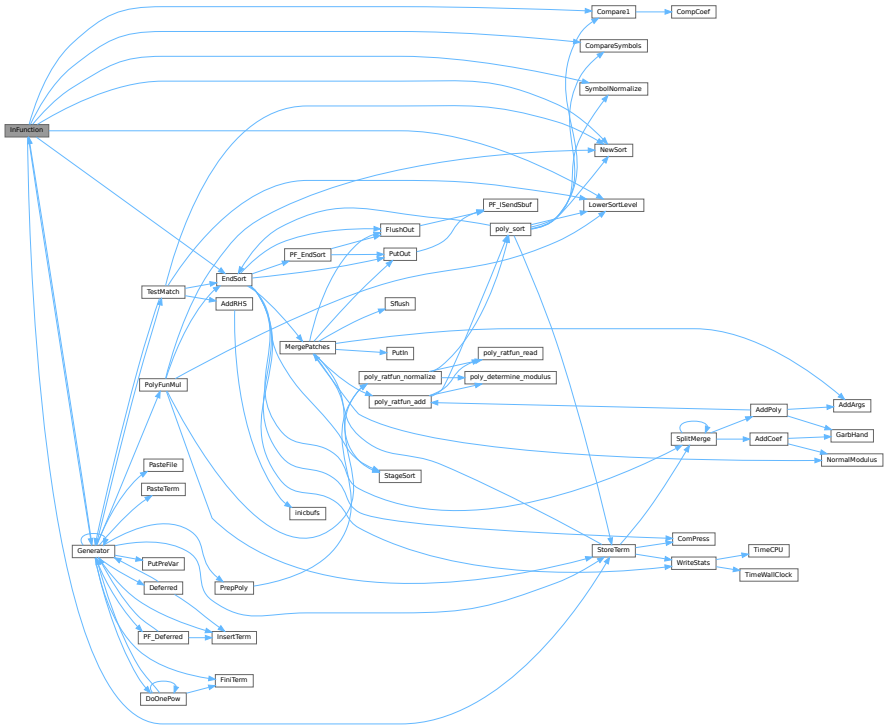
Special attention should be given to nested functions!

Definition at line 2180 of file proces.c.

References Compare1(), CompareSymbols(), EndSort(), Generator(), LowerSortLevel(), NewSort(), StoreTerm(), and SymbolNormalize().

Referenced by Generator().

Here is the call graph for this function:



4.19.4.117 IniLine()

```
void IniLine (
            WORD extrablank )  [extern]
```

Definition at line 249 of file sch.c.

4.19.4.118 IniVars()

```
void IniVars (
            void  )  [extern]
```

Definition at line 1425 of file startup.c.

4.19.4.119 InsertTerm()

```
int InsertTerm (
              PHEAD WORD * term,
              WORD replac,
              WORD extractbuff,
              WORD * position,
              WORD * termout,
              WORD tepos )  [extern]
```

Puts the terms 'term' and 'position' together into a single legal term in termout. replac is the number of the subexpression that should be replaced. It must be a positive term. When action is needed in the argument of a function all terms in that argument are dealt with recursively. The subexpression is sorted. Only one subexpression is done at a time this way.

Parameters

term	the input term
replac	number of the subexpression pointer to replace
extractbuff	number of the compiler buffer replac refers to
position	position from where to take the term in the compiler buffer
termout	the output term
tepos	offset in term where the subexpression is.

Returns

Normal conventions (OK = 0).

Definition at line 2749 of file proces.c.

Referenced by Deferred(), Generator(), and PF_Deferred().

4.19.4.120 LongToLine()

```
void LongToLine (
              UWORD * a,
              WORD na )  [extern]
```

Definition at line 285 of file sch.c.

4.19.4.121 MakeDirty()

```
int MakeDirty (
              WORD * term,
              WORD * x,
              WORD par )  [extern]
```

Definition at line 51 of file function.c.

4.19.4.122 MarkDirty()

```
void MarkDirty (
            WORD * term,
            WORD flags )  [extern]
```

Definition at line 97 of file function.c.

4.19.4.123 PolyFunDirty()

```
void PolyFunDirty (
            PHEAD WORD * term )  [extern]
```

Definition at line 139 of file function.c.

4.19.4.124 PolyFunClean()

```
void PolyFunClean (
            PHEAD WORD * term )  [extern]
```

Definition at line 174 of file function.c.

4.19.4.125 MakeModTable()

```
int MakeModTable (
            void  )  [extern]
```

Definition at line 3382 of file reken.c.

4.19.4.126 MatchE()

```
int MatchE (
            PHEAD WORD * pattern,
            WORD * fun,
            WORD * inter,
            WORD par )  [extern]
```

Definition at line 56 of file symmetr.c.

4.19.4.127 MatchCy()

```
int MatchCy (
            PHEAD WORD * pattern,
            WORD * fun,
            WORD * inter,
            WORD par )  [extern]
```

Definition at line 571 of file symmetr.c.

4.19.4.128 FunMatchCy()

```
int FunMatchCy (
            PHEAD WORD * pattern,
            WORD * fun,
            WORD * inter,
            WORD par )  [extern]
```

Definition at line 1067 of file symmetr.c.

4.19.4.129 FunMatchSy()

```
int FunMatchSy (
            PHEAD WORD * pattern,
            WORD * fun,
            WORD * inter,
            WORD par )  [extern]
```

Definition at line 1525 of file symmetr.c.

4.19.4.130 MatchArgument()

```
int MatchArgument (
            PHEAD WORD * arg,
            WORD * pat )  [extern]
```

Definition at line 2052 of file symmetr.c.

4.19.4.131 MatchFunction()

```
int MatchFunction (
            PHEAD WORD * pattern,
            WORD * interm,
            WORD * wilds )  [extern]
```

Definition at line 826 of file function.c.

4.19.4.132 MergePatches()

```
int MergePatches (
            WORD par )  [extern]
```

The general merge routine. Can be used for the large buffer and the file merging. The array S->Patches tells where the patches start S->pStop tells where they end (has to be computed first). The end of a 'line to be merged' is indicated by a zero. If the end is reached without running into a zero or a term runs over the boundary of a patch it is a file merging operation and a new piece from the file is read in.

Parameters

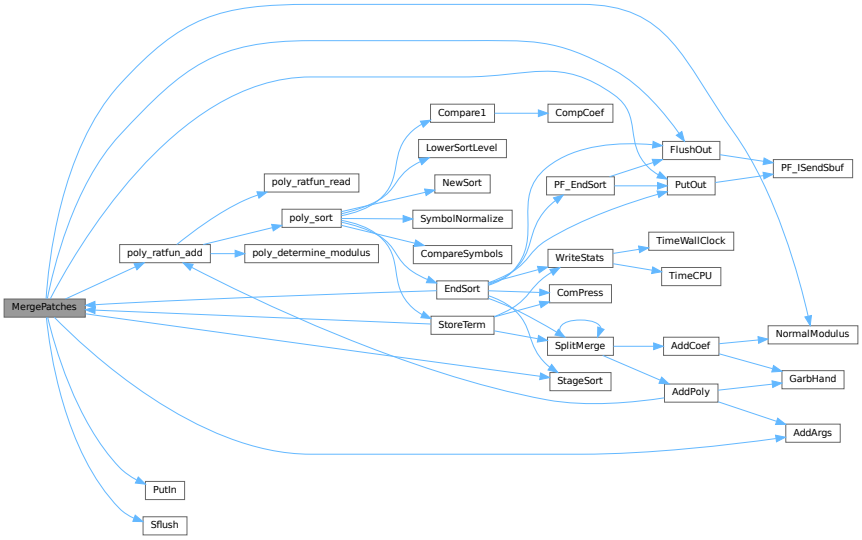
par	If par == 0 the sort is for file -> outputfile. If par == 1 the sort is for large buffer -> sortfile. If par == 2 the sort is for large buffer -> outputfile.

Definition at line 3520 of file sort.c.

References AddArgs(), FlushOut(), FiLe::handle, NormalModulus(), poly_ratfun_add(), PutIn(), PutOut(), Sflush(), and StageSort().

Referenced by EndSort(), and StoreTerm().

Here is the call graph for this function:



4.19.4.133 MesCerr()

```
int MesCerr (
            char * s,
            UBYTE * t )  [extern]
```

Definition at line 828 of file message.c.

4.19.4.134 MesComp()

```
int MesComp (
            char * s,
            UBYTE * p,
            UBYTE * q )  [extern]
```

Definition at line 847 of file message.c.

4.19.4.135 Modulus()

```
int Modulus (
            WORD * term )  [extern]
```

Definition at line 3121 of file reken.c.

4.19.4.136 MoveDummies()

```
void MoveDummies (
            PHEAD WORD * term,
            WORD shift )  [extern]
```

Definition at line 438 of file reshuf.c.

4.19.4.137 MulLong()

```
int MulLong (
            UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )  [extern]
```

Definition at line 850 of file reken.c.

4.19.4.138 MulRat()

```
int MulRat (
            PHEAD UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )  [extern]
```

Definition at line 384 of file reken.c.

4.19.4.139 Mully()

```
int Mully (
            PHEAD UWORD * a,
            WORD * na,
            UWORD * b,
            WORD nb )  [extern]
```

Definition at line 130 of file reken.c.

4.19.4.140 MultDo()

```
int MultDo (
            PHEAD WORD * term,
            WORD * pattern )  [extern]
```

Definition at line 1046 of file reshuf.c.

4.19.4.141 NewSort()

```
int NewSort (
            PHEAD0  )  [extern]
```

Starts a new sort. At the lowest level this is a 'main sort' with the struct according to the parameters in S0. At higher levels this is a sort for functions, subroutines or dollars. We prepare the arrays and structs.

Returns

Regular convention (OK -> 0)

Definition at line 397 of file sort.c.

Referenced by generate_expression(), get_expression(), InFunction(), MakeDollarInteger(), MakeDollarMod(), PF_CollectModifiedDollars(), PF_Processor(), poly_factorize_expression(), poly_sort(), poly_unfactorize_expression(), PolyFunMul(), Processor(), TakeArgContent(), and TestMatch().

4.19.4.142 ExtraSymbol()

```
int ExtraSymbol (
            WORD sym,
            WORD pow,
            WORD nsym,
            WORD * ppsym,
            WORD * ncoef )  [extern]
```

Definition at line 4262 of file normal.c.

4.19.4.143 Normalize()

```
int Normalize (
            PHEAD WORD * term )  [extern]
```

Definition at line 193 of file normal.c.

4.19.4.144 BracketNormalize()

```
int BracketNormalize (
            PHEAD WORD * term )  [extern]
```

Definition at line 5342 of file normal.c.

4.19.4.145 DropCoefficient()

```
void DropCoefficient (
            PHEAD WORD * term )  [extern]
```

Definition at line 5162 of file normal.c.

4.19.4.146 DropSymbols()

```
void DropSymbols (
            PHEAD WORD * term )  [extern]
```

Definition at line 5180 of file normal.c.

4.19.4.147 PutInside()

```
int PutInside (
            PHEAD WORD * term,
            WORD * code )  [extern]
```

Definition at line 611 of file index.c.

4.19.4.148 OpenTemp()

```
void OpenTemp (
            void  )  [extern]
```

Definition at line 48 of file store.c.

4.19.4.149 Pack()

```
void Pack (
            UWORD * a,
            WORD * na,
            UWORD * b,
            WORD nb )  [extern]
```

Definition at line 62 of file reken.c.

4.19.4.150 PasteFile()

```
LONG PasteFile (
            PHEAD WORD number,
            WORD * accum,
            POSITION * position,
            WORD ** accfill,
            RENUMBER renumber,
            WORD * freeze,
            WORD nexpr )  [extern]
```

Gets a term from stored expression expr and puts it in the accumulator at position number. It returns the length of the term that came from file.

Parameters

number	number of partial terms to skip in accum
accum	the accumulator
position	file position from where to get the stored term
accfill	returns tail position in accum
renumber	the renumber struct for the variables in the stored expression
freeze	information about if we need only the contents of a bracket
nexpr	the number of the stored expression

Returns

Normal conventions (OK = 0).

Definition at line 2885 of file proces.c.

Referenced by Generator().

4.19.4.151 Permute()

```
int Permute (
              PERM * perm,
              WORD first )  [extern]
```

Definition at line 444 of file symmetr.c.

4.19.4.152 PermuteP()

```
int PermuteP (
              PERMP * perm,
              WORD first )  [extern]
```

Definition at line 478 of file symmetr.c.

4.19.4.153 PolyFunMul()

```
int PolyFunMul (
              PHEAD WORD * term )  [extern]
```

Multiplies the arguments of multiple occurrences of the polyfun. In this routine we do the original PolyFun with one argument only. The PolyRatFun (PolyFunType = 2) is done in a dedicated routine in the file polywrap.cc The new result is written over the old result.

Parameters

term	It contains the input term and later the output.

Returns

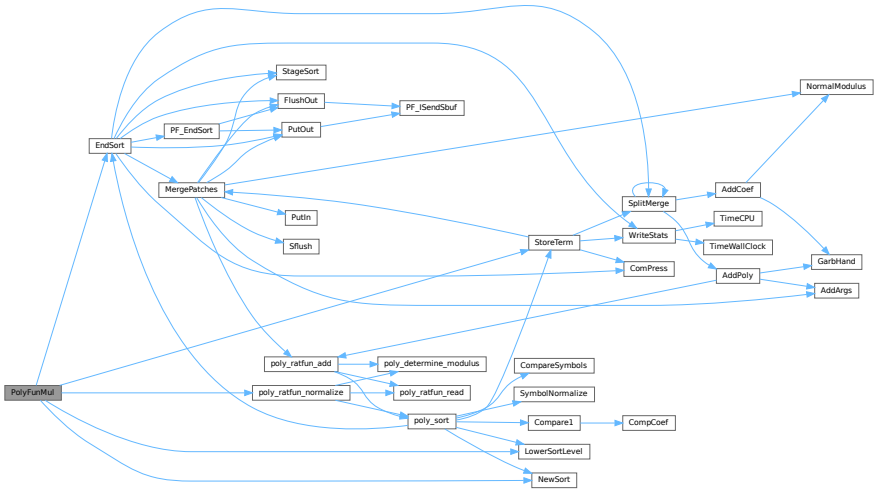
Normal conventions (OK = 0).

Definition at line 5394 of file proces.c.

References EndSort(), LowerSortLevel(), NewSort(), poly_ratfun_normalize(), and StoreTerm().

Referenced by Generator().

Here is the call graph for this function:



4.19.4.154 PopVariables()

```
int PopVariables (
            void  )  [extern]
```

Definition at line 211 of file execute.c.

4.19.4.155 PrepPoly()

```
int PrepPoly (
            PHEAD WORD * term,
            WORD par )  [extern]
```

Routine checks whether the count of function AR.PolyFun is zero or one. If it is one and it has one scalarlike argument the coefficient of the term is pulled inside the argument. If the count is zero a new function is made with the coefficient as its only argument. The function should be placed at its proper position.

When this function is active it places the PolyFun as last object before the coefficient. This is needed because otherwise the compress algorithm has problems in MergePatches.

The bracket routine should also place the PolyFun at a comparable spot. The compression should then stop at the PolyFun. It doesn't really have to stop when writing the final result but this may be too complicated.

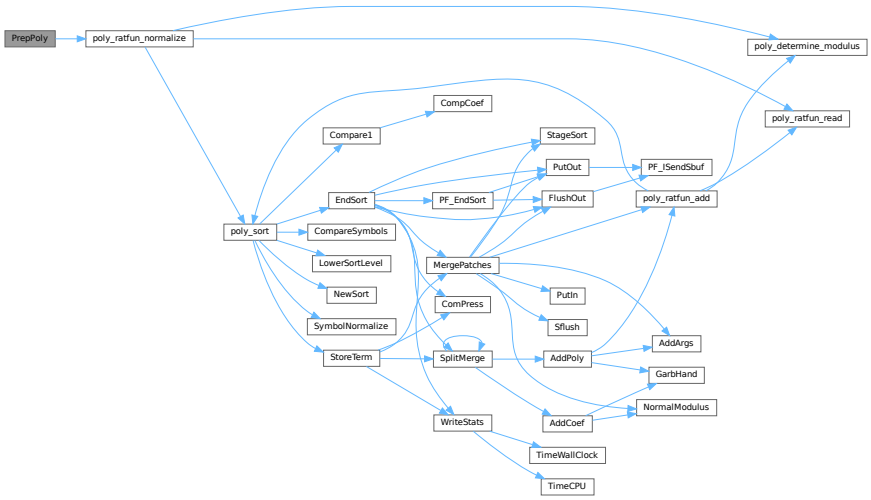
The parameter par tells whether we are at groundlevel or inside a function or dollar variable.

Definition at line 5004 of file proces.c.

References poly_ratfun_normalize().

Referenced by Generator().

Here is the call graph for this function:



4.19.4.156 Processor()

```
int Processor (
            void  )  [extern]
```

This is the central processor. It accepts a stream of Expressions which is accessed by calls to GetTerm. The expressions reside either in AR.infile or AR.hidefile The definitions of an expression are seen as an id-statement, so the primary Expressions should be written to the system of scratch files as single terms with an expression pointer. Each expression is terminated with a zero and the whole is terminated by two zeroes.

The routine DoExecute should determine whether results are to be printed, should revert the scratch I/O directions etc. In principle it is DoExecute that calls Processor.

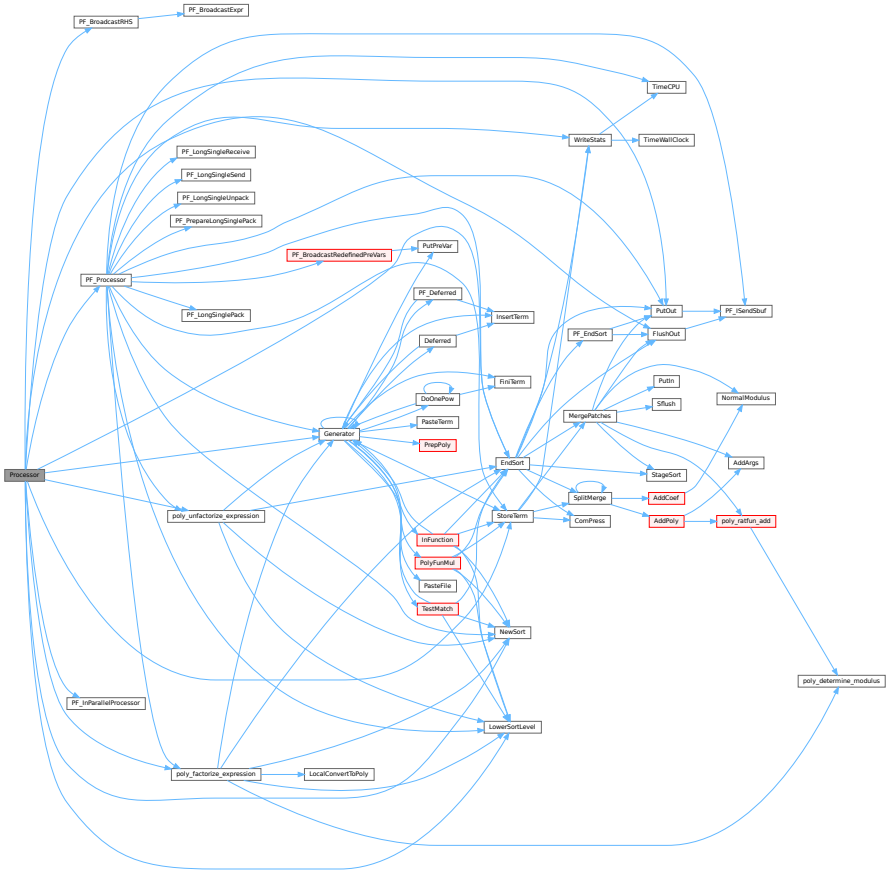
Returns

if everything OK: 0. Otherwise error. The preprocessor may continue with compilation though. Really fatal errors should return on the spot by calling Terminate.

Definition at line 64 of file proces.c.

References CbUf::Buffer, EndSort(), FlushOut(), Generator(), CbUf::lhs, LowerSortLevel(), NewSort(), PF_BroadcastRHS(), PF_InParallelProcessor(), PF_Processor(), CbUf::Pointer, poly_factorize_expression(), poly_unfactorize_expression(), PutOut(), CbUf::rhs, and StoreTerm().

Here is the call graph for this function:



4.19.4.157 Product()

```
int Product (
            UWORD * a,
            WORD * na,
            WORD b )  [extern]
```

Definition at line 1600 of file reken.c.

4.19.4.158 PrtLong()

```
void PrtLong (
            UWORD * a,
            WORD na,
            UBYTE * s )  [extern]
```

Definition at line 1727 of file reken.c.

4.19.4.159 PrtTerms()

```
void PrtTerms (
            void  )  [extern]
```

Definition at line 698 of file sch.c.

4.19.4.160 PrintDeprecation()

```
void PrintDeprecation (
            const char * feature,
            const char * issue )  [extern]
```

Prints a deprecation warning for a given feature.

Parameters

feature	The name of the deprecated feature.
issue	The associated issue, e.g., "issues/700".

Definition at line 2143 of file startup.c.

4.19.4.161 PrintFeatureList()

```
void PrintFeatureList (
            void  )  [extern]
```

Prints the list of available features.

Definition at line 131 of file features.cc.

4.19.4.162 PrintRunningTime()

```
void PrintRunningTime (
            void  )  [extern]
```

Definition at line 2167 of file startup.c.

4.19.4.163 GetRunningTime()

```
LONG GetRunningTime (
            void  )  [extern]
```

Definition at line 2208 of file startup.c.

4.19.4.164 PutBracket()

```
int PutBracket (
            PHEAD WORD * termin )  [extern]
```

Definition at line 1214 of file execute.c.

4.19.4.165 PutIn()

```
LONG PutIn (
            FILEHANDLE * file,
            POSITION * position,
            WORD * buffer,
            WORD ** take,
            int npat )  [extern]
```

Reads a new patch from position in file handle. It is put at buffer, anything after take is moved forward. This would be part of a term that hasn't been used yet. Because of this there should be some space before the start of the buffer

Parameters

file	The file system from which to read
position	The position from which to read
buffer	The buffer into which to read
take	The unused tail should be moved before the buffer
npat	The number of the patch. Is needed if the information was compressed with gzip, because each patch has its own independent gzip encoding.

Definition at line 1065 of file sort.c.

References FiLe::handle.

Referenced by MergePatches().

4.19.4.166 PutInStore()

```
int PutInStore (
            INDEXENTRY * ind,
            WORD num )  [extern]
```

Definition at line 899 of file store.c.

4.19.4.167 PutOut()

```
WORD PutOut (
            PHEAD WORD * term,
            POSITION * position,
            FILEHANDLE * fi,
            WORD ncomp )  [extern]
```

Routine writes one term to file handle at position. It returns the new value of the position.

NOTE: For 'final output' we may have to index the brackets. See the struct BRACKETINDEX. We should maintain: 1: a list with brackets array with the brackets 2: a list of objects of type BRACKETINDEX. It contains array with either pointers or offsets to the list of brackets. starting positions in the file. The index may be tied to a maximum size. In that case we may have to prune the list occasionally.

Parameters

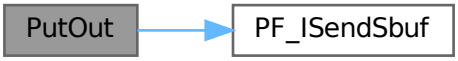
term	The term to be written
position	The position in the file. Afterwards it is updated
fi	The file (or its cache) to which should be written
ncomp	Information about what type of compression should be used

Definition at line 1217 of file sort.c.

References FiLe::handle, and PF_ISendSbuf().

Referenced by EndSort(), generate_expression(), MergePatches(), PF_EndSort(), PF_Processor(), and Processor().

Here is the call graph for this function:



4.19.4.168 Quotient()

```
UWORD Quotient (
              UWORD * a,
              WORD * na,
              WORD b )  [extern]
```

Definition at line 1635 of file reken.c.

4.19.4.169 RaisPow()

```
int RaisPow (
              PHEAD UWORD * a,
              WORD * na,
              UWORD b )  [extern]
```

Definition at line 1241 of file reken.c.

4.19.4.170 RaisPowCached()

```
void RaisPowCached (
              PHEAD WORD x,
              WORD n,
              UWORD ** c,
              WORD * nc )  [extern]
```

Computes power x^n and caches the value

4.19.5 Description

Calculates the power x^n and stores the results for caching purposes. The pointer c (i.e., the pointer, and not what it points to) is overwritten. What it points to should not be overwritten in the calling function.

4.19.6 Notes

- Caching is done in AT.small_power[]. This array is extended if necessary.

Definition at line 1309 of file reken.c.

Referenced by poly::divmod_heap(), poly::divmod_univar(), and poly::mul_heap().

4.19.6.1 RaisPowMod()

```
WORD RaisPowMod (
            WORD x,
            WORD n,
            WORD m )  [extern]
```

Definition at line 1396 of file reken.c.

4.19.6.2 NormalModulus()

```
int NormalModulus (
            UWORD * a,
            WORD * na )  [extern]
```

Brings a modular representation in the range -p/2 to +p/2 The return value tells whether anything was done. Routine made in the general modulus revamp of July 2008 (JV).

Definition at line 1416 of file reken.c.

Referenced by AddCoef(), and MergePatches().

4.19.6.3 MakeInverses()

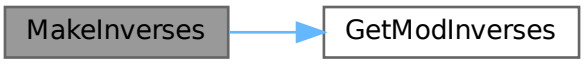
```
int MakeInverses (
            void  )  [extern]
```

Makes a table of inverses in modular calculus The modulus is in AC.cmod and AC.ncmod One should notice that the table of inverses can only be made if the modulus fits inside a single FORM word. Otherwise the table lookup becomes too difficult and the table too long.

Definition at line 1453 of file reken.c.

References GetModInverses().

Here is the call graph for this function:



4.19.6.4 GetModInverses()

```
int GetModInverses (
            WORD m1,
            WORD m2,
            WORD * im1,
            WORD * im2 )  [extern]
```

Input m1 and m2, which are relative prime. determines a∗m1+b∗m2 = 1 (and 1 is the gcd of m1 and m2) then a∗m1 = 1 mod m2 and hence im1 = a. and b∗m2 = 1 mod m1 and hence im2 = b. Set m1 = 0∗m1+1∗m2 = a1∗m1+b1∗m2 m2 = 1∗m1+0∗m2 = a2∗m1+b2∗m2 If everything is OK, the return value is zero

Definition at line 1489 of file reken.c.

Referenced by poly::divmod_heap(), poly::divmod_univar(), MakeDollarMod(), MakeInverses(), MakeMod(), TakeContent(), and TakeSymbolContent().

4.19.6.5 GetLongModInverses()

```
int GetLongModInverses (
            PHEAD UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * ia,
            WORD * nia,
            UWORD * ib,
            WORD * nib )  [extern]
```

Definition at line 1523 of file reken.c.

4.19.6.6 RatToLine()

```
void RatToLine (
            UWORD * a,
            WORD na )  [extern]
```

Definition at line 319 of file sch.c.

4.19.6.7 RatioFind()

```
int RatioFind (
            PHEAD WORD * term,
            WORD * params )  [extern]
```

Definition at line 62 of file ratio.c.

4.19.6.8 RatioGen()

```
int RatioGen (
             PHEAD WORD * term,
             WORD * params,
             WORD num,
             WORD level )  [extern]
```

Definition at line 170 of file ratio.c.

4.19.6.9 ReNumber()

```
WORD ReNumber (
             PHEAD WORD * term )  [extern]
```

Definition at line 80 of file reshuf.c.

4.19.6.10 ReadSnum()

```
WORD ReadSnum (
             UBYTE ** p )  [extern]
```

Definition at line 2005 of file tools.c.

4.19.6.11 Remain10()

```
WORD Remain10 (
             UWORD * a,
             WORD * na )  [extern]
```

Definition at line 1674 of file reken.c.

4.19.6.12 Remain4()

```
WORD Remain4 (
             UWORD * a,
             WORD * na )  [extern]
```

Definition at line 1701 of file reken.c.

4.19.6.13 ResetScratch()

```
int ResetScratch (
             void  )  [extern]
```

Definition at line 244 of file store.c.

4.19.6.14 ResolveSet()

```
int ResolveSet (
            PHEAD WORD * from,
            WORD * to,
            WORD * subs )  [extern]
```

Definition at line 1361 of file wildcard.c.

4.19.6.15 RevertScratch()

```
int RevertScratch (
            void  )  [extern]
```

Definition at line 195 of file store.c.

4.19.6.16 ScanFunctions()

```
int ScanFunctions (
            PHEAD WORD * inpat,
            WORD * inter,
            WORD par )  [extern]
```

Definition at line 1618 of file function.c.

4.19.6.17 SeekScratch()

```
void SeekScratch (
            FILEHANDLE * fi,
            POSITION * pos )  [extern]
```

Definition at line 63 of file store.c.

4.19.6.18 SetEndScratch()

```
void SetEndScratch (
            FILEHANDLE * f,
            POSITION * position )  [extern]
```

Definition at line 74 of file store.c.

4.19.6.19 SetEndHScratch()

```
void SetEndHScratch (
            FILEHANDLE * f,
            POSITION * position )  [extern]
```

Definition at line 88 of file store.c.

4.19.6.20 SetFileIndex()

```
int SetFileIndex (
            void  )  [extern]
```

Reads the next file index and puts it into AR.StoreData.Index. TODO

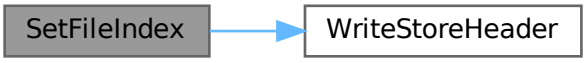
Returns

= 0 everything okay, != 0 an error occurred

Definition at line 2380 of file store.c.

References WriteStoreHeader().

Here is the call graph for this function:



4.19.6.21 Sflush()

```
int Sflush (
            FILEHANDLE * fi )  [extern]
```

Puts the contents of a buffer to output Only to be used when there is a single patch in the large buffer.

Parameters

fi	The filesystem (or its cache) to which the patch should be written

Definition at line 1131 of file sort.c.

References FiLe::handle.

Referenced by MergePatches().

4.19.6.22 Simplify()

```
int Simplify (
            PHEAD UWORD * a,
            WORD * na,
            UWORD * b,
            WORD * nb )  [extern]
```

Definition at line 539 of file reken.c.

4.19.6.23 SortWild()

```
int SortWild (
            WORD * w,
            WORD nw )  [extern]
```

Sorts the wildcard entries in the parameter w. Double entries are removed. Full space taken is nw words. Routine serves for the reading of wildcards in the compiler. The entries come in the format: (type,4,number,0) in which the zero is reserved for the future replacement of 'number'.

Parameters

w	buffer with wildcard entries.
nw	number of wildcard entries.

Returns

Normal conventions (OK -> 0)

Definition at line 4536 of file sort.c.

4.19.6.24 LocateBase()

```
FILE * LocateBase (
            char ** name,
            char ** newname,
            char * iomode )  [extern]
```

Definition at line 225 of file minos.c.

4.19.6.25 SplitMerge()

```
LONG SplitMerge (
            PHEAD WORD ** Pointer,
            LONG number )  [extern]
```

Algorithm by J.A.M.Vermaseren (31-7-1988)

Note that AN.SplitScratch and AN.InScratch are used also in GarbHand

Merge sort in memory. The input is an array of pointers. Sorting is done recursively by dividing the array in two equal parts and calling SplitMerge for each. When the parts are small enough we can do the compare and take the appropriate action. An addition is that we look for 'runs'. Sequences that are already ordered. This happens a lot when there is very little action in a module. This made FORM faster by a few percent.

Parameters

Pointer	The array of pointers to the terms to be sorted.
number	The number of pointers in Pointer.

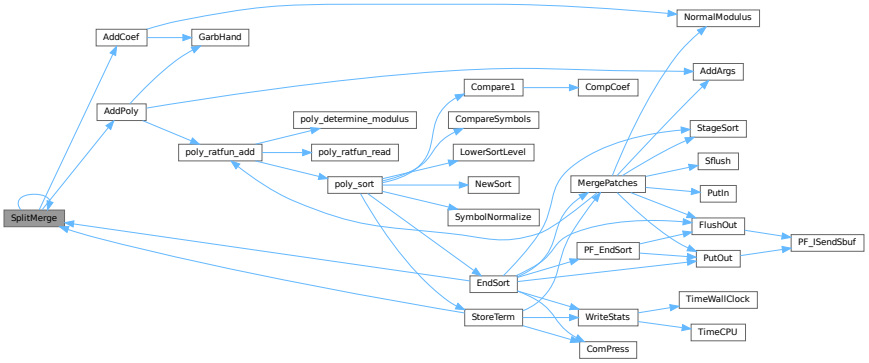
The terms are supposed to be sitting in the small buffer and there is supposed to be an extension to this buffer for when there are two terms that should be added and the result takes more space than each of the original terms. The notation guarantees that the result never needs more space than the sum of the spaces of the original terms.

Definition at line 3125 of file sort.c.

References AddCoef(), AddPoly(), and SplitMerge().

Referenced by EndSort(), SplitMerge(), and StoreTerm().

Here is the call graph for this function:



4.19.6.26 StoreTerm()

```
int StoreTerm (
            PHEAD WORD * term )  [extern]
```

The central routine to accept terms, store them and keep things at least partially sorted. A call to EndSort will then complete storing and sorting.

Parameters

term	The term to be stored

Returns

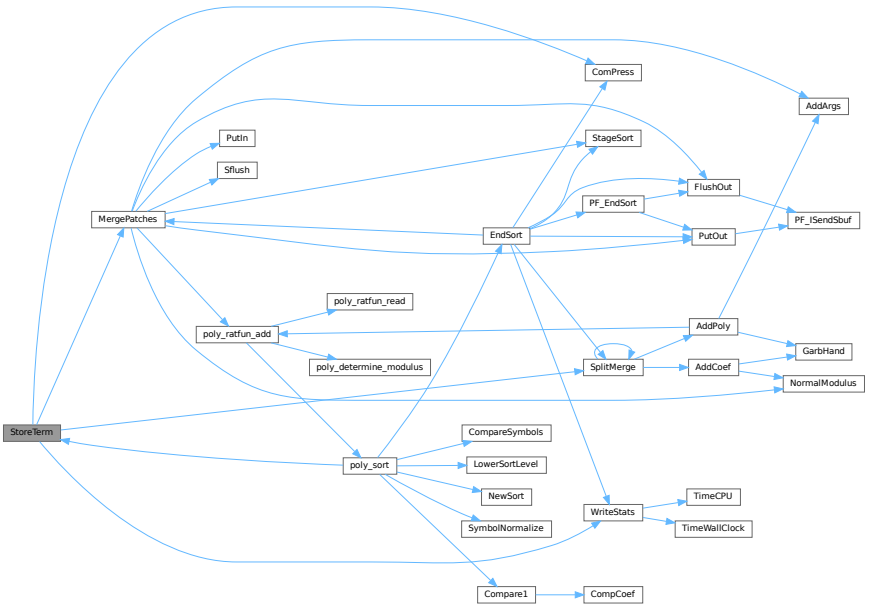
Regular return conventions (OK -> 0)

Definition at line 4311 of file sort.c.

References ComPress(), MergePatches(), SplitMerge(), and WriteStats().

Referenced by Generator(), get_expression(), InFunction(), PF_Processor(), poly_sort(), PolyFunMul(), Processor(), and TakeArgContent().

Here is the call graph for this function:



4.19.6.27 SubPLon()

```
void SubPLon (
            UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )  [extern]
```

Definition at line 812 of file reken.c.

4.19.6.28 Substitute()

```
void Substitute (
            PHEAD WORD * term,
            WORD * pattern,
            WORD power )  [extern]
```

Definition at line 643 of file pattern.c.

4.19.6.29 SymFind()

```
int SymFind (
            PHEAD WORD * term,
            WORD * params )  [extern]
```

Definition at line 633 of file function.c.

4.19.6.30 SymGen()

```
int SymGen (
              PHEAD WORD * term,
              WORD * params,
              WORD num,
              WORD level )  [extern]
```

Definition at line 518 of file function.c.

4.19.6.31 Symmetrize()

```
WORD Symmetrize (
              PHEAD WORD * func,
              WORD * Lijst,
              WORD ngroups,
              WORD gsize,
              WORD type )  [extern]
```

Definition at line 213 of file function.c.

4.19.6.32 FullSymmetrize()

```
int FullSymmetrize (
              PHEAD WORD * fun,
              int type )  [extern]
```

Definition at line 473 of file function.c.

4.19.6.33 TakeModulus()

```
int TakeModulus (
              UWORD * a,
              WORD * na,
              UWORD * cmodvec,
              WORD ncmod,
              WORD par )  [extern]
```

Definition at line 3168 of file reken.c.

4.19.6.34 TakeNormalModulus()

```
int TakeNormalModulus (
              UWORD * a,
              WORD * na,
              UWORD * c,
              WORD nc,
              WORD par )  [extern]
```

Definition at line 3340 of file reken.c.

4.19.6.35 TalToLine()

```
void TalToLine (
            UWORD x )  [extern]
```

Definition at line 504 of file sch.c.

4.19.6.36 TenVec()

```
int TenVec (
            PHEAD WORD * term,
            WORD * params,
            WORD num,
            WORD level )  [extern]
```

Definition at line 2317 of file opera.c.

4.19.6.37 TenVecFind()

```
int TenVecFind (
            PHEAD WORD * term,
            WORD * params )  [extern]
```

Definition at line 2183 of file opera.c.

4.19.6.38 TermRenumber()

```
int TermRenumber (
            WORD * term,
            RENUMBER renumber,
            WORD nexpr )  [extern]
```

!! WORD *memterm=term; static LONG ctrap=0; !!!

!! ctrap++; !!!

Definition at line 2499 of file store.c.

References ReNuMbEr::func, ReNuMbEr::funnum, ReNuMbEr::indi, ReNuMbEr::indnum, ReNuMbEr::symb, ReNuMbEr::symnum, TermRenumber(), ReNuMbEr::vecnum, and ReNuMbEr::vect.

Referenced by TermRenumber().

Here is the call graph for this function:

4.19.6.39 TestDrop()

```
void TestDrop (
            void  )  [extern]
```

Definition at line 484 of file execute.c.

4.19.6.40 PutInVflags()

```
void PutInVflags (
            WORD nexpr )  [extern]
```

Definition at line 562 of file execute.c.

4.19.6.41 TestMatch()

```
int TestMatch (
            PHEAD WORD * term,
            WORD * level )  [extern]
```

This routine governs the pattern matching. If it decides that a substitution should be made, this can be either the insertion of a right hand side (C->rhs) or the automatic generation of terms as a result of an operation (like trace). The object to be replaced is removed from term and a subexpression pointer is inserted. If the substitution is made more than once there can be more subexpression pointers. Its number is positive as it corresponds to the level at which the C->rhs can be found in the compiler output. The subexpression pointer contains the wildcard substitution information. The power is found in *AT.TMout. For operations the subexpression pointer is negative and corresponds to an address in the array AT.TMout. In this array are then the instructions for the routine to be called and its number in the array 'Operations' The format is here: length,functionnumber,length-2 parameters

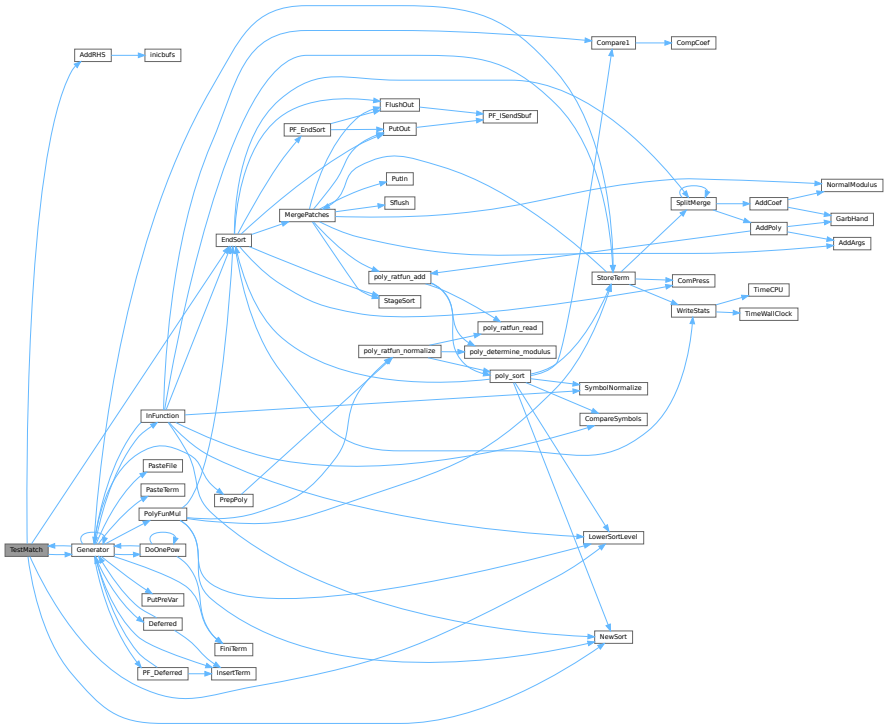
There is a certain complexity wrt repeat levels. Another complication is the poking of the wildcard values in the subexpression prototype in the compiler buffer. This was how things were done in the past with sequential FORM, but with the advent of TFORM this cannot be maintained. Now, for TFORM we make a copy of it. 7-may-2008 (JV): We cannot yet guarantee that this has been done 100% correctly. There are errors that occur in TFORM only and that may indicate problems.

Definition at line 97 of file pattern.c.

References AddRHS(), EndSort(), Generator(), CbUf::lhs, LowerSortLevel(), and NewSort().

Referenced by Generator().

Here is the call graph for this function:



4.19.6.42 TestSub()

```
WORD TestSub (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 703 of file proces.c.

4.19.6.43 TimeCPU()

```
LONG TimeCPU (
            WORD par )  [extern]
```

Returns the CPU time.

Parameters

par	If zero, the CPU time will be reset to 0.

Returns

The CPU time in milliseconds.

Definition at line 3499 of file tools.c.

Referenced by PF_GetSlaveTimes(), PF_Processor(), and WriteStats().

4.19.6.44 TimeChildren()

```
LONG TimeChildren (
            WORD par )  [extern]
```

Definition at line 3481 of file tools.c.

4.19.6.45 TimeWallClock()

```
LONG TimeWallClock (
            WORD par )  [extern]
```

Returns the wall-clock time.

Parameters

par	If zero, the wall-clock time will be reset to 0.

Returns

The wall-clock time in centiseconds.

Definition at line 3425 of file tools.c.

Referenced by DoCheckpoint(), DoRecovery(), find_Horner_MCTS(), optimize_expression_given_Horner(), optimize_greedy(), and WriteStats().

4.19.6.46 ToStorage()

```
int ToStorage (
            EXPRESSIONS e,
            POSITION * length )  [extern]
```

Definition at line 2056 of file store.c.

4.19.6.47 TokenToLine()

```
void TokenToLine (
            UBYTE * s )  [extern]
```

Definition at line 533 of file sch.c.

4.19.6.48 Trace4()

```
int Trace4 (
            PHEAD WORD * term,
            WORD * params,
            WORD num,
            WORD level )  [extern]
```

Definition at line 697 of file opera.c.

4.19.6.49 Trace4Gen()

```
int Trace4Gen (
            PHEAD TRACES * t,
            WORD number )  [extern]
```

Definition at line 860 of file opera.c.

4.19.6.50 Trace4no()

```
int Trace4no (
            WORD number,
            WORD * kron,
            TRACES * t )  [extern]
```

Definition at line 420 of file opera.c.

4.19.6.51 TraceFind()

```
int TraceFind (
            PHEAD WORD * term,
            WORD * params )  [extern]
```

Definition at line 1858 of file opera.c.

4.19.6.52 TraceN()

```
int TraceN (
            PHEAD WORD * term,
            WORD * params,
            WORD num,
            WORD level )  [extern]
```

Definition at line 1386 of file opera.c.

4.19.6.53 TraceNgen()

```
int TraceNgen (
            PHEAD TRACES * t,
            WORD number )  [extern]
```

Definition at line 1451 of file opera.c.

4.19.6.54 TraceNno()

```
WORD TraceNno (
            WORD number,
            WORD * kron,
            TRACES * t )  [extern]
```

Definition at line 1345 of file opera.c.

4.19.6.55 Traces()

```
int Traces (
            PHEAD WORD * term,
            WORD * params,
            WORD num,
            WORD level )  [extern]
```

Definition at line 1836 of file opera.c.

4.19.6.56 Trick()

```
WORD Trick (
            WORD * in,
            TRACES * t )  [extern]
```

Definition at line 333 of file opera.c.

4.19.6.57 TryDo()

```
int TryDo (
            PHEAD WORD * term,
            WORD * pattern,
            WORD level )  [extern]
```

Definition at line 1074 of file reshuf.c.

4.19.6.58 UnPack()

```
void UnPack (
            UWORD * a,
            WORD na,
            WORD * denom,
            WORD * numer )  [extern]
```

Definition at line 104 of file reken.c.

4.19.6.59 VarStore()

```
int VarStore (
            UBYTE * s,
            WORD n,
            WORD name,
            WORD namesize )  [extern]
```

Definition at line 2441 of file store.c.

4.19.6.60 WildFill()

```
WORD WildFill (
            PHEAD WORD * to,
            WORD * from,
            WORD * sub )  [extern]
```

Definition at line 65 of file wildcard.c.

4.19.6.61 WriteAll()

```
int WriteAll (
            void  )  [extern]
```

Definition at line 2490 of file sch.c.

4.19.6.62 WriteOne()

```
int WriteOne (
            UBYTE * name,
            int alreadyinline,
            int nosemi,
            WORD plus )  [extern]
```

Definition at line 2716 of file sch.c.

4.19.6.63 WriteArgument()

```
void WriteArgument (
            WORD * t )  [extern]
```

Definition at line 1406 of file sch.c.

4.19.6.64 WriteExpression()

```
int WriteExpression (
            WORD * terms,
            LONG ltot )  [extern]
```

Definition at line 2459 of file sch.c.

4.19.6.65 WriteInnerTerm()

```
int WriteInnerTerm (
            WORD * term,
            WORD first )  [extern]
```

Definition at line 1937 of file sch.c.

4.19.6.66 WriteLists()

```
void WriteLists (
            void  )  [extern]
```

Definition at line 792 of file sch.c.

4.19.6.67 WriteSetup()

```
void WriteSetup (
            void  )  [extern]
```

Definition at line 812 of file setfile.c.

4.19.6.68 WriteStats()

```
void WriteStats (
            POSITION * plspace,
            WORD par,
            WORD checkLogType )  [extern]
```

Writes the statistics.

Parameters

plspace	The size in bytes currently occupied
par	par = 0 = STATSSPLITMERGE after a splitmerge. par = 1 = STATSMERGETOFILE after merge to sortfile. par = 2 = STATSPOSTSORT after the sort
checkLogType	== CHECKLOGTYPE: The output should not go to the log file if AM.LogType is 1.

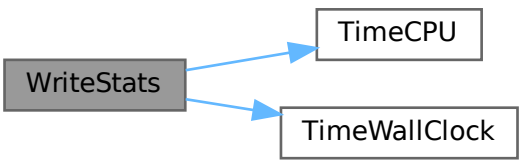
current expression is to be found in AR.CurExpr. terms are in S->TermsLeft. S->GenTerms.

Definition at line 138 of file sort.c.

References TimeCPU(), and TimeWallClock().

Referenced by EndSort(), PF_Processor(), and StoreTerm().

Here is the call graph for this function:



4.19.6.69 WriteSubTerm()

```
int WriteSubTerm (
            WORD * sterm,
            WORD first )  [extern]
```

Definition at line 1526 of file sch.c.

4.19.6.70 WriteTerm()

```
int WriteTerm (
            WORD * term,
            WORD * lbrac,
            WORD first,
            WORD prtf,
            WORD br )  [extern]
```

Definition at line 2137 of file sch.c.

4.19.6.71 execarg()

```
int execarg (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 56 of file argument.c.

4.19.6.72 execterm()

```
int execterm (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 1777 of file argument.c.

4.19.6.73 SpecialCleanup()

```
void SpecialCleanup (
            PHEAD0  )  [extern]
```

Definition at line 1728 of file execute.c.

4.19.6.74 SetMods()

```
void SetMods (
            void  )  [extern]
```

Definition at line 1742 of file execute.c.

4.19.6.75 UnSetMods()

```
void UnSetMods (
            void  )  [extern]
```

Definition at line 1760 of file execute.c.

4.19.6.76 DoExecute()

```
int DoExecute (
            WORD par,
            WORD skip )  [extern]
```

Definition at line 616 of file execute.c.

4.19.6.77 SetScratch()

```
void SetScratch (
            FILEHANDLE * f,
            POSITION * position )  [extern]
```

Definition at line 114 of file store.c.

4.19.6.78 Warning()

```
void Warning (
            char * s )  [extern]
```

Definition at line 794 of file message.c.

4.19.6.79 HighWarning()

```
void HighWarning (
            char * s )  [extern]
```

Definition at line 806 of file message.c.

4.19.6.80 SpareTable()

```
int SpareTable (
            TABLES TT )  [extern]
```

Definition at line 700 of file tables.c.

4.19.6.81 strDup1()

```
UBYTE * strDup1 (
            UBYTE * instring,
            char * ifwrong )  [extern]
```

Definition at line 1908 of file tools.c.

4.19.6.82 Malloc1()

```
void * Malloc1 (
            LONG size,
            const char * messageifwrong )  [extern]
```

Definition at line 2245 of file tools.c.

4.19.6.83 DoTail()

```
int DoTail (
            int argc,
            UBYTE ** argv )  [extern]
```

Definition at line 282 of file startup.c.

4.19.6.84 OpenInput()

```
int OpenInput (
            void  )  [extern]
```

Definition at line 589 of file startup.c.

4.19.6.85 PutPreVar()

```
int PutPreVar (
            UBYTE * name,
            UBYTE * value,
            UBYTE * args,
            int mode )  [extern]
```

Inserts/Updates a preprocessor variable in the name administration.

Parameters

name	Character string with the variable name.
value	Character string with a possible value. Special case: if this argument is zero, then we have no value. Note: This is different from having an empty argument! This should only occur when the name starts with a ?
args	Character string with possible arguments.
mode	=0: always create a new name entry, =1: try to do a redefinition if possible.

Returns

Index of used entry in name list.

Definition at line 724 of file pre.c.

Referenced by ClearOptimize(), Generator(), Optimize(), PF_BroadcastRedefinedPreVars(), StartVariables(), and TheDefine().

4.19.6.86 Error0()

```
void Error0 (
            char * s )  [extern]
```

Definition at line 56 of file message.c.

4.19.6.87 Error1()

```
void Error1 (
            char * s,
            UBYTE * t )  [extern]
```

Definition at line 67 of file message.c.

4.19.6.88 Error2()

```
void Error2 (
            char * s1,
            char * s2,
            UBYTE * t )  [extern]
```

Definition at line 78 of file message.c.

4.19.6.89 ReadFromStream()

```
UBYTE ReadFromStream (
            STREAM * stream )  [extern]
```

Definition at line 151 of file tools.c.

4.19.6.90 GetFromStream()

```
UBYTE GetFromStream (
            STREAM * stream )  [extern]
```

Definition at line 286 of file tools.c.

4.19.6.91 LookInStream()

```
UBYTE LookInStream (
            STREAM * stream )  [extern]
```

Definition at line 309 of file tools.c.

4.19.6.92 OpenStream()

```
STREAM * OpenStream (
            UBYTE * name,
            int type,
            int prevarmode,
            int raiselow )  [extern]
```

Definition at line 321 of file tools.c.

4.19.6.93 LocateFile()

```
int LocateFile (
            UBYTE ** name,
            int type )  [extern]
```

Definition at line 576 of file tools.c.

4.19.6.94 CloseStream()

```
STREAM * CloseStream (
            STREAM * stream )  [extern]
```

Definition at line 663 of file tools.c.

4.19.6.95 PositionStream()

```
void PositionStream (
            STREAM * stream,
            LONG position )  [extern]
```

Definition at line 825 of file tools.c.

4.19.6.96 ReverseStatements()

```
int ReverseStatements (
            STREAM * stream )  [extern]
```

Definition at line 857 of file tools.c.

4.19.6.97 ProcessOption()

```
int ProcessOption (
             UBYTE * s1,
             UBYTE * s2,
             int filetype )  [extern]
```

Definition at line 183 of file setfile.c.

4.19.6.98 DoSetups()

```
int DoSetups (
             void  )  [extern]
```

Definition at line 133 of file setfile.c.

4.19.6.99 TerminateImpl()

```
void TerminateImpl (
             int errorcode,
             const char * file,
             int line,
             const char * function )  [extern]
```

Definition at line 1923 of file startup.c.

4.19.6.100 GetNode()

```
NAMENODE * GetNode (
             NAMETREE * nametree,
             UBYTE * name )  [extern]
```

Definition at line 49 of file names.c.

4.19.6.101 AddName()

```
int AddName (
             NAMETREE * nametree,
             UBYTE * name,
             WORD type,
             WORD number,
             int * nodenum )  [extern]
```

Definition at line 71 of file names.c.

4.19.6.102 GetName()

```
int GetName (
            NAMETREE * nametree,
            UBYTE * namein,
            WORD * number,
            int par )  [extern]
```

Definition at line 254 of file names.c.

4.19.6.103 GetFunction()

```
UBYTE * GetFunction (
            UBYTE * s,
            WORD * funnum )  [extern]
```

Definition at line 344 of file names.c.

4.19.6.104 GetNumber()

```
UBYTE * GetNumber (
            UBYTE * s,
            WORD * num )  [extern]
```

Definition at line 390 of file names.c.

4.19.6.105 GetLastExprName()

```
int GetLastExprName (
            UBYTE * name,
            WORD * number )  [extern]
```

Definition at line 440 of file names.c.

4.19.6.106 GetAutoName()

```
int GetAutoName (
            UBYTE * name,
            WORD * number )  [extern]
```

Definition at line 481 of file names.c.

4.19.6.107 GetVar()

```
int GetVar (
            UBYTE * name,
            WORD * type,
            WORD * number,
            int wantedtype,
            int par )  [extern]
```

Definition at line 526 of file names.c.

4.19.6.108 MakeDubious()

```
int MakeDubious (
            NAMETREE * nametree,
            UBYTE * name,
            WORD * number )  [extern]
```

Definition at line 2700 of file names.c.

4.19.6.109 GetOName()

```
int GetOName (
            NAMETREE * nametree,
            UBYTE * name,
            WORD * number,
            int par )  [extern]
```

Definition at line 462 of file names.c.

4.19.6.110 DumpTree()

```
void DumpTree (
            NAMETREE * nametree )  [extern]
```

Definition at line 604 of file names.c.

4.19.6.111 DumpNode()

```
void DumpNode (
            NAMETREE * nametree,
            WORD node,
            WORD depth )  [extern]
```

Definition at line 617 of file names.c.

4.19.6.112 LinkTree()

```
void LinkTree (
            NAMETREE * tree,
            WORD offset,
            WORD numnodes )  [extern]
```

Definition at line 788 of file names.c.

4.19.6.113 CopyTree()

```
void CopyTree (
            NAMETREE * newtree,
            NAMETREE * oldtree,
            WORD node,
            WORD par )  [extern]
```

Definition at line 698 of file names.c.

4.19.6.114 CompactifyTree()

```
int CompactifyTree (
            NAMETREE * nametree,
            WORD par )  [extern]
```

Definition at line 636 of file names.c.

4.19.6.115 MakeNameTree()

```
NAMETREE * MakeNameTree (
            void  )  [extern]
```

Definition at line 819 of file names.c.

4.19.6.116 FreeNameTree()

```
void FreeNameTree (
            NAMETREE * n )  [extern]
```

Definition at line 838 of file names.c.

4.19.6.117 AddExpression()

```
int AddExpression (
            UBYTE * name,
            int x,
            int y )  [extern]
```

Definition at line 2751 of file names.c.

4.19.6.118 AddSymbol()

```
int AddSymbol (
            UBYTE * name,
            int minpow,
            int maxpow,
            int cplx,
            int dim )  [extern]
```

Definition at line 924 of file names.c.

4.19.6.119 AddDollar()

```
int AddDollar (
            UBYTE * name,
            WORD type,
            WORD * start,
            LONG size )  [extern]
```

Definition at line 2610 of file names.c.

4.19.6.120 ReplaceDollar()

```
int ReplaceDollar (
             WORD number,
             WORD newtype,
             WORD * newstart,
             LONG newsize )  [extern]
```

Definition at line 2653 of file names.c.

4.19.6.121 DollarRaiseLow()

```
int DollarRaiseLow (
             UBYTE * name,
             LONG value )  [extern]
```

Definition at line 2536 of file dollar.c.

4.19.6.122 AddVector()

```
int AddVector (
             UBYTE * name,
             int cplx,
             int dim )  [extern]
```

Definition at line 1248 of file names.c.

4.19.6.123 AddDubious()

```
int AddDubious (
             UBYTE * name )  [extern]
```

Definition at line 2686 of file names.c.

4.19.6.124 AddIndex()

```
int AddIndex (
             UBYTE * name,
             int dim,
             int dim4 )  [extern]
```

Definition at line 1092 of file names.c.

4.19.6.125 DoDimension()

```
UBYTE * DoDimension (
             UBYTE * s,
             int * dim,
             int * dim4 )  [extern]
```

Definition at line 1164 of file names.c.

4.19.6.126 AddFunction()

```
int AddFunction (
            UBYTE * name,
            int comm,
            int istensor,
            int cplx,
            int symprop,
            int dim,
            int argmax,
            int argmin )  [extern]
```

Definition at line 1330 of file names.c.

4.19.6.127 CoCommuteInSet()

```
int CoCommuteInSet (
            UBYTE * s )  [extern]
```

Definition at line 1360 of file names.c.

4.19.6.128 CoFunction()

```
int CoFunction (
            UBYTE * s,
            int comm,
            int istensor )  [extern]
```

Definition at line 1450 of file names.c.

4.19.6.129 TestName()

```
int TestName (
            UBYTE * name )  [extern]
```

Definition at line 3261 of file names.c.

4.19.6.130 AddSet()

```
int AddSet (
            UBYTE * name,
            WORD dim )  [extern]
```

Definition at line 2128 of file names.c.

4.19.6.131 DoElements()

```
int DoElements (
            UBYTE * s,
            SETS set,
            UBYTE * name )  [extern]
```

Definition at line 2161 of file names.c.

4.19.6.132 DoTempSet()

```
int DoTempSet (
            UBYTE * from,
            UBYTE * to )  [extern]
```

Definition at line 2488 of file names.c.

4.19.6.133 NameConflict()

```
int NameConflict (
            int type,
            UBYTE * name )  [extern]
```

Definition at line 2735 of file names.c.

4.19.6.134 OpenFile()

```
int OpenFile (
            char * name )  [extern]
```

Definition at line 985 of file tools.c.

4.19.6.135 OpenAddFile()

```
int OpenAddFile (
            char * name )  [extern]
```

Definition at line 1004 of file tools.c.

4.19.6.136 ReOpenFile()

```
int ReOpenFile (
            char * name )  [extern]
```

Definition at line 1025 of file tools.c.

4.19.6.137 CreateFile()

```
int CreateFile (
            char * name )  [extern]
```

Definition at line 1045 of file tools.c.

4.19.6.138 CreateLogFile()

```
int CreateLogFile (
            char * name )  [extern]
```

Definition at line 1062 of file tools.c.

4.19.6.139 CloseFile()

```
void CloseFile (
            int handle )  [extern]
```

Definition at line 1080 of file tools.c.

4.19.6.140 CopyFile()

```
int CopyFile (
            char * source,
            char * dest )  [extern]
```

Copies a file with name ∗source to a file named ∗dest. The involved files must not be open. Returns non-zero if an error occurred. Uses if possible the combined large and small sorting buffers as cache.

Definition at line 1104 of file tools.c.

Referenced by DoRecovery().

4.19.6.141 CreateHandle()

```
int CreateHandle (
            void  )  [extern]
```

Definition at line 1166 of file tools.c.

4.19.6.142 ReadFile()

```
LONG ReadFile (
            int handle,
            UBYTE * buffer,
            LONG size )  [extern]
```

Definition at line 1221 of file tools.c.

4.19.6.143 ReadPosFile()

```
LONG ReadPosFile (
            PHEAD FILEHANDLE * fi,
            UBYTE * buffer,
            LONG size,
            POSITION * pos )  [extern]
```

Definition at line 1271 of file tools.c.

4.19.6.144 WriteFileToFile()

```
LONG WriteFileToFile (
            int handle,
            UBYTE * buffer,
            LONG size )  [extern]
```

Definition at line 1337 of file tools.c.

4.19.6.145 SeekFile()

```
void SeekFile (
            int handle,
            POSITION * offset,
            int origin )  [extern]
```

Definition at line 1375 of file tools.c.

4.19.6.146 TellFile()

```
LONG TellFile (
            int handle )  [extern]
```

Definition at line 1400 of file tools.c.

4.19.6.147 FlushFile()

```
void FlushFile (
            int handle )  [extern]
```

Definition at line 1424 of file tools.c.

4.19.6.148 GetPosFile()

```
int GetPosFile (
            int handle,
            fpos_t * pospointer )  [extern]
```

Definition at line 1438 of file tools.c.

4.19.6.149 SetPosFile()

```
int SetPosFile (
            int handle,
            fpos_t * pospointer )  [extern]
```

Definition at line 1452 of file tools.c.

4.19.6.150 SynchFile()

```
void SynchFile (
            int handle )  [extern]
```

Definition at line 1472 of file tools.c.

4.19.6.151 TruncateFile()

```
void TruncateFile (
            int handle )  [extern]
```

Definition at line 1494 of file tools.c.

4.19.6.152 GetChannel()

```
int GetChannel (
            char * name,
            int mode )  [extern]
```

Definition at line 1514 of file tools.c.

4.19.6.153 GetAppendChannel()

```
int GetAppendChannel (
            char * name )  [extern]
```

Definition at line 1550 of file tools.c.

4.19.6.154 CloseChannel()

```
int CloseChannel (
            char * name )  [extern]
```

Definition at line 1580 of file tools.c.

4.19.6.155 inictable()

```
void inictable (
            void  )  [extern]
```

Definition at line 315 of file compiler.c.

4.19.6.156 findcommand()

```
KEYWORD * findcommand (
            UBYTE * in )  [extern]
```

Definition at line 341 of file compiler.c.

4.19.6.157 inicbufs()

```
int inicbufs (
            void  )  [extern]
```

Creates a new compiler buffer and returns its ID number.

Returns

The ID number for the new compiler buffer.

Definition at line 47 of file comtool.c.

References CbUf::boomlijst, CbUf::Buffer, CbUf::BufferSize, CbUf::CanCommu, CbUf::dimension, CbUf::lhs, CbUf::numdum, CbUf::NumTerms, CbUf::Pointer, CbUf::rhs, and CbUf::Top.

Referenced by AddRHS(), and StartVariables().

4.19.6.158 StartFiles()

```
void StartFiles (
            void  )  [extern]
```

Definition at line 958 of file tools.c.

4.19.6.159 MakeDate()

```
UBYTE * MakeDate (
            void  )  [extern]
```

Definition at line 2119 of file tools.c.

4.19.6.160 PreProcessor()

```
void PreProcessor (
            void  )  [extern]
```

Definition at line 953 of file pre.c.

4.19.6.161 FromList()

```
void * FromList (
            LIST * L )  [extern]
```

Definition at line 2720 of file tools.c.

4.19.6.162 From0List()

```
void * From0List (
            LIST * L )  [extern]
```

Definition at line 2746 of file tools.c.

4.19.6.163 FromVarList()

```
void * FromVarList (
            LIST * L )  [extern]
```

Definition at line 2775 of file tools.c.

4.19.6.164 DoubleList()

```
int DoubleList (
            void *** lijst,
            int * oldsize,
            int objectsize,
            char * nameoftype )  [extern]
```

Definition at line 2821 of file tools.c.

4.19.6.165 DoubleLList()

```
int DoubleLList (
            void *** lijst,
            LONG * oldsize,
            int objectsize,
            char * nameoftype )  [extern]
```

Definition at line 2873 of file tools.c.

4.19.6.166 DoubleBuffer()

```
void DoubleBuffer (
            void ** start,
            void ** stop,
            int size,
            char * text )  [extern]
```

Definition at line 2921 of file tools.c.

4.19.6.167 ExpandBuffer()

```
void ExpandBuffer (
            void ** buffer,
            LONG * oldsize,
            int type )  [extern]
```

Definition at line 2946 of file tools.c.

4.19.6.168 iexp()

```
LONG iexp (
            LONG x,
            int p )  [extern]
```

Definition at line 2970 of file tools.c.

4.19.6.169 IsLikeVector()

```
int IsLikeVector (
            WORD * arg )  [extern]
```

Definition at line 3175 of file tools.c.

4.19.6.170 AreArgsEqual()

```
int AreArgsEqual (
            WORD * arg1,
            WORD * arg2 )  [extern]
```

Definition at line 3203 of file tools.c.

4.19.6.171 CompareArgs()

```
int CompareArgs (
            WORD * arg1,
            WORD * arg2 )  [extern]
```

Definition at line 3222 of file tools.c.

4.19.6.172 SkipField()

```
UBYTE * SkipField (
            UBYTE * s,
            int level )  [extern]
```

Skips from `s` to the end of a declaration field (e.g., `x`, `1+2*[x+a]`, or `f({x,[x+a]},1)`).

Parameters

in	*s*	Pointer to a null-terminated input buffer.
in	*level*	Number of parentheses that still have to be closed.

Returns

Pointer to the character after the field, which is either a comma at level 0 or the null terminator.

Definition at line 1978 of file tools.c.

4.19.6.173 StrCmp()

```
int StrCmp (
            UBYTE * s1,
            UBYTE * s2 )  [extern]
```

Definition at line 1704 of file tools.c.

4.19.6.174 StrICmp()

```
int StrICmp (
            UBYTE * s1,
            UBYTE * s2 )  [extern]
```

Definition at line 1715 of file tools.c.

4.19.6.175 StrHICmp()

```
int StrHICmp (
            UBYTE * s1,
            UBYTE * s2 )  [extern]
```

Definition at line 1726 of file tools.c.

4.19.6.176 StrICont()

```
int StrICont (
            UBYTE * s1,
            UBYTE * s2 )  [extern]
```

Definition at line 1737 of file tools.c.

4.19.6.177 CmpArray()

```
int CmpArray (
            WORD * t1,
            WORD * t2,
            WORD n )  [extern]
```

Definition at line 1749 of file tools.c.

4.19.6.178 ConWord()

```
int ConWord (
            UBYTE * s1,
            UBYTE * s2 )  [extern]
```

Definition at line 1763 of file tools.c.

4.19.6.179 StrLen()

```
int StrLen (
            UBYTE * s )  [extern]
```

Definition at line 1775 of file tools.c.

4.19.6.180 GetPreVar()

```
UBYTE * GetPreVar (
            UBYTE * name,
            int flag )  [extern]
```

Definition at line 542 of file pre.c.

4.19.6.181 ToGeneral()

```
void ToGeneral (
            WORD * r,
            WORD * m,
            WORD par )  [extern]
```

Definition at line 3001 of file tools.c.

4.19.6.182 ToPolyFunGeneral()

```
WORD ToPolyFunGeneral (
            PHEAD WORD * term )  [extern]
```

Definition at line 3098 of file tools.c.

4.19.6.183 ToFast()

```
int ToFast (
            WORD * r,
            WORD * m )  [extern]
```

Definition at line 3045 of file tools.c.

4.19.6.184 GetSetupPar()

```
SETUPPARAMETERS * GetSetupPar (
            UBYTE * s )  [extern]
```

Definition at line 338 of file setfile.c.

4.19.6.185 RecalcSetups()

```
int RecalcSetups (
            void  )  [extern]
```

Definition at line 359 of file setfile.c.

4.19.6.186 AllocSetups()

```
int AllocSetups (
            void  )  [extern]
```

Definition at line 415 of file setfile.c.

4.19.6.187 AllocSort()

```
SORTING * AllocSort (
            LONG inLargeSize,
            LONG inSmallSize,
            LONG inSmallEsize,
            LONG inTermsInSmall,
            int inMaxPatches,
            int inMaxFpatches,
            LONG inIOsize,
            int level )  [extern]
```

Definition at line 870 of file setfile.c.

4.19.6.188 AllocSortFileName()

```
void AllocSortFileName (
            SORTING * sort )  [extern]
```

Definition at line 1020 of file setfile.c.

4.19.6.189 LoadInputFile()

```
UBYTE * LoadInputFile (
            UBYTE * filename,
            int type )  [extern]
```

Definition at line 112 of file tools.c.

4.19.6.190 GetInput()

```
UBYTE GetInput (
            void  )  [extern]
```

Definition at line 138 of file pre.c.

4.19.6.191 ClearPushback()

```
void ClearPushback (
            void  )  [extern]
```

Definition at line 167 of file pre.c.

4.19.6.192 GetChar()

```
UBYTE GetChar (
            int level )  [extern]
```

Definition at line 185 of file pre.c.

4.19.6.193 CharOut()

```
void CharOut (
            UBYTE c )  [extern]
```

Definition at line 495 of file pre.c.

4.19.6.194 UnsetAllowDelay()

```
void UnsetAllowDelay (
            void  )  [extern]
```

Definition at line 516 of file pre.c.

4.19.6.195 PopPreVars()

```
void PopPreVars (
            int tonumber )  [extern]
```

Definition at line 826 of file pre.c.

4.19.6.196 IniModule()

```
void IniModule (
            int type )  [extern]
```

Definition at line 841 of file pre.c.

4.19.6.197 IniSpecialModule()

```
void IniSpecialModule (
            int type )  [extern]
```

Definition at line 943 of file pre.c.

4.19.6.198 ModuleInstruction()

```
int ModuleInstruction (
            int * moduletype,
            int * specialtype )  [extern]
```

Definition at line 76 of file module.c.

4.19.6.199 PreProInstruction()

```
int PreProInstruction (
            void  )  [extern]
```

Definition at line 1172 of file pre.c.

4.19.6.200 LoadInstruction()

```
int LoadInstruction (
            int mode )  [extern]
```

Definition at line 1269 of file pre.c.

4.19.6.201 LoadStatement()

```
int LoadStatement (
            int type )  [extern]
```

Definition at line 1472 of file pre.c.

4.19.6.202 FindKeyWord()

```
KEYWORD * FindKeyWord (
            UBYTE * theword,
            KEYWORD * table,
            int size )  [extern]
```

Definition at line 1978 of file pre.c.

4.19.6.203 FindInKeyWord()

```
KEYWORD * FindInKeyWord (
            UBYTE * theword,
            KEYWORD * table,
            int size )  [extern]
```

Definition at line 2009 of file pre.c.

4.19.6.204 DoDefine()

```
int DoDefine (
            UBYTE * s )  [extern]
```

Definition at line 2174 of file pre.c.

4.19.6.205 DoRedefine()

```
int DoRedefine (
            UBYTE * s )  [extern]
```

Definition at line 2184 of file pre.c.

4.19.6.206 TheDefine()

```
int TheDefine (
            UBYTE * s,
            int mode )  [extern]
```

Preprocessor assignment. Possible arguments and values are treated and the new preprocessor variable is put into the name administration.

Parameters

s	Pointer to the character string following the preprocessor command.
mode	Bitmask. 0-bit clear: always create a new name entry, 0-bit set: try to redefine an existing name, 1-bit set: ignore preprocessor if/switch status.

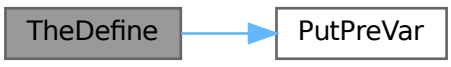
Returns

zero: no errors, negative number: errors.

Definition at line 2039 of file pre.c.

References PutPreVar().

Here is the call graph for this function:



4.19.6.207 TheUndefine()

```
int TheUndefine (
            UBYTE * name )  [extern]
```

Definition at line 2224 of file pre.c.

4.19.6.208 ClearMacro()

```
int ClearMacro (
            UBYTE * name )  [extern]
```

Definition at line 2196 of file pre.c.

4.19.6.209 DoUndefine()

```
int DoUndefine (
            UBYTE * s )  [extern]
```

Definition at line 2278 of file pre.c.

4.19.6.210 DoInclude()

```
int DoInclude (
            UBYTE * s )  [extern]
```

Definition at line 2330 of file pre.c.

4.19.6.211 DoReverseInclude()

```
int DoReverseInclude (
            UBYTE * s )  [extern]
```

Definition at line 2337 of file pre.c.

4.19.6.212 Include()

```
int Include (
            UBYTE * s,
            int type )  [extern]
```

Definition at line 2344 of file pre.c.

4.19.6.213 DoExternal()

```
int DoExternal (
            UBYTE * s )  [extern]
```

Definition at line 5515 of file pre.c.

4.19.6.214 DoToExternal()

```
int DoToExternal (
            UBYTE * s )  [extern]
```

Definition at line 6077 of file pre.c.

4.19.6.215 DoFromExternal()

```
int DoFromExternal (
            UBYTE * s )  [extern]
```

Definition at line 5882 of file pre.c.

4.19.6.216 DoPrompt()

```
int DoPrompt (
            UBYTE * s )  [extern]
```

Definition at line 5597 of file pre.c.

4.19.6.217 DoSetExternal()

```
int DoSetExternal (
            UBYTE * s )  [extern]
```

Definition at line 5630 of file pre.c.

4.19.6.218 DoSetExternalAttr()

```
int DoSetExternalAttr (
            UBYTE * s )  [extern]
```

Definition at line 5700 of file pre.c.

4.19.6.219 DoRmExternal()

```
int DoRmExternal (
            UBYTE * s )  [extern]
```

Definition at line 5817 of file pre.c.

4.19.6.220 DoFactDollar()

```
int DoFactDollar (
            UBYTE * s )  [extern]
```

Definition at line 6724 of file pre.c.

4.19.6.221 GetDollarNumber()

```
WORD GetDollarNumber (
            UBYTE ** inp,
            DOLLARS d )  [extern]
```

Definition at line 6761 of file pre.c.

4.19.6.222 DoSetRandom()

```
int DoSetRandom (
            UBYTE * s )  [extern]
```

Definition at line 6851 of file pre.c.

4.19.6.223 DoOptimize()

```
int DoOptimize (
            UBYTE * s )  [extern]
```

Definition at line 6894 of file pre.c.

4.19.6.224 DoClearOptimize()

```
int DoClearOptimize (
            UBYTE * s )  [extern]
```

Definition at line 7031 of file pre.c.

4.19.6.225 DoSkipExtraSymbols()

```
int DoSkipExtraSymbols (
            UBYTE * s )  [extern]
```

Definition at line 7051 of file pre.c.

4.19.6.226 DoTimeOutAfter()

```
int DoTimeOutAfter (
            UBYTE * s )  [extern]
```

Definition at line 7277 of file pre.c.

4.19.6.227 DoMessage()

```
int DoMessage (
            UBYTE * s )  [extern]
```

Definition at line 3752 of file pre.c.

4.19.6.228 DoPreOut()

```
int DoPreOut (
            UBYTE * s )  [extern]
```

Definition at line 3823 of file pre.c.

4.19.6.229 DoPreAppend()

```
int DoPreAppend (
            UBYTE * s )  [extern]
```

Definition at line 3880 of file pre.c.

4.19.6.230 DoPreCreate()

```
int DoPreCreate (
            UBYTE * s )  [extern]
```

Definition at line 3923 of file pre.c.

4.19.6.231 DoPreAssign()

```
int DoPreAssign (
            UBYTE * s )  [extern]
```

Definition at line 2143 of file pre.c.

4.19.6.232 DoPreBreak()

```
int DoPreBreak (
            UBYTE * s )  [extern]
```

Definition at line 4138 of file pre.c.

4.19.6.233 DoPreDefault()

```
int DoPreDefault (
            UBYTE * s )  [extern]
```

Definition at line 4201 of file pre.c.

4.19.6.234 DoPreSwitch()

```
int DoPreSwitch (
            UBYTE * s )  [extern]
```

Definition at line 4252 of file pre.c.

4.19.6.235 DoPreEndSwitch()

```
int DoPreEndSwitch (
            UBYTE * s )  [extern]
```

Definition at line 4225 of file pre.c.

4.19.6.236 DoPreCase()

```
int DoPreCase (
            UBYTE * s )  [extern]
```

Definition at line 4162 of file pre.c.

4.19.6.237 DoPreShow()

```
int DoPreShow (
            UBYTE * s )  [extern]
```

Definition at line 4306 of file pre.c.

4.19.6.238 DoPreExchange()

```
int DoPreExchange (
            UBYTE * s )  [extern]
```

Definition at line 2505 of file pre.c.

4.19.6.239 DoSystem()

```
int DoSystem (
            UBYTE * s )  [extern]
```

Definition at line 4345 of file pre.c.

4.19.6.240 DoPipe()

```
int DoPipe (
            UBYTE * s )  [extern]
```

Definition at line 3766 of file pre.c.

4.19.6.241 StartPrepro()

```
void StartPrepro (
            void  )  [extern]
```

Definition at line 4523 of file pre.c.

4.19.6.242 DoIfdef()

```
int DoIfdef (
            UBYTE * s,
            int par )  [extern]
```

Definition at line 3467 of file pre.c.

4.19.6.243 DoIfydef()

```
int DoIfydef (
            UBYTE * s )  [extern]
```

Definition at line 3492 of file pre.c.

4.19.6.244 DoIfndef()

```
int DoIfndef (
            UBYTE * s )  [extern]
```

Definition at line 3502 of file pre.c.

4.19.6.245 DoElse()

```
int DoElse (
            UBYTE * s )  [extern]
```

Definition at line 3170 of file pre.c.

4.19.6.246 DoElseif()

```
int DoElseif (
            UBYTE * s )  [extern]
```

Definition at line 3207 of file pre.c.

4.19.6.247 DoEndif()

```
int DoEndif (
            UBYTE * s )  [extern]
```

Definition at line 3388 of file pre.c.

4.19.6.248 DoTerminate()

```
int DoTerminate (
            UBYTE * s )  [extern]
```

Definition at line 2779 of file pre.c.

4.19.6.249 DoIf()

```
int DoIf (
            UBYTE * s )  [extern]
```

Definition at line 3443 of file pre.c.

4.19.6.250 DoCall()

```
int DoCall (
            UBYTE * s )  [extern]
```

Definition at line 2566 of file pre.c.

4.19.6.251 DoDebug()

```
int DoDebug (
            UBYTE * s )  [extern]
```

Definition at line 2742 of file pre.c.

4.19.6.252 DoContinueDo()

```
int DoContinueDo (
            UBYTE * s )  [extern]
```

Jumps forward to the corresponding `#enddo` of the specified number of outer `#do` loops.

Syntax:

```
#continuedo [<number>=1]
```

If `number` is omitted, it defaults to 1. If `number` is zero then the instruction has no effect.

Definition at line 2813 of file pre.c.

4.19.6.253 DoDo()

```
int DoDo (
            UBYTE * s )  [extern]
```

Definition at line 2877 of file pre.c.

4.19.6.254 DoBreakDo()

```
int DoBreakDo (
            UBYTE * s )  [extern]
```

Definition at line 3092 of file pre.c.

4.19.6.255 DoEnddo()

```
int DoEnddo (
            UBYTE * s )  [extern]
```

Definition at line 3242 of file pre.c.

4.19.6.256 DoEndprocedure()

```
int DoEndprocedure (
            UBYTE * s )  [extern]
```

Definition at line 3416 of file pre.c.

4.19.6.257 DoInside()

```
int DoInside (
            UBYTE * s )  [extern]
```

Definition at line 3527 of file pre.c.

4.19.6.258 DoEndInside()

```
int DoEndInside (
            UBYTE * s )  [extern]
```

Definition at line 3620 of file pre.c.

4.19.6.259 DoProcedure()

```
int DoProcedure (
            UBYTE * s )  [extern]
```

Definition at line 4081 of file pre.c.

4.19.6.260 DoPrePrintTimes()

```
int DoPrePrintTimes (
            UBYTE * s )  [extern]
```

Definition at line 3846 of file pre.c.

4.19.6.261 DoPreWrite()

```
int DoPreWrite (
            UBYTE * s )  [extern]
```

Definition at line 4040 of file pre.c.

4.19.6.262 DoPreClose()

```
int DoPreClose (
            UBYTE * s )  [extern]
```

Definition at line 3996 of file pre.c.

4.19.6.263 DoPreRemove()

```
int DoPreRemove (
            UBYTE * s )  [extern]
```

Definition at line 3963 of file pre.c.

4.19.6.264 DoPreSortReallocate()

```
int DoPreSortReallocate (
            UBYTE * s )  [extern]
```

Definition at line 3860 of file pre.c.

4.19.6.265 DoCommentChar()

```
int DoCommentChar (
            UBYTE * s )  [extern]
```

Definition at line 2112 of file pre.c.

4.19.6.266 DoPrcExtension()

```
int DoPrcExtension (
            UBYTE * s )  [extern]
```

Definition at line 3789 of file pre.c.

4.19.6.267 DoPreReset()

```
int DoPreReset (
            UBYTE * s )  [extern]
```

Definition at line 7090 of file pre.c.

4.19.6.268 WriteString()

```
void WriteString (
            int type,
            UBYTE * str,
            int num )  [extern]
```

Definition at line 1811 of file tools.c.

4.19.6.269 WriteUnfinString()

```
void WriteUnfinString (
            int type,
            UBYTE * str,
            int num )  [extern]
```

Definition at line 1845 of file tools.c.

4.19.6.270 AddToString()

```
UBYTE * AddToString (
            UBYTE * outstring,
            UBYTE * extrastring,
            int par )  [extern]
```

Definition at line 1870 of file tools.c.

4.19.6.271 PreCalc()

```
UBYTE * PreCalc (
            void  )  [extern]
```

Definition at line 5204 of file pre.c.

4.19.6.272 PreEval()

```
UBYTE * PreEval (
            UBYTE * s,
            LONG * x )  [extern]
```

Definition at line 5282 of file pre.c.

4.19.6.273 NumToStr()

```
void NumToStr (
            UBYTE * s,
            LONG x )  [extern]
```

Definition at line 1787 of file tools.c.

4.19.6.274 PreCmp()

```
int PreCmp (
            int type,
            int val,
            UBYTE * t,
            int type2,
            int val2,
            UBYTE * t2,
            int cmpop )  [extern]
```

Definition at line 4715 of file pre.c.

4.19.6.275 PreEq()

```
int PreEq (
            int type,
            int val,
            UBYTE * t,
            int type2,
            int val2,
            UBYTE * t2,
            int eqop )  [extern]
```

Definition at line 4739 of file pre.c.

4.19.6.276 pParseObject()

```
UBYTE * pParseObject (
            UBYTE * s,
            int * type,
            LONG * val2 )  [extern]
```

Definition at line 4768 of file pre.c.

4.19.6.277 PreIfEval()

```
UBYTE * PreIfEval (
            UBYTE * s,
            int * value )  [extern]
```

Definition at line 4586 of file pre.c.

4.19.6.278 EvalPreIf()

```
int EvalPreIf (
            UBYTE * s )  [extern]
```

Definition at line 4551 of file pre.c.

4.19.6.279 PreLoad()

```
int PreLoad (
            PRELOAD * p,
            UBYTE * start,
            UBYTE * stop,
            int mode,
            char * message )  [extern]
```

Definition at line 4385 of file pre.c.

4.19.6.280 PreSkip()

```
int PreSkip (
            UBYTE * start,
            UBYTE * stop,
            int mode )  [extern]
```

Definition at line 4468 of file pre.c.

4.19.6.281 EndOfToken()

```
UBYTE * EndOfToken (
            UBYTE * s )  [extern]
```

Skips over alphanumeric characters to find the end of the token.

Note

This function does not handle formal names (e.g., `[x+a]`). To handle them, use `SkipAName` instead.

Parameters

in	*s*	Pointer to a null-terminated input buffer.

Returns

Pointer to the first non-alphanumeric character (i.e., the character immediately after the token), or the null terminator if the token reaches the end of the string.

Definition at line 1934 of file tools.c.

4.19.6.282 SetSpecialMode()

```
void SetSpecialMode (
            int moduletype,
            int specialtype )  [extern]
```

Definition at line 258 of file module.c.

4.19.6.283 MakeGlobal()

```
void MakeGlobal (
            void  )  [extern]
```

Definition at line 383 of file execute.c.

4.19.6.284 ExecModule()

```
int ExecModule (
            int moduletype )  [extern]
```

Definition at line 275 of file module.c.

4.19.6.285 ExecStore()

```
int ExecStore (
            void  )  [extern]
```

Definition at line 300 of file module.c.

4.19.6.286 FullCleanUp()

```
void FullCleanUp (
            void  )  [extern]
```

Definition at line 315 of file module.c.

4.19.6.287 DoExecStatement()

```
int DoExecStatement (
            void  )  [extern]
```

Definition at line 797 of file module.c.

4.19.6.288 DoPipeStatement()

```
int DoPipeStatement (
            void  )  [extern]
```

Definition at line 814 of file module.c.

4.19.6.289 DoPolyfun()

```
int DoPolyfun (
            UBYTE * s )  [extern]
```

Definition at line 398 of file module.c.

4.19.6.290 DoPolyratfun()

```
int DoPolyratfun (
            UBYTE * s )  [extern]
```

Definition at line 461 of file module.c.

4.19.6.291 CompileStatement()

```
int CompileStatement (
            UBYTE * in )  [extern]
```

Definition at line 572 of file compiler.c.

4.19.6.292 ToToken()

```
UBYTE * ToToken (
            UBYTE * s )  [extern]
```

Skips over non-alphanumeric characters to find the start of a token.

Note

This function does not handle formal names (e.g., `[x+a]`). To handle them, consider simply skipping whitespace, e.g., by using `SkipSpaces`.

Parameters

in	*s*	Pointer to a null-terminated input buffer.

Returns

Pointer to the first alphanumeric character (i.e., the start of the token), or the null terminator if none is found.

Definition at line 1957 of file tools.c.

4.19.6.293 GetDollar()

```
int GetDollar (
            UBYTE * name )  [extern]
```

Definition at line 592 of file names.c.

4.19.6.294 MesWork()

```
void MesWork (
            void  )  [extern]
```

Definition at line 89 of file message.c.

4.19.6.295 MesPrint()

```
int MesPrint (
            const char * fmt,
             ...  )  [extern]
```

Definition at line 136 of file message.c.

4.19.6.296 MesCall()

```
int MesCall (
            char * s )  [extern]
```

Definition at line 818 of file message.c.

4.19.6.297 NumCopy()

```
UBYTE * NumCopy (
            WORD y,
            UBYTE * to )  [extern]
```

Definition at line 2029 of file tools.c.

4.19.6.298 LongCopy()

```
char * LongCopy (
            LONG y,
            char * to )  [extern]
```

Definition at line 2056 of file tools.c.

4.19.6.299 LongLongCopy()

```
char * LongLongCopy (
            off_t * y,
            char * to )  [extern]
```

Definition at line 2083 of file tools.c.

4.19.6.300 ReserveTempFiles()

```
void ReserveTempFiles (
            int par )  [extern]
```

Definition at line 713 of file startup.c.

4.19.6.301 PrintTerm()

```
void PrintTerm (
            WORD * term,
            char * where )  [extern]
```

Definition at line 861 of file message.c.

4.19.6.302 PrintTermC()

```
void PrintTermC (
            WORD * term,
            char * where )  [extern]
```

Definition at line 891 of file message.c.

4.19.6.303 PrintSubTerm()

```
void PrintSubTerm (
            WORD * term,
            char * where )  [extern]
```

Definition at line 925 of file message.c.

4.19.6.304 PrintWords()

```
void PrintWords (
            WORD * buffer,
            LONG number )  [extern]
```

Definition at line 947 of file message.c.

4.19.6.305 PrintSeq()

```
void PrintSeq (
            WORD * a,
            char * text )  [extern]
```

Definition at line 965 of file message.c.

4.19.6.306 ExpandTripleDots()

```
int ExpandTripleDots (
            int par )  [extern]
```

Definition at line 1610 of file pre.c.

4.19.6.307 ComPress()

```
LONG ComPress (
            WORD ** ss,
            LONG * n )  [extern]
```

Gets a list of pointers to terms and compresses the terms. In n it collects the number of terms and the return value of the function is the space that is occupied.

We have to pay some special attention to the compression of terms with a PolyFun. This PolyFun should occur only straight before the coefficient, so we can use the same trick as for the coefficient to sabotage compression of this object (Replace in the history the function pointer by zero. This is safe, because terms that would be identical otherwise would have been added).

Parameters

ss	Array of pointers to terms to be compressed.
n	Number of pointers in ss.

Returns

Total number of words needed for the compressed result.

Definition at line 2959 of file sort.c.

Referenced by EndSort(), and StoreTerm().

4.19.6.308 StageSort()

```
void StageSort (
            FILEHANDLE * fout )  [extern]
```

Prepares a stage 4 or higher sort. Stage 4 sorts occur when the sort file contains more patches than can be merged in one pass.

Definition at line 4437 of file sort.c.

References FiLe::handle.

Referenced by EndSort(), and MergePatches().

4.19.6.309 M_free()

```
void M_free (
            void * x,
            const char * where )  [extern]
```

Definition at line 2323 of file tools.c.

4.19.6.310 ClearWildcardNames()

```
void ClearWildcardNames (
            void  )  [extern]
```

Definition at line 853 of file names.c.

4.19.6.311 AddWildcardName()

```
int AddWildcardName (
            UBYTE * name )  [extern]
```

Definition at line 858 of file names.c.

4.19.6.312 GetWildcardName()

```
int GetWildcardName (
            UBYTE * name )  [extern]
```

Definition at line 902 of file names.c.

4.19.6.313 Globalize()

```
void Globalize (
            int par )  [extern]
```

Definition at line 3162 of file names.c.

4.19.6.314 ResetVariables()

```
void ResetVariables (
            int par )  [extern]
```

Definition at line 2839 of file names.c.

4.19.6.315 AddToPreTypes()

```
void AddToPreTypes (
            int type )  [extern]
```

Definition at line 5432 of file pre.c.

4.19.6.316 MessPreNesting()

```
void MessPreNesting (
            int par )  [extern]
```

Definition at line 5450 of file pre.c.

4.19.6.317 GetStreamPosition()

```
LONG GetStreamPosition (
            STREAM * stream )  [extern]
```

Definition at line 815 of file tools.c.

4.19.6.318 DoubleCbuffer()

```
WORD * DoubleCbuffer (
            int num,
            WORD * w,
            int par )  [extern]
```

Doubles a compiler buffer.

Parameters

num	The ID number for the buffer to be doubled.
w	The pointer to the end (exclusive) of the current buffer. The contents in the range of [cbuf[num].Buffer,w) will be kept.

Definition at line 143 of file comtool.c.

References CbUf::Buffer, CbUf::BufferSize, CbUf::lhs, CbUf::Pointer, CbUf::rhs, and CbUf::Top.

Referenced by AddNtoC(), and AddNtoL().

4.19.6.319 AddLHS()

```
WORD * AddLHS (
            int num )  [extern]
```

Adds an LHS to a compiler buffer and returns the pointer to a buffer for the new LHS.

Parameters

num	The ID number for the buffer to get another LHS.

Definition at line 184 of file comtool.c.

References CbUf::lhs, and CbUf::Pointer.

Referenced by AddNtoL().

4.19.6.320 AddRHS()

```
WORD * AddRHS (
            int num,
            int type )  [extern]
```

Adds an RHS to a compiler buffer and returns the pointer to a buffer for the new RHS.

Parameters

num	The ID number for the buffer to get another RHS.
type	If 0, the subexpression tree will be reallocated.

Definition at line 210 of file comtool.c.

References TaBlEs::buffers, TaBlEs::buffersfill, TaBlEs::bufferssize, TaBlEs::bufnum, CbUf::CanCommu, CbUf::dimension, inicbufs(), CbUf::numdum, CbUf::NumTerms, CbUf::Pointer, and CbUf::rhs.

Referenced by InsertArg(), StartVariables(), and TestMatch().

Here is the call graph for this function:

4.19.6.321 AddNtoL()

```
int AddNtoL (
            int n,
            WORD * array )  [extern]
```

Adds an LHS with the given data to the current compiler buffer.

Parameters

n	The length of the data.
array	The data to be added.

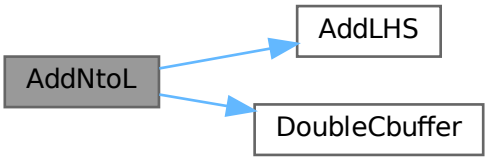
Returns

0 if succeeds.

Definition at line 284 of file comtool.c.

References AddLHS(), DoubleCbuffer(), CbUf::Pointer, and CbUf::Top.

Here is the call graph for this function:



4.19.6.322 AddNtoC()

```
int AddNtoC (
              int bufnum,
              int n,
              WORD * array,
              int par )  [extern]
```

Adds the given data to the last LHS/RHS in a compiler buffer.

Parameters

bufnum	The ID number for the buffer where the data will be added.
n	The length of the data.
array	The data to be added.

Returns

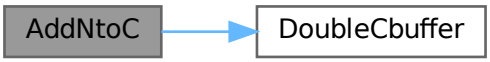
0 if succeeds.

Definition at line 313 of file comtool.c.

References DoubleCbuffer(), CbUf::Pointer, and CbUf::Top.

Referenced by InsertArg(), and StartVariables().

Here is the call graph for this function:



4.19.6.323 DoubleIfBuffers()

```
void DoubleIfBuffers (
              void  )  [extern]
```

Definition at line 1049 of file if.c.

4.19.6.324 CreateStream()

```
STREAM * CreateStream (
              UBYTE * where )  [extern]
```

Definition at line 784 of file tools.c.

4.19.6.325 setonoff()

```
int setonoff (
            UBYTE * s,
            int * flag,
            int onvalue,
            int offvalue )  [extern]
```

Definition at line 499 of file compcomm.c.

4.19.6.326 DoPrint()

```
int DoPrint (
            UBYTE * s,
            int par )  [extern]
```

Definition at line 1052 of file compcomm.c.

4.19.6.327 SetExpr()

```
int SetExpr (
            UBYTE * s,
            int setunset,
            int par )  [extern]
```

Definition at line 1523 of file compcomm.c.

4.19.6.328 AddToCom()

```
void AddToCom (
            int n,
            WORD * array )  [extern]
```

Definition at line 1660 of file compcomm.c.

4.19.6.329 Add2ComStrings()

```
int Add2ComStrings (
            int n,
            WORD * array,
            UBYTE * string1,
            UBYTE * string2 )  [extern]
```

Definition at line 1734 of file compcomm.c.

4.19.6.330 DoSymmetrize()

```
int DoSymmetrize (
            UBYTE * s,
            int par )  [extern]
```

Definition at line 2245 of file compcomm.c.

4.19.6.331 DoArgument()

```
int DoArgument (
            UBYTE * s,
            int par )  [extern]
```

Definition at line 1873 of file compcomm.c.

4.19.6.332 ArgFactorize()

```
int ArgFactorize (
            PHEAD WORD * argin,
            WORD * argout )  [extern]
```

Definition at line 2070 of file argument.c.

4.19.6.333 TakeArgContent()

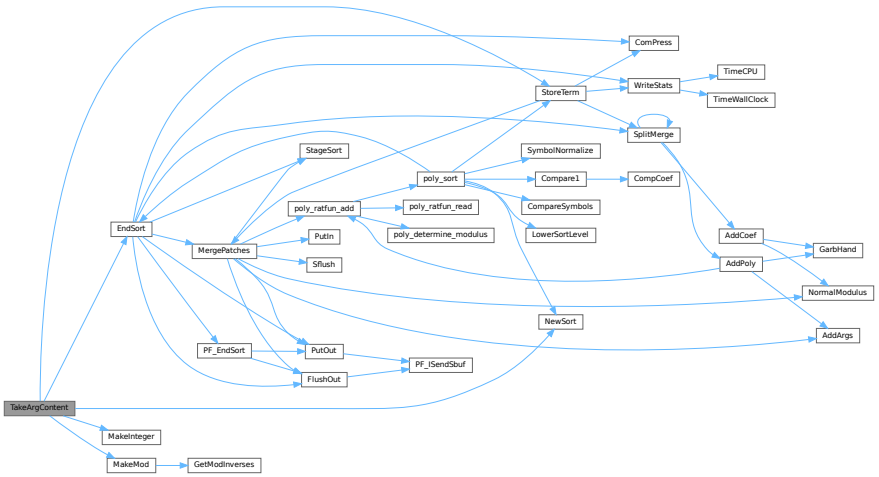
```
WORD * TakeArgContent (
            PHEAD WORD * argin,
            WORD * argout )  [extern]
```

Implements part of the old ExecArg in which we take common factors from arguments with more than one term. The common pieces are put in argout as a sequence of arguments. The part with the multiple terms that are now relative prime is put in argfree which is allocated via TermMalloc and is given as the return value. The difference with the old code is that negative powers are always removed. Hence it is as in MakeInteger in which only numerators will be left: now only zero or positive powers will be remaining.

Definition at line 2778 of file argument.c.

References EndSort(), MakeInteger(), MakeMod(), NewSort(), and StoreTerm().

Here is the call graph for this function:



4.19.6.334 MakeInteger()

```
WORD * MakeInteger (
            PHEAD WORD * argin,
            WORD * argout,
            WORD * argfree )  [extern]
```

For normalizing everything to integers we have to determine for all elements of this argument the LCM of the denominators and the GCD of the numerators. The input argument is in argin. The number that comes out should go to argout. The new pointer in the argout buffer is the return value. The normalized argument is in argfree.

Definition at line 3328 of file argument.c.

Referenced by TakeArgContent().

4.19.6.335 MakeMod()

```
WORD * MakeMod (
            PHEAD WORD * argin,
            WORD * argout,
            WORD * argfree )  [extern]
```

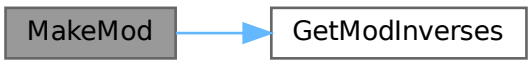
Similar to MakeInteger but now with modulus arithmetic using only a one WORD 'prime'. We make the coefficient of the first term in the argument equal to one. Already the coefficients are taken modulus AN.cmod and AN.ncmod == 1

Definition at line 3499 of file argument.c.

References GetModInverses().

Referenced by TakeArgContent().

Here is the call graph for this function:



4.19.6.336 FindArg()

```
WORD FindArg (
            PHEAD WORD * a )  [extern]
```

Looks the argument up in the (workers) table. If it is found the number in the table is returned (plus one to make it positive). If it is not found we look in the compiler provided table. If it is found - the number in the table is returned (minus one to make it negative). If in neither table we return zero.

Definition at line 2525 of file argument.c.

4.19.6.337 InsertArg()

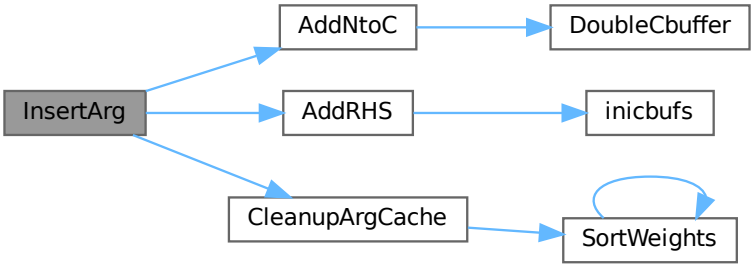
```
int InsertArg (
            PHEAD WORD * argin,
            WORD * argout,
            int par )  [extern]
```

Inserts the argument into the (workers) table. If the table is too full we eliminate half of it. The eliminated elements are the ones that have not been used most recently, weighted by their total use and age(?). If par == 0 it inserts in the regular factorization cache If par == 1 it inserts in the cache defined with the FactorCache statement

Definition at line 2549 of file argument.c.

References AddNtoC(), AddRHS(), and CleanupArgCache().

Here is the call graph for this function:



4.19.6.338 CleanupArgCache()

```
int CleanupArgCache (
            PHEAD WORD bufnum )  [extern]
```

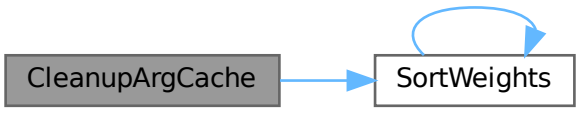
Cleans up the argument factorization cache. We throw half the elements. For a weight of what we want to keep we use the product of usage and the number in the buffer.

Definition at line 2584 of file argument.c.

References CbUf::boomlijst, CbUf::Buffer, CbUf::Pointer, CbUf::rhs, SortWeights(), and tree::usage.

Referenced by InsertArg().

Here is the call graph for this function:



4.19.6.339 ArgSymbolMerge()

```
int ArgSymbolMerge (
            WORD * t1,
            WORD * t2 )  [extern]
```

Definition at line 2655 of file argument.c.

4.19.6.340 ArgDotproductMerge()

```
int ArgDotproductMerge (
            WORD * t1,
            WORD * t2 )  [extern]
```

Definition at line 2710 of file argument.c.

4.19.6.341 SortWeights()

```
void SortWeights (
            LONG * weights,
            LONG * extraspace,
            WORD number )  [extern]
```

Sorts an array of LONGS in the same way SplitMerge (in sort.c) works We use gradual division in two.

Definition at line 3544 of file argument.c.

References SortWeights().

Referenced by CleanupArgCache(), and SortWeights().

Here is the call graph for this function:

4.19.6.342 DoBrackets()

```
int DoBrackets (
            UBYTE * inp,
            int par )  [extern]
```

Definition at line 3780 of file compcomm.c.

4.19.6.343 DoPutInside()

```
int DoPutInside (
            UBYTE * inp,
            int par )  [extern]
```

Definition at line 7080 of file compcomm.c.

4.19.6.344 CountComp()

```
WORD * CountComp (
            UBYTE * inp,
            WORD * to )  [extern]
```

Definition at line 4058 of file compcomm.c.

4.19.6.345 CoAntiBracket()

```
int CoAntiBracket (
            UBYTE * inp )  [extern]
```

Definition at line 3915 of file compcomm.c.

4.19.6.346 CoAntiSymmetrize()

```
int CoAntiSymmetrize (
            UBYTE * s )  [extern]
```

Definition at line 2372 of file compcomm.c.

4.19.6.347 DoArgPlode()

```
int DoArgPlode (
            UBYTE * s,
            int par )  [extern]
```

Definition at line 5740 of file compcomm.c.

4.19.6.348 CoArgExplode()

```
int CoArgExplode (
            UBYTE * s )  [extern]
```

Definition at line 5796 of file compcomm.c.

4.19.6.349 CoArgImplode()

```
int CoArgImplode (
            UBYTE * s )  [extern]
```

Definition at line 5803 of file compcomm.c.

4.19.6.350 CoArgument()

```
int CoArgument (
            UBYTE * s )  [extern]
```

Definition at line 2124 of file compcomm.c.

4.19.6.351 CoInside()

```
int CoInside (
            UBYTE * s )  [extern]
```

Definition at line 2157 of file compcomm.c.

4.19.6.352 ExecInside()

```
int ExecInside (
            UBYTE * s )  [extern]
```

Definition at line 1667 of file dollar.c.

4.19.6.353 CoInExpression()

```
int CoInExpression (
            UBYTE * s )  [extern]
```

Definition at line 3385 of file compcomm.c.

4.19.6.354 CoInParallel()

```
int CoInParallel (
            UBYTE * s )  [extern]
```

Definition at line 3291 of file compcomm.c.

4.19.6.355 CoNotInParallel()

```
int CoNotInParallel (
            UBYTE * s )  [extern]
```

Definition at line 3301 of file compcomm.c.

4.19.6.356 DoInParallel()

```
int DoInParallel (
            UBYTE * s,
            int par )  [extern]
```

Definition at line 3316 of file compcomm.c.

4.19.6.357 CoEndInExpression()

```
int CoEndInExpression (
            UBYTE * s )  [extern]
```

Definition at line 3438 of file compcomm.c.

4.19.6.358 CoBracket()

```
int CoBracket (
            UBYTE * inp )  [extern]
```

Definition at line 3907 of file compcomm.c.

4.19.6.359 CoPutInside()

```
int CoPutInside (
            UBYTE * inp )  [extern]
```

Definition at line 7077 of file compcomm.c.

4.19.6.360 CoAntiPutInside()

```
int CoAntiPutInside (
            UBYTE * inp )  [extern]
```

Definition at line 7078 of file compcomm.c.

4.19.6.361 CoMultiBracket()

```
int CoMultiBracket (
            UBYTE * inp )  [extern]
```

Definition at line 3926 of file compcomm.c.

4.19.6.362 CoCFunction()

```
int CoCFunction (
            UBYTE * s )  [extern]
```

Definition at line 1631 of file names.c.

4.19.6.363 CoCTensor()

```
int CoCTensor (
            UBYTE * s )  [extern]
```

Definition at line 1633 of file names.c.

4.19.6.364 CoCollect()

```
int CoCollect (
            UBYTE * s )  [extern]
```

Definition at line 437 of file compcomm.c.

4.19.6.365 CoCompress()

```
int CoCompress (
            UBYTE * s )  [extern]
```

Definition at line 515 of file compcomm.c.

4.19.6.366 CoContract()

```
int CoContract (
            UBYTE * s )  [extern]
```

Definition at line 1798 of file compcomm.c.

4.19.6.367 CoCycleSymmetrize()

```
int CoCycleSymmetrize (
            UBYTE * s )  [extern]
```

Definition at line 2379 of file compcomm.c.

4.19.6.368 CoDelete()

```
int CoDelete (
            UBYTE * s )  [extern]
```

Definition at line 951 of file compcomm.c.

4.19.6.369 CoTableBase()

```
int CoTableBase (
            UBYTE * s )  [extern]
```

Definition at line 633 of file tables.c.

4.19.6.370 CoApply()

```
int CoApply (
            UBYTE * s )  [extern]
```

Definition at line 1809 of file tables.c.

4.19.6.371 CoDenominators()

```
int CoDenominators (
            UBYTE * s )  [extern]
```

Definition at line 5892 of file compcomm.c.

4.19.6.372 CoDimension()

```
int CoDimension (
            UBYTE * s )  [extern]
```

Definition at line 1230 of file names.c.

4.19.6.373 CoDiscard()

```
int CoDiscard (
            UBYTE * s )  [extern]
```

Definition at line 1775 of file compcomm.c.

4.19.6.374 CoDisorder()

```
int CoDisorder (
            UBYTE * inp )  [extern]
```

Definition at line 420 of file comexpr.c.

4.19.6.375 CoDrop()

```
int CoDrop (
            UBYTE * s )  [extern]
```

Definition at line 1568 of file compcomm.c.

4.19.6.376 CoDropCoefficient()

```
int CoDropCoefficient (
            UBYTE * s )  [extern]
```

Definition at line 5921 of file compcomm.c.

4.19.6.377 CoDropSymbols()

```
int CoDropSymbols (
            UBYTE * s )  [extern]
```

Definition at line 5935 of file compcomm.c.

4.19.6.378 CoElse()

```
int CoElse (
            UBYTE * p )  [extern]
```

Definition at line 4895 of file compcomm.c.

4.19.6.379 CoElseIf()

```
int CoElseIf (
            UBYTE * inp )  [extern]
```

Definition at line 4922 of file compcomm.c.

4.19.6.380 CoEndArgument()

```
int CoEndArgument (
            UBYTE * s )  [extern]
```

Definition at line 2131 of file compcomm.c.

4.19.6.381 CoEndInside()

```
int CoEndInside (
            UBYTE * s )  [extern]
```

Definition at line 2164 of file compcomm.c.

4.19.6.382 CoEndIf()

```
int CoEndIf (
            UBYTE * inp )  [extern]
```

Definition at line 4949 of file compcomm.c.

4.19.6.383 CoEndRepeat()

```
int CoEndRepeat (
            UBYTE * inp )  [extern]
```

Definition at line 3742 of file compcomm.c.

4.19.6.384 CoEndTerm()

```
int CoEndTerm (
            UBYTE * s )  [extern]
```

Definition at line 5298 of file compcomm.c.

4.19.6.385 CoEndWhile()

```
int CoEndWhile (
            UBYTE * inp )  [extern]
```

Definition at line 5015 of file compcomm.c.

4.19.6.386 CoExit()

```
int CoExit (
            UBYTE * s )  [extern]
```

Definition at line 3264 of file compcomm.c.

4.19.6.387 CoFactArg()

```
int CoFactArg (
            UBYTE * s )  [extern]
```

Definition at line 2225 of file compcomm.c.

4.19.6.388 CoFactDollar()

```
int CoFactDollar (
            UBYTE * inp )  [extern]
```

Definition at line 6451 of file compcomm.c.

4.19.6.389 CoFactorize()

```
int CoFactorize (
            UBYTE * s )  [extern]
```

Definition at line 6480 of file compcomm.c.

4.19.6.390 CoNFactorize()

```
int CoNFactorize (
            UBYTE * s )  [extern]
```

Definition at line 6487 of file compcomm.c.

4.19.6.391 CoUnFactorize()

```
int CoUnFactorize (
            UBYTE * s )  [extern]
```

Definition at line 6494 of file compcomm.c.

4.19.6.392 CoNUnFactorize()

```
int CoNUnFactorize (
            UBYTE * s )  [extern]
```

Definition at line 6501 of file compcomm.c.

4.19.6.393 DoFactorize()

```
int DoFactorize (
            UBYTE * s,
            int par )  [extern]
```

Definition at line 6508 of file compcomm.c.

4.19.6.394 CoFill()

```
int CoFill (
            UBYTE * inp )  [extern]
```

Definition at line 1073 of file comexpr.c.

4.19.6.395 CoFillExpression()

```
int CoFillExpression (
            UBYTE * inp )  [extern]
```

Definition at line 1359 of file comexpr.c.

4.19.6.396 CoFixIndex()

```
int CoFixIndex (
            UBYTE * s )  [extern]
```

Definition at line 1010 of file compcomm.c.

4.19.6.397 CoFormat()

```
int CoFormat (
            UBYTE * s )  [extern]
```

Definition at line 157 of file compcomm.c.

4.19.6.398 CoGlobal()

```
int CoGlobal (
            UBYTE * inp )  [extern]
```

Definition at line 73 of file comexpr.c.

4.19.6.399 CoGlobalFactorized()

```
int CoGlobalFactorized (
            UBYTE * inp )  [extern]
```

Definition at line 87 of file comexpr.c.

4.19.6.400 CoGoTo()

```
int CoGoTo (
            UBYTE * inp )  [extern]
```

Definition at line 1827 of file compcomm.c.

4.19.6.401 CoId()

```
int CoId (
            UBYTE * inp )  [extern]
```

Definition at line 398 of file comexpr.c.

4.19.6.402 CoIdNew()

```
int CoIdNew (
            UBYTE * inp )  [extern]
```

Definition at line 409 of file comexpr.c.

4.19.6.403 CoIdOld()

```
int CoIdOld (
            UBYTE * inp )  [extern]
```

Definition at line 387 of file comexpr.c.

4.19.6.404 CoIf()

```
int CoIf (
            UBYTE * inp )  [extern]
```

Definition at line 4248 of file compcomm.c.

4.19.6.405 CoIfMatch()

```
int CoIfMatch (
            UBYTE * inp )  [extern]
```

Definition at line 453 of file comexpr.c.

4.19.6.406 CoIfNoMatch()

```
int CoIfNoMatch (
            UBYTE * inp )  [extern]
```

Definition at line 464 of file comexpr.c.

4.19.6.407 CoIndex()

```
int CoIndex (
            UBYTE * s )  [extern]
```

Definition at line 1116 of file names.c.

4.19.6.408 CoInsideFirst()

```
int CoInsideFirst (
            UBYTE * s )  [extern]
```

Definition at line 937 of file compcomm.c.

4.19.6.409 CoKeep()

```
int CoKeep (
            UBYTE * s )  [extern]
```

Definition at line 998 of file compcomm.c.

4.19.6.410 CoLabel()

```
int CoLabel (
            UBYTE * inp )  [extern]
```

Definition at line 1846 of file compcomm.c.

4.19.6.411 CoLoad()

```
int CoLoad (
            UBYTE * inp )  [extern]
```

Definition at line 594 of file store.c.

4.19.6.412 CoLocal()

```
int CoLocal (
            UBYTE * inp )  [extern]
```

Definition at line 66 of file comexpr.c.

4.19.6.413 CoLocalFactorized()

```
int CoLocalFactorized (
            UBYTE * inp )  [extern]
```

Definition at line 80 of file comexpr.c.

4.19.6.414 CoMany()

```
int CoMany (
            UBYTE * inp )  [extern]
```

Definition at line 431 of file comexpr.c.

4.19.6.415 CoMerge()

```
int CoMerge (
            UBYTE * inp )  [extern]
```

Definition at line 5581 of file compcomm.c.

4.19.6.416 CoStuffle()

```
int CoStuffle (
            UBYTE * inp )  [extern]
```

Definition at line 5637 of file compcomm.c.

4.19.6.417 CoMetric()

```
int CoMetric (
            UBYTE * s )  [extern]
```

Definition at line 1044 of file compcomm.c.

4.19.6.418 CoModOption()

```
int CoModOption (
            UBYTE * s )  [extern]
```

Definition at line 213 of file module.c.

4.19.6.419 CoModuleOption()

```
int CoModuleOption (
            UBYTE * s )  [extern]
```

Definition at line 143 of file module.c.

4.19.6.420 CoModulus()

```
int CoModulus (
            UBYTE * inp )  [extern]
```

Definition at line 3581 of file compcomm.c.

4.19.6.421 CoMulti()

```
int CoMulti (
            UBYTE * inp )  [extern]
```

Definition at line 442 of file comexpr.c.

4.19.6.422 CoMultiply()

```
int CoMultiply (
            UBYTE * inp )  [extern]
```

Definition at line 1034 of file comexpr.c.

4.19.6.423 CoNFunction()

```
int CoNFunction (
            UBYTE * s )  [extern]
```

Definition at line 1630 of file names.c.

4.19.6.424 CoNPrint()

```
int CoNPrint (
            UBYTE * s )  [extern]
```

Definition at line 1282 of file compcomm.c.

4.19.6.425 CoNTensor()

```
int CoNTensor (
            UBYTE * s )  [extern]
```

Definition at line 1632 of file names.c.

4.19.6.426 CoNWrite()

```
int CoNWrite (
            UBYTE * s )  [extern]
```

Definition at line 2418 of file compcomm.c.

4.19.6.427 CoNoDrop()

```
int CoNoDrop (
            UBYTE * s )  [extern]
```

Definition at line 1575 of file compcomm.c.

4.19.6.428 CoNoSkip()

```
int CoNoSkip (
            UBYTE * s )  [extern]
```

Definition at line 1589 of file compcomm.c.

4.19.6.429 CoNormalize()

```
int CoNormalize (
            UBYTE * s )  [extern]
```

Definition at line 2190 of file compcomm.c.

4.19.6.430 CoMakeInteger()

```
int CoMakeInteger (
            UBYTE * s )  [extern]
```

Definition at line 2197 of file compcomm.c.

4.19.6.431 CoFlags()

```
int CoFlags (
            UBYTE * s,
            int value )  [extern]
```

Definition at line 564 of file compcomm.c.

4.19.6.432 CoOff()

```
int CoOff (
            UBYTE * s )  [extern]
```

Definition at line 599 of file compcomm.c.

4.19.6.433 CoOn()

```
int CoOn (
            UBYTE * s )  [extern]
```

Definition at line 665 of file compcomm.c.

4.19.6.434 CoOnce()

```
int CoOnce (
            UBYTE * inp )  [extern]
```

Definition at line 475 of file comexpr.c.

4.19.6.435 CoOnly()

```
int CoOnly (
            UBYTE * inp )  [extern]
```

Definition at line 486 of file comexpr.c.

4.19.6.436 CoOptimizeOption()

```
int CoOptimizeOption (
            UBYTE * s )  [extern]
```

Definition at line 6641 of file compcomm.c.

4.19.6.437 CoPolyFun()

```
int CoPolyFun (
            UBYTE * s )  [extern]
```

Definition at line 5364 of file compcomm.c.

4.19.6.438 CoPolyRatFun()

```
int CoPolyRatFun (
            UBYTE * s )  [extern]
```

Definition at line 5411 of file compcomm.c.

4.19.6.439 CoPrint()

```
int CoPrint (
            UBYTE * s )  [extern]
```

Definition at line 1268 of file compcomm.c.

4.19.6.440 CoPrintB()

```
int CoPrintB (
            UBYTE * s )  [extern]
```

Definition at line 1275 of file compcomm.c.

4.19.6.441 CoProperCount()

```
int CoProperCount (
            UBYTE * s )  [extern]
```

Definition at line 944 of file compcomm.c.

4.19.6.442 CoUnitTrace()

```
int CoUnitTrace (
            UBYTE * s )  [extern]
```

Definition at line 5215 of file compcomm.c.

4.19.6.443 CoRCycleSymmetrize()

```
int CoRCycleSymmetrize (
            UBYTE * s )  [extern]
```

Definition at line 2386 of file compcomm.c.

4.19.6.444 CoRatio()

```
int CoRatio (
            UBYTE * s )  [extern]
```

Definition at line 2445 of file compcomm.c.

4.19.6.445 CoRedefine()

```
int CoRedefine (
            UBYTE * s )  [extern]
```

Definition at line 2485 of file compcomm.c.

4.19.6.446 CoRenumber()

```
int CoRenumber (
            UBYTE * s )  [extern]
```

Definition at line 2590 of file compcomm.c.

4.19.6.447 CoRepeat()

```
int CoRepeat (
            UBYTE * inp )  [extern]
```

Definition at line 3719 of file compcomm.c.

4.19.6.448 CoSave()

```
int CoSave (
            UBYTE * inp )  [extern]
```

Definition at line 378 of file store.c.

4.19.6.449 CoSelect()

```
int CoSelect (
            UBYTE * inp )  [extern]
```

Definition at line 497 of file comexpr.c.

4.19.6.450 CoSet()

```
int CoSet (
            UBYTE * s )  [extern]
```

Definition at line 2361 of file names.c.

4.19.6.451 CoSetExitFlag()

```
int CoSetExitFlag (
            UBYTE * s )  [extern]
```

Definition at line 3464 of file compcomm.c.

4.19.6.452 CoSkip()

```
int CoSkip (
            UBYTE * s )  [extern]
```

Definition at line 1582 of file compcomm.c.

4.19.6.453 CoProcessBucket()

```
int CoProcessBucket (
            UBYTE * s )  [extern]
```

Definition at line 5692 of file compcomm.c.

4.19.6.454 CoPushHide()

```
int CoPushHide (
            UBYTE * s )  [extern]
```

Definition at line 1289 of file compcomm.c.

4.19.6.455 CoPopHide()

```
int CoPopHide (
            UBYTE * s )  [extern]
```

Definition at line 1333 of file compcomm.c.

4.19.6.456 CoHide()

```
int CoHide (
            UBYTE * inp )  [extern]
```

Definition at line 1596 of file compcomm.c.

4.19.6.457 CoIntoHide()

```
int CoIntoHide (
            UBYTE * inp )  [extern]
```

Definition at line 1614 of file compcomm.c.

4.19.6.458 CoNoIntoHide()

```
int CoNoIntoHide (
            UBYTE * inp )  [extern]
```

Definition at line 1632 of file compcomm.c.

4.19.6.459 CoNoHide()

```
int CoNoHide (
            UBYTE * inp )  [extern]
```

Definition at line 1639 of file compcomm.c.

4.19.6.460 CoUnHide()

```
int CoUnHide (
            UBYTE * inp )  [extern]
```

Definition at line 1646 of file compcomm.c.

4.19.6.461 CoNoUnHide()

```
int CoNoUnHide (
            UBYTE * inp )  [extern]
```

Definition at line 1653 of file compcomm.c.

4.19.6.462 CoSort()

```
int CoSort (
            UBYTE * s )  [extern]
```

Definition at line 5325 of file compcomm.c.

4.19.6.463 CoSplitArg()

```
int CoSplitArg (
            UBYTE * s )  [extern]
```

Definition at line 2204 of file compcomm.c.

4.19.6.464 CoSplitFirstArg()

```
int CoSplitFirstArg (
            UBYTE * s )  [extern]
```

Definition at line 2211 of file compcomm.c.

4.19.6.465 CoSplitLastArg()

```
int CoSplitLastArg (
            UBYTE * s )  [extern]
```

Definition at line 2218 of file compcomm.c.

4.19.6.466 CoSum()

```
int CoSum (
            UBYTE * s )  [extern]
```

Definition at line 2611 of file compcomm.c.

4.19.6.467 CoSymbol()

```
int CoSymbol (
            UBYTE * s )  [extern]
```

Definition at line 950 of file names.c.

4.19.6.468 CoSymmetrize()

```
int CoSymmetrize (
            UBYTE * s )  [extern]
```

Definition at line 2365 of file compcomm.c.

4.19.6.469 DoTable()

```
int DoTable (
            UBYTE * s,
            int par )  [extern]
```

Definition at line 1672 of file names.c.

4.19.6.470 CoTable()

```
int CoTable (
            UBYTE * s )  [extern]
```

Definition at line 2054 of file names.c.

4.19.6.471 CoTerm()

```
int CoTerm (
            UBYTE * s )  [extern]
```

Definition at line 5250 of file compcomm.c.

4.19.6.472 CoNTable()

```
int CoNTable (
            UBYTE * s )  [extern]
```

Definition at line 2064 of file names.c.

4.19.6.473 CoCTable()

```
int CoCTable (
            UBYTE * s )  [extern]
```

Definition at line 2074 of file names.c.

4.19.6.474 EmptyTable()

```
void EmptyTable (
            TABLES T )  [extern]
```

Definition at line 2084 of file names.c.

4.19.6.475 CoToTensor()

```
int CoToTensor (
            UBYTE * s )  [extern]
```

Definition at line 2731 of file compcomm.c.

4.19.6.476 CoToVector()

```
int CoToVector (
            UBYTE * s )  [extern]
```

Definition at line 2899 of file compcomm.c.

4.19.6.477 CoTrace4()

```
int CoTrace4 (
            UBYTE * s )  [extern]
```

Definition at line 2969 of file compcomm.c.

4.19.6.478 CoTraceN()

```
int CoTraceN (
            UBYTE * s )  [extern]
```

Definition at line 3056 of file compcomm.c.

4.19.6.479 CoChisholm()

```
int CoChisholm (
            UBYTE * s )  [extern]
```

Definition at line 3116 of file compcomm.c.

4.19.6.480 CoTransform()

```
int CoTransform (
            UBYTE * in )  [extern]
```

Definition at line 87 of file transform.c.

4.19.6.481 CoClearTable()

```
int CoClearTable (
            UBYTE * s )  [extern]
```

Definition at line 5810 of file compcomm.c.

4.19.6.482 DoChain()

```
int DoChain (
            UBYTE * s,
            int option )  [extern]
```

Definition at line 3198 of file compcomm.c.

4.19.6.483 CoChainin()

```
int CoChainin (
            UBYTE * s )  [extern]
```

Definition at line 3242 of file compcomm.c.

4.19.6.484 CoChainout()

```
int CoChainout (
            UBYTE * s )  [extern]
```

Definition at line 3254 of file compcomm.c.

4.19.6.485 CoTryReplace()

```
int CoTryReplace (
            UBYTE * p )  [extern]
```

Definition at line 3478 of file compcomm.c.

4.19.6.486 CoVector()

```
int CoVector (
            UBYTE * s )  [extern]
```

Definition at line 1271 of file names.c.

4.19.6.487 CoWhile()

```
int CoWhile (
            UBYTE * inp )  [extern]
```

Definition at line 4994 of file compcomm.c.

4.19.6.488 CoWrite()

```
int CoWrite (
            UBYTE * s )  [extern]
```

Definition at line 2393 of file compcomm.c.

4.19.6.489 CoAuto()

```
int CoAuto (
            UBYTE * inp )  [extern]
```

Definition at line 2580 of file names.c.

4.19.6.490 CoSwitch()

```
int CoSwitch (
            UBYTE * s )  [extern]
```

Definition at line 7200 of file compcomm.c.

4.19.6.491 CoCase()

```
int CoCase (
            UBYTE * s )  [extern]
```

Definition at line 7246 of file compcomm.c.

4.19.6.492 CoBreak()

```
int CoBreak (
            UBYTE * s )  [extern]
```

Definition at line 7296 of file compcomm.c.

4.19.6.493 CoDefault()

```
int CoDefault (
            UBYTE * s )  [extern]
```

Definition at line 7322 of file compcomm.c.

4.19.6.494 CoEndSwitch()

```
int CoEndSwitch (
            UBYTE * s )  [extern]
```

Definition at line 7349 of file compcomm.c.

4.19.6.495 CoTBaddto()

```
int CoTBaddto (
            UBYTE * s )  [extern]
```

Definition at line 807 of file tables.c.

4.19.6.496 CoTBaudit()

```
int CoTBaudit (
            UBYTE * s )  [extern]
```

Definition at line 1482 of file tables.c.

4.19.6.497 CoTBcleanup()

```
int CoTBcleanup (
            UBYTE * s )  [extern]
```

Definition at line 1584 of file tables.c.

4.19.6.498 CoTBcreate()

```
int CoTBcreate (
            UBYTE * s )  [extern]
```

Definition at line 762 of file tables.c.

4.19.6.499 CoTBenter()

```
int CoTBenter (
            UBYTE * s )  [extern]
```

Definition at line 980 of file tables.c.

4.19.6.500 CoTBhelp()

```
int CoTBhelp (
            UBYTE * s )  [extern]
```

Definition at line 1896 of file tables.c.

4.19.6.501 CoTBload()

```
int CoTBload (
            UBYTE * ss )  [extern]
```

Definition at line 1258 of file tables.c.

4.19.6.502 CoTBoff()

```
int CoTBoff (
            UBYTE * s )  [extern]
```

Definition at line 1553 of file tables.c.

4.19.6.503 CoTBon()

```
int CoTBon (
            UBYTE * s )  [extern]
```

Definition at line 1522 of file tables.c.

4.19.6.504 CoTBopen()

```
int CoTBopen (
            UBYTE * s )  [extern]
```

Definition at line 778 of file tables.c.

4.19.6.505 CoTBreplace()

```
int CoTBreplace (
            UBYTE * s )  [extern]
```

Definition at line 1596 of file tables.c.

4.19.6.506 CoTBuse()

```
int CoTBuse (
            UBYTE * s )  [extern]
```

Definition at line 1611 of file tables.c.

4.19.6.507 CoTestUse()

```
int CoTestUse (
            UBYTE * s )  [extern]
```

Definition at line 1139 of file tables.c.

4.19.6.508 CoThreadBucket()

```
int CoThreadBucket (
            UBYTE * s )  [extern]
```

Definition at line 5710 of file compcomm.c.

4.19.6.509 AddComString()

```
int AddComString (
            int n,
            WORD * array,
            UBYTE * thestring,
            int par )  [extern]
```

Definition at line 1675 of file compcomm.c.

4.19.6.510 CompileAlgebra()

```
int CompileAlgebra (
            UBYTE * s,
            int leftright,
            WORD * prototype )  [extern]
```

Definition at line 536 of file compiler.c.

4.19.6.511 IsIdStatement()

```
int IsIdStatement (
            UBYTE * s )  [extern]
```

Definition at line 522 of file compiler.c.

4.19.6.512 IsRHS()

```
UBYTE * IsRHS (
            UBYTE * s,
            UBYTE c )  [extern]
```

Definition at line 476 of file compiler.c.

4.19.6.513 ParenthesesTest()

```
int ParenthesesTest (
            UBYTE * sin )  [extern]
```

Definition at line 385 of file compiler.c.

4.19.6.514 tokenize()

```
int tokenize (
            UBYTE * in,
            WORD leftright )  [extern]
```

Definition at line 58 of file token.c.

4.19.6.515 WriteTokens()

```
void WriteTokens (
            SBYTE * in )  [extern]
```

Definition at line 652 of file token.c.

4.19.6.516 simp1token()

```
int simp1token (
            SBYTE * s )  [extern]
```

Definition at line 700 of file token.c.

4.19.6.517 simpwtoken()

```
int simpwtoken (
            SBYTE * s )  [extern]
```

Definition at line 789 of file token.c.

4.19.6.518 simp2token()

```
int simp2token (
            SBYTE * s )  [extern]
```

Definition at line 943 of file token.c.

4.19.6.519 simp3atoken()

```
int simp3atoken (
            SBYTE * s,
            int mode )  [extern]
```

Definition at line 1191 of file token.c.

4.19.6.520 simp3btoken()

```
int simp3btoken (
            SBYTE * s,
            int mode )  [extern]
```

Definition at line 1432 of file token.c.

4.19.6.521 simp4token()

```
int simp4token (
            SBYTE * s )  [extern]
```

Definition at line 1843 of file token.c.

4.19.6.522 simp5token()

```
int simp5token (
            SBYTE * s,
            int mode )  [extern]
```

Definition at line 2005 of file token.c.

4.19.6.523 simp6token()

```
int simp6token (
            SBYTE * tokens,
            int mode )  [extern]
```

Definition at line 2051 of file token.c.

4.19.6.524 SkipAName()

```
UBYTE * SkipAName (
            UBYTE * s )  [extern]
```

Skips a name and gives a pointer to the character after the name. If there is not a proper name, it emits a compiler message and then returns a null pointer. This function supports formal names (e.g., `[x+a]`) and $-variables in addition to ordinary identifiers.

Note

The brackets are assumed to be matched already.

Parameters

in	*s*	Pointer to a null-terminated input string that starts with a name.

Returns

Pointer to the character after the name, or a null pointer if the name is invalid.

Definition at line 443 of file compiler.c.

4.19.6.525 TestTables()

```
int TestTables (
            void  )  [extern]
```

Definition at line 706 of file compiler.c.

4.19.6.526 GetLabel()

```
int GetLabel (
            UBYTE * name )  [extern]
```

Definition at line 2794 of file names.c.

4.19.6.527 CoIdExpression()

```
int CoIdExpression (
            UBYTE * inp,
            int type )  [extern]
```

Definition at line 510 of file comexpr.c.

4.19.6.528 CoAssign()

```
int CoAssign (
            UBYTE * inp )  [extern]
```

Definition at line 1929 of file comexpr.c.

4.19.6.529 DoExpr()

```
int DoExpr (
            UBYTE * inp,
            int type,
            int par )  [extern]
```

Definition at line 96 of file comexpr.c.

4.19.6.530 CompileSubExpressions()

```
int CompileSubExpressions (
            SBYTE * tokens )  [extern]
```

Definition at line 750 of file compiler.c.

4.19.6.531 CodeGenerator()

```
int CodeGenerator (
            SBYTE * tokens )  [extern]
```

Definition at line 885 of file compiler.c.

4.19.6.532 CompleteTerm()

```
int CompleteTerm (
            WORD * term,
            UWORD * numer,
            UWORD * denom,
            WORD nnum,
            WORD nden,
            int sign )  [extern]
```

Definition at line 1962 of file compiler.c.

4.19.6.533 CodeFactors()

```
int CodeFactors (
            SBYTE * s )  [extern]
```

Definition at line 1999 of file compiler.c.

4.19.6.534 GenerateFactors()

```
WORD GenerateFactors (
            WORD n,
            WORD inc )  [extern]
```

Definition at line 2300 of file compiler.c.

4.19.6.535 InsTree()

```
int InsTree (
            int bufnum,
            int h )  [extern]
```

Definition at line 351 of file comtool.c.

4.19.6.536 FindTree()

```
int FindTree (
            int bufnum,
            WORD * subexpr )  [extern]
```

Definition at line 531 of file comtool.c.

4.19.6.537 RedoTree()

```
void RedoTree (
            CBUF * C,
            int size )  [extern]
```

Definition at line 567 of file comtool.c.

4.19.6.538 ClearTree()

```
void ClearTree (
            int i )  [extern]
```

Definition at line 586 of file comtool.c.

4.19.6.539 CatchDollar()

```
int CatchDollar (
            int par )  [extern]
```

Definition at line 52 of file dollar.c.

4.19.6.540 AssignDollar()

```
int AssignDollar (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 207 of file dollar.c.

4.19.6.541 WriteDollarToBuffer()

```
UBYTE * WriteDollarToBuffer (
            WORD numdollar,
            WORD par )  [extern]
```

Definition at line 581 of file dollar.c.

4.19.6.542 WriteDollarFactorToBuffer()

```
UBYTE * WriteDollarFactorToBuffer (
            WORD numdollar,
            WORD numfac,
            WORD par )  [extern]
```

Definition at line 672 of file dollar.c.

4.19.6.543 AddToDollarBuffer()

```
void AddToDollarBuffer (
            UBYTE * s )  [extern]
```

Definition at line 747 of file dollar.c.

4.19.6.544 PutTermInDollar()

```
int PutTermInDollar (
            WORD * term,
            WORD numdollar )  [extern]
```

Definition at line 848 of file dollar.c.

4.19.6.545 TermAssign()

```
void TermAssign (
            WORD * term )  [extern]
```

Definition at line 788 of file dollar.c.

4.19.6.546 WildDollars()

```
void WildDollars (
            PHEAD WORD * term )  [extern]
```

Definition at line 876 of file dollar.c.

4.19.6.547 numcommute()

```
LONG numcommute (
            WORD * terms,
            LONG * numterms )  [extern]
```

Definition at line 657 of file comtool.c.

4.19.6.548 FullRenumber()

```
int FullRenumber (
            PHEAD WORD * term,
            WORD par )  [extern]
```

Definition at line 326 of file reshuf.c.

4.19.6.549 Lus()

```
int Lus (
            WORD * term,
            WORD funnum,
            WORD loopsize,
            WORD numargs,
            WORD outfun,
            WORD mode )  [extern]
```

Definition at line 57 of file lus.c.

4.19.6.550 FindLus()

```
int FindLus (
            int from,
            int level,
            int openindex )  [extern]
```

Definition at line 515 of file lus.c.

4.19.6.551 CoReplaceLoop()

```
int CoReplaceLoop (
            UBYTE * inp )  [extern]
```

Definition at line 5168 of file compcomm.c.

4.19.6.552 CoFindLoop()

```
int CoFindLoop (
            UBYTE * inp )  [extern]
```

Definition at line 5160 of file compcomm.c.

4.19.6.553 DoFindLoop()

```
int DoFindLoop (
            UBYTE * inp,
            int mode )  [extern]
```

Definition at line 5043 of file compcomm.c.

4.19.6.554 CoFunPowers()

```
int CoFunPowers (
            UBYTE * inp )  [extern]
```

Definition at line 5188 of file compcomm.c.

4.19.6.555 SortTheList()

```
int SortTheList (
            int * slist,
            int num )  [extern]
```

Definition at line 578 of file lus.c.

4.19.6.556 MatchIsPossible()

```
int MatchIsPossible (
            WORD * pattern,
            WORD * term )  [extern]
```

Definition at line 299 of file smart.c.

4.19.6.557 StudyPattern()

```
int StudyPattern (
            WORD * lhs )  [extern]
```

Definition at line 62 of file smart.c.

4.19.6.558 DolToTensor()

```
WORD DolToTensor (
            PHEAD WORD )  [extern]
```

Definition at line 1069 of file dollar.c.

4.19.6.559 DolToFunction()

```
WORD DolToFunction (
            PHEAD WORD )  [extern]
```

Definition at line 1130 of file dollar.c.

4.19.6.560 DolToVector()

```
WORD DolToVector (
            PHEAD WORD )  [extern]
```

Definition at line 1187 of file dollar.c.

4.19.6.561 DolToNumber()

```
WORD DolToNumber (
            PHEAD WORD )  [extern]
```

Definition at line 1251 of file dollar.c.

4.19.6.562 DolToSymbol()

```
WORD DolToSymbol (
            PHEAD WORD )  [extern]
```

Definition at line 1310 of file dollar.c.

4.19.6.563 DolToIndex()

```
WORD DolToIndex (
            PHEAD WORD )  [extern]
```

Definition at line 1364 of file dollar.c.

4.19.6.564 DolToLong()

```
LONG DolToLong (
            PHEAD WORD )  [extern]
```

Definition at line 1592 of file dollar.c.

4.19.6.565 DollarFactorize()

```
int DollarFactorize (
            PHEAD WORD )  [extern]
```

Definition at line 2939 of file dollar.c.

4.19.6.566 CoPrintTable()

```
int CoPrintTable (
            UBYTE * inp )  [extern]
```

Definition at line 1753 of file comexpr.c.

4.19.6.567 CoDeallocateTable()

```
int CoDeallocateTable (
            UBYTE * inp )  [extern]
```

Definition at line 1987 of file comexpr.c.

4.19.6.568 CleanDollarFactors()

```
void CleanDollarFactors (
            DOLLARS d )  [extern]
```

Definition at line 3538 of file dollar.c.

4.19.6.569 TakeDollarContent()

```
WORD * TakeDollarContent (
            PHEAD WORD * dollarbuffer,
            WORD ** factor )  [extern]
```

Definition at line 3560 of file dollar.c.

4.19.6.570 MakeDollarInteger()

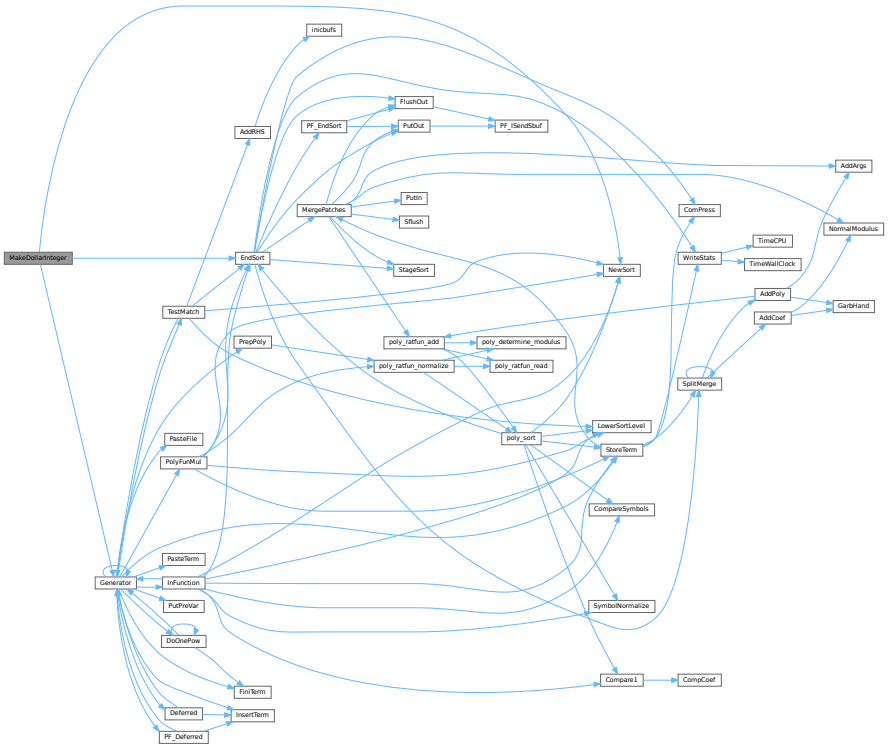
```
WORD * MakeDollarInteger (
            PHEAD WORD * bufin,
            WORD ** bufout )  [extern]
```

For normalizing everything to integers we have to determine for all elements of this argument the LCM of the denominators and the GCD of the numerators. The input argument is in bufin. The number that comes out is the return value. The normalized argument is in bufout.

Definition at line 3612 of file dollar.c.

References EndSort(), Generator(), and NewSort().

Here is the call graph for this function:



4.19.6.571 MakeDollarMod()

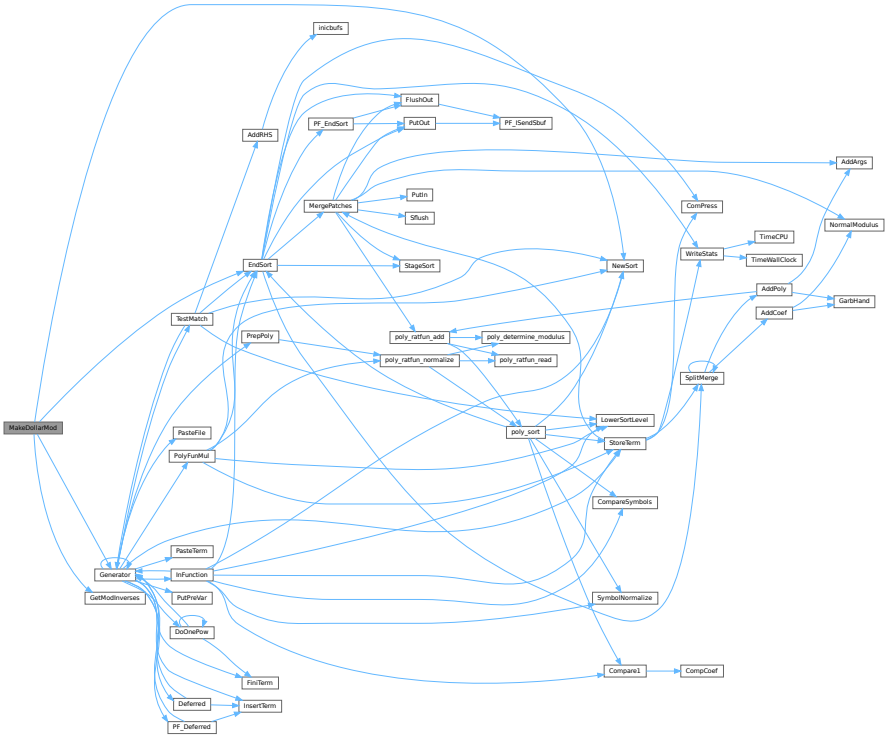
```
WORD * MakeDollarMod (
            PHEAD WORD * buffer,
            WORD ** bufout )  [extern]
```

Similar to MakeDollarInteger but now with modulus arithmetic using only a one WORD 'prime'. We make the coefficient of the first term in the argument equal to one. Already the coefficients are taken modulus AN.cmod and AN.ncmod == 1

Definition at line 3786 of file dollar.c.

References EndSort(), Generator(), GetModInverses(), and NewSort().

Here is the call graph for this function:



4.19.6.572 GetDolNum()

```
int GetDolNum (
            PHEAD WORD * t,
            WORD * tstop )  [extern]
```

Definition at line 3831 of file dollar.c.

4.19.6.573 AddPotModdollar()

```
void AddPotModdollar (
            WORD numdollar )  [extern]
```

Adds a $-variable specified by *numdollar* to the list of potentially modified $-variables unless it has already been included in the list.

Parameters

numdollar	The index of the $-variable to be added.

Definition at line 3944 of file dollar.c.

4.19.6.574 Optimize()

```
int Optimize (
              WORD exprnr,
              int do_print )  [extern]
```

Optimization of expression

4.19.7 Description

This method takes an input expression and generates optimized code to calculate its value. The following methods are called to do so:

(1) get_expression : to read to expression

(2) get_brackets : find brackets for simultaneous optimization

(3) occurrence_order or find_Horner_MCTS : to determine (the) Horner scheme(s) to use; this depends on AO.↩ optimize.horner

(4) optimize_expression_given_Horner : to do the optimizations for each Horner scheme; this method does either CSE or greedy optimizations dependings on AO.optimize.method

(5) generate_output : to format the output in Form notation and store it in a buffer

(6a) optimize_print_code : to print the expression (for "Print") or (6b) generate_expression : to modify the expression (for "#Optimize")

On ParFORM, all the processes must call this function at the same time. Then

(1) Because only the master can access to the expression to be optimized, the master broadcast the expression to all the slaves after reading the expression (PF_get_expression).

(2) get_brackets reads optimize_expr as the input and it works also on the slaves. We leave it although the bracket information is not needed on the slaves (used in (5) on the master).

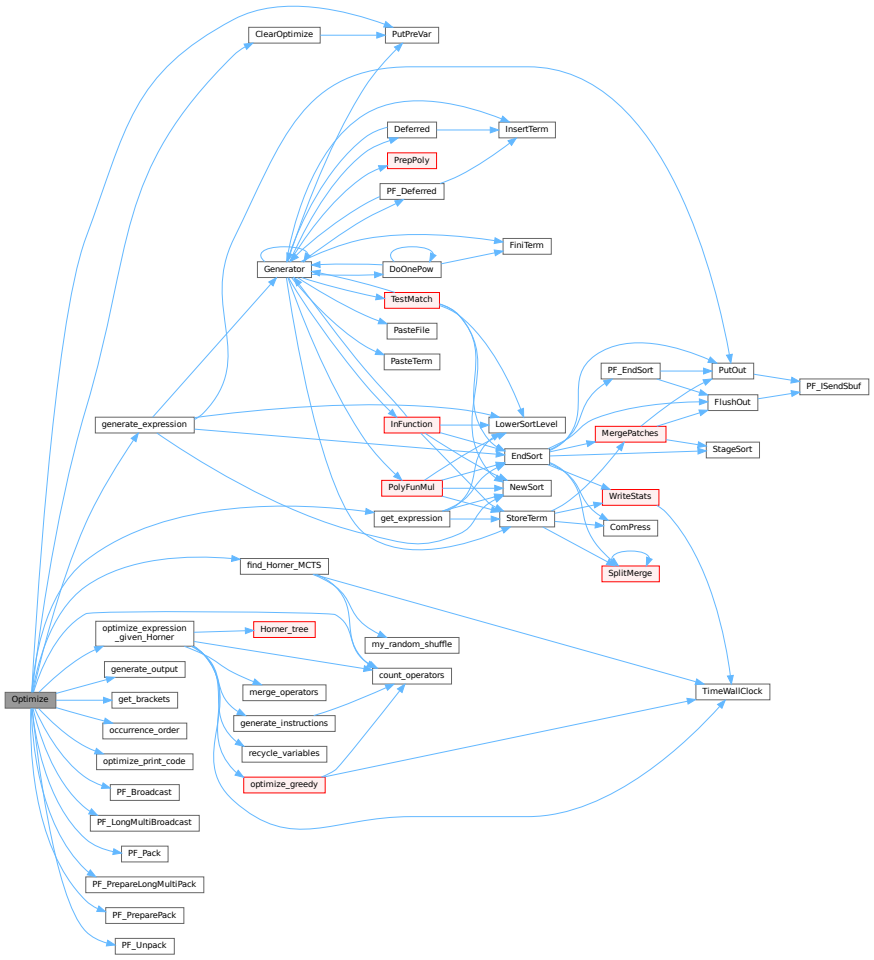
(3) and (4) find_Horner_MCTS and optimize_expression_given_Horner are parallelized.

(5), (6a) and (6b) are needed only on the master.

Definition at line 4641 of file optimize.cc.

References ClearOptimize(), count_operators(), find_Horner_MCTS(), generate_expression(), generate_output(), get_brackets(), get_expression(), occurrence_order(), optimize_expression_given_Horner(), optimize_print_code(), PF_Broadcast(), PF_LongMultiBroadcast(), PF_Pack(), PF_PrepareLongMultiPack(), PF_PreparePack(), PF_Unpack(), and PutPreVar().

Here is the call graph for this function:



4.19.7.1 ClearOptimize()

```
int ClearOptimize (
            void  )  [extern]
```

Optimization of expression

4.19.8 Description

Clears the buffers that were used for optimization output. Clears the expression from the buffers (marks it to be dropped). Note: we need to use the expression by its name, because numbers may change if we drop other expressions between when we do the optimizations and clear the results (in execute.c). Also this is not 100% safe, because we could overwrite the optimized expression. But that can be done only in a Local or Global statement and hence we only have to test there that we might have to call ClearOptimize first. (in file comexpr.c)

Definition at line 4978 of file optimize.cc.

References PutPreVar().

Referenced by Optimize().

Here is the call graph for this function:

4.19.8.1 TakeLongRoot()

```
int TakeLongRoot (
            UWORD * a,
            WORD * n,
            WORD power )  [extern]
```

Definition at line 2754 of file reken.c.

4.19.8.2 TakeRatRoot()

```
int TakeRatRoot (
            UWORD * a,
            WORD * n,
            WORD power )  [extern]
```

Definition at line 689 of file reken.c.

4.19.8.3 MakeRational()

```
int MakeRational (
            WORD a,
            WORD m,
            WORD * b,
            WORD * c )  [extern]
```

Definition at line 2874 of file reken.c.

4.19.8.4 MakeLongRational()

```
int MakeLongRational (
            PHEAD UWORD * a,
            WORD na,
            UWORD * m,
            WORD nm,
            UWORD * b,
            WORD * nb )  [extern]
```

Definition at line 2922 of file reken.c.

4.19.8.5 ClearTableTree()

```
void ClearTableTree (
            TABLES T )  [extern]
```

Definition at line 72 of file tables.c.

4.19.8.6 InsTableTree()

```
int InsTableTree (
            TABLES T,
            WORD * tp )  [extern]
```

Definition at line 103 of file tables.c.

4.19.8.7 RedoTableTree()

```
void RedoTableTree (
            TABLES T,
            int newsize )  [extern]
```

Definition at line 268 of file tables.c.

4.19.8.8 FindTableTree()

```
int FindTableTree (
            TABLES T,
            WORD * tp,
            int inc )  [extern]
```

Definition at line 293 of file tables.c.

4.19.8.9 finishcbuf()

```
void finishcbuf (
            WORD num )  [extern]
```

Frees a compiler buffer.

Parameters

num	The ID number for the buffer to be freed.

Definition at line 89 of file comtool.c.

References CbUf::boomlijst, CbUf::Buffer, CbUf::BufferSize, CbUf::CanCommu, CbUf::lhs, CbUf::NumTerms, CbUf::Pointer, CbUf::rhs, and CbUf::Top.

Referenced by PF_BroadcastCBuf().

4.19.8.10 clearcbuf()

```
void clearcbuf (
            WORD num )  [extern]
```

Clears contents in a compiler buffer.

Parameters

num	The ID number for the buffer to be cleared.

Definition at line 116 of file comtool.c.

References CbUf::boomlijst, CbUf::Buffer, CbUf::Pointer, and CbUf::rhs.

4.19.8.11 CleanUpSort()

```
void CleanUpSort (
            int num )  [extern]
```

Partially or completely frees function sort buffers.

Definition at line 4631 of file sort.c.

4.19.8.12 AllocFileHandle()

```
FILEHANDLE * AllocFileHandle (
            WORD par,
            char * name )  [extern]
```

Definition at line 1045 of file setfile.c.

4.19.8.13 DeAllocFileHandle()

```
void DeAllocFileHandle (
            FILEHANDLE * fh )  [extern]
```

Definition at line 1106 of file setfile.c.

4.19.8.14 LowerSortLevel()

```
void LowerSortLevel (
            void  )  [extern]
```

Lowers the level in the sort system.

Definition at line 4731 of file sort.c.

Referenced by generate_expression(), get_expression(), InFunction(), PF_CollectModifiedDollars(), PF_Processor(), poly_factorize_expression(), poly_sort(), poly_unfactorize_expression(), PolyFunMul(), Processor(), and TestMatch().

4.19.8.15 PolyRatFunSpecial()

```
WORD * PolyRatFunSpecial (
            PHEAD WORD * t1,
            WORD * t2 )  [extern]
```

Definition at line 4748 of file sort.c.

4.19.8.16 SimpleSplitMergeRec()

```
void SimpleSplitMergeRec (
            WORD * array,
            WORD num,
            WORD * auxarray )  [extern]
```

Definition at line 4850 of file sort.c.

4.19.8.17 SimpleSplitMerge()

```
void SimpleSplitMerge (
            WORD * array,
            WORD num )  [extern]
```

Definition at line 4878 of file sort.c.

4.19.8.18 BinarySearch()

```
WORD BinarySearch (
            WORD * array,
            WORD num,
            WORD x )  [extern]
```

Definition at line 4896 of file sort.c.

4.19.8.19 InsideDollar()

```
int InsideDollar (
            PHEAD WORD * ll,
            WORD level )  [extern]
```

Definition at line 1732 of file dollar.c.

4.19.8.20 DolToTerms()

```
DOLLARS DolToTerms (
            PHEAD WORD )  [extern]
```

Definition at line 1445 of file dollar.c.

4.19.8.21 EvalDoLoopArg()

```
WORD EvalDoLoopArg (
            PHEAD WORD * arg,
            WORD par )  [extern]
```

Evaluates one argument of a do loop. Such an argument is constructed from SNUMBERs DOLLAREXPRESSIONs and possibly DOLLAREXPR2s which indicate factors of the preceding dollar. Hence we have SNUMBER,num DOLLAREXPRESSION,numdollar DOLLAREXPRESSION,numdollar,DOLLAREXPR2,numfactor DOLLAREXPRESSION,numdollar,DOLLAREXPR2,numfactor,DOLLAREXPR2,numfactor etc. Because we have a do-loop at every stage we should have a number. The notation in DOLLAREXPR2 is that >= 0 is number of yet another dollar and < 0 is -n-1 with n the array element or zero. The return value is the (short) number. The routine works its way through the list in a recursive manner.

Definition at line 2635 of file dollar.c.

References EvalDoLoopArg().

Referenced by EvalDoLoopArg().

Here is the call graph for this function:

4.19.8.22 SetExprCases()

```
int SetExprCases (
            int par,
            int setunset,
            int val )  [extern]
```

Definition at line 1368 of file compcomm.c.

4.19.8.23 TestSelect()

```
int TestSelect (
            WORD * term,
            WORD * setp )  [extern]
```

Definition at line 1750 of file pattern.c.

4.19.8.24 SubsInAll()

```
void SubsInAll (
            PHEAD0  )  [extern]
```

Definition at line 1984 of file pattern.c.

4.19.8.25 TransferBuffer()

```
void TransferBuffer (
            int from,
            int to,
            int spectator )  [extern]
```

Definition at line 2145 of file pattern.c.

4.19.8.26 TakeIDfunction()

```
int TakeIDfunction (
            PHEAD WORD * term )  [extern]
```

Definition at line 2215 of file pattern.c.

4.19.8.27 MakeSetupAllocs()

```
int MakeSetupAllocs (
            void  )  [extern]
```

Definition at line 1123 of file setfile.c.

4.19.8.28 AllocNormData()

```
NORMDATA * AllocNormData (
            void  )  [extern]
```

Definition at line 1137 of file setfile.c.

4.19.8.29 TryFileSetups()

```
int TryFileSetups (
            void  )  [extern]
```

Definition at line 1169 of file setfile.c.

4.19.8.30 ExchangeExpressions()

```
void ExchangeExpressions (
            int num1,
            int num2 )  [extern]
```

Definition at line 1775 of file execute.c.

4.19.8.31 ExchangeDollars()

```
void ExchangeDollars (
            int num1,
            int num2 )  [extern]
```

Definition at line 1831 of file dollar.c.

4.19.8.32 GetFirstBracket()

```
int GetFirstBracket (
            WORD * term,
            int num )  [extern]
```

Definition at line 1853 of file execute.c.

4.19.8.33 GetFirstTerm()

```
int GetFirstTerm (
            WORD * term,
            int num,
            int pre )  [extern]
```

Gets the first term of an expression. Routine should be thread safe.

Parameters

out	*term*	Pointer to the buffer where the term should be written.
in	*num*	The expression number from which to get the term.
in	*pre*	Denotes a pre-processor call (1) or not (0). Used to determine the correct source file for the expression.

Returns

First WORD of term, a negative value denotes an error.

Definition at line 1972 of file execute.c.

References FiLe::handle, VaRrEnUm::lo, and ReNuMbEr::symb.

4.19.8.34 GetContent()

```
int GetContent (
            WORD * content,
            int num )  [extern]
```

Definition at line 2080 of file execute.c.

4.19.8.35 CleanupTerm()

```
int CleanupTerm (
            WORD * term )  [extern]
```

Definition at line 2197 of file execute.c.

4.19.8.36 ContentMerge()

```
WORD ContentMerge (
            PHEAD WORD * content,
            WORD * term )  [extern]
```

Definition at line 2221 of file execute.c.

4.19.8.37 PreIfDollarEval()

```
UBYTE * PreIfDollarEval (
            UBYTE * s,
            int * value )  [extern]
```

Definition at line 1960 of file dollar.c.

4.19.8.38 TermsInDollar()

```
LONG TermsInDollar (
            WORD num )  [extern]
```

Definition at line 1849 of file dollar.c.

4.19.8.39 SizeOfDollar()

```
LONG SizeOfDollar (
            WORD num )  [extern]
```

Definition at line 1898 of file dollar.c.

4.19.8.40 TermsInExpression()

```
LONG TermsInExpression (
             WORD num )  [extern]
```

Definition at line 2485 of file execute.c.

4.19.8.41 SizeOfExpression()

```
LONG SizeOfExpression (
             WORD num )  [extern]
```

Definition at line 2497 of file execute.c.

4.19.8.42 TranslateExpression()

```
WORD * TranslateExpression (
             UBYTE * s )  [extern]
```

Definition at line 2142 of file dollar.c.

4.19.8.43 IsSetMember()

```
int IsSetMember (
             WORD * buffer,
             WORD numset )  [extern]
```

Definition at line 2199 of file dollar.c.

4.19.8.44 IsMultipleOf()

```
int IsMultipleOf (
             WORD * buf1,
             WORD * buf2 )  [extern]
```

Definition at line 2380 of file dollar.c.

4.19.8.45 TwoExprCompare()

```
int TwoExprCompare (
             WORD * buf1,
             WORD * buf2,
             int oprtr )  [extern]
```

Definition at line 2455 of file dollar.c.

4.19.8.46 UpdatePositions()

```
void UpdatePositions (
            void  )  [extern]
```

Definition at line 2509 of file execute.c.

4.19.8.47 M_print()

```
void M_print (
            void  )  [extern]
```

Definition at line 2457 of file tools.c.

4.19.8.48 M_check1()

```
void M_check1 (
            void  )  [extern]
```

Definition at line 2456 of file tools.c.

4.19.8.49 PrintTime()

```
void PrintTime (
            UBYTE * mess )  [extern]
```

Definition at line 766 of file sch.c.

4.19.8.50 FindBracket()

```
POSITION * FindBracket (
            WORD nexp,
            WORD * bracket )  [extern]
```

Definition at line 65 of file index.c.

4.19.8.51 PutBracketInIndex()

```
void PutBracketInIndex (
            PHEAD WORD * term,
            POSITION * newpos )  [extern]
```

Definition at line 331 of file index.c.

4.19.8.52 ClearBracketIndex()

```
void ClearBracketIndex (
            WORD numexp )  [extern]
```

Definition at line 548 of file index.c.

4.19.8.53 OpenBracketIndex()

```
void OpenBracketIndex (
            WORD nexpr )  [extern]
```

Definition at line 580 of file index.c.

4.19.8.54 DoNoParallel()

```
int DoNoParallel (
            UBYTE * s )  [extern]
```

Definition at line 532 of file module.c.

4.19.8.55 DoParallel()

```
int DoParallel (
            UBYTE * s )  [extern]
```

Definition at line 547 of file module.c.

4.19.8.56 DoModSum()

```
int DoModSum (
            UBYTE * s )  [extern]
```

Definition at line 562 of file module.c.

4.19.8.57 DoModMax()

```
int DoModMax (
            UBYTE * s )  [extern]
```

Definition at line 582 of file module.c.

4.19.8.58 DoModMin()

```
int DoModMin (
            UBYTE * s )  [extern]
```

Definition at line 602 of file module.c.

4.19.8.59 DoModLocal()

```
int DoModLocal (
            UBYTE * s )  [extern]
```

Definition at line 622 of file module.c.

4.19.8.60 DoModDollar()

```
UBYTE * DoModDollar (
            UBYTE * s,
            int type )  [extern]
```

Definition at line 660 of file module.c.

4.19.8.61 DoProcessBucket()

```
int DoProcessBucket (
            UBYTE * s )  [extern]
```

Definition at line 642 of file module.c.

4.19.8.62 DoinParallel()

```
int DoinParallel (
            UBYTE * s )  [extern]
```

Definition at line 775 of file module.c.

4.19.8.63 DonotinParallel()

```
int DonotinParallel (
            UBYTE * s )  [extern]
```

Definition at line 785 of file module.c.

4.19.8.64 FlipTable()

```
int FlipTable (
            FUNCTIONS f,
            int type )  [extern]
```

Definition at line 677 of file tables.c.

4.19.8.65 ChainIn()

```
int ChainIn (
            PHEAD WORD * term,
            WORD funnum )  [extern]
```

Definition at line 691 of file function.c.

4.19.8.66 ChainOut()

```
int ChainOut (
            PHEAD WORD * term,
            WORD funnum )  [extern]
```

Definition at line 748 of file function.c.

4.19.8.67 ArgumentImplode()

```
int ArgumentImplode (
            PHEAD WORD * term,
            WORD * thelist )  [extern]
```

Definition at line 1853 of file argument.c.

4.19.8.68 ArgumentExplode()

```
int ArgumentExplode (
            PHEAD WORD * term,
            WORD * thelist )  [extern]
```

Definition at line 1957 of file argument.c.

4.19.8.69 DenToFunction()

```
int DenToFunction (
            WORD * term,
            WORD numfun )  [extern]
```

Definition at line 2565 of file wildcard.c.

4.19.8.70 HowMany()

```
WORD HowMany (
            PHEAD WORD * ifcode,
            WORD * term )  [extern]
```

Definition at line 856 of file if.c.

4.19.8.71 RemoveDollars()

```
void RemoveDollars (
            void  )  [extern]
```

Definition at line 3135 of file names.c.

4.19.8.72 CountTerms1()

```
LONG CountTerms1 (
            PHEAD0  )  [extern]
```

Definition at line 2569 of file execute.c.

4.19.8.73 TermsInBracket()

```
LONG TermsInBracket (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 2677 of file execute.c.

4.19.8.74 Crash()

```
int Crash (
            void  )  [extern]
```

Definition at line 3784 of file tools.c.

4.19.8.75 str_dup()

```
char * str_dup (
            char * str )  [extern]
```

Definition at line 101 of file minos.c.

4.19.8.76 convertblock()

```
void convertblock (
            INDEXBLOCK * in,
            INDEXBLOCK * out,
            int mode )  [extern]
```

Definition at line 120 of file minos.c.

4.19.8.77 convertnamesblock()

```
void convertnamesblock (
            NAMESBLOCK * in,
            NAMESBLOCK * out,
            int mode )  [extern]
```

Definition at line 171 of file minos.c.

4.19.8.78 convertiniinfo()

```
void convertiniinfo (
             INIINFO * in,
             INIINFO * out,
             int mode )  [extern]
```

Definition at line 197 of file minos.c.

4.19.8.79 ReadIndex()

```
int ReadIndex (
             DBASE * d )  [extern]
```

Definition at line 288 of file minos.c.

4.19.8.80 WriteIndexBlock()

```
int WriteIndexBlock (
             DBASE * d,
             MLONG num )  [extern]
```

Definition at line 409 of file minos.c.

4.19.8.81 WriteNamesBlock()

```
int WriteNamesBlock (
             DBASE * d,
             MLONG num )  [extern]
```

Definition at line 434 of file minos.c.

4.19.8.82 WriteIndex()

```
int WriteIndex (
             DBASE * d )  [extern]
```

Definition at line 461 of file minos.c.

4.19.8.83 WriteIniInfo()

```
int WriteIniInfo (
             DBASE * d )  [extern]
```

Definition at line 532 of file minos.c.

4.19.8.84 ReadIniInfo()

```
int ReadIniInfo (
            DBASE * d )  [extern]
```

Definition at line 551 of file minos.c.

4.19.8.85 AddToIndex()

```
int AddToIndex (
            DBASE * d,
            MLONG number )  [extern]
```

Definition at line 1109 of file minos.c.

4.19.8.86 GetDbase()

```
DBASE * GetDbase (
            char * filename,
            MLONG rwmode )  [extern]
```

Definition at line 578 of file minos.c.

4.19.8.87 OpenDbase()

```
DBASE * OpenDbase (
            char * filename )  [extern]
```

Definition at line 862 of file minos.c.

4.19.8.88 ReadObject()

```
char * ReadObject (
            DBASE * d,
            MLONG tablenumber,
            char * arguments )  [extern]
```

Definition at line 1350 of file minos.c.

4.19.8.89 ReadijObject()

```
char * ReadijObject (
            DBASE * d,
            MLONG i,
            MLONG j,
            char * arguments )  [extern]
```

Definition at line 1435 of file minos.c.

4.19.8.90 ExistsObject()

```
int ExistsObject (
             DBASE * d,
             MLONG tablenumber,
             char * arguments )  [extern]
```

Definition at line 1491 of file minos.c.

4.19.8.91 DeleteObject()

```
int DeleteObject (
             DBASE * d,
             MLONG tablenumber,
             char * arguments )  [extern]
```

Definition at line 1523 of file minos.c.

4.19.8.92 WriteObject()

```
int WriteObject (
             DBASE * d,
             MLONG tablenumber,
             char * arguments,
             char * rhs,
             MLONG number )  [extern]
```

Definition at line 1246 of file minos.c.

4.19.8.93 AddObject()

```
MLONG AddObject (
             DBASE * d,
             MLONG tablenumber,
             char * arguments,
             char * rhs )  [extern]
```

Definition at line 1204 of file minos.c.

4.19.8.94 NewDbase()

```
DBASE * NewDbase (
             char * name,
             MLONG number )  [extern]
```

Definition at line 634 of file minos.c.

4.19.8.95 FreeTableBase()

```
void FreeTableBase (
            DBASE * d )  [extern]
```

Definition at line 781 of file minos.c.

4.19.8.96 ComposeTableNames()

```
int ComposeTableNames (
            DBASE * d )  [extern]
```

Definition at line 813 of file minos.c.

4.19.8.97 PutTableNames()

```
int PutTableNames (
            DBASE * d )  [extern]
```

Definition at line 1011 of file minos.c.

4.19.8.98 AddTableName()

```
MLONG AddTableName (
            DBASE * d,
            char * name,
            TABLES T )  [extern]
```

Definition at line 896 of file minos.c.

4.19.8.99 GetTableName()

```
MLONG GetTableName (
            DBASE * d,
            char * name )  [extern]
```

Definition at line 979 of file minos.c.

4.19.8.100 FindTableNumber()

```
MLONG FindTableNumber (
            DBASE * d,
            char * name )  [extern]
```

Definition at line 1218 of file minos.c.

4.19.8.101 TryEnvironment()

```
int TryEnvironment (
            void  )  [extern]
```

Definition at line 1258 of file setfile.c.

4.19.8.102 CopyExpression()

```
int CopyExpression (
            FILEHANDLE * from,
            FILEHANDLE * to )  [extern]
```

Definition at line 3379 of file store.c.

4.19.8.103 set_in()

```
int set_in (
            UBYTE ch,
            set_of_char set )  [extern]
```

Definition at line 2135 of file tools.c.

4.19.8.104 set_set()

```
one_byte set_set (
            UBYTE ch,
            set_of_char set )  [extern]
```

Definition at line 2155 of file tools.c.

4.19.8.105 set_del()

```
one_byte set_del (
            UBYTE ch,
            set_of_char set )  [extern]
```

Definition at line 2176 of file tools.c.

4.19.8.106 set_sub()

```
one_byte set_sub (
            set_of_char set,
            set_of_char set1,
            set_of_char set2 )  [extern]
```

Definition at line 2198 of file tools.c.

4.19.8.107 DoPreAddSeparator()

```
int DoPreAddSeparator (
            UBYTE * s )  [extern]
```

Definition at line 5473 of file pre.c.

4.19.8.108 DoPreRmSeparator()

```
int DoPreRmSeparator (
            UBYTE * s )  [extern]
```

Definition at line 5498 of file pre.c.

4.19.8.109 openExternalChannel()

```
int openExternalChannel (
            UBYTE * cmd,
            int daemonize,
            UBYTE * shellname,
            UBYTE * stderrname )  [extern]
```

Definition at line 230 of file extcmd.c.

4.19.8.110 initPresetExternalChannels()

```
int initPresetExternalChannels (
            UBYTE * theline,
            int thetimeout )  [extern]
```

Definition at line 232 of file extcmd.c.

4.19.8.111 closeExternalChannel()

```
int closeExternalChannel (
            int n )  [extern]
```

Definition at line 233 of file extcmd.c.

4.19.8.112 selectExternalChannel()

```
int selectExternalChannel (
            int n )  [extern]
```

Definition at line 234 of file extcmd.c.

4.19.8.113 getCurrentExternalChannel()

```
int getCurrentExternalChannel (
            void  )  [extern]
```

Definition at line 235 of file extcmd.c.

4.19.8.114 closeAllExternalChannels()

```
void closeAllExternalChannels (
            void  )  [extern]
```

Definition at line 236 of file extcmd.c.

4.19.8.115 defineChannel()

```
UBYTE * defineChannel (
            UBYTE * s,
            HANDLERS * h )  [extern]
```

Definition at line 6124 of file pre.c.

4.19.8.116 writeToChannel()

```
int writeToChannel (
            int wtype,
            UBYTE * s,
            HANDLERS * h )  [extern]
```

Definition at line 6159 of file pre.c.

4.19.8.117 ReleaseTB()

```
int ReleaseTB (
            void  )  [extern]
```

Definition at line 2329 of file tables.c.

4.19.8.118 SymbolNormalize()

```
int SymbolNormalize (
            WORD * term )  [extern]
```

Routine normalizes terms that contain only symbols. Regular minimum and maximum properties are ignored.

We check whether there are negative powers in the output. This is not allowed.

Definition at line 5210 of file normal.c.

Referenced by InFunction(), and poly_sort().

4.19.8.119 TestFunFlag()

```
int TestFunFlag (
            PHEAD WORD * tfun )  [extern]
```

Definition at line 5311 of file normal.c.

4.19.8.120 CompareSymbols()

```
WORD CompareSymbols (
            PHEAD WORD * term1,
            WORD * term2,
            WORD par )  [extern]
```

Compares the terms, based on the value of AN.polysortflag. If term1 $<$ term2 the return value is -1 If term1 $>$ term2 the return value is 1 If term1 = term2 the return value is 0 The coefficients may differ. The terms contain only a single subterm of type SYMBOL. If AN.polysortflag = 0 it is a 'regular' compare. If AN.polysortflag = 1 the sum of the powers is more important par is a dummy parameter to make the parameter field identical to that of Compare1 which is the regular compare routine in sort.c

Definition at line 2856 of file sort.c.

Referenced by InFunction(), and poly_sort().

4.19.8.121 CompareHSymbols()

```
WORD CompareHSymbols (
            PHEAD WORD * term1,
            WORD * term2,
            WORD par )  [extern]
```

Compares terms that can have only SYMBOL and HAAKJE subterms. If term1 $<$ term2 the return value is -1 If term1 $>$ term2 the return value is 1 If term1 = term2 the return value is 0 par is a dummy parameter to make the parameter field identical to that of Compare1 which is the regular compare routine in sort.c

Definition at line 2906 of file sort.c.

4.19.8.122 NextPrime()

```
WORD NextPrime (
            PHEAD WORD num )  [extern]
```

Gives the next prime number in the list of prime numbers.

If the list isn't long enough we expand it. For ease in ParForm and because these lists shouldn't be very big we let each worker keep its own list.

The list is cut off at MAXPOWER, because we don't want to get into trouble that the power of a variable gets larger than the prime number.

Definition at line 3685 of file reken.c.

4.19.8.123 Moebius()

```
WORD Moebius (
            PHEAD WORD )  [extern]
```

Definition at line 3751 of file reken.c.

4.19.8.124 wranf()

```
UWORD wranf (
            PHEAD0  )  [extern]
```

Definition at line 3891 of file reken.c.

4.19.8.125 iranf()

```
UWORD iranf (
            PHEAD UWORD )  [extern]
```

Definition at line 3910 of file reken.c.

4.19.8.126 iniwranf()

```
void iniwranf (
            PHEAD0  )  [extern]
```

Definition at line 3842 of file reken.c.

4.19.8.127 PreRandom()

```
UBYTE * PreRandom (
            UBYTE * s )  [extern]
```

Definition at line 3935 of file reken.c.

4.19.8.128 TreatPolyRatFun()

```
int TreatPolyRatFun (
            PHEAD WORD * prf )  [extern]
```

Definition at line 5026 of file normal.c.

4.19.8.129 ReadSaveHeader()

```
int ReadSaveHeader (
            void  )  [extern]
```

Reads the header in the save file and sets function pointers and flags according to the information found there. Must be called before any other ReadSave... function.

Currently works only for the exchange between 32bit and 64bit machines (WORD size must be 2 or 4 bytes)!

It is called by CoLoad().

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 4136 of file store.c.

4.19.8.130 ReadSaveIndex()

```
LONG ReadSaveIndex (
            FILEINDEX * fileind )  [extern]
```

Reads a FILEINDEX from the open save file specified by AO.SaveData.Handle. Translations for adjusting endianness and data sizes are done if necessary.

Depends on the assumption that sizeof(FILEINDEX) is the same everywhere. If FILEINDEX or INDEXENTRY change, then this functions has to be adjusted.

Called by CoLoad() and FindInIndex().

Parameters

fileind	contains the read FILEINDEX after successful return. must point to allocated, big enough memory.

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 4257 of file store.c.

References INFILEINDEX, FiLeInDeX::next, FiLeInDeX::number, and FuNcTiOn::number.

4.19.8.131 ReadSaveExpression()

```
LONG ReadSaveExpression (
            UBYTE * buffer,
            UBYTE * top,
            LONG * size,
            LONG * outsize )  [extern]
```

Reads an expression from the open file specified by AO.SaveData.Handle. The endianness flip and a resizing without renumbering is done in this function. Thereafter the buffer consists of chunks with a uniform maximal word size (32bit at the moment). The actual renumbering is then done by calling the function ReadSaveTerm32(). The result is returned in *buffer*.

If the translation at some point doesn't fit into the buffer anymore, the function returns and must be called again. In any case *size* returns the number of successfully read bytes, *outsize* returns the number of successfully written bytes, and the file will be positioned at the next byte after the successfully read data.

It is called by PutInStore().

Parameters

buffer	output buffer, holds the (translated) expression
top	end of buffer
size	number of read bytes
outsize	number of written bytes

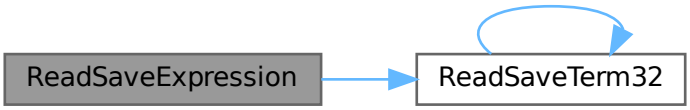
Returns

= 0 everything okay, != 0 an error occurred

Definition at line 5228 of file store.c.

References ReadSaveTerm32().

Here is the call graph for this function:



4.19.8.132 ReadSaveTerm32()

```
UBYTE * ReadSaveTerm32 (
            UBYTE * bin,
            UBYTE * binend,
            UBYTE ** bout,
            UBYTE * boutend,
            UBYTE * top,
            int terminbuf )  [extern]
```

Reads a single term from the given buffer at *bin* and write the translated term back to this buffer at *bout*.

ReadSaveTerm32() is currently the only instantiation of a ReadSaveTerm-function. It only deals with data that already has the correct endianness and that is resized to 32bit words but without being renumbered or translated in any other way. It uses the compress buffer AR.CompressBuffer.

The function is reentrant in order to cope with nested function arguments. It is called by ReadSaveExpression() and itself.

The *return value* indicates the position in the input buffer up to which the data has already been successfully processed. The parameter *bout* returns the corresponding position in the output buffer.

Parameters

bin	start of the input buffer
binend	end of the input buffer
bout	as input points to the beginning of the output buffer, as output points behind the already translated data in the output buffer
boutend	end of already decompressed data in output buffer
top	end of output buffer
terminbuf	flag whether decompressed data is already in the output buffer. used in recursive calls

Returns

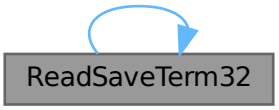
pointer to the next unprocessed data in the input buffer

Definition at line 4814 of file store.c.

References ReadSaveTerm32().

Referenced by ReadSaveExpression(), and ReadSaveTerm32().

Here is the call graph for this function:



4.19.8.133 ReadSaveVariables()

```
LONG ReadSaveVariables (
             UBYTE * buffer,
             UBYTE * top,
             LONG * size,
             LONG * outsize,
             INDEXENTRY * ind,
             LONG * stage )  [extern]
```

Reads the variables from the open file specified by AO.SaveData.Handle. It reads the ∗size bytes and writes them to the ∗buffer. It is called by PutInStore().

If translation is necessary, the data might shrink or grow in size, then ∗*size* is adjusted so that the reading and writing fits into the memory from the buffer to the top. The actual number of read bytes is returned in ∗*size*, the number of written bytes is returned in ∗*outsize*.

If the ∗size is smaller than the actual size of the variables, this function will be called several times and needs to remember the current position in the variable structure. The parameter *stage* does this job. When ReadSaveVariables() is called for the first time, this parameter should have the value -1.

The parameter *ind* is used to get the number of variables.

Parameters

buffer	read variables are written into this allocated memory
top	upper end of allocated memory
size	number of bytes to read. might return a smaller number of read bytes if translation was necessary
outsize	if translation has be done, outsize contains the number of written bytes
ind	pointer of INDEXENTRY for the current expression. read-only
stage	should be -1 for the first call, will be increased by ReadSaveVariables to memorize the position in the variable structure

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 4445 of file store.c.

References InDeXeNtRy::nfunctions, InDeXeNtRy::nindices, InDeXeNtRy::nsymbols, InDeXeNtRy::nvectors, and FuNcTiOn::spec.

4.19.8.134 WriteStoreHeader()

```
LONG WriteStoreHeader (
            WORD handle )  [extern]
```

Writes header with information about system architecture and FORM revision to an open store file.

Called by SetFileIndex().

Parameters

handle	specifies open file to which header will be written

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 4040 of file store.c.

References STOREHEADER::endianness, STOREHEADER::lenLONG, STOREHEADER::lenPOINTER, STOREHEADER::lenPOS, STOREHEADER::lenWORD, STOREHEADER::maxpower, STOREHEADER::sFun, STOREHEADER::sInd, STOREHEADER::sSym, STOREHEADER::sVec, and STOREHEADER::wildoffset.

Referenced by SetFileIndex().

4.19.8.135 InitRecovery()

```
void InitRecovery (
            void  )  [extern]
```

Sets up the strings for the filenames of the recovery files. This functions should only be called once to avoid memory leaks and after AM.TempDir has been initialized.

Definition at line 399 of file checkpoint.c.

4.19.8.136 CheckRecoveryFile()

```
int CheckRecoveryFile (
            void  )  [extern]
```

Checks whether a snapshot/recovery file exists. Returns 1 if it exists, 0 otherwise.

Definition at line 278 of file checkpoint.c.

References RecoveryFilename().

Here is the call graph for this function:

4.19.8.137 DeleteRecoveryFile()

```
void DeleteRecoveryFile (
            void  )  [extern]
```

Deletes the recovery files. It is called by CleanUp() in the case of a successful completion.

Definition at line 333 of file checkpoint.c.

4.19.8.138 RecoveryFilename()

```
char * RecoveryFilename (
            void  )  [extern]
```

Returns pointer to recovery filename.

Definition at line 364 of file checkpoint.c.

Referenced by CheckRecoveryFile(), and DoRecovery().

4.19.8.139 DoRecovery()

```
int DoRecovery (
            int * moduletype )  [extern]
```

Reads from the recovery file and restores all necessary variables and states in FORM, so that the execution can recommence in preprocessor() as if no restart of FORM had occurred.

The recovery file is read into memory as a whole. The pointer p then points into this memory at the next non-processed data. The macros by which variables are restored, like R_SET, automatically increase p appropriately.

If something goes wrong, the function returns with a non-zero value.

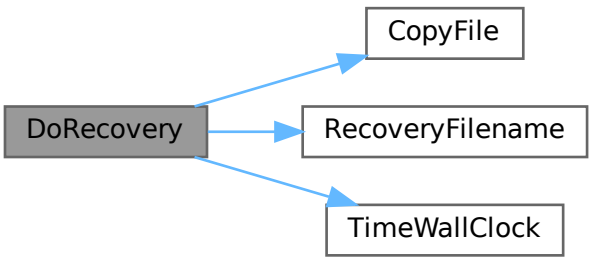
Allocated memory that would be lost when overwriting the global structs with data from the file is freed first. A major part of the code deals with the restoration of pointers. The idiom we use is to memorize the original pointer value (org), allocate new memory and copy the data from the file into this memory, calculate the offset between the old pointer value and the new allocated memory position (ofs), and then correct all affected pointers (+=ofs).

We rely on the fact that several variables (especially in AM) are already assigned the correct values by the startup functions. That means, in principle, that a change in the setup files between snapshot creation and recovery will be noticed.

Definition at line 1402 of file checkpoint.c.

References TaBlEs::argtail, TaBlEs::boomlijst, BrAcKeTiNfO::bracketbuffer, TaBlEs::buffers, TaBlEs::bufferssize, CopyFile(), TaBlEs::flags, ReNuMbEr::func, ReNuMbEr::funnum, VaRrEnUm::hi, BrAcKeTiNfO::indexbuffer, ReNuMbEr::indi, ReNuMbEr::indnum, VaRrEnUm::lo, TaBlEs::MaxTreeSize, TaBlEs::mm, TaBlEs::numind, TaBlEs::pattern, TaBlEs::prototype, TaBlEs::prototypeSize, RecoveryFilename(), ExPrEsSiOn::renumlists, TaBlEs::reserved, TaBlEs::spare, TaBlEs::sparse, VaRrEnUm::start, ReNuMbEr::symb, ReNuMbEr::symnum, TaBlEs::tablepointers, TimeWallClock(), TaBlEs::totind, ReNuMbEr::vecnum, and ReNuMbEr::vect.

Here is the call graph for this function:



4.19.8.140 DoCheckpoint()

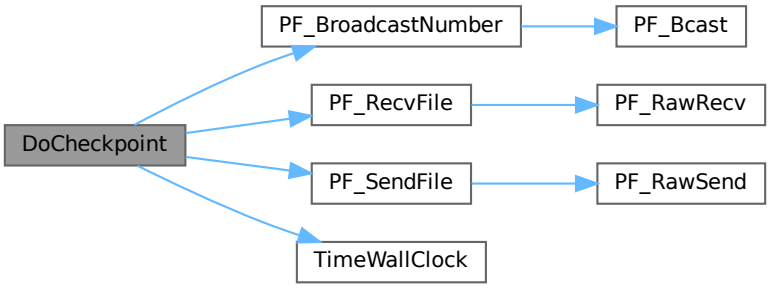
```
void DoCheckpoint (
            int moduletype )  [extern]
```

Checks whether a snapshot should be done. Calls DoSnapshot() to create the snapshot.

Definition at line 3122 of file checkpoint.c.

References PF_BroadcastNumber(), PF_RecvFile(), PF_SendFile(), and TimeWallClock().

Here is the call graph for this function:



4.19.8.141 NumberMallocAddMemory()

```
void NumberMallocAddMemory (
            PHEAD0  )  [extern]
```

Definition at line 2594 of file tools.c.

4.19.8.142 CacheNumberMallocAddMemory()

```
void CacheNumberMallocAddMemory (
            PHEAD0  )  [extern]
```

Definition at line 2668 of file tools.c.

4.19.8.143 TermMallocAddMemory()

```
void TermMallocAddMemory (
            PHEAD0  )  [extern]
```

Definition at line 2490 of file tools.c.

4.19.8.144 ExprStatus()

```
void ExprStatus (
            EXPRESSIONS e )  [extern]
```

Definition at line 66 of file bugtool.c.

4.19.8.145 iniTools()

```
void iniTools (
            void  )  [extern]
```

Definition at line 2224 of file tools.c.

4.19.8.146 TestTerm()

```
int TestTerm (
            WORD * term )  [extern]
```

Tests the consistency of the term. Returns 0 when the term is OK. Any nonzero value is trouble. In the current version the testing isn't 100% complete. For instance, we don't check the validity of the symbols nor do we check the range of their powers. Etc. This should be extended when the need is there.

Parameters

term	the term to be tested

Definition at line 3812 of file tools.c.

References TestTerm().

Referenced by TestTerm().

Here is the call graph for this function:

4.19.8.147 RunTransform()

```
int RunTransform (
            PHEAD WORD * term,
            WORD * params )  [extern]
```

Definition at line 750 of file transform.c.

4.19.8.148 RunEncode()

```
int RunEncode (
            PHEAD WORD * fun,
            WORD * args,
            WORD * info )  [extern]
```

Definition at line 1008 of file transform.c.

4.19.8.149 RunDecode()

```
int RunDecode (
            PHEAD WORD * fun,
            WORD * args,
            WORD * info )  [extern]
```

Definition at line 1197 of file transform.c.

4.19.8.150 RunReplace()

```
int RunReplace (
            PHEAD WORD * fun,
            WORD * args,
            WORD * info )  [extern]
```

Definition at line 1369 of file transform.c.

4.19.8.151 RunImplode()

```
int RunImplode (
            WORD * fun,
            WORD * args )  [extern]
```

Definition at line 1849 of file transform.c.

4.19.8.152 RunExplode()

```
int RunExplode (
            PHEAD WORD * fun,
            WORD * args )  [extern]
```

Definition at line 2050 of file transform.c.

4.19.8.153 TestArgNum()

```
int TestArgNum (
            int n,
            int totarg,
            WORD * args )  [extern]
```

Definition at line 3315 of file transform.c.

4.19.8.154 PutArgInScratch()

```
WORD PutArgInScratch (
            WORD * arg,
            UWORD * scrat )  [extern]
```

Definition at line 3384 of file transform.c.

4.19.8.155 ReadRange()

```
UBYTE * ReadRange (
            UBYTE * s,
            WORD * out,
            int par )  [extern]
```

Definition at line 3420 of file transform.c.

4.19.8.156 FindRange()

```
int FindRange (
            PHEAD WORD * args,
            WORD * arg1,
            WORD * arg2,
            WORD totarg )  [extern]
```

Definition at line 3580 of file transform.c.

4.19.8.157 RunPermute()

```
int RunPermute (
            PHEAD WORD * fun,
            WORD * args,
            WORD * info )  [extern]
```

Definition at line 2164 of file transform.c.

4.19.8.158 RunReverse()

```
int RunReverse (
            PHEAD WORD * fun,
            WORD * args )  [extern]
```

Definition at line 2391 of file transform.c.

4.19.8.159 RunCycle()

```
int RunCycle (
            PHEAD WORD * fun,
            WORD * args,
            WORD * info )  [extern]
```

Definition at line 2567 of file transform.c.

4.19.8.160 RunAddArg()

```
int RunAddArg (
            PHEAD WORD * fun,
            WORD * args )  [extern]
```

Definition at line 2719 of file transform.c.

4.19.8.161 RunMulArg()

```
int RunMulArg (
            PHEAD WORD * fun,
            WORD * args )  [extern]
```

Definition at line 2808 of file transform.c.

4.19.8.162 RunIsLyndon()

```
int RunIsLyndon (
            PHEAD WORD * fun,
            WORD * args,
            int par )  [extern]
```

Definition at line 2949 of file transform.c.

4.19.8.163 RunToLyndon()

```
WORD RunToLyndon (
            PHEAD WORD * fun,
            WORD * args,
            int par )  [extern]
```

Definition at line 3027 of file transform.c.

4.19.8.164 RunDropArg()

```
int RunDropArg (
            PHEAD WORD * fun,
            WORD * args )  [extern]
```

Definition at line 3139 of file transform.c.

4.19.8.165 RunSelectArg()

```
int RunSelectArg (
            PHEAD WORD * fun,
            WORD * args )  [extern]
```

Definition at line 3165 of file transform.c.

4.19.8.166 RunDedup()

```
int RunDedup (
            PHEAD WORD * fun,
            WORD * args )  [extern]
```

Definition at line 2478 of file transform.c.

4.19.8.167 RunZtoHArg()

```
int RunZtoHArg (
            PHEAD WORD * fun,
            WORD * args )  [extern]
```

Definition at line 3193 of file transform.c.

4.19.8.168 RunHtoZArg()

```
int RunHtoZArg (
            PHEAD WORD * fun,
            WORD * args )  [extern]
```

Definition at line 3249 of file transform.c.

4.19.8.169 NormPolyTerm()

```
int NormPolyTerm (
            PHEAD WORD * term )  [extern]
```

Definition at line 53 of file notation.c.

4.19.8.170 ConvertToPoly()

```
int ConvertToPoly (
            PHEAD WORD * term,
            WORD * outterm,
            WORD * comlist,
            WORD par )  [extern]
```

Definition at line 309 of file notation.c.

4.19.8.171 LocalConvertToPoly()

```
int LocalConvertToPoly (
            PHEAD WORD * term,
            WORD * outterm,
            WORD startebuf,
            WORD par )  [extern]
```

Converts a generic term to polynomial notation in which there are only symbols and brackets. During conversion there will be only symbols. Brackets are stripped. Objects that need 'translation' are put inside a special compiler buffer and represented by a symbol. The numbering of the extra symbols is down from the maximum. In principle there can be a problem when running into the already assigned ones. This uses the FindTree for searching in the global tree and then looks further in the AT.ebufnum. This allows fully parallel processing. Hence we need no locks. Cannot be used in the same module as ConvertToPoly.

Definition at line 514 of file notation.c.

Referenced by poly_factorize_expression().

4.19.8.172 ConvertFromPoly()

```
int ConvertFromPoly (
            PHEAD WORD * term,
            WORD * outterm,
            WORD from,
            WORD to,
            WORD offset,
            WORD par )  [extern]
```

Definition at line 664 of file notation.c.

4.19.8.173 FindSubterm()

```
int FindSubterm (
            WORD * subterm )  [extern]
```

Definition at line 777 of file notation.c.

4.19.8.174 FindLocalSubterm()

```
int FindLocalSubterm (
            PHEAD WORD * subterm,
            WORD startebuf )  [extern]
```

Definition at line 847 of file notation.c.

4.19.8.175 PrintSubtermList()

```
void PrintSubtermList (
            int from,
            int to )  [extern]
```

Definition at line 901 of file notation.c.

4.19.8.176 PrintExtraSymbol()

```
void PrintExtraSymbol (
             int num,
             WORD * terms,
             int par )  [extern]
```

Definition at line 1013 of file notation.c.

4.19.8.177 FindSubexpression()

```
int FindSubexpression (
             WORD * subexpr )  [extern]
```

Definition at line 1100 of file notation.c.

4.19.8.178 UpdateMaxSize()

```
void UpdateMaxSize (
             void  )  [extern]
```

Definition at line 1614 of file tools.c.

4.19.8.179 CoToPolynomial()

```
int CoToPolynomial (
             UBYTE * inp )  [extern]
```

Definition at line 5958 of file compcomm.c.

4.19.8.180 CoFromPolynomial()

```
int CoFromPolynomial (
             UBYTE * inp )  [extern]
```

Definition at line 6033 of file compcomm.c.

4.19.8.181 CoArgToExtraSymbol()

```
int CoArgToExtraSymbol (
             UBYTE * s )  [extern]
```

Definition at line 6059 of file compcomm.c.

4.19.8.182 CoExtraSymbols()

```
int CoExtraSymbols (
             UBYTE * inp )  [extern]
```

Definition at line 6109 of file compcomm.c.

4.19.8.183 GetDoParam()

```
UBYTE * GetDoParam (
             UBYTE * inp,
             WORD ** wp,
             int par )  [extern]
```

Definition at line 6240 of file compcomm.c.

4.19.8.184 GetIfDollarFactor()

```
WORD * GetIfDollarFactor (
             UBYTE ** inp,
             WORD * w )  [extern]
```

Definition at line 6182 of file compcomm.c.

4.19.8.185 CoDo()

```
int CoDo (
             UBYTE * inp )  [extern]
```

Definition at line 6317 of file compcomm.c.

4.19.8.186 CoEndDo()

```
int CoEndDo (
             UBYTE * inp )  [extern]
```

Definition at line 6420 of file compcomm.c.

4.19.8.187 ExtraSymFun()

```
int ExtraSymFun (
             PHEAD WORD * term,
             WORD level )  [extern]
```

Definition at line 1153 of file notation.c.

4.19.8.188 PruneExtraSymbols()

```
int PruneExtraSymbols (
             WORD downto )  [extern]
```

Definition at line 1221 of file notation.c.

4.19.8.189 IniFbuffer()

```
int IniFbuffer (
            WORD bufnum )  [extern]
```

Initialize a factorization cache buffer. We set the size of the rhs and boomlijst buffers immediately to their final values.

Definition at line 612 of file comtool.c.

References tree::blnce, CbUf::boomlijst, CbUf::CanCommu, CbUf::dimension, tree::left, CbUf::numdum, CbUf::NumTerms, tree::parent, CbUf::rhs, tree::right, tree::usage, and tree::value.

4.19.8.190 GCDfunction()

```
int GCDfunction (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 609 of file ratio.c.

4.19.8.191 GCDfunction3()

```
WORD * GCDfunction3 (
            PHEAD WORD * in1,
            WORD * in2 )  [extern]
```

Definition at line 1147 of file ratio.c.

4.19.8.192 ReadPolyRatFun()

```
int ReadPolyRatFun (
            PHEAD WORD * term )  [extern]
```

Definition at line 2321 of file ratio.c.

4.19.8.193 FromPolyRatFun()

```
int FromPolyRatFun (
            PHEAD WORD * fun,
            WORD ** numout,
            WORD ** denout )  [extern]
```

Definition at line 2440 of file ratio.c.

4.19.8.194 PolyDiv()

```
WORD * PolyDiv (
            PHEAD WORD * a,
            WORD * b,
            char * text )  [extern]
```

Definition at line 2800 of file ratio.c.

4.19.8.195 GCDclean()

```
void GCDclean (
            PHEAD WORD * num,
            WORD * den )  [extern]
```

Definition at line 2703 of file ratio.c.

4.19.8.196 TakeSymbolContent()

```
WORD * TakeSymbolContent (
            PHEAD WORD * in,
            WORD * term )  [extern]
```

Implements part of the old ExecArg in which we take common factors from arguments with more than one term. We allow only symbols as this code is used for the polyratfun only. We have a special routine, because the generic TakeContent does too much work and speed is at a premium here. Input: in is the input expression as a sequence of terms. Output: term: the content return value: the contentfree expression. it is in new allocation, made by TermMalloc. (should be in a TermMalloc space?)

Definition at line 2493 of file ratio.c.

References GetModInverses().

Here is the call graph for this function:

4.19.8.197 GCDterms()

```
int GCDterms (
            PHEAD WORD * term1,
            WORD * term2,
            WORD * termout )  [extern]
```

Definition at line 2133 of file ratio.c.

4.19.8.198 PutExtraSymbols()

```
WORD * PutExtraSymbols (
            PHEAD WORD * in,
            WORD startebuf,
            int * actionflag )  [extern]
```

Definition at line 1249 of file ratio.c.

4.19.8.199 TakeExtraSymbols()

```
WORD * TakeExtraSymbols (
            PHEAD WORD * in,
            WORD startebuf )  [extern]
```

Definition at line 1278 of file ratio.c.

4.19.8.200 MultiplyWithTerm()

```
WORD * MultiplyWithTerm (
            PHEAD WORD * in,
            WORD * term,
            WORD par )  [extern]
```

Definition at line 1317 of file ratio.c.

4.19.8.201 TakeContent()

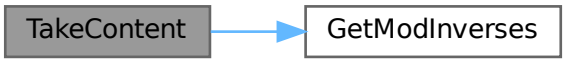
```
WORD * TakeContent (
            PHEAD WORD * in,
            WORD * term )  [extern]
```

Implements part of the old ExecArg in which we take common factors from arguments with more than one term. Here the input is a sequence of terms in 'in' and the answer is a content-free sequence of terms. This sequence has been allocated by the Malloc1 routine in a call to EndSort, unless the expression was already content-free. In that case the input pointer is returned. The content is returned in term. This is supposed to be a separate allocation, made by TermMalloc in the calling routine.

Definition at line 1382 of file ratio.c.

References GetModInverses().

Here is the call graph for this function:



4.19.8.202 MergeSymbolLists()

```
int MergeSymbolLists (
            PHEAD WORD * old,
            WORD * extra,
            int par )  [extern]
```

Definition at line 1830 of file ratio.c.

4.19.8.203 MergeDotproductLists()

```
int MergeDotproductLists (
            PHEAD WORD * old,
            WORD * extra,
            int par )  [extern]
```

Definition at line 1950 of file ratio.c.

4.19.8.204 CreateExpression()

```
WORD * CreateExpression (
            PHEAD WORD )  [extern]
```

Definition at line 2077 of file ratio.c.

4.19.8.205 DIVfunction()

```
int DIVfunction (
            PHEAD WORD * term,
            WORD level,
            int par )  [extern]
```

Definition at line 2847 of file ratio.c.

4.19.8.206 MULfunc()

```
WORD * MULfunc (
            PHEAD WORD * p1,
            WORD * p2 )  [extern]
```

Definition at line 3020 of file ratio.c.

4.19.8.207 ConvertArgument()

```
WORD * ConvertArgument (
            PHEAD WORD * arg,
            int * type )  [extern]
```

Definition at line 3083 of file ratio.c.

4.19.8.208 ExpandRat()

```
int ExpandRat (
            PHEAD WORD * fun )  [extern]
```

Definition at line 3179 of file ratio.c.

4.19.8.209 InvPoly()

```
int InvPoly (
             PHEAD WORD * inpoly,
             WORD maxpow,
             WORD sym )  [extern]
```

Definition at line 3649 of file ratio.c.

4.19.8.210 TestDoLoop()

```
WORD TestDoLoop (
             PHEAD WORD * lhsbuf,
             WORD level )  [extern]
```

Definition at line 2746 of file dollar.c.

4.19.8.211 TestEndDoLoop()

```
WORD TestEndDoLoop (
             PHEAD WORD * lhsbuf,
             WORD level )  [extern]
```

Definition at line 2824 of file dollar.c.

4.19.8.212 poly_gcd()

```
WORD * poly_gcd (
             PHEAD WORD * a,
             WORD * b,
             WORD fit )  [extern]
```

Polynomial gcd

4.19.9 Description

This method calculates the greatest common divisor of two polynomials, given by two zero-terminated Form-style term lists.

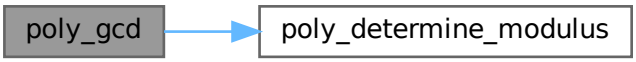
4.19.10 Notes

- The result is written at newly allocated memory
- Called from ratio.c
- Calls polygcd::gcd

Definition at line 124 of file polywrap.cc.

References poly_determine_modulus().

Here is the call graph for this function:



4.19.10.1 poly_div()

```
WORD * poly_div (
             PHEAD WORD * a,
             WORD * b,
             WORD fit )  [extern]
```

Definition at line 437 of file polywrap.cc.

4.19.10.2 poly_rem()

```
WORD * poly_rem (
             PHEAD WORD * a,
             WORD * b,
             WORD fit )  [extern]
```

Definition at line 465 of file polywrap.cc.

4.19.10.3 poly_inverse()

```
WORD * poly_inverse (
             PHEAD WORD * arga,
             WORD * argb )  [extern]
```

Definition at line 1757 of file polywrap.cc.

4.19.10.4 poly_mul()

```
WORD * poly_mul (
            PHEAD WORD * a,
            WORD * b )  [extern]
```

Definition at line 1962 of file polywrap.cc.

4.19.10.5 poly_ratfun_add()

```
WORD * poly_ratfun_add (
            PHEAD WORD * t1,
            WORD * t2 )  [extern]
```

Addition of PolyRatFuns

4.19.11 Description

This method gets two pointers to polyratfuns with up to two arguments each and calculates the sum.

4.19.12 Notes

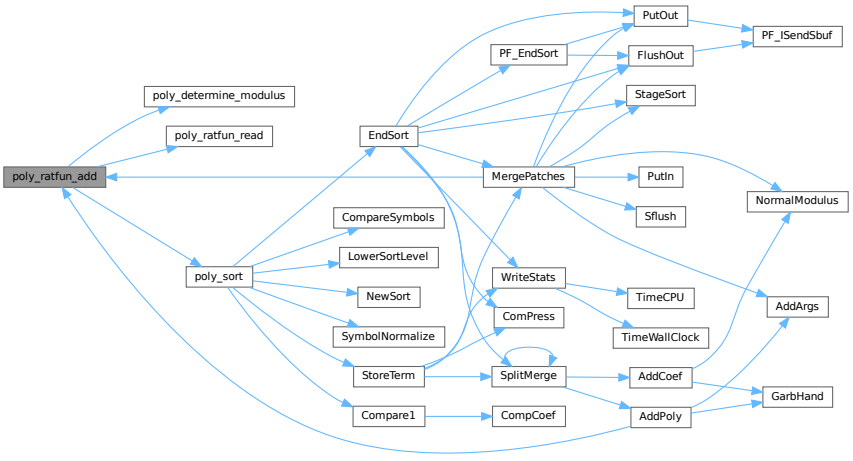
- The result is written at the workpointer
- Called from sort.c and threads.c
- Calls poly::operators and polygcd::gcd

Definition at line 635 of file polywrap.cc.

References poly_determine_modulus(), poly_ratfun_read(), and poly_sort().

Referenced by AddPoly(), and MergePatches().

Here is the call graph for this function:



4.19.12.1 poly_ratfun_normalize()

```
int poly_ratfun_normalize (
            PHEAD WORD * term )  [extern]
```

Multiplication/normalization of PolyRatFuns

4.19.13 Description

This method searches a term for multiple polyratfuns and multiplies their contents. The result is properly normalized. Normalization also works for terms with a single polyratfun.

4.19.14 Notes

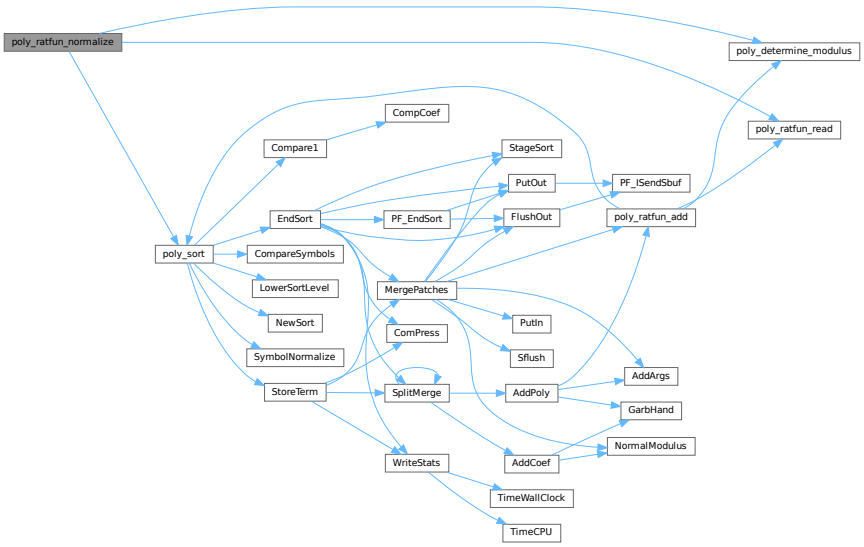
- The result overwrites the original term
- Called from proces.c
- Calls poly::operators and polygcd::gcd

Definition at line 771 of file polywrap.cc.

References poly_determine_modulus(), poly_ratfun_read(), and poly_sort().

Referenced by PolyFunMul(), and PrepPoly().

Here is the call graph for this function:



4.19.14.1 poly_factorize_argument()

```
int poly_factorize_argument (
            PHEAD WORD * argin,
            WORD * argout )  [extern]
```

Factorization of function arguments

4.19.15 Description

This method factorizes the Form-style argument argin.

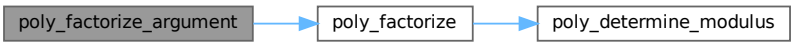
4.19.16 Notes

- The result is written at argout
- Called from argument.c
- Calls poly_factorize

Definition at line 1114 of file polywrap.cc.

References poly_factorize().

Here is the call graph for this function:



4.19.16.1 poly_factorize_dollar()

```
WORD * poly_factorize_dollar (
            PHEAD WORD * argin )  [extern]
```

Factorization of dollar variables

4.19.17 Description

This method factorizes a dollar variable.

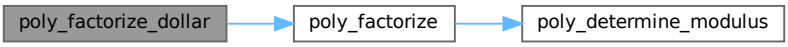
4.19.18 Notes

- The result is written at newly allocated memory.
- Called from dollar.c
- Calls poly_factorize

Definition at line 1148 of file polywrap.cc.

References poly_factorize().

Here is the call graph for this function:



4.19.18.1 poly_factorize_expression()

```
int poly_factorize_expression (
            EXPRESSIONS expr )  [extern]
```

Factorization of expressions

4.19.19 Description

This method factorizes an expression.

4.19.20 Notes

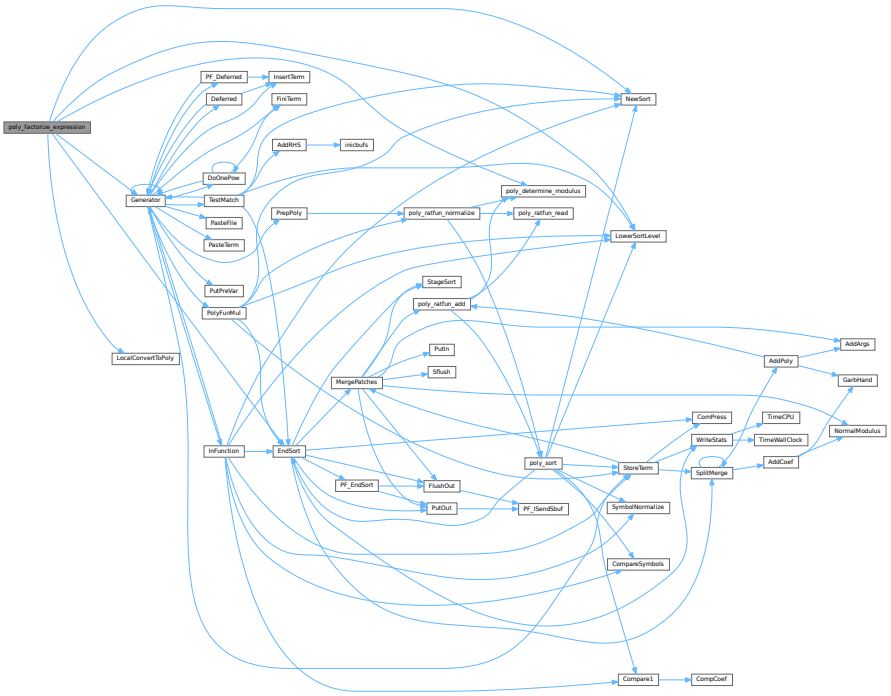
- The result overwrites the input expression
- Called from proces.c
- Calls polyfact::factorize

Definition at line 1180 of file polywrap.cc.

References EndSort(), Generator(), FiLe::handle, LocalConvertToPoly(), LowerSortLevel(), NewSort(), and poly_determine_modulus().

Referenced by PF_Processor(), and Processor().

Here is the call graph for this function:



4.19.20.1 poly_unfactorize_expression()

```
int poly_unfactorize_expression (
            EXPRESSIONS expr )  [extern]
```

Unfactorization of expressions

4.19.21 Description

This method expands a factorized expression.

4.19.22 Notes

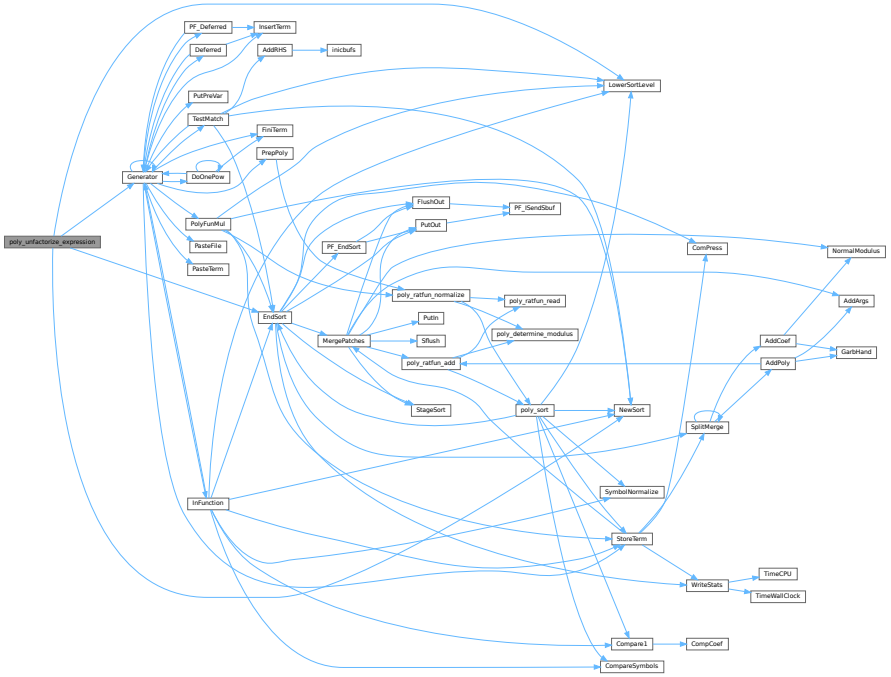
- The result overwrites the input expression
- Called from proces.c

Definition at line 1545 of file polywrap.cc.

References EndSort(), Generator(), FiLe::handle, LowerSortLevel(), and NewSort().

Referenced by PF_Processor(), and Processor().

Here is the call graph for this function:



4.19.22.1 poly_free_poly_vars()

```
void poly_free_poly_vars (
            PHEAD const char * text )  [extern]
```

Definition at line 2028 of file polywrap.cc.

4.19.22.2 optimize_print_code()

```
void optimize_print_code (
            int print_expr )  [extern]
```

Print optimized code

4.19.23 Description

This method prints the optimized code via "PrintExtraSymbol". Depending on the flag, the original expression is printed (for "Print") or not (for "#Optimize / #write "O").

Definition at line 4528 of file optimize.cc.

Referenced by Optimize().

4.19.23.1 DoPreAdd()

```
int DoPreAdd (
            UBYTE * s )  [extern]
```

Definition at line 1067 of file dict.c.

4.19.23.2 DoPreUseDictionary()

```
int DoPreUseDictionary (
            UBYTE * s )  [extern]
```

Definition at line 1022 of file dict.c.

4.19.23.3 DoPreCloseDictionary()

```
int DoPreCloseDictionary (
            UBYTE * s )  [extern]
```

Definition at line 998 of file dict.c.

4.19.23.4 DoPreOpenDictionary()

```
int DoPreOpenDictionary (
            UBYTE * s )  [extern]
```

Definition at line 959 of file dict.c.

4.19.23.5 RemoveDictionary()

```
void RemoveDictionary (
            DICTIONARY * dict )  [extern]
```

Definition at line 902 of file dict.c.

4.19.23.6 UnSetDictionary()

```
void UnSetDictionary (
            void  )  [extern]
```

Definition at line 883 of file dict.c.

4.19.23.7 SetDictionaryOptions()

```
int SetDictionaryOptions (
            UBYTE * options )  [extern]
```

Definition at line 790 of file dict.c.

4.19.23.8 AddToDictionary()

```
int AddToDictionary (
            DICTIONARY * dict,
            UBYTE * left,
            UBYTE * right )  [extern]
```

Definition at line 491 of file dict.c.

4.19.23.9 AddDictionary()

```
int AddDictionary (
            UBYTE * name )  [extern]
```

Definition at line 449 of file dict.c.

4.19.23.10 FindDictionary()

```
int FindDictionary (
            UBYTE * name )  [extern]
```

Definition at line 434 of file dict.c.

4.19.23.11 IsExponentSign()

```
UBYTE * IsExponentSign (
            void  )  [extern]
```

Definition at line 238 of file dict.c.

4.19.23.12 IsMultiplySign()

```
UBYTE * IsMultiplySign (
            void  )  [extern]
```

Definition at line 218 of file dict.c.

4.19.23.13 TransformRational()

```
void TransformRational (
            UWORD * a,
            WORD na )  [extern]
```

Definition at line 71 of file dict.c.

4.19.23.14 WriteDictionary()

```
void WriteDictionary (
            DICTIONARY * dict )  [extern]
```

Definition at line 1289 of file sch.c.

4.19.23.15 ShrinkDictionary()

```
void ShrinkDictionary (
            DICTIONARY * dict )  [extern]
```

Definition at line 945 of file dict.c.

4.19.23.16 MultiplyToLine()

```
void MultiplyToLine (
            void  )  [extern]
```

Definition at line 653 of file sch.c.

4.19.23.17 FindSymbol()

```
UBYTE * FindSymbol (
            WORD num )  [extern]
```

Definition at line 258 of file dict.c.

4.19.23.18 FindVector()

```
UBYTE * FindVector (
            WORD num )  [extern]
```

Definition at line 279 of file dict.c.

4.19.23.19 FindIndex()

```
UBYTE * FindIndex (
            WORD num )  [extern]
```

Definition at line 301 of file dict.c.

4.19.23.20 FindFunction()

```
UBYTE * FindFunction (
            WORD num )  [extern]
```

Definition at line 323 of file dict.c.

4.19.23.21 FindFunWithArgs()

```
UBYTE * FindFunWithArgs (
            WORD * t )  [extern]
```

Definition at line 345 of file dict.c.

4.19.23.22 FindExtraSymbol()

```
UBYTE * FindExtraSymbol (
            WORD num )  [extern]
```

Definition at line 377 of file dict.c.

4.19.23.23 DictToBytes()

```
LONG DictToBytes (
            DICTIONARY * dict,
            UBYTE * buf )  [extern]
```

Definition at line 1119 of file dict.c.

4.19.23.24 DictFromBytes()

```
DICTIONARY * DictFromBytes (
            UBYTE * buf )  [extern]
```

Definition at line 1152 of file dict.c.

4.19.23.25 CoCreateSpectator()

```
int CoCreateSpectator (
            UBYTE * inp )  [extern]
```

Definition at line 89 of file spectator.c.

4.19.23.26 CoToSpectator()

```
int CoToSpectator (
            UBYTE * inp )  [extern]
```

Definition at line 181 of file spectator.c.

4.19.23.27 CoRemoveSpectator()

```
int CoRemoveSpectator (
            UBYTE * inp )  [extern]
```

Definition at line 233 of file spectator.c.

4.19.23.28 CoEmptySpectator()

```
int CoEmptySpectator (
            UBYTE * inp )  [extern]
```

Definition at line 293 of file spectator.c.

4.19.23.29 CoCopySpectator()

```
int CoCopySpectator (
            UBYTE * inp )  [extern]
```

Definition at line 463 of file spectator.c.

4.19.23.30 PutInSpectator()

```
int PutInSpectator (
            WORD * term,
            WORD specnum )  [extern]
```

Definition at line 345 of file spectator.c.

4.19.23.31 ClearSpectators()

```
void ClearSpectators (
            WORD par )  [extern]
```

Definition at line 685 of file spectator.c.

4.19.23.32 GetFromSpectator()

```
WORD GetFromSpectator (
            WORD * term,
            WORD specnum )  [extern]
```

Definition at line 605 of file spectator.c.

4.19.23.33 FlushSpectators()

```
void FlushSpectators (
            void  )  [extern]
```

Definition at line 404 of file spectator.c.

4.19.23.34 ReadFromScratch()

```
int ReadFromScratch (
            FILEHANDLE * fi,
            POSITION * pos,
            UBYTE * buffer,
            POSITION * length )  [extern]
```

Definition at line 286 of file store.c.

4.19.23.35 AddToScratch()

```
int AddToScratch (
            FILEHANDLE * fi,
            POSITION * pos,
            UBYTE * buffer,
            POSITION * length,
            int withflush )  [extern]
```

Definition at line 315 of file store.c.

4.19.23.36 DoPreAppendPath()

```
int DoPreAppendPath (
            UBYTE * s )  [extern]
```

Appends the given path (absolute or relative to the current file directory) to the FORM path.

Syntax: #appendpath <path>

Definition at line 7248 of file pre.c.

4.19.23.37 DoPrePrependPath()

```
int DoPrePrependPath (
            UBYTE * s )  [extern]
```

Prepends the given path (absolute or relative to the current file directory) to the FORM path.

Syntax: #prependpath <path>

Definition at line 7265 of file pre.c.

4.19.23.38 DoSwitch()

```
int DoSwitch (
            PHEAD WORD * term,
            WORD * lhs )  [extern]
```

Definition at line 1086 of file if.c.

4.19.23.39 DoEndSwitch()

```
int DoEndSwitch (
             PHEAD WORD * term,
             WORD * lhs )  [extern]
```

Definition at line 1102 of file if.c.

4.19.23.40 FindCase()

```
SWITCHTABLE * FindCase (
             WORD nswitch,
             WORD ncase )  [extern]
```

Definition at line 1113 of file if.c.

4.19.23.41 SwitchSplitMergeRec()

```
void SwitchSplitMergeRec (
             SWITCHTABLE * array,
             WORD num,
             SWITCHTABLE * auxarray )  [extern]
```

Definition at line 1200 of file if.c.

4.19.23.42 SwitchSplitMerge()

```
void SwitchSplitMerge (
             SWITCHTABLE * array,
             WORD num )  [extern]
```

Definition at line 1229 of file if.c.

4.19.23.43 DoubleSwitchBuffers()

```
int DoubleSwitchBuffers (
             void  )  [extern]
```

Definition at line 1151 of file if.c.

4.19.23.44 DistrN()

```
int DistrN (
             int n,
             int * cpl,
             int ncpl,
             int * scratch )  [extern]
```

Definition at line 4028 of file tools.c.

4.19.23.45 DoNamespace()

```
int DoNamespace (
            UBYTE * s )  [extern]
```

Definition at line 7329 of file pre.c.

4.19.23.46 DoEndNamespace()

```
int DoEndNamespace (
            UBYTE * s )  [extern]
```

Definition at line 7377 of file pre.c.

4.19.23.47 DoUse()

```
int DoUse (
            UBYTE * s )  [extern]
```

Definition at line 7594 of file pre.c.

4.19.23.48 SkipName()

```
UBYTE * SkipName (
            UBYTE * s )  [extern]
```

Definition at line 7400 of file pre.c.

4.19.23.49 ConstructName()

```
UBYTE * ConstructName (
            UBYTE * s,
            UBYTE type )  [extern]
```

Definition at line 7500 of file pre.c.

4.19.23.50 DoSetUserFlag()

```
int DoSetUserFlag (
            UBYTE * s )  [extern]
```

Definition at line 7759 of file pre.c.

4.19.23.51 DoClearUserFlag()

```
int DoClearUserFlag (
            UBYTE * s )  [extern]
```

Definition at line 7749 of file pre.c.

4.19.23.52 CreateVertex()

```
VERTEX * CreateVertex (
             MODEL * m )
```

Definition at line 76 of file model.c.

4.19.23.53 ReadParticle()

```
UBYTE * ReadParticle (
             UBYTE * s,
             VERTEX * v,
             MODEL * m,
             int par )
```

Definition at line 106 of file model.c.

4.19.23.54 CoModel()

```
int CoModel (
             UBYTE * s )
```

Definition at line 149 of file model.c.

4.19.23.55 CoParticle()

```
int CoParticle (
             UBYTE * s )
```

Definition at line 224 of file model.c.

4.19.23.56 CoVertex()

```
int CoVertex (
             UBYTE * s )
```

Definition at line 306 of file model.c.

4.19.23.57 CoEndModel()

```
int CoEndModel (
             UBYTE * s )
```

Definition at line 418 of file model.c.

4.19.23.58 LoadModel()

```
int LoadModel (
            MODEL * m )
```

Definition at line 58 of file diawrap.cc.

4.19.23.59 CoSetUserFlag()

```
int CoSetUserFlag (
            UBYTE * s )
```

Definition at line 7417 of file compcomm.c.

4.19.23.60 CoClearUserFlag()

```
int CoClearUserFlag (
            UBYTE * s )
```

Definition at line 7445 of file compcomm.c.

4.19.23.61 CoCreateAllLoops()

```
int CoCreateAllLoops (
            UBYTE * s )
```

Definition at line 7482 of file compcomm.c.

4.19.23.62 CoCreateAllPaths()

```
int CoCreateAllPaths (
            UBYTE * s )
```

Definition at line 7602 of file compcomm.c.

4.19.23.63 CoCreateAll()

```
int CoCreateAll (
            UBYTE * s )
```

Definition at line 7730 of file compcomm.c.

4.19.23.64 AllLoops()

```
int AllLoops (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 650 of file lus.c.

4.19.23.65 StartLoops()

```
LONG StartLoops (
             PHEAD WORD * term,
             WORD level,
             LONG vert,
             WORD nvert,
             WORD * arglist,
             WORD nargs,
             WORD * loop,
             WORD nloop )
```

Definition at line 894 of file lus.c.

4.19.23.66 GenLoops()

```
LONG GenLoops (
             PHEAD WORD * term,
             WORD level,
             LONG vert,
             WORD nvert,
             WORD * arglist,
             WORD nargs,
             WORD * loop,
             WORD nloop )
```

Definition at line 982 of file lus.c.

4.19.23.67 LoopOutput()

```
void LoopOutput (
             PHEAD WORD * term,
             WORD level,
             WORD * loop,
             WORD nloop )
```

Definition at line 1061 of file lus.c.

4.19.23.68 AllPaths()

```
int AllPaths (
             PHEAD WORD * term,
             WORD level )
```

Definition at line 1125 of file lus.c.

4.19.23.69 GenPaths()

```
LONG GenPaths (
              PHEAD WORD * term,
              WORD level,
              LONG vert,
              WORD nvert,
              WORD * arglist,
              WORD nargs,
              WORD * path,
              WORD npath )
```

Definition at line 1415 of file lus.c.

4.19.23.70 PathOutput()

```
void PathOutput (
              PHEAD WORD * term,
              WORD level,
              WORD * path,
              WORD npath )
```

Definition at line 1488 of file lus.c.

4.20 declare.h

Go to the documentation of this file.

```
00001 #ifndef __FDECLARE__
00002 #define __FDECLARE__
00003
00009 /* #[ License : */
00010 /*
00011  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00012  *   When using this file you are requested to refer to the publication
00013  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00014  *   This is considered a matter of courtesy as the development was paid
00015  *   for by FOM the Dutch physics granting agency and we would like to
00016  *   be able to track its scientific use to convince FOM of its value
00017  *   for the community.
00018  *
00019  *   This file is part of FORM.
00020  *
00021  *   FORM is free software: you can redistribute it and/or modify it under the
00022  *   terms of the GNU General Public License as published by the Free Software
00023  *   Foundation, either version 3 of the License, or (at your option) any later
00024  *   version.
00025  *
00026  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00027  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00028  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00029  *   details.
00030  *
00031  *   You should have received a copy of the GNU General Public License along
00032  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00033  */
00034 /* #] License : */
00035
00036 /*
00037     #[ Macro's :
00038 */
00039
00040 #define MaX(x,y) ((x) > (y) ? (x): (y))
00041 #define MiN(x,y) ((x) < (y) ? (x): (y))
00042 #define ABS(x) ( (x) < 0 ? -(x): (x) )
00043 #define SGN(x) ( (x) > 0 ? 1 : (x) < 0 ? -1 : 0 )
00044 #define REDLENG(x) ((((x)<0)?((x)+1):((x)-1))/2)
00045 #define INCLENG(x) (((x)<0)?(((x)*2)-1):(((x)*2)+1))
```

```
00046 #define GETCOEF(x,y) x += *x;y = x[-1];x -= ABS(y);y=REDLENG(y)
00047 #define GETSTOP(x,y) y=x+(*x)-1;y -= ABS(*y)-1
00048 #define StuffAdd(x,y)  (((x)<0?-1:1)*(y)+((y)<0?-1:1)*(x))
00049
00050 #define EXCHN(t1,t2,n) { WORD a,i; for(i=0;i<n;i++){a=t1[i];t1[i]=t2[i];t2[i]=a;} }
00051 #define EXCH(x,y) { WORD a = (x); (x) = (y); (y) = a; }
00052
00053 #define TOKENTOLINE(x,y) if ( AC.OutputSpaces == NOSPACEFORMAT ) { \
00054         TokenToLine((UBYTE *)(y)); } else { TokenToLine((UBYTE *)(x)); }
00055
00056 #define UngetFromStream(stream,c) ((stream)->nextchar[(stream)->isnextchar++]=c)
00057 #ifdef WITHRETURN
00058 #define AddLineFeed(s,n) { (s)[(n)++] = CARRIAGERETURN; (s)[(n)++] = LINEFEED; }
00059 #else
00060 #define AddLineFeed(s,n) { (s)[(n)++] = LINEFEED; }
00061 #endif
00062 #define TryRecover(x) Terminate(-1)
00063 #define UngetChar(c) { pushbackchar = c; }
00064 #define ParseNumber(x,s) {(x)=0;while(*(s)>='0'&&*(s)<='9')(x)=10*(x)+*(s)++ -'0';}
00065 #define ParseSign(sgn,s) {(sgn)=0;while(*(s)=='-'||*(s)=='+'){\
00066           if ( *(s)++ == '-' ) sgn ^= 1;}}
00067 #define ParseSignedNumber(x,s) { int sgn; ParseSign(sgn,s)\
00068           ParseNumber(x,s) if ( sgn ) x = -x; }
00069
00070 /* (n) is necessary here, since the macro is sometimes passed dereferenced pointers for n */
00071 #define NCOPY(s,t,n) { while ( (n)-- > 0 ) { *s++ = *t++; } }
00072 #define NCOPYI(s,t,n) { while ( (n)-- > 0 ) { *s++ = *t++; } }
00073 #define NCOPYB(s,t,n) { while ( (n)-- > 0 ) { *s++ = *t++; } }
00074 #define NCOPYI32(s,t,n) { while ( (n)-- > 0 ) { *s++ = *t++; } }
00075 #define WCOPY(s,t,n) { int nn=n; WORD *ss=(WORD *)s, *tt=(WORD *)t; while ( (nn)-- > 0 ) { *ss++ =
      *tt++; } }
00076
00077 /* Use memmove not memcpy: we can't guarantee that s, t do not overlap */
00078 /* Using memmove produces no measurable performance improvement or regression. */
00079 /*#define NCOPY(s,t,n) { memmove(s,t,(n)*sizeof(*s)); s+=n; t+=n; n=0; }
00080 #define NCOPYI(s,t,n) { NCOPY(s,t,n); }
00081 #define NCOPYB(s,t,n) { NCOPY(s,t,n); }
00082 #define NCOPYI32(s,t,n) { NCOPY(s,t,n); }
00083 #define WCOPY(s,t,n) { int nn=n; WORD *ss=(WORD *)s, *tt=(WORD *)t; NCOPY(ss,tt,nn); }*/
00084
00085 #define NeedNumber(x,s,err) { int sgn = 1;                                  \
00086         while ( *s == ' ' || *s == '\t' || *s == '-' || *s == '+' ) {       \
00087             if ( *s == '-' ) {sgn = -sgn;} s++; }                           \
00088         if ( chartype[*s] != 1 ) goto err;                                  \
00089         ParseNumber(x,s)                                                    \
00090         if ( sgn < 0 ) {(x) = -(x);} while ( *s == ' ' || *s == '\t' ) s++;\
00091     }
00092 #define SKIPBLANKS(s) { while ( *(s) == ' ' || *(s) == '\t' ) (s)++; }
00093 #define FLUSHCONSOLE if ( AP.InOutBuf > 0 ) CharOut(LINEFEED)
00094
00095 #define SKIPBRA1(s) { int lev1=0; s++; while(*s) { if(*s=='['){lev1++;} \
00096                     else {if(*s==']'&&--lev1<0){break;}} s++;} }
00097 #define SKIPBRA2(s) { int lev2=0; s++; while(*s) { if(*s=='{'){lev2++;} \
00098                     else {if(*s=='}'&&--lev2<0){break;} \
00099                     else {if(*s=='['){SKIPBRA1(s)}}} s++;} }
00100 #define SKIPBRA3(s) { int lev3=0; s++; while(*s) { if(*s=='('){lev3++;} \
00101                     else {if(*s==')'&&--lev3<0){break;} \
00102                     else {if(*s=='{'){SKIPBRA2(s)} \
00103                     else {if(*s=='['){SKIPBRA1(s)}}}} s++;} }
00104 #define SKIPBRA4(s) { int lev4=0; s++; while(*s) { if(*s=='('){lev4++;} \
00105                     else {if(*s==')'&&--lev4<0){break;} \
00106                     else {if(*s=='['){SKIPBRA1(s)}}} s++;} }
00107 #define SKIPBRA5(s) { int lev5=0; s++; while(*s) { if(*s=='{'){lev5++;} \
00108                     else {if(*s=='}'&&--lev5<0){break;} \
00109                     else {if(*s=='('){SKIPBRA4(s)} \
00110                     else {if(*s=='['){SKIPBRA1(s)}}} s++;} }
00111
00112 /*
00113 #define CYCLE1(a,i) {WORD iX,jX; iX=*a; for(jX=1;jX<i;jX++)a[jX-1]=a[jX]; a[i-1]=iX;}
00114 */
00115 #define CYCLE1(t,a,i) {t iX=*a; WORD jX; for(jX=1;jX<i;jX++)a[jX-1]=a[jX]; a[i-1]=iX;}
00116
00117 #define AddToCB(c,wx) if(c->Pointer>=c->Top) \
00118         {DoubleCbuffer(c-cbuf,c->Pointer,21);} \
00119         *(c->Pointer)++ = wx;
00120
00121 #define EXCHINOUT { FILEHANDLE *ffFi = AR.outfile; \
00122     AR.outfile = AR.infile; AR.infile = ffFi; }
00123 #define BACKINOUT { FILEHANDLE *ffFi = AR.outfile; POSITION posi; \
00124     AR.outfile = AR.infile; AR.infile = ffFi; \
00125     SetEndScratch(AR.infile,&posi); }
00126
00127 #define CopyArg(to,from) { if ( *from > 0 ) { int ica = *from; NCOPY(to,from,ica) } \
00128         else if ( *from <= -FUNCTION ) *to++ = *from++;  \
00129         else { *to++ = *from++; *to++ = *from++; } }
00130
00131 #if ARGHEAD > 2
```

```
00132 #define FILLARG(w) { int i = ARGHEAD-2; while ( --i >= 0 ) *w++ = 0; }
00133 #define COPYARG(w,t) { int i = ARGHEAD-2; while ( --i >= 0 ) *w++ = *t++; }
00134 #define ZEROARG(w) { int i; for ( i = 2; i < ARGHEAD; i++ ) w[i] = 0; }
00135 #else
00136 #define FILLARG(w)
00137 #define COPYARG(w,t)
00138 #define ZEROARG(w)
00139 #endif
00140
00141 #if FUNHEAD > 2
00142 #define FILLFUN(w) { *w++ = 0; FILLFUN3(w) }
00143 #define COPYFUN(w,t) { *w++ = *t++; COPYFUN3(w,t) }
00144 #else
00145 #define FILLFUN(w)
00146 #define COPYFUN(w,t)
00147 #endif
00148
00149 #if FUNHEAD > 3
00150 #define FILLFUN3(w) { int ie = FUNHEAD-3; while ( --ie >= 0 ) *w++ = 0; }
00151 #define COPYFUN3(w,t) { int ie = FUNHEAD-3; while ( --ie >= 0 ) *w++ = *t++; }
00152 #else
00153 #define COPYFUN3(w,t)
00154 #define FILLFUN3(w)
00155 #endif
00156
00157 #if SUBEXPSIZE > 5
00158 #define FILLSUB(w) { int ie = SUBEXPSIZE-5; while ( --ie >= 0 ) *w++ = 0; }
00159 #define COPYSUB(w,ww) { int ie = SUBEXPSIZE-5; while ( --ie >= 0 ) *w++ = *ww++; }
00160 #else
00161 #define FILLSUB(w)
00162 #define COPYSUB(w,ww)
00163 #endif
00164
00165 #if EXPRHEAD > 4
00166 #define FILLEXPR(w) { int ie = EXPRHEAD-4; while ( --ie >= 0 ) *w++ = 0; }
00167 #else
00168 #define FILLEXPR(w)
00169 #endif
00170
00171 #define NEXTARG(x) if(*x>0) x += *x; else if(*x <= -FUNCTION)x++; else x += 2;
00172 #define COPY1ARG(s1,t1) { int ica; if ( (ica=*t1) > 0 ) { NCOPY(s1,t1,ica) } \
00173         else if(*t1<=-FUNCTION){*s1++=*t1++;} else{*s1++=*t1++;*s1++=*t1++;} }
00174
00182 #define ZeroFillRange(w,begin,end) do { \
00183     int tmp_i; \
00184     for ( tmp_i = begin; tmp_i < end; tmp_i++ ) { (w)[tmp_i] = 0; } \
00185 } while (0)
00186
00187 #define TABLESIZE(a,b) (((WORD)sizeof(a))/((WORD)sizeof(b)))
00188 #define WORDDIF(x,y) (WORD)(x-y)
00189 #define wsizeof(a) ((WORD)sizeof(a))
00190 #define VARNAME(type,num) (AC.varnames->namebuffer+type[num].name)
00191 #define DOLLARNAME(type,num) (AC.dollarnames->namebuffer+type[num].name)
00192 #define EXPRNAME(num) (AC.exprnames->namebuffer+Expressions[num].name)
00193
00194 #define PREV(x) prevorder?prevorder:x
00195
00196 #define Terminate(x) do { TerminateImpl(x, __FILE__, __LINE__, __FUNCTION__); } while(0)
00197 #define SETERROR(x) { Terminate(-1); return(-1); }
00198
00199 /* use this macro to avoid the unused parameter warning */
00200 #define DUMMYUSE(x) (void)(x);
00201
00202 #ifdef _FILE_OFFSET_BITS
00203 #if _FILE_OFFSET_BITS==64
00204 /*:[19mar2004 mt]*/
00205
00206 #define ADDPOS(pp,x) (pp).p1 = ((pp).p1+(off_t)(x))
00207 #define SETBASELENGTH(ss,x) (ss).p1 = (off_t)(x)
00208 #define SETBASEPOSITION(pp,x) (pp).p1 = (off_t)(x)
00209 #define ISEQUALPOSINC(pp1,pp2,x) ( (pp1).p1 == ((pp2).p1+(off_t)(x)) )
00210 #define ISGEPOSINC(pp1,pp2,x) ( (pp1).p1 >= ((pp2).p1+(off_t)(x)) )
00211 #define DIVPOS(pp,n) ( (pp).p1/(off_t)(n) )
00212 #define MULPOS(pp,n) (pp).p1 *= (off_t)(n)
00213
00214 #else
00215
00216 #define ADDPOS(pp,x) (pp).p1 = ((pp).p1+(x))
00217 #define SETBASELENGTH(ss,x) (ss).p1 = (x)
00218 #define SETBASEPOSITION(pp,x) (pp).p1 = (x)
00219 #define ISEQUALPOSINC(pp1,pp2,x) ( (pp1).p1 == ((pp2).p1+(LONG)(x)) )
00220 #define ISGEPOSINC(pp1,pp2,x) ( (pp1).p1 >= ((pp2).p1+(LONG)(x)) )
00221 #define DIVPOS(pp,n) ( (pp).p1/(n) )
00222 #define MULPOS(pp,n) (pp).p1 *= (n)
00223 #endif
00224 #else
00225
```

```
00226 #define ADDPOS(pp,x) (pp).p1 = ((pp).p1+(LONG)(x))
00227 #define SETBASELENGTH(ss,x) (ss).p1 = (LONG)(x)
00228 #define SETBASEPOSITION(pp,x) (pp).p1 = (LONG)(x)
00229 #define ISEQUALPOSINC(pp1,pp2,x) ( (pp1).p1 == ((pp2).p1+(LONG)(x)) )
00230 #define ISGEPOSINC(pp1,pp2,x) ( (pp1).p1 >= ((pp2).p1+(LONG)(x)) )
00231 #define DIVPOS(pp,n) ( (pp).p1/(LONG)(n) )
00232 #define MULPOS(pp,n) (pp).p1 *= (LONG)(n)
00233
00234 #endif
00235 #define DIFPOS(ss,pp1,pp2) (ss).p1 = ((pp1).p1-(pp2).p1)
00236 #define DIFBASE(pp1,pp2) ((pp1).p1-(pp2).p1)
00237 #define ADD2POS(pp1,pp2) (pp1).p1 += (pp2).p1
00238 #define PUTZERO(pp) (pp).p1 = 0
00239 #define BASEPOSITION(pp) ((pp).p1)
00240 #define SETSTARTPOS(pp) (pp).p1 = -2
00241 #define NOTSTARTPOS(pp) ( (pp).p1 > -2 )
00242 #define ISMINPOS(pp) ( (pp).p1 == -1 )
00243 #define ISEQUALPOS(pp1,pp2) ( (pp1).p1 == (pp2).p1 )
00244 #define ISNOTEQUALPOS(pp1,pp2) ( (pp1).p1 != (pp2).p1 )
00245 #define ISLESSPOS(pp1,pp2) ( (pp1).p1 < (pp2).p1 )
00246 #define ISGEPOS(pp1,pp2) ( (pp1).p1 >= (pp2).p1 )
00247 #define ISNOTZEROPOS(pp) ( (pp).p1 != 0 )
00248 #define ISZEROPOS(pp) ( (pp).p1 == 0 )
00249 #define ISPOSPOS(pp) ( (pp).p1 > 0 )
00250 #define ISNEGPOS(pp) ( (pp).p1 < 0 )
00251 extern void TELLFILE(int,POSITION *);
00252
00253 #define TOLONG(x) ((LONG)(x))
00254
00255 #define Add2Com(x) { WORD cod[2]; cod[0] = x; cod[1] = 2; AddNtoL(2,cod); }
00256 #define Add3Com(x1,x2) { WORD cod[3]; cod[0] = x1; cod[1] = 3; cod[2] = x2; AddNtoL(3,cod); }
00257 #define Add4Com(x1,x2,x3) { WORD cod[4]; cod[0] = x1; cod[1] = 4; \
00258    cod[2] = x2; cod[3] = x3; AddNtoL(4,cod); }
00259 #define Add5Com(x1,x2,x3,x4) { WORD cod[5]; cod[0] = x1; cod[1] = 5; \
00260    cod[2] = x2; cod[3] = x3; cod[4] = x4; AddNtoL(5,cod); }
00261
00262 /*
00263     The temporary variable ppp is to avoid a compiler warning about strict aliassing
00264 */
00265 #define WantAddPointers(x) while((AT.pWorkPointer+(x))>AR.pWorkSize){WORD ***ppp=&AT.pWorkSpace;\
00266     ExpandBuffer((void **)ppp,&AR.pWorkSize,(int)(sizeof(WORD *)));}
00267 #define WantAddLongs(x) while((AT.lWorkPointer+(x))>AR.lWorkSize){LONG **ppp=&AT.lWorkSpace;\
00268     ExpandBuffer((void **)ppp,&AR.lWorkSize,sizeof(LONG));}
00269 #define WantAddPositions(x) while((AT.posWorkPointer+(x))>AR.posWorkSize){POSITION
    **ppp=&AT.posWorkSpace;\
00270     ExpandBuffer((void **)ppp,&AR.posWorkSize,sizeof(POSITION));}
00271
00272 /* inline in form3.h (or config.h). */
00273 #define FORM_INLINE inline
00274
00275 /*
00276     Macro's for memory management. This can be done by routines, but that
00277     would be slower. Inline routines could do this, but we don't want to
00278     leave this to the friendliness of the compiler(s).
00279     The routines can be found in the file tools.c
00280 */
00281 #define MEMORYMACROS
00282
00283 #ifdef MEMORYMACROS
00284
00285 #define TermMalloc(x) ( (AT.TermMemTop <= 0 ) ? TermMallocAddMemory(BHEAD0),
    AT.TermMemHeap[--AT.TermMemTop]: AT.TermMemHeap[--AT.TermMemTop] )
00286 #define NumberMalloc(x) ( (AT.NumberMemTop <= 0 ) ? NumberMallocAddMemory(BHEAD0),
    AT.NumberMemHeap[--AT.NumberMemTop]: AT.NumberMemHeap[--AT.NumberMemTop] )
00287 #define CacheNumberMalloc(x) ( (AT.CacheNumberMemTop <= 0 ) ? CacheNumberMallocAddMemory(BHEAD0),
    AT.CacheNumberMemHeap[--AT.CacheNumberMemTop]: AT.CacheNumberMemHeap[--AT.CacheNumberMemTop] )
00288 #define TermFree(TermMem,x) AT.TermMemHeap[AT.TermMemTop++] = (WORD *)(TermMem)
00289 #define NumberFree(NumberMem,x) AT.NumberMemHeap[AT.NumberMemTop++] = (UWORD *)(NumberMem)
00290 #define CacheNumberFree(NumberMem,x) AT.CacheNumberMemHeap[AT.CacheNumberMemTop++] = (UWORD
    *)(NumberMem)
00291
00292 #else
00293
00294 #define TermMalloc(x) TermMalloc2(BHEAD (char *)(x))
00295 #define NumberMalloc(x) NumberMalloc2(BHEAD (char *)(x))
00296 #define CacheNumberMalloc(x) CacheNumberMalloc2(BHEAD (char *)(x))
00297 #define TermFree(x,y) TermFree2(BHEAD (WORD *)(x),(char *)(y))
00298 #define NumberFree(x,y) NumberFree2(BHEAD (UWORD *)(x),(char *)(y))
00299 #define CacheNumberFree(x,y) CacheNumberFree2(BHEAD (UWORD *)(x),(char *)(y))
00300
00301 #endif
00302
00303 /*
00304  * Macros for checking nesting levels in the compiler, used as follows:
00305  *
00306  *   AC.IfSumCheck[AC.IfLevel] = NestingChecksum();
00307  *   AC.IfLevel++;
```

```
00308  *
00309  *   AC.IfLevel--;
00310  *   if ( AC.IfSumCheck[AC.IfLevel] != NestingChecksum() ) {
00311  *     MesNesting();
00312  *   }
00313  *
00314  * Note that NestingChecksum() also contains AC.IfLevel and so in this case
00315  * using increment/decrement operators on it in the left-hand side may be
00316  * confusing.
00317  */
00318 #define NestingChecksum() (AC.IfLevel + AC.RepLevel + AC.arglevel + AC.insidelevel + AC.termlevel +
      AC.inexprlevel + AC.dolooplevel +AC.SwitchLevel)
00319 #define MesNesting() MesPrint("&Illegal nesting of if, repeat, argument, inside, term, inexpression
      and do")
00320
00321 #define MarkPolyRatFunDirty(T) {if(*T&&AR.PolyFunType==2){WORD *TP,*TT;TT=T+*T;TT-=ABS(TT[-1]);\
00322 TP=T+1;while(TP<TT){if(*TP==AR.PolyFun){TP[2]|=(DIRTYFLAG|MUSTCLEANPRF);}TP+=TP[1];}}}
00323
00324 /*
00325     Macros for nesting input levels for #$name = ...; assign instructions.
00326     Note that the level should never go below zero.
00327 */
00328 #define PUSHPREASSIGNLEVEL AP.PreAssignLevel++; { GETIDENTITY \
00329     if ( AP.PreAssignLevel >= AP.MaxPreAssignLevel ) { int i; \
00330         LONG *ap = (LONG *)Malloc1(2*AP.MaxPreAssignLevel*sizeof(LONG *),"PreAssignStack"); \
00331         for ( i = 0; i < AP.MaxPreAssignLevel; i++ ) ap[i] = AP.PreAssignStack[i]; \
00332         M_free(AP.PreAssignStack,"PreAssignStack"); \
00333         AP.MaxPreAssignLevel *= 2; AP.PreAssignStack = ap; \
00334     } \
00335     *AT.WorkPointer++ = AP.PreContinuation; AP.PreContinuation = 0; \
00336     AP.PreAssignStack[AP.PreAssignLevel] = AC.iPointer - AC.iBuffer; }
00337
00338 #define POPPREASSIGNLEVEL if ( AP.PreAssignLevel > 0 ) { GETIDENTITY \
00339     AC.iPointer = AC.iBuffer + AP.PreAssignStack[AP.PreAssignLevel--]; \
00340     AP.PreContinuation = *--AT.WorkPointer; \
00341     *AC.iPointer = 0; }
00342
00343 #ifdef WITHFLOAT
00344 /*
00345     The following macro's are needed to avoid problems with the compilers
00346     and gmp.h. For the C++ files we need the Form include files to be inside
00347     and extern C {} environment, but then the structs.h needs gmp.h to
00348     recognise the mpf_t datatype. This causes no end of problems.
00349     Hence we collect the sensitive objects as (void *) and cast them to
00350     something usable with the macro's below. This way the gmp.h file
00351     needs to be included only in a very limited number of .c files.
00352 */
00353 #define mpftab1 ((mpf_t *)(AT.mpf_tab1))
00354 #define mpftab2 ((mpf_t *)(AT.mpf_tab2))
00355 #define mpfaux_ ((mpf_t *)(AT.aux_))
00356 #define aux1 (((mpf_t *)(AT.aux_))[0])
00357 #define aux2 (((mpf_t *)(AT.aux_))[1])
00358 #define aux3 (((mpf_t *)(AT.aux_))[2])
00359 #define aux4 (((mpf_t *)(AT.aux_))[3])
00360 #define aux5 (((mpf_t *)(AT.aux_))[4])
00361 #define auxjm (((mpf_t *)(AT.aux_))[5])
00362 #define auxjjm (((mpf_t *)(AT.aux_))[6])
00363 #define auxsum (((mpf_t *)(AT.aux_))[7])
00364
00365 #define mpfdelta1 (((mpf_t *)(AS.delta_1))[0])
00366
00367 #endif
00368
00369
00370 /*
00371     MesPrint("P-level popped to %d with %d",AP.PreAssignLevel,(WORD)(AC.iPointer - AC.iBuffer));
00372
00373     #] Macro's :
00374     #[ Inline functions :
00375 */
00376
00377 /*
00378  * The following three functions give the unsigned absolute value of a signed
00379  * integer even for the most negative integer. This is beyond the scope of
00380  * the standard abs() function and its family, whose return-values are signed.
00381  * In short, we should not use the unary minus operator with signed numbers
00382  * unless we are sure that there are no integer overflows. Instead, we rely on
00383  * two well-defined operations: (i) signed-to-unsigned conversion and
00384  * (ii) unary minus of unsigned operands.
00385  *
00386  * See also:
00387  *   https://stackoverflow.com/a/4536188   (Unary minus and signed-to-unsigned conversion)
00388  *   https://stackoverflow.com/q/8026694   (C: unary minus operator behavior with unsigned operands)
00389  *   https://stackoverflow.com/q/1610947   (Why does stdlib.h's abs() family of functions return a
      signed value?)
00390  *   https://blog.regehr.org/archives/226  (A Guide to Undefined Behavior in C and C++, Part 2)
00391  */
```

```
00392 static inline unsigned int IntAbs(int x)
00393 {
00394     if ( x >= 0 ) return (unsigned int)x;
00395     return(-((unsigned int)x));
00396 }
00397
00398 static inline UWORD WordAbs(WORD x)
00399 {
00400     if ( x >= 0 ) return (UWORD)x;
00401     return(-((UWORD)x));
00402 }
00403
00404 static inline ULONG LongAbs(LONG x)
00405 {
00406     if ( x >= 0 ) return (ULONG)x;
00407     return(-((ULONG)x));
00408 }
00409
00410 /*
00411  * The following functions provide portable unsigned-to-signed conversions
00412  * (to avoid the implementation-defined behaviour), which is expected to be
00413  * optimized to a no-op.
00414  *
00415  * See also:
00416  *   https://stackoverflow.com/a/13208789  (Efficient unsigned-to-signed cast avoiding
      implementation-defined behavior)
00417  */
00418 static inline int UnsignedToInt(unsigned int x)
00419 {
00420     extern void TerminateImpl(int, const char*, int, const char*) NORETURN;
00421     if ( x <= INT_MAX ) return((int)x);
00422     if ( x >= (unsigned int)INT_MIN )
00423         return((int)(x - (unsigned int)INT_MIN) + INT_MIN);
00424     Terminate(1);
00425     return(0);
00426 }
00427
00428 static inline WORD UWordToWord(UWORD x)
00429 {
00430     extern void TerminateImpl(int, const char*, int, const char*) NORETURN;
00431     if ( x <= WORD_MAX_VALUE ) return((WORD)x);
00432     if ( x >= (UWORD)WORD_MIN_VALUE )
00433         return((WORD)(x - (UWORD)WORD_MIN_VALUE) + WORD_MIN_VALUE);
00434     Terminate(1);
00435     return(0);
00436 }
00437
00438 static inline LONG ULongToLong(ULONG x)
00439 {
00440     extern void TerminateImpl(int, const char*, int, const char*) NORETURN;
00441     if ( x <= LONG_MAX_VALUE ) return((LONG)x);
00442     if ( x >= (ULONG)LONG_MIN_VALUE )
00443         return((LONG)(x - (ULONG)LONG_MIN_VALUE) + LONG_MIN_VALUE);
00444     Terminate(1);
00445     return(0);
00446 }
00447
00448 /*
00449     #] Inline functions :
00450     #[ Thread objects :
00451 */
00452
00459 #ifdef WITHPTHREADS
00460
00461 #define EXTERNLOCK(x) extern pthread_mutex_t x;
00462 #define INILOCK(x)    pthread_mutex_t x = PTHREAD_MUTEX_INITIALIZER;
00463 #define EXTERNRWLOCK(x) extern pthread_rwlock_t x;
00464 #define INIRWLOCK(x)    pthread_rwlock_t x = PTHREAD_RWLOCK_INITIALIZER;
00465 #define INIRECLOCK(x) do { pthread_mutexattr_t attrib; \
00466     pthread_mutexattr_init(&attrib); \
00467     pthread_mutexattr_settype(&attrib, PTHREAD_MUTEX_RECURSIVE); \
00468     pthread_mutex_init(&(x), &attrib); \
00469     pthread_mutexattr_destroy(&attrib); \
00470     } while(0)
00471 #ifdef DEBUGGINGLOCKS
00472 #include <asm/errno.h>
00473 #define LOCK(x)       while ( pthread_mutex_trylock(&(x)) == EBUSY ) {}
00474 #define RWLOCKR(x)      while ( pthread_rwlock_tryrdlock(&(x)) == EBUSY ) {}
00475 #define RWLOCKW(x)      while ( pthread_rwlock_trywrlock(&(x)) == EBUSY ) {}
00476 #else
00477 #define LOCK(x)       pthread_mutex_lock(&(x))
00478 #define RWLOCKR(x)      pthread_rwlock_rdlock(&(x))
00479 #define RWLOCKW(x)      pthread_rwlock_wrlock(&(x))
00480 #endif
00481 #define UNLOCK(x)     pthread_mutex_unlock(&(x))
00482 #define UNRWLOCK(x)     pthread_rwlock_unlock(&(x))
00483 #define MLOCK(x)      LOCK(x)
```

```
00484 #define MUNLOCK(x)    UNLOCK(x)
00485
00486 #define GETBIDENTITY
00487 #define GETIDENTITY   int identity = WhoAmI(); ALLPRIVATES *B = AB[identity];
00488 #else
00489
00490 #define EXTERNLOCK(x)
00491 #define INILOCK(x)
00492 #define LOCK(x)
00493 #define UNLOCK(x)
00494 #define EXTERNRWLOCK(x)
00495 #define INIRWLOCK(x)
00496 #define RWLOCKR(x)
00497 #define RWLOCKW(x)
00498 #define UNRWLOCK(x)
00499 #ifdef WITHMPI
00500 #define MLOCK(x)      do { if ( PF.me != MASTER ) PF_MLock(); } while (0)
00501 #define MUNLOCK(x)    do { if ( PF.me != MASTER ) PF_MUnlock(); } while (0)
00502 #else
00503 #define MLOCK(x)
00504 #define MUNLOCK(x)
00505 #endif
00506 #define GETIDENTITY
00507 #define GETBIDENTITY
00508
00509 #endif
00510
00511 /*
00512     #] Thread objects :
00513     #[ Declarations :
00514 */
00515
00516 #ifdef TERMMALLOCDEBUG
00517 extern WORD **DebugHeap1, **DebugHeap2;
00518 #endif
00519
00524 extern void   StartVariables(void);
00525 extern void   setSignalHandlers(void);
00526 extern UBYTE *CodeToLine(WORD,UBYTE *);
00527 extern UBYTE *AddArrayIndex(WORD ,UBYTE *);
00528 extern INDEXENTRY *FindInIndex(WORD,FILEDATA *,WORD,WORD);
00529 extern INDEXENTRY *NextFileIndex(POSITION *);
00530 extern WORD  *PasteTerm(PHEAD WORD,WORD *,WORD *,WORD,WORD);
00531 extern UBYTE *StrCopy(UBYTE *,UBYTE *);
00532 extern UBYTE *WrtPower(UBYTE *,WORD);
00533 extern int    AccumGCD(PHEAD UWORD *,WORD *,UWORD *,WORD);
00534 extern void   AddArgs(PHEAD WORD *,WORD *,WORD *);
00535 extern int    AddCoef(PHEAD WORD **,WORD **);
00536 extern int    AddLong(UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
00537 extern int    AddPLon(UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
00538 extern int    AddPoly(PHEAD WORD **,WORD **);
00539 extern int    AddRat(PHEAD UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
00540 extern void   AddToLine(UBYTE *);
00541 extern int    AddWild(PHEAD WORD,WORD,WORD);
00542 extern int    BigLong(UWORD *,WORD,UWORD *,WORD);
00543 extern int    BinomGen(PHEAD WORD *,WORD,WORD **,WORD,WORD,WORD,WORD,WORD,UWORD *,WORD);
00544 extern int    CheckWild(PHEAD WORD,WORD,WORD,WORD *);
00545 extern int    Chisholm(PHEAD WORD *,WORD);
00546 extern int    CleanExpr(WORD);
00547 extern void   CleanUp(WORD);
00548 extern void   ClearWild(PHEAD0);
00549 extern int    CompareFunctions(WORD *,WORD *);
00550 extern int    Commute(WORD *,WORD *);
00551 extern WORD   DetCommu(WORD *);
00552 extern WORD   DoesCommu(WORD *);
00553 extern int    CompArg(WORD *,WORD *);
00554 extern WORD   CompCoef(WORD *, WORD *);
00555 extern int    CompGroup(PHEAD WORD,WORD **,WORD *,WORD *,WORD);
00556 extern WORD   Compare1(PHEAD WORD *,WORD *,WORD);
00557 extern WORD   CountDo(WORD *,WORD *);
00558 extern WORD   CountFun(WORD *,WORD *);
00559 extern WORD   DimensionSubterm(WORD *);
00560 extern WORD   DimensionTerm(WORD *);
00561 extern WORD   DimensionExpression(PHEAD WORD *);
00562 extern int    Deferred(PHEAD WORD *,WORD);
00563 extern int    DeleteStore(WORD);
00564 extern WORD   DetCurDum(PHEAD WORD *);
00565 extern void   DetVars(WORD *,WORD);
00566 extern int    Distribute(DISTRIBUTE *,WORD);
00567 extern int    DivLong(UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *,UWORD *,WORD *);
00568 extern int    DivRat(PHEAD UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
00569 extern int    Divvy(PHEAD UWORD *,WORD *,UWORD *,WORD);
00570 extern int    DoDelta(WORD *);
00571 extern int    DoDelta3(PHEAD WORD *,WORD);
00572 extern int    TestPartitions(WORD *, PARTI *);
00573 extern int    DoPartitions(PHEAD WORD *,WORD);
00574 extern int    CoCanonicalize(UBYTE *);
```

```
00575 extern int    DoCanonicalize(PHEAD WORD *, WORD *);
00576 extern int    GenDiagrams(PHEAD WORD *,WORD);
00577 extern int    DoTopologyCanonicalize(PHEAD WORD *,WORD,WORD,WORD *);
00578 extern int    DoShattering(PHEAD WORD *,WORD *,WORD *,WORD);
00579 extern int    DoTableExpansion(WORD *,WORD);
00580 extern int    DoDistrib(PHEAD WORD *,WORD);
00581 extern int    DoShuffle(WORD *,WORD,WORD,WORD);
00582 extern int    DoPermutations(PHEAD WORD *,WORD);
00583 extern int    Shuffle(WORD *, WORD *, WORD *);
00584 extern int    FinishShuffle(WORD *);
00585 extern int    DoStuffle(WORD *,WORD,WORD,WORD);
00586 extern int    Stuffle(WORD *, WORD *, WORD *);
00587 extern int    FinishStuffle(WORD *);
00588 extern WORD  *StuffRootAdd(WORD *, WORD *, WORD *);
00589 extern int    TestUse(WORD *,WORD);
00590 extern DBASE *FindTB(UBYTE *);
00591 extern int    CheckTableDeclarations(DBASE *);
00592 extern void   Apply(WORD *,WORD);
00593 extern int    ApplyExec(WORD *,int,WORD);
00594 extern void   ApplyReset(WORD);
00595 extern void   TableReset(void);
00596 extern void   ReWorkT(WORD *,WORD *,WORD);
00597 extern WORD   GetIfDollarNum(WORD *, WORD *);
00598 extern int    FindVar(WORD *,WORD *);
00599 extern int    DoIfStatement(PHEAD WORD *,WORD *);
00600 extern int    DoOnePow(PHEAD WORD *,WORD,WORD,WORD *,WORD *,WORD,WORD *);
00601 extern void   DoRevert(WORD *,WORD *);
00602 extern int    DoSumF1(PHEAD WORD *,WORD *,WORD,WORD);
00603 extern int    DoSumF2(PHEAD WORD *,WORD *,WORD,WORD);
00604 extern int    DoTheta(PHEAD WORD *);
00605 extern LONG   EndSort(PHEAD WORD *,int);
00606 extern int    EntVar(WORD,UBYTE *,WORD,WORD,WORD,WORD);
00607 extern int    EpfCon(PHEAD WORD *,WORD *,WORD,WORD);
00608 extern int    EpfFind(PHEAD WORD *,WORD *);
00609 extern WORD   EpfGen(WORD,WORD *,WORD *,WORD *,WORD);
00610 extern int    EqualArg(WORD *,WORD,WORD);
00611 extern int    Factorial(PHEAD WORD,UWORD *,WORD *);
00612 extern int    Bernoulli(WORD,UWORD *,WORD *);
00613 extern int    FactorIn(PHEAD WORD *,WORD);
00614 extern int    FactorInExpr(PHEAD WORD *,WORD);
00615 extern int    FindAll(PHEAD WORD *,WORD *,WORD,WORD *);
00616 extern WORD   FindMulti(PHEAD WORD *,WORD *);
00617 extern int    FindOnce(PHEAD WORD *,WORD *);
00618 extern int    FindOnly(PHEAD WORD *,WORD *);
00619 extern int    FindRest(PHEAD WORD *,WORD *);
00620 extern void   FindSpecial(WORD *);
00621 extern WORD   FindrNumber(WORD,VARRENUM *);
00622 extern void   FiniLine(void);
00623 extern int    FiniTerm(PHEAD WORD *,WORD *,WORD *,WORD,WORD);
00624 extern int    FlushOut(POSITION *,FILEHANDLE *,int);
00625 extern void   FunLevel(PHEAD WORD *);
00626 extern void   AdjustRenumScratch(PHEAD0);
00627 extern void   GarbHand(void);
00628 extern int    GcdLong(PHEAD UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
00629 extern int    LcmLong(PHEAD UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
00630 extern void   GCD(UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
00631 extern ULONG  GCD2(ULONG,ULONG);
00632 extern int    Generator(PHEAD WORD *,WORD);
00633 extern int    GetBinom(UWORD *,WORD *,WORD,WORD);
00634 extern WORD   GetFromStore(WORD *,POSITION *,RENUMBER,WORD *,WORD);
00635 extern int    GetLong(UBYTE *,UWORD *,WORD *);
00636 extern WORD   GetMoreTerms(WORD *);
00637 extern int    GetMoreFromMem(WORD *,WORD **);
00638 extern WORD   GetOneTerm(PHEAD WORD *,FILEHANDLE *,POSITION *,int);
00639 extern RENUMBER GetTable(WORD,POSITION *,WORD);
00640 extern WORD   GetTerm(PHEAD WORD *);
00641 extern int    Glue(PHEAD WORD *,WORD *,WORD *,WORD);
00642 extern int    InFunction(PHEAD WORD *,WORD *);
00643 extern void   IniLine(WORD);
00644 extern void   IniVars(void);
00645 extern int    InsertTerm(PHEAD WORD *,WORD,WORD,WORD *,WORD *,WORD);
00646 extern void   LongToLine(UWORD *,WORD);
00647 extern int    MakeDirty(WORD *,WORD *,WORD);
00648 extern void   MarkDirty(WORD *,WORD);
00649 extern void   PolyFunDirty(PHEAD WORD *);
00650 extern void   PolyFunClean(PHEAD WORD *);
00651 extern int    MakeModTable(void);
00652 extern int    MatchE(PHEAD WORD *,WORD *,WORD *,WORD);
00653 extern int    MatchCy(PHEAD WORD *,WORD *,WORD *,WORD);
00654 extern int    FunMatchCy(PHEAD WORD *,WORD *,WORD *,WORD);
00655 extern int    FunMatchSy(PHEAD WORD *,WORD *,WORD *,WORD);
00656 extern int    MatchArgument(PHEAD WORD *,WORD *);
00657 extern int    MatchFunction(PHEAD WORD *,WORD *,WORD *);
00658 extern int    MergePatches(WORD);
00659 extern int    MesCerr(char *, UBYTE *);
00660 extern int    MesComp(char *, UBYTE *, UBYTE *);
00661 extern int    Modulus(WORD *);
```

```
00662 extern void   MoveDummies(PHEAD WORD *,WORD);
00663 extern int    MulLong(UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
00664 extern int    MulRat(PHEAD UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
00665 extern int    Mully(PHEAD UWORD *,WORD *,UWORD *,WORD);
00666 extern int    MultDo(PHEAD WORD *,WORD *);
00667 extern int    NewSort(PHEAD0);
00668 extern int    ExtraSymbol(WORD,WORD,WORD,WORD *,WORD *);
00669 extern int    Normalize(PHEAD WORD *);
00670 extern int    BracketNormalize(PHEAD WORD *);
00671 extern void   DropCoefficient(PHEAD WORD *);
00672 extern void   DropSymbols(PHEAD WORD *);
00673 extern int    PutInside(PHEAD WORD *, WORD *);
00674 extern void   OpenTemp(void);
00675 extern void   Pack(UWORD *,WORD *,UWORD *,WORD );
00676 extern LONG   PasteFile(PHEAD WORD,WORD *,POSITION *,WORD **,RENUMBER,WORD *,WORD);
00677 extern int    Permute(PERM *,WORD);
00678 extern int    PermuteP(PERMP *,WORD);
00679 extern int    PolyFunMul(PHEAD WORD *);
00680 extern int    PopVariables(void);
00681 extern int    PrepPoly(PHEAD WORD *,WORD);
00682 extern int    Processor(void);
00683 extern int    Product(UWORD *,WORD *,WORD);
00684 extern void   PrtLong(UWORD *,WORD,UBYTE *);
00685 extern void   PrtTerms(void);
00686 extern void   PrintDeprecation(const char *,const char *);
00687 extern void   PrintFeatureList(void);
00688 extern void   PrintRunningTime(void);
00689 extern LONG   GetRunningTime(void);
00690 extern int    PutBracket(PHEAD WORD *);
00691 extern LONG   PutIn(FILEHANDLE *,POSITION *,WORD *,WORD **,int);
00692 extern int    PutInStore(INDEXENTRY *,WORD);
00693 extern WORD   PutOut(PHEAD WORD *,POSITION *,FILEHANDLE *,WORD);
00694 extern UWORD  Quotient(UWORD *,WORD *,WORD);
00695 extern int    RaisPow(PHEAD UWORD *,WORD *,UWORD);
00696 extern void   RaisPowCached (PHEAD WORD, WORD, UWORD **, WORD *);
00697 extern WORD   RaisPowMod (WORD, WORD, WORD);
00698 extern int    NormalModulus(UWORD *,WORD *);
00699 extern int    MakeInverses(void);
00700 extern int    GetModInverses(WORD,WORD,WORD *,WORD *);
00701 extern int    GetLongModInverses(PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *, UWORD *, WORD
      *);
00702 extern void   RatToLine(UWORD *,WORD);
00703 extern int    RatioFind(PHEAD WORD *,WORD *);
00704 extern int    RatioGen(PHEAD WORD *,WORD *,WORD,WORD);
00705 extern WORD   ReNumber(PHEAD WORD *);
00706 extern WORD   ReadSnum(UBYTE **);
00707 extern WORD   Remain10(UWORD *,WORD *);
00708 extern WORD   Remain4(UWORD *,WORD *);
00709 extern int    ResetScratch(void);
00710 extern int    ResolveSet(PHEAD WORD *,WORD *,WORD *);
00711 extern int    RevertScratch(void);
00712 extern int    ScanFunctions(PHEAD WORD *,WORD *,WORD);
00713 extern void   SeekScratch(FILEHANDLE *,POSITION *);
00714 extern void   SetEndScratch(FILEHANDLE *,POSITION *);
00715 extern void   SetEndHScratch(FILEHANDLE *,POSITION *);
00716 extern int    SetFileIndex(void);
00717 extern int    Sflush(FILEHANDLE *);
00718 extern int    Simplify(PHEAD UWORD *,WORD *,UWORD *,WORD *);
00719 extern int    SortWild(WORD *,WORD);
00720 extern FILE  *LocateBase(char **,char **,char *);
00721 extern LONG   SplitMerge(PHEAD WORD **,LONG);
00722 extern int    StoreTerm(PHEAD WORD *);
00723 extern void   SubPLon(UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
00724 extern void   Substitute(PHEAD WORD *,WORD *,WORD);
00725 extern int    SymFind(PHEAD WORD *,WORD *);
00726 extern int    SymGen(PHEAD WORD *,WORD *,WORD,WORD);
00727 extern WORD   Symmetrize(PHEAD WORD *,WORD *,WORD,WORD,WORD);
00728 extern int    FullSymmetrize(PHEAD WORD *,int);
00729 extern int    TakeModulus(UWORD *,WORD *,UWORD *,WORD,WORD);
00730 extern int    TakeNormalModulus(UWORD *,WORD *,UWORD *,WORD,WORD);
00731 extern void   TalToLine(UWORD);
00732 extern int    TenVec(PHEAD WORD *,WORD *,WORD,WORD);
00733 extern int    TenVecFind(PHEAD WORD *,WORD *);
00734 extern int    TermRenumber(WORD *,RENUMBER,WORD);
00735 extern void   TestDrop(void);
00736 extern void   PutInVflags(WORD);
00737 extern int    TestMatch(PHEAD WORD *,WORD *);
00738 extern WORD   TestSub(PHEAD WORD *,WORD);
00739 extern LONG   TimeCPU(WORD);
00740 extern LONG   TimeChildren(WORD);
00741 extern LONG   TimeWallClock(WORD);
00742 extern LONG   Timer(int);
00743 extern int    GetTimerInfo(LONG **,LONG **);
00744 extern void   WriteTimerInfo(LONG *,LONG *);
00745 extern LONG   GetWorkerTimes(void);
00746 extern int    ToStorage(EXPRESSIONS,POSITION *);
00747 extern void   TokenToLine(UBYTE *);
```

```
00748 extern int    Trace4(PHEAD WORD *,WORD *,WORD,WORD);
00749 extern int    Trace4Gen(PHEAD TRACES *,WORD);
00750 extern int    Trace4no(WORD,WORD *,TRACES *);
00751 extern int    TraceFind(PHEAD WORD *,WORD *);
00752 extern int    TraceN(PHEAD WORD *,WORD *,WORD,WORD);
00753 extern int    TraceNgen(PHEAD TRACES *,WORD);
00754 extern WORD   TraceNno(WORD,WORD *,TRACES *);
00755 extern int    Traces(PHEAD WORD *,WORD *,WORD,WORD);
00756 extern WORD   Trick(WORD *,TRACES *);
00757 extern int    TryDo(PHEAD WORD *,WORD *,WORD);
00758 extern void   UnPack(UWORD *,WORD,WORD *,WORD *);
00759 extern int    VarStore(UBYTE *,WORD,WORD,WORD);
00760 extern WORD   WildFill(PHEAD WORD *,WORD *,WORD *);
00761 extern int    WriteAll(void);
00762 extern int    WriteOne(UBYTE *,int,int,WORD);
00763 extern void   WriteArgument(WORD *);
00764 extern int    WriteExpression(WORD *,LONG);
00765 extern int    WriteInnerTerm(WORD *,WORD);
00766 extern void   WriteLists(void);
00767 extern void   WriteSetup(void);
00768 extern void   WriteStats(POSITION *,WORD,WORD);
00769 extern int    WriteSubTerm(WORD *,WORD);
00770 extern int    WriteTerm(WORD *,WORD *,WORD,WORD,WORD);
00771 extern int    execarg(PHEAD WORD *,WORD);
00772 extern int    execterm(PHEAD WORD *,WORD);
00773 extern void   SpecialCleanup(PHEAD0);
00774 extern void   SetMods(void);
00775 extern void   UnSetMods(void);
00776
00777 /*--------------------------------------------------------------------*/
00778
00779 extern int    DoExecute(WORD,WORD);
00780 extern void   SetScratch(FILEHANDLE *,POSITION *);
00781 extern void   Warning(char *);
00782 extern void   HighWarning(char *);
00783 extern int    SpareTable(TABLES);
00784
00785 extern UBYTE *strDup1(UBYTE *,char *);
00786 extern void  *Malloc1(LONG,const char *);
00787 extern int    DoTail(int,UBYTE **);
00788 extern int    OpenInput(void);
00789 extern int    PutPreVar(UBYTE *,UBYTE *,UBYTE *,int);
00790 extern void   Error0(char *);
00791 extern void   Error1(char *,UBYTE *);
00792 extern void   Error2(char *,char *,UBYTE *);
00793 extern UBYTE  ReadFromStream(STREAM *);
00794 extern UBYTE  GetFromStream(STREAM *);
00795 extern UBYTE  LookInStream(STREAM *);
00796 extern STREAM *OpenStream(UBYTE *,int,int,int);
00797 extern int    LocateFile(UBYTE **,int);
00798 extern STREAM *CloseStream(STREAM *);
00799 extern void   PositionStream(STREAM *,LONG);
00800 extern int    ReverseStatements(STREAM *);
00801 extern int    ProcessOption(UBYTE *,UBYTE *,int);
00802 extern int    DoSetups(void);
00803 extern void   TerminateImpl(int, const char *,int, const char *) NORETURN;
00804 extern NAMENODE *GetNode(NAMETREE *,UBYTE *);
00805 extern int    AddName(NAMETREE *,UBYTE *,WORD,WORD,int *);
00806 extern int    GetName(NAMETREE *,UBYTE *,WORD *,int);
00807 extern UBYTE  *GetFunction(UBYTE *,WORD *);
00808 extern UBYTE  *GetNumber(UBYTE *,WORD *);
00809 extern int    GetLastExprName(UBYTE *,WORD *);
00810 extern int    GetAutoName(UBYTE *,WORD *);
00811 extern int    GetVar(UBYTE *,WORD *,WORD *,int,int);
00812 extern int    MakeDubious(NAMETREE *,UBYTE *,WORD *);
00813 extern int    GetONname(NAMETREE *,UBYTE *,WORD *,int);
00814 extern void   DumpTree(NAMETREE *);
00815 extern void   DumpNode(NAMETREE *,WORD,WORD);
00816 extern void   LinkTree(NAMETREE *,WORD,WORD);
00817 extern void   CopyTree(NAMETREE *,NAMETREE *,WORD,WORD);
00818 extern int    CompactifyTree(NAMETREE *,WORD);
00819 extern NAMETREE *MakeNameTree(void);
00820 extern void   FreeNameTree(NAMETREE *);
00821 extern int    AddExpression(UBYTE *,int,int);
00822 extern int    AddSymbol(UBYTE *,int,int,int,int);
00823 extern int    AddDollar(UBYTE *,WORD,WORD *,LONG);
00824 extern int    ReplaceDollar(WORD,WORD,WORD *,LONG);
00825 extern int    DollarRaiseLow(UBYTE *,LONG);
00826 extern int    AddVector(UBYTE *,int,int);
00827 extern int    AddDubious(UBYTE *);
00828 extern int    AddIndex(UBYTE *,int,int);
00829 extern UBYTE *DoDimension(UBYTE *,int *,int *);
00830 extern int    AddFunction(UBYTE *,int,int,int,int,int,int,int);
00831 extern int    CoCommuteInSet(UBYTE *);
00832 extern int    CoFunction(UBYTE *,int,int);
00833 extern int    TestName(UBYTE *);
00834 extern int    AddSet(UBYTE *,WORD);
```

```
00835 extern int    DoElements(UBYTE *,SETS,UBYTE *);
00836 extern int    DoTempSet(UBYTE *,UBYTE *);
00837 extern int    NameConflict(int,UBYTE *);
00838 extern int    OpenFile(char *);
00839 extern int    OpenAddFile(char *);
00840 extern int    ReOpenFile(char *);
00841 extern int    CreateFile(char *);
00842 extern int    CreateLogFile(char *);
00843 extern void   CloseFile(int);
00844 extern int    CopyFile(char *, char *);
00845 extern int    CreateHandle(void);
00846 extern LONG   ReadFile(int,UBYTE *,LONG);
00847 extern LONG   ReadPosFile(PHEAD FILEHANDLE *,UBYTE *,LONG,POSITION *);
00848 extern LONG   WriteFileToFile(int,UBYTE *,LONG);
00849 extern void   SeekFile(int,POSITION *,int);
00850 extern LONG   TellFile(int);
00851 extern void   FlushFile(int);
00852 extern int    GetPosFile(int,fpos_t *);
00853 extern int    SetPosFile(int,fpos_t *);
00854 extern void   SynchFile(int);
00855 extern void   TruncateFile(int);
00856 extern int    GetChannel(char *,int);
00857 extern int    GetAppendChannel(char *);
00858 extern int    CloseChannel(char *);
00859 extern void   inictable(void);
00860 extern KEYWORD *findcommand(UBYTE *);
00861 extern int    inicbufs(void);
00862 extern void   StartFiles(void);
00863 extern UBYTE *MakeDate(void);
00864 extern void   PreProcessor(void);
00865 extern void  *FromList(LIST *);
00866 extern void  *From0List(LIST *);
00867 extern void  *FromVarList(LIST *);
00868 extern int    DoubleList(void ***,int *,int,char *);
00869 extern int    DoubleLList(void ***,LONG *,int,char *);
00870 extern void   DoubleBuffer(void **,void **,int,char *);
00871 extern void   ExpandBuffer(void **,LONG *,int);
00872 extern LONG   iexp(LONG,int);
00873 extern int    IsLikeVector(WORD *);
00874 extern int    AreArgsEqual(WORD *,WORD *);
00875 extern int    CompareArgs(WORD *,WORD *);
00876 extern UBYTE *SkipField(UBYTE *,int);
00877 extern int    StrCmp(UBYTE *,UBYTE *);
00878 extern int    StrICmp(UBYTE *,UBYTE *);
00879 extern int    StrHICmp(UBYTE *,UBYTE *);
00880 extern int    StrICont(UBYTE *,UBYTE *);
00881 extern int    CmpArray(WORD *,WORD *,WORD);
00882 extern int    ConWord(UBYTE *,UBYTE *);
00883 extern int    StrLen(UBYTE *);
00884 extern UBYTE *GetPreVar(UBYTE *,int);
00885 extern void   ToGeneral(WORD *,WORD *,WORD);
00886 extern WORD   ToPolyFunGeneral(PHEAD WORD *);
00887 extern int    ToFast(WORD *,WORD *);
00888 extern SETUPPARAMETERS *GetSetupPar(UBYTE *);
00889 extern int    RecalcSetups(void);
00890 extern int    AllocSetups(void);
00891 extern SORTING *AllocSort(LONG,LONG,LONG,LONG,int,int,LONG,int);
00892 extern void   AllocSortFileName(SORTING *);
00893 extern UBYTE *LoadInputFile(UBYTE *,int);
00894 extern UBYTE  GetInput(void);
00895 extern void   ClearPushback(void);
00896 extern UBYTE  GetChar(int);
00897 extern void   CharOut(UBYTE);
00898 extern void   UnsetAllowDelay(void);
00899 extern void   PopPreVars(int);
00900 extern void   IniModule(int);
00901 extern void   IniSpecialModule(int);
00902 extern int    ModuleInstruction(int *,int *);
00903 extern int    PreProInstruction(void);
00904 extern int    LoadInstruction(int);
00905 extern int    LoadStatement(int);
00906 extern KEYWORD *FindKeyWord(UBYTE *,KEYWORD *,int);
00907 extern KEYWORD *FindInKeyWord(UBYTE *,KEYWORD *,int);
00908 extern int    DoDefine(UBYTE *);
00909 extern int    DoRedefine(UBYTE *);
00910 extern int    TheDefine(UBYTE *,int);
00911 extern int    TheUndefine(UBYTE *);
00912 extern int    ClearMacro(UBYTE *);
00913 extern int    DoUndefine(UBYTE *);
00914 extern int    DoInclude(UBYTE *);
00915 extern int    DoReverseInclude(UBYTE *);
00916 extern int    Include(UBYTE *,int);
00917 /*[14apr2004 mt]:*/
00918 extern int    DoExternal(UBYTE *);
00919 extern int    DoToExternal(UBYTE *);
00920 extern int    DoFromExternal(UBYTE *);
00921 extern int    DoPrompt(UBYTE *);
```

```
00922 extern int    DoSetExternal(UBYTE *);
00923 /*[10may2006 mt]:*/
00924 extern int    DoSetExternalAttr(UBYTE *);
00925 /*:[10may2006 mt]*/
00926 extern int    DoRmExternal(UBYTE *);
00927 /*:[14apr2004 mt]*/
00928 extern int    DoFactDollar(UBYTE *);
00929 extern WORD   GetDollarNumber(UBYTE **,DOLLARS);
00930 extern int    DoSetRandom(UBYTE *);
00931 extern int    DoOptimize(UBYTE *);
00932 extern int    DoClearOptimize(UBYTE *);
00933 extern int    DoSkipExtraSymbols(UBYTE *);
00934 extern int    DoTimeOutAfter(UBYTE *);
00935 extern int    DoMessage(UBYTE *);
00936 extern int    DoPreOut(UBYTE *);
00937 extern int    DoPreAppend(UBYTE *);
00938 extern int    DoPreCreate(UBYTE *);
00939 extern int    DoPreAssign(UBYTE *);
00940 extern int    DoPreBreak(UBYTE *);
00941 extern int    DoPreDefault(UBYTE *);
00942 extern int    DoPreSwitch(UBYTE *);
00943 extern int    DoPreEndSwitch(UBYTE *);
00944 extern int    DoPreCase(UBYTE *);
00945 extern int    DoPreShow(UBYTE *);
00946 extern int    DoPreExchange(UBYTE *);
00947 extern int    DoSystem(UBYTE *);
00948 extern int    DoPipe(UBYTE *);
00949 extern void   StartPrepro(void);
00950 extern int    DoIfdef(UBYTE *,int);
00951 extern int    DoIfydef(UBYTE *);
00952 extern int    DoIfndef(UBYTE *);
00953 extern int    DoElse(UBYTE *);
00954 extern int    DoElseif(UBYTE *);
00955 extern int    DoEndif(UBYTE *);
00956 extern int    DoTerminate(UBYTE *);
00957 extern int    DoIf(UBYTE *);
00958 extern int    DoCall(UBYTE *);
00959 extern int    DoDebug(UBYTE *);
00960 extern int    DoContinueDo(UBYTE *);
00961 extern int    DoDo(UBYTE *);
00962 extern int    DoBreakDo(UBYTE *);
00963 extern int    DoEnddo(UBYTE *);
00964 extern int    DoEndprocedure(UBYTE *);
00965 extern int    DoInside(UBYTE *);
00966 extern int    DoEndInside(UBYTE *);
00967 extern int    DoProcedure(UBYTE *);
00968 extern int    DoPrePrintTimes(UBYTE *);
00969 extern int    DoPreWrite(UBYTE *);
00970 extern int    DoPreClose(UBYTE *);
00971 extern int    DoPreRemove(UBYTE *);
00972 extern int    DoPreSortReallocate(UBYTE *);
00973 extern int    DoCommentChar(UBYTE *);
00974 extern int    DoPrcExtension(UBYTE *);
00975 extern int    DoPreReset(UBYTE *);
00976 extern void   WriteString(int,UBYTE *,int);
00977 extern void   WriteUnfinString(int,UBYTE *,int);
00978 extern UBYTE *AddToString(UBYTE *,UBYTE *,int);
00979 extern UBYTE *PreCalc(void);
00980 extern UBYTE *PreEval(UBYTE *,LONG *);
00981 extern void   NumToStr(UBYTE *,LONG);
00982 extern int    PreCmp(int,int,UBYTE *,int,int,UBYTE *,int);
00983 extern int    PreEq(int,int,UBYTE *,int,int,UBYTE *,int);
00984 extern UBYTE *pParseObject(UBYTE *,int *,LONG *);
00985 extern UBYTE *PreIfEval(UBYTE *,int *);
00986 extern int    EvalPreIf(UBYTE *);
00987 extern int    PreLoad(PRELOAD *,UBYTE *,UBYTE *,int,char *);
00988 extern int    PreSkip(UBYTE *,UBYTE *,int);
00989 extern UBYTE *EndOfToken(UBYTE *);
00990 extern void   SetSpecialMode(int,int);
00991 extern void   MakeGlobal(void);
00992 extern int    ExecModule(int);
00993 extern int    ExecStore(void);
00994 extern void   FullCleanUp(void);
00995 extern int    DoExecStatement(void);
00996 extern int    DoPipeStatement(void);
00997 extern int    DoPolyfun(UBYTE *);
00998 extern int    DoPolyratfun(UBYTE *);
00999 extern int    CompileStatement(UBYTE *);
01000 extern UBYTE *ToToken(UBYTE *);
01001 extern int    GetDollar(UBYTE *);
01002 extern void   MesWork(void) NORETURN;
01003 extern int    MesPrint(const char *,...);
01004 extern int    MesCall(char *);
01005 extern UBYTE *NumCopy(WORD,UBYTE *);
01006 extern char  *LongCopy(LONG,char *);
01007 extern char  *LongLongCopy(off_t *,char *);
01008 extern void   ReserveTempFiles(int);
```

```
01009 extern void   PrintTerm(WORD *,char *);
01010 extern void   PrintTermC(WORD *,char *);
01011 extern void   PrintSubTerm(WORD *,char *);
01012 extern void   PrintWords(WORD *,LONG);
01013 extern void   PrintSeq(WORD *,char *);
01014 extern int    ExpandTripleDots(int);
01015 extern LONG   ComPress(WORD **,LONG *);
01016 extern void   StageSort(FILEHANDLE *);
01017
01018 #define M_alloc(x)      malloc((size_t)(x))
01019
01020 extern void   M_free(void *,const char *);
01021 extern void   ClearWildcardNames(void);
01022 extern int    AddWildcardName(UBYTE *);
01023 extern int    GetWildcardName(UBYTE *);
01024 extern void   Globalize(int);
01025 extern void   ResetVariables(int);
01026 extern void   AddToPreTypes(int);
01027 extern void   MessPreNesting(int);
01028 extern LONG   GetStreamPosition(STREAM *);
01029 extern WORD  *DoubleCbuffer(int,WORD *,int);
01030 extern WORD  *AddLHS(int);
01031 extern WORD  *AddRHS(int,int);
01032 extern int    AddNtoL(int,WORD *);
01033 extern int    AddNtoC(int,int,WORD *,int);
01034 extern void   DoubleIfBuffers(void);
01035 extern STREAM *CreateStream(UBYTE *);
01036
01037 extern int    setonoff(UBYTE *,int *,int,int);
01038 extern int    DoPrint(UBYTE *,int);
01039 extern int    SetExpr(UBYTE *,int,int);
01040 extern void   AddToCom(int,WORD *);
01041 extern int    Add2ComStrings(int,WORD *,UBYTE *,UBYTE *);
01042 extern int    DoSymmetrize(UBYTE *,int);
01043 extern int    DoArgument(UBYTE *,int);
01044 extern int    ArgFactorize(PHEAD WORD *,WORD *);
01045 extern WORD  *TakeArgContent(PHEAD WORD *, WORD *);
01046 extern WORD  *MakeInteger(PHEAD WORD *,WORD *,WORD *);
01047 extern WORD  *MakeMod(PHEAD WORD *,WORD *,WORD *);
01048 extern WORD   FindArg(PHEAD WORD *);
01049 extern int    InsertArg(PHEAD WORD *,WORD *,int);
01050 extern int    CleanupArgCache(PHEAD WORD);
01051 extern int    ArgSymbolMerge(WORD *, WORD *);
01052 extern int    ArgDotproductMerge(WORD *, WORD *);
01053 extern void   SortWeights(LONG *,LONG *,WORD);
01054 extern int    DoBrackets(UBYTE *,int);
01055 extern int    DoPutInside(UBYTE *,int);
01056 extern WORD  *CountComp(UBYTE *,WORD *);
01057 extern int    CoAntiBracket(UBYTE *);
01058 extern int    CoAntiSymmetrize(UBYTE *);
01059 extern int    DoArgPlode(UBYTE *,int);
01060 extern int    CoArgExplode(UBYTE *);
01061 extern int    CoArgImplode(UBYTE *);
01062 extern int    CoArgument(UBYTE *);
01063 extern int    CoInside(UBYTE *);
01064 extern int    ExecInside(UBYTE *);
01065 extern int    CoInExpression(UBYTE *);
01066 extern int    CoInParallel(UBYTE *);
01067 extern int    CoNotInParallel(UBYTE *);
01068 extern int    DoInParallel(UBYTE *,int);
01069 extern int    CoEndInExpression(UBYTE *);
01070 extern int    CoBracket(UBYTE *);
01071 extern int    CoPutInside(UBYTE *);
01072 extern int    CoAntiPutInside(UBYTE *);
01073 extern int    CoMultiBracket(UBYTE *);
01074 extern int    CoCFunction(UBYTE *);
01075 extern int    CoCTensor(UBYTE *);
01076 extern int    CoCollect(UBYTE *);
01077 extern int    CoCompress(UBYTE *);
01078 extern int    CoContract(UBYTE *);
01079 extern int    CoCycleSymmetrize(UBYTE *);
01080 extern int    CoDelete(UBYTE *);
01081 extern int    CoTableBase(UBYTE *);
01082 extern int    CoApply(UBYTE *);
01083 extern int    CoDenominators(UBYTE *);
01084 extern int    CoDimension(UBYTE *);
01085 extern int    CoDiscard(UBYTE *);
01086 extern int    CoDisorder(UBYTE *);
01087 extern int    CoDrop(UBYTE *);
01088 extern int    CoDropCoefficient(UBYTE *);
01089 extern int    CoDropSymbols(UBYTE *);
01090 extern int    CoElse(UBYTE *);
01091 extern int    CoElseIf(UBYTE *);
01092 extern int    CoEndArgument(UBYTE *);
01093 extern int    CoEndInside(UBYTE *);
01094 extern int    CoEndIf(UBYTE *);
01095 extern int    CoEndRepeat(UBYTE *);
```

```
01096 extern int    CoEndTerm(UBYTE *);
01097 extern int    CoEndWhile(UBYTE *);
01098 extern int    CoExit(UBYTE *);
01099 extern int    CoFactArg(UBYTE *);
01100 extern int    CoFactDollar(UBYTE *);
01101 extern int    CoFactorize(UBYTE *);
01102 extern int    CoNFactorize(UBYTE *);
01103 extern int    CoUnFactorize(UBYTE *);
01104 extern int    CoNUnFactorize(UBYTE *);
01105 extern int    DoFactorize(UBYTE *,int);
01106 extern int    CoFill(UBYTE *);
01107 extern int    CoFillExpression(UBYTE *);
01108 extern int    CoFixIndex(UBYTE *);
01109 extern int    CoFormat(UBYTE *);
01110 extern int    CoGlobal(UBYTE *);
01111 extern int    CoGlobalFactorized(UBYTE *);
01112 extern int    CoGoTo(UBYTE *);
01113 extern int    CoId(UBYTE *);
01114 extern int    CoIdNew(UBYTE *);
01115 extern int    CoIdOld(UBYTE *);
01116 extern int    CoIf(UBYTE *);
01117 extern int    CoIfMatch(UBYTE *);
01118 extern int    CoIfNoMatch(UBYTE *);
01119 extern int    CoIndex(UBYTE *);
01120 extern int    CoInsideFirst(UBYTE *);
01121 extern int    CoKeep(UBYTE *);
01122 extern int    CoLabel(UBYTE *);
01123 extern int    CoLoad(UBYTE *);
01124 extern int    CoLocal(UBYTE *);
01125 extern int    CoLocalFactorized(UBYTE *);
01126 extern int    CoMany(UBYTE *);
01127 extern int    CoMerge(UBYTE *);
01128 extern int    CoStuffle(UBYTE *);
01129 extern int    CoMetric(UBYTE *);
01130 extern int    CoModOption(UBYTE *);
01131 extern int    CoModuleOption(UBYTE *);
01132 extern int    CoModulus(UBYTE *);
01133 extern int    CoMulti(UBYTE *);
01134 extern int    CoMultiply(UBYTE *);
01135 extern int    CoNFunction(UBYTE *);
01136 extern int    CoNPrint(UBYTE *);
01137 extern int    CoNTensor(UBYTE *);
01138 extern int    CoNWrite(UBYTE *);
01139 extern int    CoNoDrop(UBYTE *);
01140 extern int    CoNoSkip(UBYTE *);
01141 extern int    CoNormalize(UBYTE *);
01142 extern int    CoMakeInteger(UBYTE *);
01143 extern int    CoFlags(UBYTE *,int);
01144 extern int    CoOff(UBYTE *);
01145 extern int    CoOn(UBYTE *);
01146 extern int    CoOnce(UBYTE *);
01147 extern int    CoOnly(UBYTE *);
01148 extern int    CoOptimizeOption(UBYTE *);
01149 extern int    CoPolyFun(UBYTE *);
01150 extern int    CoPolyRatFun(UBYTE *);
01151 extern int    CoPrint(UBYTE *);
01152 extern int    CoPrintB(UBYTE *);
01153 extern int    CoProperCount(UBYTE *);
01154 extern int    CoUnitTrace(UBYTE *);
01155 extern int    CoRCycleSymmetrize(UBYTE *);
01156 extern int    CoRatio(UBYTE *);
01157 extern int    CoRedefine(UBYTE *);
01158 extern int    CoRenumber(UBYTE *);
01159 extern int    CoRepeat(UBYTE *);
01160 extern int    CoSave(UBYTE *);
01161 extern int    CoSelect(UBYTE *);
01162 extern int    CoSet(UBYTE *);
01163 extern int    CoSetExitFlag(UBYTE *);
01164 extern int    CoSkip(UBYTE *);
01165 extern int    CoProcessBucket(UBYTE *);
01166 extern int    CoPushHide(UBYTE *);
01167 extern int    CoPopHide(UBYTE *);
01168 extern int    CoHide(UBYTE *);
01169 extern int    CoIntoHide(UBYTE *);
01170 extern int    CoNoIntoHide(UBYTE *);
01171 extern int    CoNoHide(UBYTE *);
01172 extern int    CoUnHide(UBYTE *);
01173 extern int    CoNoUnHide(UBYTE *);
01174 extern int    CoSort(UBYTE *);
01175 extern int    CoSplitArg(UBYTE *);
01176 extern int    CoSplitFirstArg(UBYTE *);
01177 extern int    CoSplitLastArg(UBYTE *);
01178 extern int    CoSum(UBYTE *);
01179 extern int    CoSymbol(UBYTE *);
01180 extern int    CoSymmetrize(UBYTE *);
01181 extern int    DoTable(UBYTE *,int);
01182 extern int    CoTable(UBYTE *);
```

```
01183 extern int    CoTerm(UBYTE *);
01184 extern int    CoNTable(UBYTE *);
01185 extern int    CoCTable(UBYTE *);
01186 extern void   EmptyTable(TABLES);
01187 extern int    CoToTensor(UBYTE *);
01188 extern int    CoToVector(UBYTE *);
01189 extern int    CoTrace4(UBYTE *);
01190 extern int    CoTraceN(UBYTE *);
01191 extern int    CoChisholm(UBYTE *);
01192 extern int    CoTransform(UBYTE *);
01193 extern int    CoClearTable(UBYTE *);
01194 extern int    DoChain(UBYTE *,int);
01195 extern int    CoChainin(UBYTE *);
01196 extern int    CoChainout(UBYTE *);
01197 extern int    CoTryReplace(UBYTE *);
01198 extern int    CoVector(UBYTE *);
01199 extern int    CoWhile(UBYTE *);
01200 extern int    CoWrite(UBYTE *);
01201 extern int    CoAuto(UBYTE *);
01202 extern int    CoSwitch(UBYTE *);
01203 extern int    CoCase(UBYTE *);
01204 extern int    CoBreak(UBYTE *);
01205 extern int    CoDefault(UBYTE *);
01206 extern int    CoEndSwitch(UBYTE *);
01207 extern int    CoTBaddto(UBYTE *);
01208 extern int    CoTBaudit(UBYTE *);
01209 extern int    CoTBcleanup(UBYTE *);
01210 extern int    CoTBcreate(UBYTE *);
01211 extern int    CoTBenter(UBYTE *);
01212 extern int    CoTBhelp(UBYTE *);
01213 extern int    CoTBload(UBYTE *);
01214 extern int    CoTBoff(UBYTE *);
01215 extern int    CoTBon(UBYTE *);
01216 extern int    CoTBopen(UBYTE *);
01217 extern int    CoTBreplace(UBYTE *);
01218 extern int    CoTBuse(UBYTE *);
01219 extern int    CoTestUse(UBYTE *);
01220 extern int    CoThreadBucket(UBYTE *);
01221 extern int    AddComString(int,WORD *,UBYTE *,int);
01222 extern int    CompileAlgebra(UBYTE *,int,WORD *);
01223 extern int    IsIdStatement(UBYTE *);
01224 extern UBYTE *IsRHS(UBYTE *,UBYTE);
01225 extern int    ParenthesesTest(UBYTE *);
01226 extern int    tokenize(UBYTE *,WORD);
01227 extern void   WriteTokens(SBYTE *);
01228 extern int    simp1token(SBYTE *);
01229 extern int    simpwtoken(SBYTE *);
01230 extern int    simp2token(SBYTE *);
01231 extern int    simp3atoken(SBYTE *,int);
01232 extern int    simp3btoken(SBYTE *,int);
01233 extern int    simp4token(SBYTE *);
01234 extern int    simp5token(SBYTE *,int);
01235 extern int    simp6token(SBYTE *,int);
01236 extern UBYTE *SkipAName(UBYTE *);
01237 extern int    TestTables(void);
01238 extern int    GetLabel(UBYTE *);
01239 extern int    CoIdExpression(UBYTE *,int);
01240 extern int    CoAssign(UBYTE *);
01241 extern int    DoExpr(UBYTE *,int,int);
01242 extern int    CompileSubExpressions(SBYTE *);
01243 extern int    CodeGenerator(SBYTE *);
01244 extern int    CompleteTerm(WORD *,UWORD *,UWORD *,WORD,WORD,int);
01245 extern int    CodeFactors(SBYTE *s);
01246 extern WORD   GenerateFactors(WORD,WORD);
01247 extern int    InsTree(int,int);
01248 extern int    FindTree(int,WORD *);
01249 extern void   RedoTree(CBUF *,int);
01250 extern void   ClearTree(int);
01251 extern int    CatchDollar(int);
01252 extern int    AssignDollar(PHEAD WORD *,WORD);
01253 extern UBYTE *WriteDollarToBuffer(WORD,WORD);
01254 extern UBYTE *WriteDollarFactorToBuffer(WORD,WORD,WORD);
01255 extern void   AddToDollarBuffer(UBYTE *);
01256 extern int    PutTermInDollar(WORD *,WORD);
01257 extern void   TermAssign(WORD *);
01258 extern void   WildDollars(PHEAD WORD *);
01259 extern LONG   numcommute(WORD *,LONG *);
01260 extern int    FullRenumber(PHEAD WORD *,WORD);
01261 extern int    Lus(WORD *,WORD,WORD,WORD,WORD,WORD);
01262 extern int    FindLus(int,int,int);
01263 extern int    CoReplaceLoop(UBYTE *);
01264 extern int    CoFindLoop(UBYTE *);
01265 extern int    DoFindLoop(UBYTE *,int);
01266 extern int    CoFunPowers(UBYTE *);
01267 extern int    SortTheList(int *,int);
01268 extern int    MatchIsPossible(WORD *,WORD *);
01269 extern int    StudyPattern(WORD *);
```

```
01270 extern WORD   DolToTensor(PHEAD WORD);
01271 extern WORD   DolToFunction(PHEAD WORD);
01272 extern WORD   DolToVector(PHEAD WORD);
01273 extern WORD   DolToNumber(PHEAD WORD);
01274 extern WORD   DolToSymbol(PHEAD WORD);
01275 extern WORD   DolToIndex(PHEAD WORD);
01276 extern LONG   DolToLong(PHEAD WORD);
01277 extern int    DollarFactorize(PHEAD WORD);
01278 extern int    CoPrintTable(UBYTE *);
01279 extern int    CoDeallocateTable(UBYTE *);
01280 extern void   CleanDollarFactors(DOLLARS);
01281 extern WORD   *TakeDollarContent(PHEAD WORD *,WORD **);
01282 extern WORD   *MakeDollarInteger(PHEAD WORD *,WORD **);
01283 extern WORD   *MakeDollarMod(PHEAD WORD *,WORD **);
01284 extern int    GetDolNum(PHEAD WORD *, WORD *);
01285 extern void   AddPotModdollar(WORD);
01286 // Returns 1 if the ModOptdollar type implies a thread-local copy when running
01287 // in TFORM (WITHPTHREADS), and 0 otherwise.
01288 static inline int DollarLocalCopy(WORD type) {
01289     if ( type == MODLOCAL ) { return 1; }
01290     if ( type == MODMIN   ) { return 1; }
01291     if ( type == MODMAX   ) { return 1; }
01292     return 0;
01293 }
01294
01295 extern int    Optimize(WORD, int);
01296 extern int    ClearOptimize(void);
01297 extern int    TakeLongRoot(UWORD *,WORD *,WORD);
01298 extern int    TakeRatRoot(UWORD *,WORD *,WORD);
01299 extern int    MakeRational(WORD ,WORD , WORD *, WORD *);
01300 extern int    MakeLongRational(PHEAD UWORD *,WORD ,UWORD *,WORD ,UWORD *,WORD *);
01301 extern void   ClearTableTree(TABLES);
01302 extern int    InsTableTree(TABLES,WORD *);
01303 extern void   RedoTableTree(TABLES,int);
01304 extern int    FindTableTree(TABLES,WORD *,int);
01305 extern void   finishcbuf(WORD);
01306 extern void   clearcbuf(WORD);
01307 extern void   CleanUpSort(int);
01308 extern FILEHANDLE *AllocFileHandle(WORD,char *);
01309 extern void   DeAllocFileHandle(FILEHANDLE *);
01310 extern void   LowerSortLevel(void);
01311 extern WORD  *PolyRatFunSpecial(PHEAD WORD *, WORD *);
01312 extern void   SimpleSplitMergeRec(WORD *,WORD,WORD *);
01313 extern void   SimpleSplitMerge(WORD *,WORD);
01314 extern WORD   BinarySearch(WORD *,WORD,WORD);
01315 extern int    InsideDollar(PHEAD WORD *,WORD);
01316 extern DOLLARS DolToTerms(PHEAD WORD);
01317 extern WORD   EvalDoLoopArg(PHEAD WORD *,WORD);
01318 extern int    SetExprCases(int,int,int);
01319 extern int    TestSelect(WORD *,WORD *);
01320 extern void   SubsInAll(PHEAD0);
01321 extern void   TransferBuffer(int,int,int);
01322 extern int    TakeIDfunction(PHEAD WORD *);
01323 extern int    MakeSetupAllocs(void);
01324 extern NORMDATA *AllocNormData(void);
01325 extern int    TryFileSetups(void);
01326 extern void   ExchangeExpressions(int,int);
01327 extern void   ExchangeDollars(int,int);
01328 extern int    GetFirstBracket(WORD *,int);
01329 extern int    GetFirstTerm(WORD *,int,int);
01330 extern int    GetContent(WORD *,int);
01331 extern int    CleanupTerm(WORD *);
01332 extern WORD   ContentMerge(PHEAD WORD *,WORD *);
01333 extern UBYTE *PreIfDollarEval(UBYTE *,int *);
01334 extern LONG   TermsInDollar(WORD);
01335 extern LONG   SizeOfDollar(WORD);
01336 extern LONG   TermsInExpression(WORD);
01337 extern LONG   SizeOfExpression(WORD);
01338 extern WORD   *TranslateExpression(UBYTE *);
01339 extern int    IsSetMember(WORD *,WORD);
01340 extern int    IsMultipleOf(WORD *,WORD *);
01341 extern int    TwoExprCompare(WORD *,WORD *,int);
01342 extern void   UpdatePositions(void);
01343 extern void   M_check(void);
01344 extern void   M_print(void);
01345 extern void   M_check1(void);
01346 extern void   PrintTime(UBYTE *);
01347
01348 extern POSITION *FindBracket(WORD,WORD *);
01349 extern void   PutBracketInIndex(PHEAD WORD *,POSITION *);
01350 extern void   ClearBracketIndex(WORD);
01351 extern void   OpenBracketIndex(WORD);
01352 extern int    DoNoParallel(UBYTE *);
01353 extern int    DoParallel(UBYTE *);
01354 extern int    DoModSum(UBYTE *);
01355 extern int    DoModMax(UBYTE *);
01356 extern int    DoModMin(UBYTE *);
```

```
01357 extern int    DoModLocal(UBYTE *);
01358 extern UBYTE *DoModDollar(UBYTE *,int);
01359 extern int    DoProcessBucket(UBYTE *);
01360 extern int    DoinParallel(UBYTE *);
01361 extern int    DonotinParallel(UBYTE *);
01362
01363 extern int    FlipTable(FUNCTIONS,int);
01364 extern int    ChainIn(PHEAD WORD *,WORD);
01365 extern int    ChainOut(PHEAD WORD *,WORD);
01366 extern int    ArgumentImplode(PHEAD WORD *,WORD *);
01367 extern int    ArgumentExplode(PHEAD WORD *,WORD *);
01368 extern int    DenToFunction(WORD *,WORD);
01369
01370 extern WORD   HowMany(PHEAD WORD *,WORD *);
01371 extern void   RemoveDollars(void);
01372 extern LONG   CountTerms1(PHEAD0);
01373 extern LONG   TermsInBracket(PHEAD WORD *,WORD);
01374 extern int    Crash(void);
01375
01376 extern char  *str_dup(char *);
01377 extern void   convertblock(INDEXBLOCK *,INDEXBLOCK *,int);
01378 extern void   convertnamesblock(NAMESBLOCK *,NAMESBLOCK *,int);
01379 extern void   convertiniinfo(INIINFO *,INIINFO *,int);
01380 extern int    ReadIndex(DBASE *);
01381 extern int    WriteIndexBlock(DBASE *,MLONG);
01382 extern int    WriteNamesBlock(DBASE *,MLONG);
01383 extern int    WriteIndex(DBASE *);
01384 extern int    WriteIniInfo(DBASE *);
01385 extern int    ReadIniInfo(DBASE *);
01386 extern int    AddToIndex(DBASE *,MLONG);
01387 extern DBASE *GetDbase(char *,MLONG);
01388 extern DBASE *OpenDbase(char *);
01389 extern char  *ReadObject(DBASE *,MLONG,char *);
01390 extern char  *ReadijObject(DBASE *,MLONG,MLONG,char *);
01391 extern int    ExistsObject(DBASE *,MLONG,char *);
01392 extern int    DeleteObject(DBASE *,MLONG,char *);
01393 extern int    WriteObject(DBASE *,MLONG,char *,char *,MLONG);
01394 extern MLONG  AddObject(DBASE *,MLONG,char *,char *);
01395 extern DBASE *NewDbase(char *,MLONG);
01396 extern void   FreeTableBase(DBASE *);
01397 extern int    ComposeTableNames(DBASE *);
01398 extern int    PutTableNames(DBASE *);
01399 extern MLONG  AddTableName(DBASE *,char *,TABLES);
01400 extern MLONG  GetTableName(DBASE *,char *);
01401 extern MLONG  FindTableNumber(DBASE *,char *);
01402 extern int    TryEnvironment(void);
01403
01404 #ifdef WITHZLIB
01405 extern int    SetupOutputGZIP(FILEHANDLE *);
01406 extern int    PutOutputGZIP(FILEHANDLE *);
01407 extern int    FlushOutputGZIP(FILEHANDLE *);
01408 extern int    SetupAllInputGZIP(SORTING *);
01409 extern LONG   FillInputGZIP(FILEHANDLE *,POSITION *,UBYTE *,LONG,int);
01410 extern void   ClearSortGZIP(FILEHANDLE *f);
01411 #endif
01412
01413 #ifdef WITHPTHREADS
01414 extern void   BeginIdentities(void);
01415 extern int    WhoAmI(void);
01416 extern int    StartAllThreads(int);
01417 extern void   StartHandleLock(void);
01418 extern void   TerminateAllThreads(void);
01419 extern int    GetAvailableThread(void);
01420 extern int    ConditionalGetAvailableThread(void);
01421 extern int    BalanceRunThread(PHEAD int,WORD *,WORD);
01422 extern void   WakeupThread(int,int);
01423 extern int    MasterWait(void);
01424 extern int    InParallelProcessor(void);
01425 extern int    ThreadsProcessor(EXPRESSIONS,WORD,WORD);
01426 extern int    MasterMerge(void);
01427 extern int    PutToMaster(PHEAD WORD *);
01428 extern void   SetWorkerFiles(void);
01429 extern int    MakeThreadBuckets(int,int);
01430 extern int    SendOneBucket(int);
01431 extern int    LoadOneThread(int,int,THREADBUCKET *,int);
01432 extern void  *RunSortBot(void *);
01433 extern void   MasterWaitAllSortBots(void);
01434 extern int    SortBotMerge(PHEAD0);
01435 extern int    SortBotOut(PHEAD WORD *);
01436 extern void   DefineSortBotTree(void);
01437 extern int    SortBotMasterMerge(void);
01438 extern int    SortBotWait(int);
01439 extern void   StartIdentity(void);
01440 extern void   FinishIdentity(void *);
01441 extern int    SetIdentity(int *);
01442 extern ALLPRIVATES *InitializeOneThread(int);
01443 extern void   FinalizeOneThread(int);
```

```
01444 extern void   ClearAllThreads(void);
01445 extern void  *RunThread(void *);
01446 extern void   IAmAvailable(int);
01447 extern int    ThreadWait(int);
01448 extern int    ThreadClaimedBlock(int);
01449 extern int    GetThread(int);
01450 extern int    UpdateOneThread(int);
01451 extern void   MasterWaitAll(void);
01452 extern void   MasterWaitAllBlocks(void);
01453 extern int    MasterWaitThread(int);
01454 extern void   WakeupMasterFromThread(int,int);
01455 extern int    LoadReadjusted(void);
01456 extern int    IniSortBlocks(int);
01457 extern int    UpdateSortBlocks(int);
01458 extern int    TreatIndexEntry(PHEAD LONG);
01459 extern WORD   GetTerm2(PHEAD WORD *);
01460 extern void   SetHideFiles(void);
01461
01462 #endif
01463
01464 extern int    CopyExpression(FILEHANDLE *,FILEHANDLE *);
01465
01466 extern int    set_in(UBYTE, set_of_char);
01467 extern one_byte set_set(UBYTE, set_of_char);
01468 extern one_byte set_del(UBYTE, set_of_char);
01469 extern one_byte set_sub (set_of_char, set_of_char, set_of_char);
01470 extern int    DoPreAddSeparator(UBYTE *);
01471 extern int    DoPreRmSeparator(UBYTE *);
01472
01473 /*See the file extcmd.c*/
01474 extern int    openExternalChannel(UBYTE *,int,UBYTE *,UBYTE *);
01475 extern int    initPresetExternalChannels(UBYTE *, int);
01476 extern int    closeExternalChannel(int);
01477 extern int    selectExternalChannel(int);
01478 extern int    getCurrentExternalChannel(void);
01479 extern void   closeAllExternalChannels(void);
01480
01481 typedef int (*WRITEBUFTOEXTCHANNEL)(char *,size_t);
01482 typedef int (*GETCFROMEXTCHANNEL)(void);
01483 typedef int (*SETTERMINATORFOREXTERNALCHANNEL)(char *);
01484 typedef int (*SETKILLMODEFOREXTERNALCHANNEL)(int,int);
01485 typedef LONG (*WRITEFILE)(int,UBYTE *,LONG);
01486 typedef WORD (*GETTERM)(PHEAD WORD *);
01487
01488 #define CompareTerms ((COMPARE)AR.CompareRoutine)
01489 #define FiniShuffle AN.SHvar.finishuf
01490 #define DoShtuffle ((DO_UFFLE)AN.SHvar.do_uffle)
01491
01492 extern UBYTE *defineChannel(UBYTE*, HANDLERS*);
01493 extern int    writeToChannel(int,UBYTE *,HANDLERS*);
01494 #ifdef WITHEXTERNALCHANNEL
01495 extern LONG   WriteToExternalChannel(int,UBYTE *,LONG);
01496 #endif
01497 extern int    writeBufToExtChannelOk(char *,size_t);
01498 extern int    getcFromExtChannelOk(void);
01499 extern int    setKillModeForExternalChannelOk(int,int);
01500 extern int    setTerminatorForExternalChannelOk(char *);
01501 extern int    getcFromExtChannelFailure(void);
01502 extern int    setKillModeForExternalChannelFailure(int,int);
01503 extern int    setTerminatorForExternalChannelFailure(char *);
01504 extern int    writeBufToExtChannelFailure(char *,size_t);
01505
01506 extern int    ReleaseTB(void);
01507
01508 extern int    SymbolNormalize(WORD *);
01509 extern int    TestFunFlag(PHEAD WORD *);
01510 extern WORD   CompareSymbols(PHEAD WORD *,WORD *,WORD);
01511 extern WORD   CompareHSymbols(PHEAD WORD *,WORD *,WORD);
01512 extern WORD   NextPrime(PHEAD WORD);
01513 extern WORD   Moebius(PHEAD WORD);
01514 extern UWORD  wranf(PHEAD0);
01515 extern UWORD  iranf(PHEAD UWORD);
01516 extern void   iniwranf(PHEAD0);
01517 extern UBYTE *PreRandom(UBYTE *);
01518
01519 extern WORD  *PolyNormPoly(PHEAD WORD);
01520 extern WORD  *EvaluateGcd(PHEAD WORD *);
01521 extern int    TreatPolyRatFun(PHEAD WORD *);
01522
01523 extern int    ReadSaveHeader(void);
01524 extern LONG   ReadSaveIndex(FILEINDEX *);
01525 extern LONG   ReadSaveExpression(UBYTE *,UBYTE *,LONG *,LONG *);
01526 extern UBYTE *ReadSaveTerm32(UBYTE *,UBYTE *,UBYTE **,UBYTE *,UBYTE *,int);
01527 extern LONG   ReadSaveVariables(UBYTE *,UBYTE *,LONG *,LONG *,INDEXENTRY *,LONG *);
01528 extern LONG   WriteStoreHeader(WORD);
01529
01530 extern void   InitRecovery(void);
```

```
01531 extern int    CheckRecoveryFile(void);
01532 extern void   DeleteRecoveryFile(void);
01533 extern char  *RecoveryFilename(void);
01534 extern int    DoRecovery(int *);
01535 extern void   DoCheckpoint(int);
01536
01537 extern void NumberMallocAddMemory(PHEAD0);
01538 extern void CacheNumberMallocAddMemory(PHEAD0);
01539 extern void TermMallocAddMemory(PHEAD0);
01540 #ifndef MEMORYMACROS
01541 extern WORD *TermMalloc2(PHEAD char *text);
01542 extern void TermFree2(PHEAD WORD *term,char *text);
01543 extern UWORD *NumberMalloc2(PHEAD char *text);
01544 extern UWORD *CacheNumberMalloc2(PHEAD char *text);
01545 extern void NumberFree2(PHEAD UWORD *NumberMem,char *text);
01546 extern void CacheNumberFree2(PHEAD UWORD *NumberMem,char *text);
01547 #endif
01548
01549 extern void ExprStatus(EXPRESSIONS);
01550 extern void iniTools(void);
01551 extern int TestTerm(WORD *);
01552
01553 extern int RunTransform(PHEAD WORD *term, WORD *params);
01554 extern int RunEncode(PHEAD WORD *fun, WORD *args, WORD *info);
01555 extern int RunDecode(PHEAD WORD *fun, WORD *args, WORD *info);
01556 extern int RunReplace(PHEAD WORD *fun, WORD *args, WORD *info);
01557 extern int RunImplode(WORD *fun, WORD *args);
01558 extern int RunExplode(PHEAD WORD *fun, WORD *args);
01559 extern int TestArgNum(int n, int totarg, WORD *args);
01560 extern WORD PutArgInScratch(WORD *arg,UWORD *scrat);
01561 extern UBYTE *ReadRange(UBYTE *s, WORD *out, int par);
01562 extern int FindRange(PHEAD WORD *,WORD *,WORD *,WORD);
01563 extern int RunPermute(PHEAD WORD *fun, WORD *args, WORD *info);
01564 extern int RunReverse(PHEAD WORD *fun, WORD *args);
01565 extern int RunCycle(PHEAD WORD *fun, WORD *args, WORD *info);
01566 extern int RunAddArg(PHEAD WORD *fun, WORD *args);
01567 extern int RunMulArg(PHEAD WORD *fun, WORD *args);
01568 extern int RunIsLyndon(PHEAD WORD *fun, WORD *args, int par);
01569 extern WORD RunToLyndon(PHEAD WORD *fun, WORD *args, int par);
01570 extern int RunDropArg(PHEAD WORD *fun, WORD *args);
01571 extern int RunSelectArg(PHEAD WORD *fun, WORD *args);
01572 extern int RunDedup(PHEAD WORD *fun, WORD *args);
01573 extern int RunZtoHArg(PHEAD WORD *fun, WORD *args);
01574 extern int RunHtoZArg(PHEAD WORD *fun, WORD *args);
01575
01576 extern int NormPolyTerm(PHEAD WORD *);
01577 extern WORD ComparePoly(WORD *, WORD *, WORD);
01578 extern int ConvertToPoly(PHEAD WORD *, WORD *,WORD *,WORD);
01579 extern int LocalConvertToPoly(PHEAD WORD *, WORD *, WORD,WORD);
01580 extern int ConvertFromPoly(PHEAD WORD *, WORD *, WORD, WORD, WORD, WORD);
01581 extern int FindSubterm(WORD *);
01582 extern int FindLocalSubterm(PHEAD WORD *, WORD);
01583 extern void PrintSubtermList(int,int);
01584 extern void PrintExtraSymbol(int,WORD *,int);
01585 extern int FindSubexpression(WORD *);
01586
01587 extern void UpdateMaxSize(void);
01588
01589 extern int CoToPolynomial(UBYTE *);
01590 extern int CoFromPolynomial(UBYTE *);
01591 extern int CoArgToExtraSymbol(UBYTE *);
01592 extern int CoExtraSymbols(UBYTE *);
01593 extern UBYTE *GetDoParam(UBYTE *, WORD **, int);
01594 extern WORD *GetIfDollarFactor(UBYTE **, WORD *);
01595 extern int CoDo(UBYTE *);
01596 extern int CoEndDo(UBYTE *);
01597 extern int ExtraSymFun(PHEAD WORD *,WORD);
01598 extern int PruneExtraSymbols(WORD);
01599 extern int IniFbuffer(WORD);
01600 extern void IniFbufs(void);
01601 extern int GCDfunction(PHEAD WORD *,WORD);
01602 extern WORD *GCDfunction3(PHEAD WORD *,WORD *);
01603 extern int ReadPolyRatFun(PHEAD WORD *);
01604 extern int FromPolyRatFun(PHEAD WORD *, WORD **, WORD **);
01605 extern WORD *PolyDiv(PHEAD WORD *,WORD *,char *);
01606 extern void GCDclean(PHEAD WORD *, WORD *);
01607 extern WORD *TakeSymbolContent(PHEAD WORD *,WORD *);
01608 extern int GCDterms(PHEAD WORD *,WORD *,WORD *);
01609 extern WORD *PutExtraSymbols(PHEAD WORD *,WORD,int *);
01610 extern WORD *TakeExtraSymbols(PHEAD WORD *,WORD);
01611 extern WORD *MultiplyWithTerm(PHEAD WORD *, WORD *,WORD);
01612 extern WORD *TakeContent(PHEAD WORD *, WORD *);
01613 extern int MergeSymbolLists(PHEAD WORD *, WORD *, int);
01614 extern int MergeDotproductLists(PHEAD WORD *, WORD *, int);
01615 extern WORD *CreateExpression(PHEAD WORD);
01616 extern int DIVfunction(PHEAD WORD *,WORD,int);
01617 extern WORD *MULfunc(PHEAD WORD *, WORD *);
```

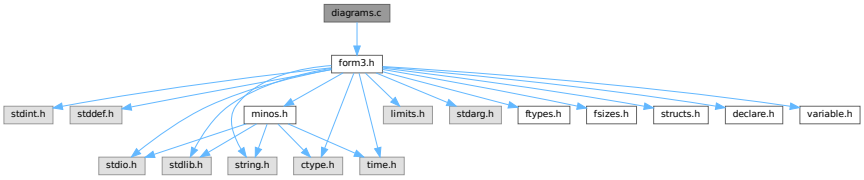
```
01618 extern WORD *ConvertArgument(PHEAD WORD *,int *);
01619 extern int ExpandRat(PHEAD WORD *);
01620 extern int InvPoly(PHEAD WORD *,WORD,WORD);
01621 extern WORD TestDoLoop(PHEAD WORD *,WORD);
01622 extern WORD TestEndDoLoop(PHEAD WORD *,WORD);
01623
01624 extern WORD  *poly_gcd(PHEAD WORD *, WORD *, WORD);
01625 extern WORD  *poly_div(PHEAD WORD *, WORD *, WORD);
01626 extern WORD  *poly_rem(PHEAD WORD *, WORD *, WORD);
01627 extern WORD  *poly_inverse(PHEAD WORD *, WORD *);
01628 extern WORD  *poly_mul(PHEAD WORD *, WORD *);
01629 extern WORD *poly_ratfun_add(PHEAD WORD *, WORD *);
01630 extern int   poly_ratfun_normalize(PHEAD WORD *);
01631 extern int   poly_factorize_argument(PHEAD WORD *, WORD *);
01632 extern WORD *poly_factorize_dollar(PHEAD WORD *);
01633 extern int   poly_factorize_expression(EXPRESSIONS);
01634 extern int   poly_unfactorize_expression(EXPRESSIONS);
01635 extern void  poly_free_poly_vars(PHEAD const char *);
01636
01637 #ifdef WITHFLINT
01638 extern void  flint_final_cleanup_thread(void);
01639 extern void  flint_final_cleanup_master(void);
01640 extern WORD* flint_div(PHEAD WORD *, WORD *, const WORD);
01641 extern int   flint_factorize_argument(PHEAD WORD *, WORD *);
01642 extern WORD* flint_factorize_dollar(PHEAD WORD *);
01643 extern WORD* flint_gcd(PHEAD WORD *, WORD *, const WORD);
01644 extern WORD* flint_inverse(PHEAD WORD *, WORD *);
01645 extern WORD* flint_mul(PHEAD WORD *, WORD *);
01646 extern WORD* flint_ratfun_add(PHEAD WORD *, WORD *);
01647 extern int   flint_ratfun_normalize(PHEAD WORD *);
01648 extern WORD* flint_rem(PHEAD WORD *, WORD *, const WORD);
01649 extern void  flint_check_version(void);
01650 #endif
01651
01652 extern void optimize_print_code (int);
01653
01654 #ifdef WITHPTHREADS
01655 extern void find_Horner_MCTS_expand_tree(void);
01656 extern void find_Horner_MCTS_expand_tree_threaded(void);
01657 extern void optimize_expression_given_Horner(void);
01658 extern void optimize_expression_given_Horner_threaded(void);
01659 #endif
01660
01661 extern int DoPreAdd(UBYTE *s);
01662 extern int DoPreUseDictionary(UBYTE *s);
01663 extern int DoPreCloseDictionary(UBYTE *s);
01664 extern int DoPreOpenDictionary(UBYTE *s);
01665 extern void RemoveDictionary(DICTIONARY *dict);
01666 extern void UnSetDictionary(void);
01667 extern int SetDictionaryOptions(UBYTE *options);
01668 extern int AddToDictionary(DICTIONARY *dict,UBYTE *left,UBYTE *right);
01669 extern int AddDictionary(UBYTE *name);
01670 extern int FindDictionary(UBYTE *name);
01671 extern UBYTE *IsExponentSign(void);
01672 extern UBYTE *IsMultiplySign(void);
01673 extern void TransformRational(UWORD *a, WORD na);
01674 extern void WriteDictionary(DICTIONARY *);
01675 extern void ShrinkDictionary(DICTIONARY *);
01676 extern void MultiplyToLine(void);
01677 extern UBYTE *FindSymbol(WORD num);
01678 extern UBYTE *FindVector(WORD num);
01679 extern UBYTE *FindIndex(WORD num);
01680 extern UBYTE *FindFunction(WORD num);
01681 extern UBYTE *FindFunWithArgs(WORD *t);
01682 extern UBYTE *FindExtraSymbol(WORD num);
01683 extern LONG DictToBytes(DICTIONARY *dict,UBYTE *buf);
01684 extern DICTIONARY *DictFromBytes(UBYTE *buf);
01685 extern int CoCreateSpectator(UBYTE *inp);
01686 extern int CoToSpectator(UBYTE *inp);
01687 extern int CoRemoveSpectator(UBYTE *inp);
01688 extern int CoEmptySpectator(UBYTE *inp);
01689 extern int CoCopySpectator(UBYTE *inp);
01690 extern int PutInSpectator(WORD *,WORD);
01691 extern void ClearSpectators(WORD);
01692 extern WORD GetFromSpectator(WORD *,WORD);
01693 extern void FlushSpectators(void);
01694
01695 extern WORD *PreGCD(PHEAD WORD *, WORD *,int);
01696 extern int InvPoly(PHEAD WORD *,WORD,WORD);
01697
01698 extern int ReadFromScratch(FILEHANDLE *,POSITION *,UBYTE *,POSITION *);
01699 extern int AddToScratch(FILEHANDLE *,POSITION *,UBYTE *,POSITION *,int);
01700
01701 extern int DoPreAppendPath(UBYTE *);
01702 extern int DoPrePrependPath(UBYTE *);
01703
01704 extern int DoSwitch(PHEAD WORD *, WORD *);
```

```
01705 extern int DoEndSwitch(PHEAD WORD *, WORD *);
01706 extern SWITCHTABLE *FindCase(WORD, WORD);
01707 extern void SwitchSplitMergeRec(SWITCHTABLE *, WORD, SWITCHTABLE *);
01708 extern void SwitchSplitMerge(SWITCHTABLE *, WORD);
01709 extern int DoubleSwitchBuffers(void);
01710
01711 extern int DistrN(int, int *, int, int *);
01712
01713 extern int DoNamespace(UBYTE *);
01714 extern int DoEndNamespace(UBYTE *);
01715 extern int DoUse(UBYTE *);
01716 extern UBYTE *SkipName(UBYTE *);
01717 extern UBYTE *ConstructName(UBYTE *,UBYTE);
01718 extern int DoSetUserFlag(UBYTE *);
01719 extern int DoClearUserFlag(UBYTE *);
01720
01721 VERTEX *CreateVertex(MODEL *);
01722 UBYTE *ReadParticle(UBYTE *, VERTEX *,MODEL *, int);
01723 int CoModel(UBYTE *);
01724 int CoParticle(UBYTE *);
01725 int CoVertex(UBYTE *);
01726 int CoEndModel(UBYTE *);
01727 int LoadModel(MODEL *);
01728
01729 int CoSetUserFlag(UBYTE *);
01730 int CoClearUserFlag(UBYTE *);
01731 int CoCreateAllLoops(UBYTE *);
01732 int CoCreateAllPaths(UBYTE *);
01733 int CoCreateAll(UBYTE *);
01734
01735 int AllLoops(PHEAD WORD *,WORD);
01736 LONG StartLoops(PHEAD WORD *,WORD,LONG,WORD,WORD *,WORD,WORD *,WORD);
01737 LONG GenLoops(PHEAD WORD *,WORD,LONG,WORD,WORD *,WORD,WORD *,WORD);
01738 void LoopOutput(PHEAD WORD *,WORD,WORD *,WORD);
01739 int AllPaths(PHEAD WORD *,WORD);
01740 LONG GenPaths(PHEAD WORD *,WORD,LONG,WORD,WORD *,WORD,WORD *,WORD);
01741 void PathOutput(PHEAD WORD *,WORD,WORD *,WORD);
01742
01743 #ifdef WITHFLOAT
01744 int DoStartFloat(UBYTE *);
01745 int DoEndFloat(UBYTE *);
01746 int SetFloatPrecision(WORD);
01747 void SetupMZVTables(void);
01748 void SetupMPFTables(void);
01749 void ClearMZVTables(void);
01750 int EvaluateEuler(PHEAD WORD *,WORD,WORD);
01751 int EvaluateSqrt(PHEAD WORD *,WORD,WORD);
01752 int CoEvaluate(UBYTE *);
01753 int PrintFloat(WORD *fun,int numdigits);
01754 int CoToRat(UBYTE *);
01755 int CoToFloat(UBYTE *);
01756 int CoChop(UBYTE *);
01757 int ToRat(PHEAD WORD *,WORD);
01758 int ToFloat(PHEAD WORD *,WORD);
01759 int Chop(PHEAD WORD *, WORD);
01760 WORD FloatFunToRat(PHEAD UWORD *,WORD *);
01761 int AddWithFloat(PHEAD WORD **,WORD **);
01762 int MergeWithFloat(PHEAD WORD **,WORD **);
01763 void ClearfFloat(void);
01764 int TestFloat(WORD *);
01765 SBYTE *ReadFloat(SBYTE *);
01766 UBYTE *CheckFloat(UBYTE *,int *);
01767 void SetfFloatPrecision(LONG);
01768 int EvaluateFun(PHEAD WORD *, WORD, WORD *);
01769 int CoStrictRounding(UBYTE *);
01770 int StrictRounding(PHEAD WORD *, WORD, WORD, WORD);
01771 #endif
01772
01773 /*
01774     #] Declarations :
01775 */
01776 #endif
```

4.21 diagrams.c File Reference

```
#include "form3.h"
```

Include dependency graph for diagrams.c:



Functions

- int CoCanonicalize (UBYTE ∗s)
- int DoCanonicalize (PHEAD WORD ∗term, WORD ∗params)
- int DoTopologyCanonicalize (PHEAD WORD ∗term, WORD vert, WORD edge, WORD ∗args)
- int DoShattering (PHEAD WORD ∗connect, WORD ∗environ, WORD ∗partitions, WORD nvert)

4.21.1 Detailed Description

Contains the wrapper routines for diagram manipulations.

Definition in file diagrams.c.

4.21.2 Function Documentation

4.21.2.1 CoCanonicalize()

```
int CoCanonicalize (
            UBYTE * s )
```

Definition at line 64 of file diagrams.c.

4.21.2.2 DoCanonicalize()

```
int DoCanonicalize (
            PHEAD WORD * term,
            WORD * params )
```

Definition at line 185 of file diagrams.c.

4.21.2.3 DoTopologyCanonicalize()

```
int DoTopologyCanonicalize (
              PHEAD WORD * term,
              WORD vert,
              WORD edge,
              WORD * args )
```

Definition at line 346 of file diagrams.c.

4.21.2.4 DoShattering()

```
int DoShattering (
              PHEAD WORD * connect,
              WORD * environ,
              WORD * partitions,
              WORD nvert )
```

Definition at line 555 of file diagrams.c.

4.22 diagrams.c

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
00026  *
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
00031 /*
00032     #[ Includes : diagrams.c
00033 */
00034 /* UNFINISHED_FEATURE_EXCL_START */
00035 #include "form3.h"
00036
00037 static WORD one = 1;
00038
00039 /*
00040     #] Includes :
00041     #[ CoCanonicalize :
00042
00043     Syntax:
00044         Canonicalize,mainoption,...
00045     With the main options currently:
00046         Canonicalize,topology,vertexfunction,edgefunction,OutDollar,extraoptions;
00047         Canonicalize,polynomial,InDollar,set_or_setname_or_dollar,OutDollar,extraoptions;
00048     The vertex function needs to have the format (assume it is called vx):
00049         vx(p1,p2,-p3)  or vx(-p1,p2,p3,-p4) etc.
```

```
00050     The external lines have a vertex with only one line.
00051     All momenta that form connections should be unique.
00052     In principle the - signs are not relevant for the topology, but they
00053     may exist already in the remaining part of the diagram. They are also
00054     part of the canonical form.
00055     The edge function can be used in different ways, depending on the options.
00056     The extraoption(s) should be nonnegative integers or $variables that evaluate
00057     into nonnegative integers (integers less than 2^31).
00058     The Indollar variable contains the polynomial to be canonicalized.
00059     The OutDollar variable should be the name of a $-variable (as in $out) which
00060     will be filled with a replace_ function. The canonicalization can then
00061     be executed in the whole term with the   Multiply $out;  command.
00062 */
00063
00064 int CoCanonicalize(UBYTE *s)
00065 {
00066     WORD args[10], *a, num;
00067     UBYTE *t, c;
00068     args[0] = TYPECANONICALIZE;
00069     a = args+2;
00070     while ( *s == ',' || *s == '\t' || *s == ' ' ) s++;
00071     t = s; while ( FG.cTable[*s] == 0 ) s++;
00072     c = *s; *s = 0;
00073     if ( StrICmp(t,(UBYTE *)("topology")) == 0 ) {
00074         *s = c; *a++ = 0;
00075         while ( *s == ',' || *s == '\t' || *s == ' ' ) s++;
00076         s = GetFunction(s,a);
00077         while ( *s == ',' || *s == '\t' || *s == ' ' ) s++;
00078         s = GetFunction(s,a+1);
00079         if ( *a == 0 || a[1] == 0 ) return(1);
00080         a += 2;
00081     }
00082     else if ( StrICmp(t,(UBYTE *)("polynomial")) == 0 ) {
00083         *s = c; *a++ = 1;
00084         while ( *s == ',' || *s == '\t' || *s == ' ' ) s++;
00085 /*
00086         Now get the name of the input $-variable
00087 */
00088         if ( *s != '$' ) {
00089             MesPrint("&Canonicalize statement needs a $-variable for its input.");
00090             return(1);
00091         }
00092         s++; t = s; while ( FG.cTable[*s] < 2 ) s++;
00093         c = *s; *s = 0;
00094         if ( GetName(AC.dollarnames,t,&num,NOAUTO) == CDOLLAR ) *a++ = num;
00095         else { *a++ = AddDollar(t,DOLINDEX,&one,1); }
00096         *s = c;
00097 /*
00098         Now the set
00099 */
00100         if ( *s == '{' ) {
00101             t = s+1; SKIPBRA2(s)
00102             c = *s; *s = 0;
00103             *a++ = DoTempSet(t,s);
00104             *s++ = c;
00105         }
00106         else if ( FG.cTable[*s] == 0 || *s == '[' ) {
00107             t = s;
00108             if ( ( s = SkipAName(s) ) == 0 ) {
00109                 MesPrint("&Illegal name for set in Canonicalize statement: %s",t);
00110                 return(1);
00111             }
00112             c = *s; *s = 0;
00113             if ( GetName(AC.varnames,t,a,WITHAUTO) == CSET ) {
00114                 if ( Sets[*a].type != CSYMBOL ) {
00115                     MesPrint("&In Canonicalize: %s is not a set of symbols.",t);
00116                     return(1);
00117                 }
00118             }
00119             else {
00120                 MesPrint("&In Canonicalize: %s is not a set.",t);
00121                 return(1);
00122             }
00123             *s = c; a++;
00124         }
00125         else if ( *s == '$' ) {
00126             s++; t = s; while ( FG.cTable[*s] < 2 ) s++;
00127             c = *s; *s = 0;
00128             if ( GetName(AC.dollarnames,t,&num,NOAUTO) == CDOLLAR ) *a++ = -num-2;
00129             else {
00130                 MesPrint("&In Canonicalize: %s is undefined.",t-1);
00131                 return(1);
00132             }
00133             *s = c;
00134         }
00135         else {
00136             MesPrint("&In Canonicalize: Illegal third(=set) argument.");
```

```
00137             return(1);
00138         }
00139     }
00140     else {
00141         MesPrint("&Unrecognized option in Canonicalize statement: %s",t);
00142         return(1);
00143     }
00144 /*
00145     Now get the name of the output $-variable
00146 */
00147     while ( *s == ',' || *s == '\t' || *s == ' ' ) s++;
00148     if ( *s != '$' ) {
00149         MesPrint("&Canonicalize statement needs a $-variable for its output.");
00150         return(1);
00151     }
00152     s++; t = s; while ( FG.cTable[*s] < 2 ) s++;
00153     c = *s; *s = 0;
00154     if ( GetName(AC.dollarnames,t,&num,NOAUTO) == CDOLLAR ) *a++ = num;
00155     else { *a++ = AddDollar(t,DOLINDEX,&one,1); }
00156     *s = c;
00157 /*
00158     Now the options. At the moment we just do one of them.
00159     (the first extra option is relevant to determine the use of the edge function)
00160     In the future we may have to be more flexible.
00161 */
00162     a[0] = 0;  /* default value */
00163     while ( *s == ',' || *s == '\t' || *s == ' ' ) s++;
00164     if ( *s != 0 ) {
00165         s = GetNumber(s,a);
00166         if ( *a == -1 ) return(1);
00167         while ( *s == ',' || *s == '\t' || *s == ' ' ) s++;
00168         a++;
00169     }
00170 /*
00171     Now complete the args string and put it in the compiler buffer
00172 */
00173     args[1] = a-args;
00174     AddNtoL(args[1],args);
00175     return(0);
00176 }
00177
00178 /*
00179     #] CoCanonicalize :
00180     #[ DoCanonicalize :
00181
00182     Does the canonicalization. The output term overwrites the input term.
00183 */
00184
00185 int DoCanonicalize(PHEAD WORD *term, WORD *params)
00186 {
00187     WORD args[10];
00188     int i;
00189 /*
00190     First check whether we need to expand dollars;
00191 */
00192     for ( i = 0; i < params[1]; i++ ) args[i] = params[i];
00193     if ( args[2] == 0 ) { /* topology */
00194         for ( i = 3; i < 5; i++ ) {
00195             if ( args[i] < 0 ) { /* This is a dollar */
00196                 args[i] = DolToFunction(BHEAD -args[i]-2);
00197                 if ( args[i] == 0 ) {
00198                     MLOCK(ErrorMessageLock);
00199                     MesPrint("Value of $-variable in Canonicalize statement should be a function.");
00200                     MUNLOCK(ErrorMessageLock);
00201                     Terminate(-1);
00202                 }
00203             }
00204         }
00205         for ( i = 6; i < args[1]; i++ ) { /* Extra options */
00206             if ( args[i] < 0 ) { /* This is a dollar */
00207                 args[i] = DolToNumber(BHEAD -args[i]-2);
00208                 if ( args[i] < 0 ) {
00209                     MLOCK(ErrorMessageLock);
00210                     MesPrint("Value of $-variable in Canonicalize statement should be a nonnegative
      number < %l.",(LONG)MAXPOSITIVE);
00211                     MUNLOCK(ErrorMessageLock);
00212                     Terminate(-1);
00213                 }
00214             }
00215         }
00216         switch ( args[6] ) {
00217             case 1: {/* pass the vertex and edge functions. */
00218                 WORD *tstop, *t, *tedge, *te;
00219                 tstop = term + *term; tstop -= ABS(tstop[-1]);
00220                 t = term+1;
00221                 tedge = AT.WorkPointer; te = tedge+1;
00222                 while ( t < tstop ) {
```

```
00223                   if ( *t != args[3] && *t != args[4] ) { t += t[1]; continue; }
00224                   for ( i = 0; i < t[1]; i++ ) te[i] = t[i];
00225                   te += t[1]; t += t[1];
00226               }
00227               *te++ = 1; *te++ = 1; *te++ = 3;
00228               tedge[0] = te-tedge;
00229               AT.WorkPointer = te;
00230 /*
00231               DoVertexCanonicalize(BHEAD term,tedge,args[3],args[4],args[5],args[6]);
00232 */
00233               AT.WorkPointer = tedge;
00234           } break;
00235           case 2: {/* pass the edge functions only */
00236               WORD *tstop, *t, *tedge, *te;
00237               tstop = term + *term; tstop -= ABS(tstop[-1]);
00238               t = term+1;
00239               tedge = AT.WorkPointer; te = tedge+1;
00240               while ( t < tstop ) {
00241                   if ( *t != args[4] ) { t += t[1]; continue; }
00242                   for ( i = 0; i < t[1]; i++ ) te[i] = t[i];
00243                   te += t[1]; t += t[1];
00244               }
00245               *te++ = 1; *te++ = 1; *te++ = 3;
00246               tedge[0] = te-tedge;
00247               AT.WorkPointer = te;
00248 /*
00249               DoEdgeCanonicalize(BHEAD term,tedge,args[5]);
00250 */
00251               AT.WorkPointer = tedge;
00252           } break;
00253           default: {
00254               DoTopologyCanonicalize(BHEAD term,args[3],args[4],args+5);
00255           } break;
00256       }
00257 /*
00258       Call here the topology canonicalization
00259       We will have the arguments:
00260           args[3]: The function used as vertex function
00261           args[4]: The function used as edge function
00262           args[5]: The number of the dollar to be used for the output
00263           args[6]: Potentially other options (like saying how to use args[4]).
00264           term:    The term in which the topology resides.
00265 */
00266
00267   }
00268   else if ( args[2] == 1 ) { /* polynomial */
00269       WORD *symlist, nsymlist;
00270       for ( i = 6; i < args[1]; i++ ) { /* Extra options */
00271           if ( args[i] < 0 ) { /* This is a dollar */
00272               args[i] = DolToNumber(BHEAD -args[i]-2);
00273               if ( args[i] < 0 ) {
00274                   MLOCK(ErrorMessageLock);
00275                   MesPrint("Value of $-variable in Canonicalize statement should be a nonnegative
      number < %l.",(LONG)MAXPOSITIVE);
00276                   MUNLOCK(ErrorMessageLock);
00277                   Terminate(-1);
00278               }
00279           }
00280       }
00281 /*
00282       Now we sort out the set. We create a pointer to the list of set
00283       elements, and we determine the number of elements in the set.
00284 */
00285       symlist = AT.WorkPointer;
00286       if ( args[4] < -1 ) { /* Dollar that should expand into a list of symbols */
00287           DOLLARS d = Dollars - args[4] - 2;
00288           WORD *ds, *insym;
00289           if ( d->type != DOLWILDARGS ) {
00290 NoWildArg:
00291               MLOCK(ErrorMessageLock);
00292               MesPrint("Value of $-variable in Canonicalize statement should be a argument wildcard
      of symbol arguments.");
00293               MUNLOCK(ErrorMessageLock);
00294               Terminate(-1);
00295           }
00296           insym = symlist; ds = d->where+1;
00297           while ( *ds ) {
00298               if ( *ds != -SYMBOL ) goto NoWildArg;
00299               *insym++ = ds[1];
00300               ds += 2;
00301           }
00302           nsymlist = insym-symlist;
00303       }
00304       else { /*if ( args[4] >= 0 ) */
00305           WORD *ss, *sy, n;
00306           ss = (WORD *)(AC.SetElementList.lijst)+Sets[args[4]].first;
00307           nsymlist = n = Sets[args[4]].last-Sets[args[4]].first;
```

```
00308             sy = symlist = AT.WorkPointer;
00309             NCOPY(sy,ss,n);
00310         }
00311         AT.WorkPointer = symlist+nsymlist;
00312 /*
00313         Call here the polynomial canonicalization
00314         We will have the arguments:
00315             args[3]: The number of the dollar to be used for the input
00316             symlist: an array of symbols
00317             nsymlist: the number of symbols in symlist
00318             args[5]: The number of the dollar to be used for the output
00319             args[6]: Potentially other options.
00320 */
00321 /*
00322         DoPolynomialCanonicalize(BHEAD args[3],symlist,nsymlist,args[5],args[6]);
00323 */
00324         AT.WorkPointer = symlist;
00325     }
00326     return(0);
00327 }
00328
00329 /*
00330     #] DoCanonicalize :
00331     #[ DoTopologyCanonicalize :
00332
00333     term: The term
00334     vert: the vertex function
00335     edge: the edge function
00336     args[0]: the number of the output dollar
00337     args[1]: options
00338     return value should be zero if all is correct.
00339
00340     The external lines connect to an 'external vertex' which has only one line.
00341     The vertices are of a type: vertex_(p1,p2,-p3)*vertex_(p3,p4,p5) etc.
00342     The edges indicate noninteger powers of the lines:
00343         edge_(p1,2) means 1/p1.p1^(2*ep)
00344 */
00345
00346 int DoTopologyCanonicalize(PHEAD WORD *term,WORD vert,WORD edge,WORD *args)
00347 {
00348     int nvert = 0, nvert2, i, ii, jj, flipnames = 0, nparts/*, level, num */;
00349     WORD *tstop, *t, *tt, *tend, *td;
00350     WORD *oldworkpointer = AT.WorkPointer;
00351     WORD *termcopy = TermMalloc("TopologyCanonize1");
00352     WORD *vet= TermMalloc("TopologyCanonize2");
00353     WORD *partition, *environ, *connect, *pparts, *p;
00354 /*
00355     WORD *pparts;
00356 */
00357     WORD momenta[150],flips[50],nmomenta = 0, nflips = 0;
00358 /*
00359     Step one: the vertices should get a number. We copy the term for this.
00360               We need a high number for the vertex function to make sure
00361               that it comes after the edge function in the sorting.
00362 */
00363     if ( args[0] < args[1] ) { flipnames = 1; }
00364     tend = term + *term; tend -= ABS(tend[-1]); t = term+1; tt = termcopy+1;
00365     while ( t < tend ) {
00366         if ( *t == vert ) {
00367             for ( i = FUNHEAD; i < t[1]; i += 2 ) {
00368                 if ( t[i] == -VECTOR || ( t[i] == -INDEX && t[i+1] < 0 ) ) {
00369                     momenta[nmomenta++] = -VECTOR;
00370                     momenta[nmomenta++] = t[i+1];
00371                 }
00372                 else if ( t[i] == -MINVECTOR ) {
00373                     momenta[nmomenta++] = -MINVECTOR;
00374                     momenta[nmomenta++] = t[i+1];
00375                 }
00376                 else goto notgoodvertex;
00377                 momenta[nmomenta++] = nvert;
00378             }
00379             ii = FUNHEAD; i = t[1]-FUNHEAD;
00380             NCOPY(tt,t,ii)
00381             if ( flipnames ) tt[-FUNHEAD] = edge;
00382             tt[-FUNHEAD+1] += 2;
00383             *tt++ = -CNUMBER; *tt++ = nvert++;
00384         }
00385         else if ( *t == edge && flipnames ) {
00386             i = t[1] - 1; *tt++ = vert; t++;
00387         }
00388         else {
00389 notgoodvertex:
00390             i = t[1];
00391         }
00392         NCOPY(tt,t,i)
00393     }
00394     while ( t < tend ) *tt++ = *t++;
```

```
00395     termcopy[0] = tt - termcopy;
00396     if ( flipnames ) EXCH(edge,vert)
00397     nvert2 = nvert*nvert;
00398 /*
00399     Sort the momenta. Keep the sign order.
00400 */
00401     for ( i = 0; i < nmomenta-3; i+=3 ) {
00402         jj = i;
00403         while ( jj >= 0 && momenta[jj+4] > momenta[jj+1] ) {
00404             EXCH(momenta[jj+5],momenta[jj+2])
00405             EXCH(momenta[jj+4],momenta[jj+1])
00406             EXCH(momenta[jj+3],momenta[jj])
00407             jj -= 3;
00408         }
00409     }
00410 /*
00411     Step two: make now the edge functions in the proper notation.
00412 */
00413     t = vet+1;
00414     for ( i = 0; i < nmomenta; i += 6 ) {
00415         if ( momenta[i] == -VECTOR && momenta[i+3] == -MINVECTOR
00416              && momenta[i+1] == momenta[i+4] ) {
00417         }
00418         else if ( momenta[i] == -MINVECTOR && momenta[i+3] == -VECTOR
00419              && momenta[i+1] == momenta[i+4] ) {
00420             flips[nflips++] = momenta[i+1];
00421             DUMMYUSE(flips[nflips-1]);
00422         }
00423         else { /* something wrong with the momenta */
00424             MLOCK(ErrorMessageLock);
00425             MesPrint("No momentum conservation or wrong momenta in Canonicalize statement");
00426             MUNLOCK(ErrorMessageLock);
00427             Terminate(-1);
00428         }
00429         *t++ = EDGE; *t++ = FUNHEAD+10; FILLFUN(t)
00430         *t++ = -SNUMBER; *t++ = momenta[i+2];
00431         *t++ = -SNUMBER; *t++ = momenta[i+5];
00432         *t++ = -VECTOR; *t++ = momenta[i+1];
00433         *t++ = -SNUMBER; *t++ = 0; /* provisional power/color, multiple of ep */
00434         *t++ = -SNUMBER; *t++ = 0; /* provisional power/color, integer */
00435     }
00436     tend = t;
00437     *t++ = 1; *t++ = 1; *t++ = 3; vet[0] = t-vet; *t = 0;
00438 /*
00439     Now the powers of the denominators
00440 */
00441     tstop = termcopy++*termcopy; tstop -= ABS(tstop[-1]); td = termcopy+1;
00442     while ( td < tstop ) {
00443         if ( *td == edge && td[1] == FUNHEAD+4 ) {
00444             if ( td[FUNHEAD+2] == -SNUMBER && ( td[FUNHEAD] == -VECTOR || td[FUNHEAD] == -INDEX
00445             || td[FUNHEAD] == -MINVECTOR ) ) {}
00446             else {
00447                 MLOCK(ErrorMessageLock);
00448                 MesPrint("Illegal argument in edge function in Canonicalize statement");
00449                 MUNLOCK(ErrorMessageLock);
00450                 Terminate(-1);
00451             }
00452             tt = vet+1;
00453             while ( tt < tend ) {
00454                 if ( tt[FUNHEAD+5] == td[FUNHEAD+1] ) { tt[FUNHEAD+7] = td[FUNHEAD+3]; break; }
00455                 tt += tt[1];
00456             }
00457         }
00458         else if ( *td == DOTPRODUCT ) break;
00459         td += td[1];
00460     }
00461     if ( td < tstop ) {
00462         tt = vet+1;
00463         while ( tt < tend ) {
00464 /*
00465             tt[FUNHEAD+5] is a vector. We look for dotproducts with twice tt[FUNHEAD+5]
00466 */
00467             for ( i = 2; i < td[1]; i += 3 ) {
00468                 if ( td[i] == tt[FUNHEAD+5] && td[i+1] == tt[FUNHEAD+5] ) {
00469                     tt[FUNHEAD+9] = td[i+2];
00470                     break;
00471                 }
00472             }
00473             tt += tt[1];
00474         }
00475     }
00476     Normalize(BHEAD vet);
00477 /*
00478     Now we have a term `vet' in the proper notation and we can start.
00479     To keep track of the shattering we use an array of 2*nvert.
00480     each entry is Number,marker
00481     When the marker is zero, the vertices are in the same partition.
```

```
00482     For the environment we need a matrix that is nvert x nvert
00483     At the same time we keep the connectivity matrix, because that will
00484     save much time later.
00485     The partitions are stored in a matrix as well. This allows them to be
00486     treated as a stack. The entries are separated by 0 if they belong to
00487     the same part, and by a 1 when they belong to different parts.
00488 */
00489     partition = AT.WorkPointer; AT.WorkPointer += 2*nvert2;
00490     for ( i = 0; i < nvert; i++ ) { partition[2*i] = i; partition[2*i+1] = 0; }
00491     partition[2*i-1] = -1;  /* end of the first part which is currently all vertices */
00492     nparts = 1;
00493
00494     connect = AT.WorkPointer; AT.WorkPointer += nvert2;
00495     for ( i = 0; i < nvert2; i++ ) connect[i] = 0;
00496     tstop = vet+*vet; tstop -= ABS(tstop[-1]); t = vet+1;
00497     while ( t < tstop ) {
00498         if ( *t == EDGE ) {
00499             connect[t[FUNHEAD+1]*nvert+t[FUNHEAD+3]]++;
00500             connect[t[FUNHEAD+3]*nvert+t[FUNHEAD+1]]++;
00501         }
00502         t += t[1];
00503     }
00504 for ( i = 0; i < nvert; i++ ) {
00505 MesPrint("connectivity: %d -- %a",i,nvert,connect+i*nvert);
00506 }
00507 /*
00508     Create the environment matrix and sort it.
00509 */
00510     environ = AT.WorkPointer; AT.WorkPointer += nvert2;
00511 /*
00512     And now the refinement process starts.
00513 */
00514     WantAddPointers(nvert+1);
00515     for ( i = 0; i < nvert2; i++ ) environ[i] = 0;
00516     /* level = 0; */
00517     pparts = partition;
00518     while ( nparts < nvert ) {
00519         nparts = DoShattering(BHEAD connect,environ,pparts,nvert);
00520         if ( nparts < nvert ) { /* raise level and make a copy and split a part */
00521             p = pparts + 2*nvert;
00522             /* level++; */
00523             for ( i = 0; i < 2*nvert; i++ ) p[i] = pparts[i];
00524             for ( ii = 0; ii < 2*nvert; ii += 2 ) {
00525                 if ( p[ii+1] == 0 ) { /* found a part with more than one */
00526                     /* num = 2; */ i = ii+2;
00527                     while ( p[i+1] == 0 ) { /* num++; */ i += 2; }
00528                     p[ii+1] = -1; pparts = p;
00529                     break;
00530                 }
00531             }
00532
00533
00534         }
00535 MesPrint("partition: %d -- %a",nparts,2*nvert,pparts);
00536
00537     }
00538 /*
00539     Just for now
00540 */
00541     PutTermInDollar(vet,args[0]);
00542
00543
00544     TermFree(vet,"TopologyCanonize2");
00545     TermFree(termcopy,"TopologyCanonize1");
00546     AT.WorkPointer = oldworkpointer;
00547     return(0);
00548 }
00549
00550 /*
00551     #] DoTopologyCanonicalize :
00552     #[ DoShattering :
00553 */
00554
00555 int DoShattering(PHEAD WORD *connect, WORD *environ, WORD *partitions, WORD nvert)
00556 {
00557     int nparts, i, j, ii, jj, iii, jjj, newmarker;
00558     WORD **p = AT.pWorkSpace + AT.pWorkPointer, *part, *endpart;
00559     WORD *poin1, *poin2;
00560 #ifdef SHATBUG
00561 MesPrint("Entering DoShattering. partitions = %a",2*nvert,partitions);
00562 #endif
00563 restart:
00564 /*
00565     Determine the number of parts
00566     p will be an array with pointers to the parts.
00567     We made space for this array in the calling routine and because this
00568     routine is not calling any other routines we do not need to raise
```

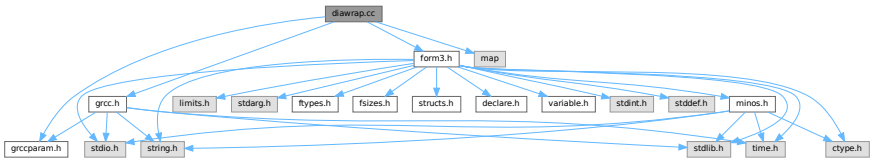
```
00569     the pointer in this stack (AT.pWorkPointer).
00570 */
00571     nparts = 0; newmarker = 0;
00572     part = partitions; endpart = part + 2*nvert;
00573     p[0] = part;
00574     while ( part < endpart ) {
00575         if ( part[1] != 0 ) { p[++nparts] = part+2; }
00576         part += 2;
00577     }
00578     for ( i = 0; i < nparts; i++ )
00579         AT.WorkPointer[i] = (p[i+1]-p[i])/2;
00580 #ifdef SHATBUG
00581 MesPrint("DoShattering: calculated the pointers");
00582 MesPrint("DoShattering: sizes: %a",nparts,AT.WorkPointer);
00583 MesPrint("DoShattering: p[0]: %a, p[1]: %a",6,p[0],6,p[1]);
00584 #endif
00585     for ( i = 0; i < nparts; i++ ) {
00586      if ( AT.WorkPointer[i] > 1 ) {
00587       for ( j = 0; j < nparts; j++ ) {
00588 /*
00589         Shatter part i wrt part j.
00590         if there is action, go to restart.
00591 */
00592         for ( ii = 0; ii < AT.WorkPointer[i]; ii++ ) {
00593           for ( jj = 0; jj < AT.WorkPointer[j]; jj++ ) {
00594             environ[ii*AT.WorkPointer[j]+jj] += connect[p[i][2*ii]*nvert+p[j][2*jj]];
00595           }
00596         }
00597 #ifdef SHATBUG
00598 for ( ii = 0; ii < AT.WorkPointer[i]; ii++ ) {
00599 MesPrint("Environ(%d,%d): %a",i,j,AT.WorkPointer[j],environ+ii*AT.WorkPointer[j]);
00600 }
00601 #endif
00602 /*
00603         Sort the rows internally, then sort the rows wrt each other
00604         and finally place new markers. If a new marker, we restart.
00605         Don't forget to clean up the environ array.
00606 */
00607         for ( ii = 0; ii < nvert; ii++ ) {
00608           poin1 = environ+ii*AT.WorkPointer[j];
00609           for ( jj = 0; jj < AT.WorkPointer[j]-1; jj++ ) {
00610             jjj = jj;
00611             while ( jjj >= 0 && poin1[jjj+1] > poin1[jjj] ) {
00612                 EXCH(poin1[jjj+1],poin1[jjj])
00613                 jjj--;
00614             }
00615           }
00616         }
00617 #ifdef SHATBUG
00618 for ( ii = 0; ii < AT.WorkPointer[i]; ii++ ) {
00619 MesPrint("environ(%d,%d): %a",i,j,AT.WorkPointer[j],environ+ii*AT.WorkPointer[j]);
00620 }
00621 #endif
00622         for ( ii = 0; ii < AT.WorkPointer[i]-1; ii++ ) {
00623             poin2 = environ+ii*AT.WorkPointer[j]; poin1 = poin2+AT.WorkPointer[j];
00624             iii = ii;
00625             while ( iii >= 0 && ( CmpArray(poin1,poin2,AT.WorkPointer[j]) < 0 ) ) {
00626                 EXCHN(poin2,poin1,AT.WorkPointer[j])
00627                 EXCH(p[i][2*iii+2],p[i][2*iii])
00628                 iii--; poin1 = poin2; poin2 = poin1-AT.WorkPointer[j];
00629             }
00630         }
00631 #ifdef SHATBUG
00632 for ( ii = 0; ii < AT.WorkPointer[i]; ii++ ) {
00633 MesPrint("environ(%d,%d): %a",i,j,AT.WorkPointer[j],environ+ii*AT.WorkPointer[j]);
00634 }
00635 MesPrint("partitions = %a",2*nvert,partitions);
00636 #endif
00637         for ( ii = 0; ii < AT.WorkPointer[i]-1; ii++ ) {
00638             poin2 = environ+ii*AT.WorkPointer[j]; poin1 = poin2+AT.WorkPointer[j];
00639             if ( CmpArray(poin1,poin2,AT.WorkPointer[j]) == 0 ) continue;
00640             p[i][2*ii+1] = -1; nparts++; newmarker++;
00641         }
00642 #ifdef SHATBUG
00643 MesPrint("partitions = %a",2*nvert,partitions);
00644 #endif
00645 /*
00646         Clear environ. This is probably faster than just clearing the whole array.
00647         Maybe in the future a test could be done on nvert to decide how to clear.
00648 */
00649         for ( ii = 0; ii < AT.WorkPointer[i]; ii++ ) {
00650           for ( jj = 0; jj < AT.WorkPointer[j]; jj++ ) {
00651             environ[ii*AT.WorkPointer[j]+jj] = 0;
00652           }
00653         }
00654         if ( newmarker ) { goto restart; }
00655       }
```

```
00656       }
00657     }
00658     return(nparts);
00659 }
00660
00661 /*
00662     #] DoShattering :
00663 */
00664
00665 /* UNFINISHED_FEATURE_EXCL_STOP */
```

4.23 diawrap.cc File Reference

```
#include "form3.h"
#include "grccparam.h"
#include "grcc.h"
#include <map>
```

Include dependency graph for diawrap.cc:



Data Structures

- struct ToPoTyPe

Macros

- #define MAXPOINTS 120
- #define **WITHEARLYVETO**

Typedefs

- typedef struct ToPoTyPe **TOPOTYPE**

Functions

- int LoadModel (MODEL ∗m)
- int ConvertParticle (Model ∗model, int formnum)
- int ReConvertParticle (Model ∗model, int grccnum)
- int numParticle (MODEL ∗m, WORD n)
- Bool fendMG (EGraph ∗eg, void ∗ti)
- Bool ProcessTopology (EGraph ∗eg, void ∗ti)
- void SetDualOpts (int ∗opt, const WORD num, const int key, const char ∗key_name, const int dual, const char ∗dual_name, const int val, const int dval)
- int GenDiagrams (PHEAD WORD ∗term, WORD level)

4.23.1 Detailed Description

Functions with interface FORM with grcc, to implement the diagrams_ function.

Definition in file diawrap.cc.

4.23.2 Macro Definition Documentation

4.23.2.1 MAXPOINTS

```
#define MAXPOINTS 120
```

Definition at line 41 of file diawrap.cc.

4.23.3 Function Documentation

4.23.3.1 LoadModel()

```
int LoadModel (
            MODEL * m )
```

Definition at line 58 of file diawrap.cc.

4.23.3.2 ConvertParticle()

```
int ConvertParticle (
            Model * model,
            int formnum )
```

Definition at line 205 of file diawrap.cc.

4.23.3.3 ReConvertParticle()

```
int ReConvertParticle (
            Model * model,
            int grccnum )
```

Definition at line 223 of file diawrap.cc.

4.23.3.4 numParticle()

```
int numParticle (
            MODEL * m,
            WORD n )
```

Definition at line 235 of file diawrap.cc.

4.23.3.5 fendMG()

```
Bool fendMG (
            EGraph * eg,
            void * ti )
```

Definition at line 506 of file diawrap.cc.

4.23.3.6 ProcessTopology()

```
Bool ProcessTopology (
            EGraph * eg,
            void * ti )
```

Definition at line 516 of file diawrap.cc.

4.23.3.7 SetDualOpts()

```
void SetDualOpts (
            int * opt,
            const WORD num,
            const int key,
            const char * key_name,
            const int dual,
            const char * dual_name,
            const int val,
            const int dval )
```

Definition at line 779 of file diawrap.cc.

4.23.3.8 GenDiagrams()

```
int GenDiagrams (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 806 of file diawrap.cc.

4.24 diawrap.cc

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
```

```
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
00026  *
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
00031 //  #[ Includes : diawrap.cc
00032
00033 extern "C" {
00034 #include "form3.h"
00035 }
00036
00037 #include "grccparam.h"
00038 #include "grcc.h"
00039 #include <map>
00040
00041 #define MAXPOINTS 120
00042
00043 typedef struct ToPoTyPe {
00044     WORD *vert;
00045     WORD *vertmax;
00046     Options *opt;
00047     int cldeg[MAXPOINTS], clnum[MAXPOINTS], clext[MAXPOINTS];
00048     int cmind[MAXLEGS+1],cmaxd[MAXLEGS+1];
00049     int ncl, nvert;
00050 } TOPOTYPE;
00051
00052 static void ProcessDiagram(EGraph *eg, void *ti);
00053 static int processVertex(TOPOTYPE *TopoInf, int pointsremaining, int level);
00054
00055 //  #] Includes :
00056 //  #[ LoadModel :
00057
00058 int LoadModel(MODEL *m)
00059 {
00060 //
00061 //  First check whether there was a model already.
00062 //  In the Kaneko setup there can be only one at the same time.
00063 //  Hence if there was one we need to remove it first.
00064 //  Unless of course, it was already the model we want.
00065 //
00066     if ( m->grccmodel != NULL ) return(0);
00067
00068     int i,j,k;
00069 //
00070 //  First the information that goes into the Model struct.
00071 //  Note that new Model takes over (does not copy) minp.cnamlist.
00072 //
00073     MInput minp;
00074     PInput pinp;
00075     IInput iinp;
00076     if ( m->ncouplings > GRCC_MAXNCPLG ) {
00077         MesPrint("Too many coupling constants in model. Current limit is %d.",(WORD)GRCC_MAXNCPLG);
00078         MesPrint("Suggestion: recompile Form with a larger value for GRCC_MAXNCPLG");
00079         return(-1);
00080     }
00081     minp.defpart = GRCC_DEFBYCODE;
00082     minp.name = (char *)(m->name);
00083     minp.ncouple = m->ncouplings;
00084     for ( i = 0; i < GRCC_MAXNCPLG; i++ ) minp.cnamlist[i] = NULL;
00085     for ( i = 0; i < minp.ncouple; i++ )
00086         minp.cnamlist[i] = (char *)(VARNAME(symbols,m->couplings[i]));
00087     Model *mdl = new Model(&minp);
00088 //
00089 //  Now the particles
00090 //
00091     for ( i = 0; i < m->nparticles; i++ ) {
00092         if ( minp.defpart == GRCC_DEFBYCODE ) {
00093             pinp.name = NULL;
00094             pinp.aname = NULL;
00095             pinp.pcode = m->vertices[i]->particles[0].number;
00096             pinp.acode = m->vertices[i]->particles[1].number;
00097             switch ( m->vertices[i]->particles[0].spin ) {
00098               case 1:
00099                 pinp.ptypec = GRCC_PT_Scalar;
00100               break;
00101               case -1:
00102                 pinp.ptypec = GRCC_PT_Ghost;
00103               break;
00104               case 2:
```

```
00105                 if ( m->vertices[i]->particles[0].type == 0 )
00106                     pinp.ptypec = GRCC_PT_Majorana;
00107                 else
00108                     pinp.ptypec = GRCC_PT_Undef;
00109               break;
00110               case -2:
00111                 if ( m->vertices[i]->particles[0].type == 0 )
00112                     pinp.ptypec = GRCC_PT_Majorana;
00113                 else
00114                     pinp.ptypec = GRCC_PT_Dirac;
00115               break;
00116               case 3:
00117                 pinp.ptypec = GRCC_PT_Vector;
00118               break;
00119               default:
00120                 pinp.ptypec = GRCC_PT_Undef;
00121               break;
00122             }
00123         }
00124         else {
00125             pinp.name   = (char *)(VARNAME(functions,m->vertices[i]->particles[0].number));
00126             pinp.aname  = (char *)(VARNAME(functions,m->vertices[i]->particles[1].number));
00127             switch ( m->vertices[i]->particles[0].spin ) {
00128               case 1:
00129                 pinp.ptypen = "scalar";
00130               break;
00131               case -1:
00132                 pinp.ptypen = "ghost";
00133               break;
00134               case 2:
00135                 if ( m->vertices[i]->particles[0].type == 0 )
00136                     pinp.ptypen = "majorana";
00137                 else
00138                     pinp.ptypen = "undef";
00139               break;
00140               case -2:
00141                 if ( m->vertices[i]->particles[0].type == 0 )
00142                     pinp.ptypen = "majorana";
00143                 else
00144                     pinp.ptypen = "dirac";
00145               break;
00146               case 3:
00147                 pinp.ptypen = "vector";
00148               break;
00149               default:
00150                 pinp.ptypen = "undef";
00151               break;
00152             }
00153         }
00154         pinp.extonly = m->vertices[i]->externonly;
00155         mdl->addParticle(&pinp);
00156     }
00157     mdl->addParticleEnd();
00158 //
00159 //  Now the vertices
00160 //
00161     for ( i = m->nparticles; i < m->invertices; i++ ) {
00162         VERTEX *v = m->vertices[i];
00163         if ( minp.defpart == GRCC_DEFBYCODE ) {
00164             iinp.icode = NODEFUNCTION+i;
00165             iinp.name = NULL;
00166         }
00167         else {
00168             iinp.name = "node_";
00169         }
00170         iinp.nplistn = v->nparticles;
00171         for ( j = 0; j < iinp.nplistn; j++ ) {
00172             if ( minp.defpart == GRCC_DEFBYCODE ) {
00173                 iinp.plistc[j] = v->particles[j].number;
00174             }
00175             else {
00176                 iinp.plistn[j] = (char *)(VARNAME(functions,v->particles[j].number));
00177             }
00178         }
00179 //
00180 //      We need a properly ordered list of coupling constants
00181 //      The ordered list is in m->couplings.
00182 //      For each vertex they are in v->couplings.
00183 //
00184 //      iinp.cvallist = (int *)Malloc1(m->ncouplings*sizeof(int),"couplings");
00185
00186         for ( j = 0; j < m->ncouplings; j++ ) {
00187             iinp.cvallist[j] = 0;
00188             for ( k = 0; k < v->ncouplings; k += 2 ) {
00189                 if ( v->couplings[k] == m->couplings[j] ) {
00190                     iinp.cvallist[j] = v->couplings[k+1];
00191                     break;
```

```
00192                 }
00193             }
00194         }
00195         mdl->addInteraction(&iinp);
00196     }
00197     mdl->addInteractionEnd();
00198     m->grccmodel = (void *)mdl;
00199     return(0);
00200 }
00201
00202 //  #] LoadModel :
00203 //  #[ ConvertParticle :
00204
00205 int ConvertParticle(Model *model,int formnum)
00206 {
00207 //
00208 //  Returns the grcc number of the particle, because grcc does not convert
00209 //
00210     int i;
00211     for ( i = 0; i < model->nParticles; i++ ) {
00212         if ( model->particles[i]->pcode == formnum ) { return(i); }
00213         else if ( model->particles[i]->acode == formnum ) { return(-i); }
00214     }
00215     MesPrint("Particle %d not found in model %s",formnum,model->name);
00216     Terminate(-1);
00217     return(0);
00218 }
00219
00220 //  #] ConvertParticle :
00221 //  #[ ReConvertParticle :
00222
00223 int ReConvertParticle(Model *model,int grccnum)
00224 {
00225 //
00226 //  Returns the grcc number of the particle, because grcc does not convert
00227 //
00228     if ( grccnum < 0 ) { return(model->particles[-grccnum]->acode); }
00229     else { return(model->particles[grccnum]->pcode); }
00230 }
00231
00232 //  #] ReConvertParticle :
00233 //  #[ numParticle :
00234
00235 int numParticle(MODEL *m,WORD n)
00236 {
00237     int i;
00238     for ( i = 0; i < m->nparticles; i++ ) {
00239         if ( m->vertices[i]->particles[0].number == n ) return(i);
00240         if ( m->vertices[i]->particles[1].number == n ) return(i);
00241     }
00242     MesPrint("numParticle: particle %d not found in model",n);
00243     Terminate(-1);
00244     return(-1);
00245 }
00246
00247 //  #] numParticle :
00248 //  #[ ProcessDiagram :
00249
00250 void ProcessDiagram(EGraph *eg, void *ti)
00251 {
00252 //
00253 //  This is the routine that gets a complete diagram and passes it on
00254 //  to Form (Generator) for further algebraic manipulations.
00255 //  The term is picked up from AT.diaterm and a new term is constructed
00256 //  in the workspace.
00257 //
00258     GETIDENTITY
00259     TERMINFO *info = (TERMINFO *)ti;
00260
00261     if ( ( info->flags & TOPOLOGIESONLY ) == TOPOLOGIESONLY ) return;
00262
00263     WORD *term = info->term, *newterm, *oldworkpointer = AT.WorkPointer;
00264     WORD *tdia = term + info->diaoffset;
00265     WORD *tail = tdia + tdia[1];
00266     WORD *tend = term + *term;
00267     WORD *fill, *startfill, *cfill, *afill;
00268     int i, j, intr;
00269     Model *model = (Model *)info->currentModel;
00270     MODEL *m = (MODEL *)info->currentMODEL;
00271     int numlegs, vect, edge, maxmom = 0;
00272
00273     newterm = term + *term;
00274     for ( i = 1; i < info->diaoffset; i++ ) newterm[i] = term[i];
00275     fill = newterm + info->diaoffset;
00276 //
00277 //  Now get the nodes
00278 //
```

```
00279     if ( ( info->flags & WITHOUTNODES ) == 0 ) {
00280       for ( i = 0; i < eg->nNodes; i++ ) {
00281 //
00282 //      node_(number,coupling,particle_1(momentum_1),...,particle_n(momentum_n))
00283 //
00284         numlegs = eg->nodes[i]->deg;
00285         startfill = fill;
00286         *fill++ = NODEFUNCTION;
00287         *fill++ = 0;
00288         FILLFUN(fill)
00289         *fill++ = -SNUMBER; *fill++ = i+1;
00290 //
00291 //      Now we put the coupling constants. This is done inside the
00292 //      function for when we want to work with counterterms.
00293 //
00294         if ( !eg->isExternal(i) ) {
00295             afill = fill; *fill++ = 0; *fill++ = 1; FILLARG(fill)
00296             cfill = fill; *fill++ = 0;
00297             intr = eg->nodes[i]->intrct;
00298             for ( j = 0; j < model->interacts[intr]->nclist; j++ ) {
00299                 if ( model->interacts[intr]->clist[j] != 0 ) {
00300                     *fill++ = SYMBOL; *fill++ = 4;
00301                     *fill++ = m->couplings[j];
00302                     *fill++ = model->interacts[intr]->clist[j];
00303                 }
00304             }
00305             *fill++ = 1; *fill++ = 1; *fill++ = 3;
00306             *cfill = fill - cfill;
00307             *afill = fill - afill;
00308         }
00309         else {
00310             *fill++ = -SNUMBER;
00311             *fill++ = 1;
00312         }
00313 //
00314 //      Now the particles and their momenta.
00315 //
00316         for ( j = 0; j < numlegs; j++ ) {
00317             *fill++ = ARGHEAD+FUNHEAD+6;
00318             *fill++ = 0;
00319             FILLARG(fill)
00320             edge = eg->nodes[i]->edges[j];
00321             vect = ABS(edge)-1;
00322             *fill++ = 6+FUNHEAD;
00323             int a;
00324             if ( edge < 0 ) { a = ReConvertParticle(model,-eg->edges[vect]->ptcl); }
00325             else {            a = ReConvertParticle(model,eg->edges[vect]->ptcl); }
00326             *fill++ = a;
00327             *fill++ = FUNHEAD+2;
00328             FILLFUN(fill)
00329             *fill++ = edge < 0 ? -MINVECTOR: -VECTOR;
00330             if ( numlegs == 1 || vect < info->numextern ) { // Look up in set of external momenta
00331                 *fill++ = SetElements[Sets[info->externalset].first+vect];
00332             }
00333             else { // Look up in set of internal momenta set
00334                 *fill++ = SetElements[Sets[info->internalset].first+(vect-eg->nExtern)];
00335                 // determine the number of momenta required from internalset:
00336                 maxmom = MaX(maxmom, vect-eg->nExtern);
00337             }
00338             *fill++ = 1; *fill++ = 1; *fill++ = 3;
00339         }
00340         startfill[1] = fill-startfill;
00341       }
00342     }
00343     if ( ( info->flags & WITHEDGES ) == WITHEDGES ) {
00344         for ( i = 0; i < eg->nEdges; i++ ) {
00345             int n1 = eg->edges[i]->nodes[0];
00346             int n2 = eg->edges[i]->nodes[1];
00347 //          int l1 = eg->edges[i]->nlegs[0];
00348 //          int l2 = eg->edges[i]->nlegs[1];
00349
00350             startfill = fill;
00351             *fill++ = EDGE;
00352             *fill++ = 0;
00353             FILLFUN(fill)
00354             *fill++ = -SNUMBER; *fill++ = i+1; // number of the edge
00355 //
00356             *fill++ = ARGHEAD+FUNHEAD+6;
00357             *fill++ = 0;
00358             FILLARG(fill)
00359             *fill++ = 6+FUNHEAD;
00360             int a = ReConvertParticle(model,eg->edges[i]->ptcl);
00361             *fill++ = a;
00362             *fill++ = FUNHEAD+2;
00363             FILLFUN(fill)
00364                 *fill++ = -VECTOR;
00365             // Look up in set of internal momenta set
```

```
00366             if ( i < info->numextern ) { // Look up in set of external momenta
00367                 *fill++ = SetElements[Sets[info->externalset].first+i];
00368             }
00369             else { // Look up in set of internal momenta set
00370                 *fill++ = SetElements[Sets[info->internalset].first+(i-eg->nExtern)];
00371                 maxmom = MaX(maxmom, i-eg->nExtern);
00372             }
00373             *fill++ = 1; *fill++ = 1; *fill++ = 3;
00374 //
00375             *fill++ = -SNUMBER; *fill++ = n1+1; // number of the node from
00376             *fill++ = -SNUMBER; *fill++ = n2+1; // number of the node to
00377             startfill[1] = fill - startfill;
00378         }
00379     }
00380     if ( ( info->flags & WITHBLOCKS ) == WITHBLOCKS ) {
00381         for ( i = 0; i < eg->econn->nblocks; i++ ) {
00382             startfill = fill;
00383             *fill++ = BLOCK;
00384             *fill++ = 0;
00385             FILLFUN(fill);
00386             *fill++ = -SNUMBER;
00387             *fill++ = i+1;
00388             *fill++ = -SNUMBER;
00389             *fill++ = eg->econn->blocks[i].loop;
00390 //
00391 //          Now we have to make a list of all nodes inside this block
00392 //
00393             int bnodes[GRCC_MAXNODES], k;
00394             WORD *argfill = fill, *funfill;
00395             *fill++ = 0; *fill++ = 0; FILLARG(fill)
00396             *fill++ = 0;
00397             for ( k = 0; k < GRCC_MAXNODES; k++ ) bnodes[k] = 0;
00398             for ( k = 0; k < eg->econn->blocks[i].nmedges; k++ ) {
00399                 bnodes[eg->econn->blocks[i].edges[k][0]] = 1;
00400                 bnodes[eg->econn->blocks[i].edges[k][1]] = 1;
00401             }
00402             for ( k = 0; k < GRCC_MAXNODES; k++ ) {
00403                 if ( bnodes[k] == 0 ) continue;
00404 //
00405 //              Now we put the node inside this argument.
00406 //
00407                 funfill = fill;
00408                 *fill++ = NODEFUNCTION;
00409                 *fill++ = 0;
00410                 FILLFUN(fill)
00411                 *fill++ = -SNUMBER; *fill++ = k+1;
00412                 numlegs = eg->nodes[k]->deg;
00413                 for ( j = 0; j < numlegs; j++ ) {
00414                     edge = eg->nodes[k]->edges[j];
00415                     vect = ABS(edge)-1;
00416                     *fill++ = -VECTOR;
00417                     if ( numlegs == 1 || vect < info->numextern ) { // Look up in set of external
    momenta
00418                         *fill++ = SetElements[Sets[info->externalset].first+vect];
00419                     }
00420                     else { // Look up in set of internal momenta set
00421                         *fill++ = SetElements[Sets[info->internalset].first+(vect-info->numextern)];
00422                         maxmom = MaX(maxmom, vect-info->numextern);
00423                     }
00424                 }
00425                 funfill[1] = fill-funfill;
00426             }
00427             *fill++ = 1; *fill++ = 1; *fill++ = 3;
00428             *argfill = fill - argfill;
00429             argfill[ARGHEAD] = argfill[0] - ARGHEAD;
00430             startfill[1] = fill-startfill;
00431         }
00432     }
00433     if ( ( info->flags & WITHONEPISETS ) == WITHONEPISETS ) {
00434         for ( i = 0; i < eg->econn->nopic; i++ ) {
00435             startfill = fill;
00436             *fill++ = ONEPI;
00437             *fill++ = 0;
00438             FILLFUN(fill);
00439             *fill++ = -SNUMBER;
00440             *fill++ = i+1;
00441             *fill++ = -ONEPI;
00442             for ( j = 0; j < eg->econn->opics[i].nnodes; j++ ) {
00443                 *fill++ = -SNUMBER;
00444                 *fill++ = eg->econn->opics[i].nodes[j]+1;
00445             }
00446             startfill[1] = fill-startfill;
00447         }
00448     }
00449 //
00450 //  Topology counter. We have exaggerated a bit with the eye on the far future.
00451 //
```

```
00452     if ( info->numtopo-1 < MAXPOSITIVE ) {
00453         *fill++ = TOPO; *fill++ = FUNHEAD+2; FILLFUN(fill)
00454         *fill++ = -SNUMBER; *fill++ = (WORD)(info->numtopo-1);
00455     }
00456     else if ( info->numtopo-1 < FULLMAX-1 ) {
00457         *fill++ = TOPO; *fill++ = FUNHEAD+ARGHEAD+4; FILLFUN(fill)
00458         *fill++ = ARGHEAD+4; *fill++ = 0; FILLARG(fill)
00459         *fill++ = 4;
00460         *fill++ = (WORD)((info->numtopo-1) & WORDMASK);
00461         *fill++ = 1; *fill++ = 3;
00462     }
00463     else {  // for now: science fiction
00464         *fill++ = TOPO; *fill++ = FUNHEAD+ARGHEAD+6; FILLFUN(fill)
00465         *fill++ = ARGHEAD+6; *fill++ = 0; FILLARG(fill)
00466         *fill++ = 6; *fill++ = (WORD)((info->numtopo-1) >> BITSINWORD);
00467         *fill++ = (WORD)((info->numtopo-1) & WORDMASK);
00468         *fill++ = 0; *fill++ = 1; *fill++ = 5;
00469     }
00470 //
00471 //  Symmetry factors. We let Normalize do the multiplication.
00472 //
00473     if ( eg->nsym != 1 ) {
00474         *fill++ = SNUMBER; *fill++ = 4; *fill++ = (WORD)eg->nsym; *fill++ = -1;
00475     }
00476     if ( eg->esym != 1 ) {
00477         *fill++ = SNUMBER; *fill++ = 4; *fill++ = (WORD)eg->esym; *fill++ = -1;
00478     }
00479     if ( eg->extperm != 1 ) {
00480         *fill++ = SNUMBER; *fill++ = 4; *fill++ = (WORD)eg->extperm; *fill++ = 1;
00481     }
00482 //
00483 //  verify internalset has sufficient momenta:
00484 //
00485     if ( maxmom >= Sets[info->internalset].last - Sets[info->internalset].first ) {
00486         MLOCK(ErrorMessageLock);
00487         MesPrint("&Insufficient internal momenta in diagrams_");
00488         MUNLOCK(ErrorMessageLock);
00489         Terminate(-1);
00490     }
00491 //
00492 //  finish it off
00493 //
00494     while ( tail < tend ) *fill++ = *tail++;
00495     if ( eg->fsign < 0 ) fill[-1] = -fill[-1];
00496     *newterm = fill - newterm;
00497     AT.WorkPointer = fill;
00498
00499     Generator(BHEAD newterm,info->level);
00500     AT.WorkPointer = oldworkpointer;
00501 }
00502
00503 //  #] ProcessDiagram :
00504 //  #[ fendMG :
00505
00506 Bool fendMG(EGraph *eg, void *ti)
00507 {
00508     DUMMYUSE(eg);
00509     DUMMYUSE(ti);
00510     return True;
00511 }
00512
00513 //  #] fendMG :
00514 //  #[ ProcessTopology :
00515
00516 Bool ProcessTopology(EGraph *eg, void *ti)
00517 {
00518 //
00519 //  This routine is called for each new topology.
00520 //  New convention: return True;  generate the corresponding diagrams if needed
00521 //                  return False; skip diagram generation (when asked for).
00522 //
00523     TERMINFO *info = (TERMINFO *)ti;
00524
00525 // This seems to work properly. It was disabled before.
00526 #define WITHEARLYVETO
00527 #ifdef WITHEARLYVETO
00528     if ( ( ( info->flags & CHECKEXTERN ) == CHECKEXTERN ) && info->currentMODEL != NULL ) {
00529         int i, j;
00530         int numlegs, vect, edge;
00531         for ( i = 0; i < eg->nNodes; i++ ) {
00532             if ( eg->isExternal(i) ) continue;
00533             numlegs = eg->nodes[i]->deg;
00534             for ( j = 0; j < numlegs; j++ ) {
00535                 edge = eg->nodes[i]->edges[j];
00536                 vect = ABS(edge)-1;
00537                 if ( vect < info->numextern && info->legcouple[vect][numlegs] == 0 ) {
00538
```

```
00539 //                  This cannot be.
00540
00541                     info->numtopo++;
00542                     return False;
00543                 }
00544             }
00545         }
00546     }
00547 #endif
00548
00549     if ( ( info->flags & TOPOLOGIESONLY ) == 0 ) {
00550         info->numtopo++;
00551         return True;
00552     }
00553 //
00554 //  Now we are just generating topologies.
00555 //
00556     GETIDENTITY
00557     WORD *term = info->term, *newterm, *oldworkpointer = AT.WorkPointer;
00558     WORD *tdia = term + info->diaoffset;
00559     WORD *tail = tdia + tdia[1];
00560     WORD *tend = term + *term;
00561     WORD *fill, *startfill;
00562     Model *model = (Model *)info->currentModel;
00563     MODEL *m = (MODEL *)info->currentMODEL;
00564     int i, j;
00565     int numlegs, vect, edge, maxmom = 0;
00566
00567     newterm = term + *term;
00568     for ( i = 1; i < info->diaoffset; i++ ) newterm[i] = term[i];
00569     fill = newterm + info->diaoffset;
00570 //
00571 //  Now get the nodes
00572 //
00573     for ( i = 0; i < eg->nNodes; i++ ) {
00574 //
00575 //      node_(number,momentum_1,...,momentum_n)
00576 //
00577         numlegs = eg->nodes[i]->deg;
00578         startfill = fill;
00579         *fill++ = NODEFUNCTION;
00580         *fill++ = 0;
00581         FILLFUN(fill)
00582         *fill++ = -SNUMBER; *fill++ = i+1;
00583         if ( model != NULL && m != NULL ) {
00584             if ( !eg->isExternal(i) ) {
00585                 WORD *afill = fill; *fill++ = 0; *fill++ = 1; FILLARG(fill)
00586                 WORD *cfill = fill; *fill++ = 0;
00587                 int cpl = 2*eg->nodes[i]->extloop+eg->nodes[i]->deg-2;
00588                 *fill++ = SYMBOL; *fill++ = 4;
00589                 *fill++ = m->couplings[0];
00590                 *fill++ = cpl;
00591                 *fill++ = 1; *fill++ = 1; *fill++ = 3;
00592                 *cfill = fill - cfill;
00593                 *afill = fill - afill;
00594                 if ( *afill == ARGHEAD+8 && afill[ARGHEAD+4] == 1 ) {
00595                     fill = afill; *fill++ = -SYMBOL; *fill++ = afill[ARGHEAD+3];
00596                 }
00597             }
00598             else {
00599                 *fill++ = -SNUMBER;
00600                 *fill++ = 1;
00601             }
00602         }
00603 //
00604 //      Now the momenta.
00605 //
00606         for ( j = 0; j < numlegs; j++ ) {
00607             edge = eg->nodes[i]->edges[j];
00608             vect = ABS(edge)-1;
00609             *fill++ = edge < 0 ? -MINVECTOR: -VECTOR;
00610             if ( numlegs == 1 || vect < info->numextern ) { // Look up in set of external momenta
00611                 *fill++ = SetElements[Sets[info->externalset].first+vect];
00612             }
00613             else { // Look up in set of internal momenta set
00614                 *fill++ = SetElements[Sets[info->internalset].first+(vect-info->numextern)];
00615                 // determine the number of momenta required from internalset:
00616                 maxmom = MaX(maxmom, vect-info->numextern);
00617             }
00618         }
00619         startfill[1] = fill-startfill;
00620     }
00621     if ( ( info->flags & WITHEDGES ) == WITHEDGES ) {
00622         for ( i = 0; i < eg->nEdges; i++ ) {
00623             int n1 = eg->edges[i]->nodes[0];
00624             int n2 = eg->edges[i]->nodes[1];
00625 //          int l1 = eg->edges[i]->nlegs[0];
```

```
00626 //          int l2 = eg->edges[i]->nlegs[1];
00627             startfill = fill;
00628             *fill++ = EDGE;
00629             *fill++ = 0;
00630             FILLFUN(fill)
00631             *fill++ = -SNUMBER; *fill++ = i+1; // number of the edge
00632 //
00633             *fill++ = -VECTOR;
00634             if ( i < info->numextern ) { // Look up in set of external momenta
00635                 *fill++ = SetElements[Sets[info->externalset].first+i];
00636             }
00637             else { // Look up in set of internal momenta set
00638                 *fill++ = SetElements[Sets[info->internalset].first+(i-eg->nExtern)];
00639                 maxmom = MaX(maxmom, i-eg->nExtern);
00640             }
00641 //
00642             *fill++ = -SNUMBER; *fill++ = n1+1; // number of the node from
00643             *fill++ = -SNUMBER; *fill++ = n2+1; // number of the node to
00644             startfill[1] = fill - startfill;
00645         }
00646     }
00647 //
00648 //  Block information
00649 //
00650     if ( ( info->flags & WITHBLOCKS ) == WITHBLOCKS ) {
00651         for ( i = 0; i < eg->econn->nblocks; i++ ) {
00652             startfill = fill;
00653             *fill++ = BLOCK;
00654             *fill++ = 0;
00655             FILLFUN(fill);
00656             *fill++ = -SNUMBER;
00657             *fill++ = i+1;
00658             *fill++ = -SNUMBER;
00659             *fill++ = eg->econn->blocks[i].loop;
00660 //
00661 //          Now we have to make a list of all nodes inside this block
00662 //
00663             int bnodes[GRCC_MAXNODES], k;
00664             WORD *argfill = fill, *funfill;
00665             *fill++ = 0; *fill++ = 0; FILLARG(fill)
00666             *fill++ = 0;
00667             for ( k = 0; k < GRCC_MAXNODES; k++ ) bnodes[k] = 0;
00668             for ( k = 0; k < eg->econn->blocks[i].nmedges; k++ ) {
00669                 bnodes[eg->econn->blocks[i].edges[k][0]] = 1;
00670                 bnodes[eg->econn->blocks[i].edges[k][1]] = 1;
00671             }
00672             for ( k = 0; k < GRCC_MAXNODES; k++ ) {
00673                 if ( bnodes[k] == 0 ) continue;
00674 //
00675 //              Now we put the node inside this argument.
00676 //
00677                 funfill = fill;
00678                 *fill++ = NODEFUNCTION;
00679                 *fill++ = 0;
00680                 FILLFUN(fill)
00681                 *fill++ = -SNUMBER; *fill++ = k+1;
00682                 numlegs = eg->nodes[k]->deg;
00683                 for ( j = 0; j < numlegs; j++ ) {
00684                     edge = eg->nodes[k]->edges[j];
00685                     vect = ABS(edge)-1;
00686                     *fill++ = -VECTOR;
00687                     if ( numlegs == 1 || vect < info->numextern ) { // Look up in set of external
    momenta
00688                         *fill++ = SetElements[Sets[info->externalset].first+vect];
00689                     }
00690                     else { // Look up in set of internal momenta set
00691                         *fill++ = SetElements[Sets[info->internalset].first+(vect-info->numextern)];
00692                         maxmom = MaX(maxmom, vect-info->numextern);
00693                     }
00694                 }
00695                 funfill[1] = fill-funfill;
00696             }
00697             *fill++ = 1; *fill++ = 1; *fill++ = 3;
00698             *argfill = fill - argfill;
00699             argfill[ARGHEAD] = argfill[0] - ARGHEAD;
00700             startfill[1] = fill-startfill;
00701         }
00702
00703 //      if ( eg->econn->narticuls > 0 ) {
00704 //          startfill = fill;
00705 //          *fill++ = BLOCK;
00706 //          *fill++ = 0;
00707 //          FILLFUN(fill);
00708 //          for ( i = 0; i < eg->econn->snodes; i++ ) {
00709 //              if ( eg->econn->articuls[i] != 0 ) {
00710 //                  *fill++ = -SNUMBER;
00711 //                  *fill++ = i+1;
```

```
00712 //              }
00713 //          }
00714 //          startfill[1] = fill-startfill;
00715 //      }
00716     }
00717     if ( ( info->flags & WITHONEPISETS ) == WITHONEPISETS ) {
00718         for ( i = 0; i < eg->econn->nopic; i++ ) {
00719             startfill = fill;
00720             *fill++ = ONEPI;
00721             *fill++ = 0;
00722             FILLFUN(fill);
00723             *fill++ = -SNUMBER;
00724             *fill++ = i+1;
00725             *fill++ = -ONEPI;
00726             for ( j = 0; j < eg->econn->opics[i].nnodes; j++ ) {
00727                 *fill++ = -SNUMBER;
00728                 *fill++ = eg->econn->opics[i].nodes[j]+1;
00729             }
00730             startfill[1] = fill-startfill;
00731         }
00732     }
00733 //
00734 //  Topology counter. We have exaggerated a bit with the eye on the far future.
00735 //
00736     if ( info->numtopo < MAXPOSITIVE ) {
00737         *fill++ = TOPO; *fill++ = FUNHEAD+2; FILLFUN(fill)
00738         *fill++ = -SNUMBER; *fill++ = (WORD)(info->numtopo);
00739     }
00740     else if ( info->numtopo < FULLMAX-1 ) {
00741         *fill++ = TOPO; *fill++ = FUNHEAD+ARGHEAD+4; FILLFUN(fill)
00742         *fill++ = ARGHEAD+4; *fill++ = 0; FILLARG(fill)
00743         *fill++ = 4;
00744         *fill++ = (WORD)(info->numtopo & WORDMASK);
00745         *fill++ = 1; *fill++ = 3;
00746     }
00747     else {  // for now: science fiction
00748         *fill++ = TOPO; *fill++ = FUNHEAD+ARGHEAD+6; FILLFUN(fill)
00749         *fill++ = ARGHEAD+6; *fill++ = 0; FILLARG(fill)
00750         *fill++ = 6; *fill++ = (WORD)(info->numtopo » BITSINWORD);
00751         *fill++ = (WORD)(info->numtopo & WORDMASK);
00752         *fill++ = 0; *fill++ = 1; *fill++ = 5;
00753     }
00754 //
00755 //  verify internalset has sufficient momenta:
00756 //
00757     if ( maxmom >= Sets[info->internalset].last - Sets[info->internalset].first ) {
00758         MLOCK(ErrorMessageLock);
00759         MesPrint("&Insufficient internal momenta in diagrams_");
00760         MUNLOCK(ErrorMessageLock);
00761         Terminate(-1);
00762     }
00763 //
00764 //  finish it off
00765 //
00766     while ( tail < tend ) *fill++ = *tail++;
00767     if ( eg->fsign < 0 ) fill[-1] = -fill[-1];
00768     *newterm = fill - newterm;
00769     AT.WorkPointer = fill;
00770
00771     Generator(BHEAD newterm,info->level);
00772     AT.WorkPointer = oldworkpointer;
00773     info->numtopo++;
00774     return False;
00775 }
00776
00777 //  #] ProcessTopology :
00778 // #[ SetDualOpts :
00779 void SetDualOpts(int *opt, const WORD num, const int key, const char* key_name,
00780     const int dual, const char* dual_name, const int val, const int dval) {
00781
00782     if ( ( num & key ) == key ) {
00783         if ( ( num & dual ) == dual ) {
00784             MLOCK(ErrorMessageLock);
00785             MesPrint("&Conflicting diagram filters: %s and %s.", key_name, dual_name);
00786             MUNLOCK(ErrorMessageLock);
00787             Terminate(-1);
00788         }
00789         else {
00790             *opt = val;
00791         }
00792     }
00793     else {
00794         if ( ( num & dual ) == dual ) {
00795             *opt = dval;
00796         }
00797         else {
00798             // The default value is always 0.
```

```
00799             *opt = 0;
00800         }
00801     }
00802 }
00803 // #] SetDualOpts :
00804 //  #[ GenDiagrams :
00805
00806 int GenDiagrams(PHEAD WORD *term, WORD level)
00807 {
00808     Model *model;
00809     MODEL *m;
00810     Options *opt;
00811     Process *proc;
00812     int pid = 1, x;
00813     int babble = AC.GrccVerbose ? 2 : 0;
00814     TERMINFO info;
00815     WORD inset,outset,*coupl,setnum,optionnumber = 0;
00816     int i, j, cpl[GRCC_MAXNCPLG];
00817     int ninitl, initlPart[GRCC_MAXLEGS], nfinal, finalPart[GRCC_MAXLEGS];
00818     for ( i = 0; i < GRCC_MAXNCPLG; i++ ) cpl[i] = 0;
00819     std::map<int,int> momlist;
00820 //
00821 //  Here we create an object of type Option and load it up.
00822 //  Next we run the diagram generation on it.
00823 //
00824     info.term = term;
00825     info.level = level;
00826     info.diaoffset = AR.funoffset;
00827     info.externalset = term[info.diaoffset+FUNHEAD+7];
00828     info.internalset = term[info.diaoffset+FUNHEAD+9];
00829     info.flags = 0;
00830     inset = term[info.diaoffset+FUNHEAD+3];
00831     outset = term[info.diaoffset+FUNHEAD+5];
00832     coupl = term + info.diaoffset + FUNHEAD + 10;
00833     if ( *coupl < 0 ) {
00834         if ( term[info.diaoffset+1] > FUNHEAD + 12 ) {
00835             optionnumber = term[info.diaoffset+FUNHEAD+13];
00836         }
00837     }
00838     else {
00839         if ( term[info.diaoffset+1] > *coupl+FUNHEAD+10 )
00840             optionnumber = term[info.diaoffset+*coupl+FUNHEAD+11];
00841     }
00842     setnum = term[info.diaoffset+FUNHEAD+1];
00843
00844     m = AC.models[SetElements[Sets[setnum].first]];
00845     LoadModel(m);
00846     model = (Model *)m->grccmodel;
00847
00848     info.currentModel = (void *)model;
00849     info.currentMODEL = (void *)m;
00850     info.numdia = 0;
00851     info.numtopo = 1;
00852     info.flags = optionnumber;
00853
00854     opt = new Options();
00855
00856     opt->setOutAG(ProcessDiagram, &info);
00857     opt->setOutMG(ProcessTopology, &info);
00858
00859     opt->values[GRCC_OPT_SymmInitial] = ( optionnumber & WITHSYMMETRIZEI ) == WITHSYMMETRIZEI;
00860     opt->values[GRCC_OPT_SymmFinal]   = ( optionnumber & WITHSYMMETRIZEF ) == WITHSYMMETRIZEF;
00861
00862     // WITHBLOCKS controls output formatting. We could introduce an extra filtering option
00863     // corresponding to GRCC_OPT_Block, which is somewhat like Qgraf "onevi" but not quite
00864     // the same currently.
00865     //opt->values[GRCC_OPT_Block] = ;
00866
00867     // Now the "qgraf-compatible filtering options":
00868     int qgopt[GRCC_QGRAF_OPT_Size];
00869     SetDualOpts(&qgopt[GRCC_QGRAF_OPT_ONEPI],    optionnumber,ONEPARTI, "ONEPI_",    ONEPARTR,
    "ONEPR_",    1,-1);
00870     SetDualOpts(&qgopt[GRCC_QGRAF_OPT_ONSHELL],  optionnumber,ONSHELL,  "ONSHELL_",  OFFSHELL,
    "OFFSHELL_", 1,-1);
00871     SetDualOpts(&qgopt[GRCC_QGRAF_OPT_NOSIGMA],  optionnumber,NOSIGMA,  "NOSIGMA_",  SIGMA,
    "SIGMA_",    1,-1);
00872     SetDualOpts(&qgopt[GRCC_QGRAF_OPT_NOSNAIL],  optionnumber,NOSNAIL,  "NOSNAIL_",  SNAIL,
    "SNAIL_",    1,-1);
00873     SetDualOpts(&qgopt[GRCC_QGRAF_OPT_NOTADPOLE],optionnumber,NOTADPOLE,"NOTADPOLE_",TADPOLE ,
    "TADPOLE_",  1,-1);
00874     SetDualOpts(&qgopt[GRCC_QGRAF_OPT_SIMPLE],   optionnumber,SIMPLE,   "SIMPLE_",
    NOTSIMPLE,"NOTSIMPLE_",1,-1);
00875     SetDualOpts(&qgopt[GRCC_QGRAF_OPT_BIPART],   optionnumber,BIPART,   "BIPART_",
    NONBIPART,"NONBIPART_",1,-1);
00876     SetDualOpts(&qgopt[GRCC_QGRAF_OPT_CYCLI],    optionnumber,CYCLI,    "CYCLI_",    CYCLR,
    "CYCLR_",    1,-1);
00877     SetDualOpts(&qgopt[GRCC_QGRAF_OPT_FLOOP],    optionnumber,FLOOP,    "FLOOP_",    NOTFLOOP,
```

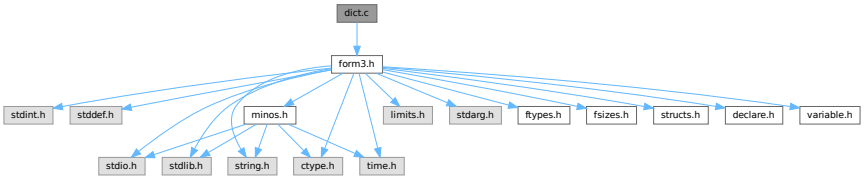
```
      "NOTFLOOP_", 1,-1);
00878     // Now set the options internally:
00879     opt->setQGrafOpt(qgopt);
00880 
00881     opt->setOutputF(False,"");
00882     opt->setOutputP(False,"");
00883     opt->printLevel(babble);
00884 
00885 //  Load the various arrays.
00886     ninitl = Sets[inset].last - Sets[inset].first;
00887     for ( i = 0; i < ninitl; i++ ) {
00888         x = SetElements[Sets[inset].first+i];
00889         initlPart[i] = ConvertParticle(model,x);
00890         info.legcouple[i] = m->vertices[numParticle(m,x)]->couplings;
00891     }
00892     nfinal = Sets[outset].last - Sets[outset].first;
00893     for ( i = 0; i < nfinal; i++ ) {
00894         x = SetElements[Sets[outset].first+i];
00895         finalPart[i] = ConvertParticle(model,x);
00896         info.legcouple[i+ninitl] = m->vertices[numParticle(m,x)]->couplings;
00897     }
00898     info.numextern = ninitl + nfinal;
00899     for ( i = 2; i <= MAXLEGS; i++ ) {
00900         if ( m->legcouple[i] == 1 ) {
00901             for ( j = 0; j < info.numextern; j++ ) {
00902                 if ( info.legcouple[j][i] == 0 ) { info.flags |= CHECKEXTERN; goto Go_on; }
00903             }
00904         }
00905     }
00906 
00907     // Check that we have sufficient external momenta in the set:
00908     if ( info.numextern > Sets[info.externalset].last - Sets[info.externalset].first ) {
00909         MLOCK(ErrorMessageLock);
00910         MesPrint("&Insufficient external momenta in diagrams_");
00911         MUNLOCK(ErrorMessageLock);
00912         Terminate(-1);
00913     }
00914 
00915     // Check that none of the supplied momenta are negative or repeated:
00916     for ( i = 0; i < Sets[info.externalset].last - Sets[info.externalset].first; i++ ) {
00917         const int momcode = SetElements[Sets[info.externalset].first + i];
00918         if ( momcode < AM.OffsetVector ) {
00919             MLOCK(ErrorMessageLock);
00920             MesPrint("&Invalid negative external momentum in diagrams_: -%s",
00921                 VARNAME(vectors, momcode+WILDMASK-AM.OffsetVector));
00922             MUNLOCK(ErrorMessageLock);
00923             Terminate(-1);
00924         }
00925         momlist[momcode]++;
00926         if ( momlist[momcode] != 1 ) {
00927             MLOCK(ErrorMessageLock);
00928             MesPrint("&Invalid repeated momentum in diagrams_: %s",
00929                 VARNAME(vectors, momcode-AM.OffsetVector));
00930             MUNLOCK(ErrorMessageLock);
00931             Terminate(-1);
00932         }
00933     }
00934     for ( i = 0; i < Sets[info.internalset].last - Sets[info.internalset].first; i++ ) {
00935         const int momcode = SetElements[Sets[info.internalset].first + i];
00936         if ( momcode < AM.OffsetVector ) {
00937             MLOCK(ErrorMessageLock);
00938             MesPrint("&Invalid negative internal momentum in diagrams_: -%s",
00939                 VARNAME(vectors, momcode+WILDMASK-AM.OffsetVector));
00940             MUNLOCK(ErrorMessageLock);
00941             Terminate(-1);
00942         }
00943         momlist[momcode]++;
00944         if ( momlist[momcode] != 1 ) {
00945             MLOCK(ErrorMessageLock);
00946             MesPrint("&Invalid repeated momentum in diagrams_: %s",
00947                 VARNAME(vectors, momcode-AM.OffsetVector));
00948             MUNLOCK(ErrorMessageLock);
00949             Terminate(-1);
00950         }
00951     }
00952 
00953 Go_on:;
00954 //
00955 //  Now we have to sort out the coupling constants.
00956 //  The argument at coupl can be of type -SNUMBER, -SYMBOL or generic
00957 //  It has however already be tested for syntax.
00958 //  Note that one cannot have 1 for the coupling constants.
00959 //  In that case one should select 0 loops or something equivalent.
00960 //
00961     if ( *coupl == -SNUMBER ) { // Number of loops
00962 //
00963 //      This is the complicated case.
```

```
00964 //      We have to compute the number of coupling constants and then
00965 //      generate diagrams for all combinations with the proper power.
00966 //
00967         int nc = coupl[1]*2 + ninitl + nfinal - 2;
00968         int *scratch = (int *)Malloc1(nc*sizeof(int),"DistrN");
00969         scratch[0] = -2; // indicating startup cq first call.
00970
00971         if ( ( info.flags & TOPOLOGIESONLY ) == 0 ) {
00972             while ( DistrN(nc,cpl,m->ncouplings,scratch) ) {
00973                 proc = new Process(pid, model, opt, ninitl, initlPart, nfinal, finalPart, cpl);
00974                 delete proc;
00975                 info.numtopo = 1;
00976             }
00977         }
00978         else {
00979             cpl[0] = nc;
00980             proc = new Process(pid, model, opt, ninitl, initlPart, nfinal, finalPart, cpl);
00981             delete proc;
00982         }
00983         M_free(scratch,"DistrN");
00984         opt->end();
00985         delete opt;
00986         return(0);
00987     }
00988     else if ( *coupl == -SYMBOL ) { // Just a single power of one constant
00989         for ( i = 0; i < m->ncouplings; i++ ) {
00990             if ( m->couplings[i] == coupl[1] ) {
00991                 cpl[i] = 1;
00992                 break;
00993             }
00994         }
00995     }
00996     else {  // One term with powers of coupling constants
00997         WORD *t, *tstop;
00998         t = coupl + ARGHEAD+3;
00999         tstop = coupl+*coupl; tstop -= ABS(tstop[-1]);
01000         while ( t < tstop ) {
01001             for ( i = 0; i < m->ncouplings; i++ ) {
01002                 if ( m->couplings[i] == *t ) {
01003                     cpl[i] = t[1];
01004                     break;
01005                 }
01006             }
01007             t += 2;
01008         }
01009     }
01010 /*
01011     And now the generation:
01012 */
01013     proc = new Process(pid, model, opt, ninitl, initlPart, nfinal, finalPart, cpl);
01014     opt->end();
01015     delete proc;
01016     delete opt;
01017     return(0);
01018 }
01019
01020 //  #] GenDiagrams :
01021 //  #[ processVertex :
01022
01023 //  Routine is to be used recursively to work its way through a list
01024 //  of possible vertices. The array of vertices is in TopoInf->vert
01025 //  with TopoInf->nvert the number of possible vertices.
01026 //  Currently we allow in TopoInf->vert only vertices with 3 or more edges.
01027 //
01028 //  We work with a point system. Each n-point vertex contributes n-2 points.
01029 //  When all points are assigned, we can call mgraph->generate().
01030 //
01031 //  The number of vertices of a given number of edges is stored in
01032 //  TopoInf->clnum[..] but the loop that determines how many there are
01033 //  may be limited by the corresponding element in TopoInf->vertmax[level]
01034
01035 int processVertex(TOPOTYPE *TopoInf, int pointsremaining, int level)
01036 {
01037     int i, j;
01038
01039     for ( i = pointsremaining, j = 0; i >= 0; i -= TopoInf->vert[level]-2, j++ ) {
01040         if ( TopoInf->vertmax && TopoInf->vertmax[level] >= 0
01041                     && j > TopoInf->vertmax[level] ) break;
01042         if ( i == 0 ) { // We got one!
01043             TopoInf->cldeg[TopoInf->ncl] = TopoInf->vert[level];
01044             TopoInf->clnum[TopoInf->ncl] = j;
01045             TopoInf->clext[TopoInf->ncl] = 0;
01046             TopoInf->ncl++;
01047
01048             MGraph *mgraph = new MGraph(1, TopoInf->ncl, TopoInf->cldeg,
01049                                 TopoInf->clnum, TopoInf->clext,
01050                                 TopoInf->cmind, TopoInf->cmaxd, TopoInf->opt);
```

```
01051
01052             mgraph->generate();
01053
01054             delete mgraph;
01055
01056             TopoInf->ncl--;
01057
01058             break;
01059         }
01060         if ( level < TopoInf->nvert-1 ) {
01061             if ( j > 0 ) {
01062                 TopoInf->cldeg[TopoInf->ncl] = TopoInf->vert[level];
01063                 TopoInf->clnum[TopoInf->ncl] = j;
01064                 TopoInf->clext[TopoInf->ncl] = 0;
01065                 TopoInf->ncl++;
01066             }
01067             if ( processVertex(TopoInf,i,level+1) < 0 ) return(-1);
01068             if ( j > 0 ) { TopoInf->ncl--; }
01069         }
01070     }
01071     return(0);
01072 }
01073
01074 //  #] processVertex :
01075
```

4.25 dict.c File Reference

`#include "form3.h"`
Include dependency graph for dict.c:



Functions

- void TransformRational (UWORD *a, WORD na)
- UBYTE * IsMultiplySign (void)
- UBYTE * IsExponentSign (void)
- UBYTE * FindSymbol (WORD num)
- UBYTE * FindVector (WORD num)
- UBYTE * FindIndex (WORD num)
- UBYTE * FindFunction (WORD num)
- UBYTE * FindFunWithArgs (WORD *t)
- UBYTE * FindExtraSymbol (WORD num)
- int FindDictionary (UBYTE *name)
- int AddDictionary (UBYTE *name)
- int AddToDictionary (DICTIONARY *dict, UBYTE *left, UBYTE *right)
- int UseDictionary (UBYTE *name, UBYTE *options)
- int SetDictionaryOptions (UBYTE *options)
- void UnSetDictionary (void)
- void RemoveDictionary (DICTIONARY *dict)
- void ShrinkDictionary (DICTIONARY *dict)
- int DoPreOpenDictionary (UBYTE *s)

- int DoPreCloseDictionary (UBYTE *s)
- int DoPreUseDictionary (UBYTE *s)
- int DoPreAdd (UBYTE *s)
- LONG DictToBytes (DICTIONARY *dict, UBYTE *buf)
- DICTIONARY * DictFromBytes (UBYTE *buf)

4.25.1 Detailed Description

Contains the code pertaining to dictionaries Commands are: #opendictionary name #closedictionary #selectdictionary name <options> There can be several dictionaries, but only one can be active. Defining elements is done with #add object: "replacement" Replacements are strings when a dictionary is for output translation. Objects can be 1: a number (rational) 2: a variable 3: * ^ 4: a function with arguments

Definition in file dict.c.

4.25.2 Function Documentation

4.25.2.1 TransformRational()

```
void TransformRational (
            UWORD * a,
            WORD na )
```

Definition at line 71 of file dict.c.

4.25.2.2 IsMultiplySign()

```
UBYTE * IsMultiplySign (
            void  )
```

Definition at line 218 of file dict.c.

4.25.2.3 IsExponentSign()

```
UBYTE * IsExponentSign (
            void  )
```

Definition at line 238 of file dict.c.

4.25.2.4 FindSymbol()

```
UBYTE * FindSymbol (
            WORD num )
```

Definition at line 258 of file dict.c.

4.25.2.5 FindVector()

```
UBYTE * FindVector (
            WORD num )
```

Definition at line 279 of file dict.c.

4.25.2.6 FindIndex()

```
UBYTE * FindIndex (
            WORD num )
```

Definition at line 301 of file dict.c.

4.25.2.7 FindFunction()

```
UBYTE * FindFunction (
            WORD num )
```

Definition at line 323 of file dict.c.

4.25.2.8 FindFunWithArgs()

```
UBYTE * FindFunWithArgs (
            WORD * t )
```

Definition at line 345 of file dict.c.

4.25.2.9 FindExtraSymbol()

```
UBYTE * FindExtraSymbol (
            WORD num )
```

Definition at line 377 of file dict.c.

4.25.2.10 FindDictionary()

```
int FindDictionary (
            UBYTE * name )
```

Definition at line 434 of file dict.c.

4.25.2.11 AddDictionary()

```
int AddDictionary (
            UBYTE * name )
```

Definition at line 449 of file dict.c.

4.25.2.12 AddToDictionary()

```
int AddToDictionary (
            DICTIONARY * dict,
            UBYTE * left,
            UBYTE * right )
```

Definition at line 491 of file dict.c.

4.25.2.13 UseDictionary()

```
int UseDictionary (
            UBYTE * name,
            UBYTE * options )
```

Definition at line 766 of file dict.c.

4.25.2.14 SetDictionaryOptions()

```
int SetDictionaryOptions (
            UBYTE * options )
```

Definition at line 790 of file dict.c.

4.25.2.15 UnSetDictionary()

```
void UnSetDictionary (
            void  )
```

Definition at line 883 of file dict.c.

4.25.2.16 RemoveDictionary()

```
void RemoveDictionary (
            DICTIONARY * dict )
```

Definition at line 902 of file dict.c.

4.25.2.17 ShrinkDictionary()

```
void ShrinkDictionary (
            DICTIONARY * dict )
```

Definition at line 945 of file dict.c.

4.25.2.18 DoPreOpenDictionary()

```
int DoPreOpenDictionary (
            UBYTE * s )
```

Definition at line 959 of file dict.c.

4.25.2.19 DoPreCloseDictionary()

```
int DoPreCloseDictionary (
            UBYTE * s )
```

Definition at line 998 of file dict.c.

4.25.2.20 DoPreUseDictionary()

```
int DoPreUseDictionary (
            UBYTE * s )
```

Definition at line 1022 of file dict.c.

4.25.2.21 DoPreAdd()

```
int DoPreAdd (
            UBYTE * s )
```

Definition at line 1067 of file dict.c.

4.25.2.22 DictToBytes()

```
LONG DictToBytes (
            DICTIONARY * dict,
            UBYTE * buf )
```

Definition at line 1119 of file dict.c.

4.25.2.23 DictFromBytes()

```
DICTIONARY * DictFromBytes (
            UBYTE * buf )
```

Definition at line 1152 of file dict.c.

4.26 dict.c

Go to the documentation of this file.

```
00001
00018 /* #[ License : */
00019 /*
00020  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00021  *   When using this file you are requested to refer to the publication
00022  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00023  *   This is considered a matter of courtesy as the development was paid
00024  *   for by FOM the Dutch physics granting agency and we would like to
00025  *   be able to track its scientific use to convince FOM of its value
00026  *   for the community.
00027  *
00028  *   This file is part of FORM.
00029  *
00030  *   FORM is free software: you can redistribute it and/or modify it under the
00031  *   terms of the GNU General Public License as published by the Free Software
00032  *   Foundation, either version 3 of the License, or (at your option) any later
00033  *   version.
00034  *
00035  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00036  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00037  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00038  *   details.
00039  *
00040  *   You should have received a copy of the GNU General Public License along
00041  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00042  */
00043 /* #] License : */
00044 /*
00045     #[ Includes : ratio.c
00046
00047     Data setup:
00048         AO.Dictionaries          Array of pointers to DICTIONARY
00049         AO.NumDictionaries
00050         AO.SizeDictionaries
00051         AO.CurrentDictionary
00052         AO.CurDictNumbers
00053         AO.CurDictVariables
00054         AO.CurDictSpecials
00055         AP.OpenDictionary
00056 */
00057
00058 #include "form3.h"
00059
00060 /*
00061     #] Includes :
00062     #[ TransformRational:
00063
00064     Tries to transform the rational number a according to the rules of
00065     the current dictionary. Whatever cannot be translated goes to the
00066     regular output.
00067     Options for AO.CurDictNumbers are:
00068         DICT_ALLNUMBERS, DICT_RATIONALONLY, DICT_INTEGERONLY, DICT_NONUMBERS
00069 */
00070
00071 void TransformRational(UWORD *a, WORD na)
00072 {
00073     DICTIONARY *dict;
00074     WORD i, j, nb, i1, i2; UWORD *b;
00075     if ( AO.CurrentDictionary <= 0 ) goto NoAction;
00076     dict = AO.Dictionaries[AO.CurrentDictionary-1];
00077     if ( na < 0 ) na = -na;
00078     switch ( AO.CurDictNumbers ) {
00079         case DICT_NONUMBERS:
00080             goto NoAction;
00081         case DICT_INTEGERONLY:
00082             if ( a[na] != 1 ) goto NoAction;
00083             if ( na > 1 ) {
00084                 for ( i = 1; i < na; i++ ) {
00085                     if ( a[na+i] != 0 ) goto NoAction;
00086                 }
00087             }
00088 Numeratoronly:;
00089             for ( i = dict->numelements-1; i >= 0; i-- ) {
00090                 if ( dict->elements[i]->type == DICT_INTEGERNUMBER ) {
00091                     if ( dict->elements[i]->size == na ) {
00092                         for ( j = 0; j < na; j++ ) {
00093                             if ( (UWORD)(dict->elements[i]->lhs[j]) != a[j] ) break;
00094                         }
00095                         if ( j == na ) {    /* Got it */
00096                             TokenToLine((UBYTE *)(dict->elements[i]->rhs));
00097                             return;
00098                         }
```

```
00099                   }
00100               }
00101           }
00102           goto NotFound;
00103       case DICT_RATIONALONLY:
00104           nb = 2*na;
00105           for ( i = dict->numelements-1; i >= 0; i-- ) {
00106               if ( dict->elements[i]->type == DICT_RATIONALNUMBER ) {
00107                   if ( dict->elements[i]->size == nb+2 ) {
00108                       for ( j = 0; j < nb; j++ ) {
00109                           if ( (UWORD)(dict->elements[i]->lhs[j+1]) != a[j] ) break;
00110                       }
00111                       if ( j == nb ) {    /* Got it */
00112                           TokenToLine((UBYTE *)(dict->elements[i]->rhs));
00113                           return;
00114                       }
00115                   }
00116               }
00117           }
00118           goto NotFound;
00119       case DICT_ALLNUMBERS:
00120 /*
00121           First fish for rationals
00122 */
00123           nb = 2*na;
00124           for ( i = dict->numelements-1; i >= 0; i-- ) {
00125               if ( dict->elements[i]->type == DICT_RATIONALNUMBER ) {
00126                   if ( dict->elements[i]->size == nb+2 ) {
00127                       for ( j = 0; j < nb; j++ ) {
00128                           if ( (UWORD)(dict->elements[i]->lhs[j+1]) != a[j] ) break;
00129                       }
00130                       if ( j == nb ) {    /* Got it */
00131                           TokenToLine((UBYTE *)(dict->elements[i]->rhs));
00132                           return;
00133                       }
00134                   }
00135               }
00136           }
00137 /*
00138           Now look for element[j1]/element[j2]
00139 */
00140           nb = na; b = a+na;
00141           while ( b[nb-1] == 0 ) nb--;
00142           if ( nb == 1 && b[0] == 1 ) goto Numeratoronly;
00143           while ( a[na-1] == 0 ) na--;
00144           for ( i1 = dict->numelements-1; i1 >= 0; i1-- ) {
00145               if ( dict->elements[i1]->type == DICT_INTEGERNUMBER ) {
00146                   if ( dict->elements[i1]->size == na ) {
00147                       for ( j = 0; j < na; j++ ) {
00148                           if ( (UWORD)(dict->elements[i1]->lhs[j]) != a[j] ) break;
00149                       }
00150                       if ( j == na ) break;
00151                   }
00152               }
00153           }
00154           for ( i2 = dict->numelements-1; i2 >= 0; i2-- ) {
00155               if ( dict->elements[i2]->type == DICT_INTEGERNUMBER ) {
00156                   if ( dict->elements[i2]->size == nb ) {
00157                       for ( j = 0; j < nb; j++ ) {
00158                           if ( (UWORD)(dict->elements[i2]->lhs[j]) != b[j] ) break;
00159                       }
00160                       if ( j == nb ) break;
00161                   }
00162               }
00163           }
00164           if ( i1 < 0 ) {
00165               if ( i2 < 0 ) goto NotFound;
00166               else {  /* number/replacement[i2] */
00167                   LongToLine(a,na);
00168                   if ( na > 1 || ( AO.DoubleFlag & 4 ) == 4 ) {
00169                       if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
00170                            || AC.OutputMode == CMODE ) {
00171                           if ( ( AO.DoubleFlag & 2 ) == 2 ) { AddToLine((UBYTE *)".Q0/"); }
00172                           else if ( ( AO.DoubleFlag & 1 ) == 1 ) { AddToLine((UBYTE *)".D0/"); }
00173                           else { AddToLine((UBYTE *)"/"); }
00174                       }
00175                   }
00176                   else AddToLine((UBYTE *)("/"));
00177                   TokenToLine((UBYTE *)(dict->elements[i2]->rhs));
00178               }
00179           }
00180           else if ( i2 < 0 ) { /* replacement[i1]/number */
00181               TokenToLine((UBYTE *)(dict->elements[i1]->rhs));
00182               AddToLine((UBYTE *)("/"));
00183               LongToLine((UWORD *)(b),nb);
00184               if ( nb > 1 || ( AO.DoubleFlag & 4 ) == 4 ) {
00185                   if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
```

```
00186                          || AC.OutputMode == CMODE ) {
00187                         if ( ( AO.DoubleFlag & 2 ) == 2 ) { AddToLine((UBYTE *)".Q0"); }
00188                         else if ( ( AO.DoubleFlag & 1 ) == 1 ) { AddToLine((UBYTE *)".D0"); }
00189                     }
00190                 }
00191             }
00192             else { /* replacement[i1]/replacement[i2] */
00193                 TokenToLine((UBYTE *)(dict->elements[i1]->rhs));
00194                 AddToLine((UBYTE *)("/"));
00195                 TokenToLine((UBYTE *)(dict->elements[i2]->rhs));
00196             }
00197             break;
00198         default:
00199             MesPrint("Illegal code in TransformRational: %d",AO.CurDictNumbers);
00200             Terminate(-1);
00201     }
00202     return;
00203 NotFound:
00204     if ( na != 1 || a[1] != 1 ) {
00205     if ( AO.CurDictNumberWarning ) {
00206         MesPrint("»»»»»Could not translate coefficient with dictionary %s«««««««",dict->name);
00207     } }
00208 NoAction:
00209     RatToLine(a,na);
00210     return;
00211 }
00212
00213 /*
00214     #] TransformRational:
00215     #[ IsMultiplySign:
00216 */
00217
00218 UBYTE *IsMultiplySign(void)
00219 {
00220     DICTIONARY *dict;
00221     int i;
00222     if ( AO.CurrentDictionary <= 0 ) return(0);
00223     dict = AO.Dictionaries[AO.CurrentDictionary-1];
00224     if ( dict->characters == 0 ) return(0);
00225     for ( i = dict->numelements-1; i >= 0; i-- ) {
00226         if ( ( dict->elements[i]->type == DICT_SPECIALCHARACTER )
00227             && ( dict->elements[i]->lhs[0] == (WORD)('*') ) )
00228         return((UBYTE *)(dict->elements[i]->rhs));
00229     }
00230     return(0);
00231 }
00232
00233 /*
00234     #] IsMultiplySign:
00235     #[ IsExponentSign:
00236 */
00237
00238 UBYTE *IsExponentSign(void)
00239 {
00240     DICTIONARY *dict;
00241     int i;
00242     if ( AO.CurrentDictionary <= 0 ) return(0);
00243     dict = AO.Dictionaries[AO.CurrentDictionary-1];
00244     if ( dict->characters == 0 ) return(0);
00245     for ( i = dict->numelements-1; i >= 0; i-- ) {
00246         if ( ( dict->elements[i]->type == DICT_SPECIALCHARACTER )
00247             && ( dict->elements[i]->lhs[0] == (WORD)('^') ) )
00248         return((UBYTE *)(dict->elements[i]->rhs));
00249     }
00250     return(0);
00251 }
00252
00253 /*
00254     #] IsExponentSign:
00255     #[ FindSymbol :
00256 */
00257
00258 UBYTE *FindSymbol(WORD num)
00259 {
00260     if ( AO.CurrentDictionary > 0 ) {
00261         DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];
00262         int i;
00263         if ( dict->variables > 0 && AO.CurDictVariables == DICT_DOVARIABLES ) {
00264             for ( i = dict->numelements-1; i >= 0; i-- ) {
00265                 if ( dict->elements[i]->type == DICT_SYMBOL &&
00266                      dict->elements[i]->lhs[0] == num )
00267                     return((UBYTE *)(dict->elements[i]->rhs));
00268             }
00269         }
00270     }
00271     return(VARNAME(symbols,num));
00272 }
```

```
00273
00274 /*
00275     #] FindSymbol :
00276     #[ FindVector :
00277 */
00278
00279 UBYTE *FindVector(WORD num)
00280 {
00281     if ( AO.CurrentDictionary > 0 ) {
00282         DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];
00283         int i;
00284         if ( dict->variables > 0 && AO.CurDictVariables == DICT_DOVARIABLES ) {
00285             for ( i = dict->numelements-1; i >= 0; i-- ) {
00286                 if ( dict->elements[i]->type == DICT_VECTOR &&
00287                      dict->elements[i]->lhs[0] == num )
00288                     return((UBYTE *)(dict->elements[i]->rhs));
00289             }
00290         }
00291     }
00292     num -= AM.OffsetVector;
00293     return(VARNAME(vectors,num));
00294 }
00295
00296 /*
00297     #] FindVector :
00298     #[ FindIndex :
00299 */
00300
00301 UBYTE *FindIndex(WORD num)
00302 {
00303     if ( AO.CurrentDictionary > 0 ) {
00304         DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];
00305         int i;
00306         if ( dict->variables > 0 && AO.CurDictVariables == DICT_DOVARIABLES ) {
00307             for ( i = dict->numelements-1; i >= 0; i-- ) {
00308                 if ( dict->elements[i]->type == DICT_INDEX &&
00309                      dict->elements[i]->lhs[0] == num )
00310                     return((UBYTE *)(dict->elements[i]->rhs));
00311             }
00312         }
00313     }
00314     num -= AM.OffsetIndex;
00315     return(VARNAME(indices,num));
00316 }
00317
00318 /*
00319     #] FindIndex :
00320     #[ FindFunction :
00321 */
00322
00323 UBYTE *FindFunction(WORD num)
00324 {
00325     if ( AO.CurrentDictionary > 0 ) {
00326         DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];
00327         int i;
00328         if ( dict->variables > 0 && AO.CurDictVariables == DICT_DOVARIABLES ) {
00329             for ( i = dict->numelements-1; i >= 0; i-- ) {
00330                 if ( dict->elements[i]->type == DICT_FUNCTION &&
00331                      dict->elements[i]->lhs[0] == num )
00332                     return((UBYTE *)(dict->elements[i]->rhs));
00333             }
00334         }
00335     }
00336     num -= FUNCTION;
00337     return(VARNAME(functions,num));
00338 }
00339
00340 /*
00341     #] FindFunction :
00342     #[ FindFunWithArgs :
00343 */
00344
00345 UBYTE *FindFunWithArgs(WORD *t)
00346 {
00347     if ( AO.CurrentDictionary > 0 ) {
00348         DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];
00349         int i, j;
00350         if ( dict->funwith > 0
00351                     && AO.CurDictFunWithArgs == DICT_DOFUNWITHARGS ) {
00352             for ( i = dict->numelements-1; i >= 0; i-- ) {
00353                 if ( dict->elements[i]->type == DICT_FUNCTION_WITH_ARGUMENTS &&
00354                      (WORD)(dict->elements[i]->lhs[0]) == t[0] &&
00355                      (WORD)(dict->elements[i]->lhs[1]) == t[1] ) {
00356                     for ( j = 2; j < t[1]; j++ ) {
00357                         if ( (WORD)(dict->elements[i]->lhs[j]) != t[j] ) break;
00358                     }
00359                     if ( j >= t[1] ) return((UBYTE *)(dict->elements[i]->rhs));
```

```
00360                 }
00361             }
00362         }
00363     }
00364     return(0);
00365 }
00366
00367 /*
00368     #] FindFunWithArgs :
00369     #[ FindExtraSymbol :
00370
00371     The extra symbol is constructed in the WorkSpace. This way we do not
00372     have to worry about Malloc and freeing the object later.
00373     The input value num is already the number of the extra symbol.
00374     We do NOT need num = MAXVARIABLES-num;
00375 */
00376
00377 UBYTE *FindExtraSymbol(WORD num)
00378 {
00379     GETIDENTITY;
00380     UBYTE *out = (UBYTE *)(AT.WorkPointer);
00381     *out = 0;
00382     if ( AO.CurrentDictionary > 0 ) {
00383         DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];
00384         int i;
00385         if ( dict->ranges > 0 && AO.CurDictVariables == DICT_DOVARIABLES ) {
00386             for ( i = dict->numelements-1; i >= 0; i-- ) {
00387                 if ( dict->elements[i]->type == DICT_RANGE
00388                     && num >= dict->elements[i]->lhs[0]
00389                     && num <= dict->elements[i]->lhs[1] ) {
00390 /*
00391                         Now we have to translate the rhs
00392                         %#  gives the number
00393                         %@  gives the number as its position in the range
00394 */
00395                         UBYTE *r = (UBYTE *)(dict->elements[i]->rhs);
00396                         while ( *r ) {
00397                             if ( *r == (UBYTE)'%' && ( r[1] == (UBYTE)'#'
00398                             || r[1] == (UBYTE)'@' ) ) {
00399                                 if ( r[1] == (UBYTE)'#' ) {
00400                                     out = NumCopy(num,out);
00401                                 }
00402                                 else {
00403                                     out = NumCopy(num-dict->elements[i]->lhs[0]+1,out);
00404                                 }
00405                                 r += 2;
00406                             }
00407                             else {
00408                                 *out++ = *r++;
00409                             }
00410                         }
00411                         *out = 0;
00412                         return((UBYTE *)(AT.WorkPointer));
00413                 }
00414             }
00415         }
00416     }
00417
00418     out = StrCopy((UBYTE *)AC.extrasym,out);
00419     if ( AC.extrasymbols == 0 ) {
00420         out = NumCopy(num,out);
00421         out = StrCopy((UBYTE *)"_",out);
00422     }
00423     else if ( AC.extrasymbols == 1 ) {
00424         out = AddArrayIndex(num,out);
00425     }
00426     return((UBYTE *)(AT.WorkPointer));
00427 }
00428
00429 /*
00430     #] FindExtraSymbol :
00431     #[ FindDictionary :
00432 */
00433
00434 int FindDictionary(UBYTE *name)
00435 {
00436     int i;
00437     for ( i = 0; i < AO.NumDictionaries; i++ ) {
00438         if ( StrCmp(AO.Dictionaries[i]->name,name) == 0 )
00439             return(i+1);
00440     }
00441     return(0);
00442 }
00443
00444 /*
00445     #] FindDictionary :
00446     #[ AddDictionary :
```

```
00447 */
00448
00449 int AddDictionary(UBYTE *name)
00450 {
00451     DICTIONARY *dict;
00452 /*
00453     First make space for the pointer in the list.
00454 */
00455     if ( AO.NumDictionaries >= AO.SizeDictionaries-1 ) {
00456         DICTIONARY **d;
00457         int i;
00458         if ( AO.SizeDictionaries <= 0 ) AO.SizeDictionaries = 10;
00459         else AO.SizeDictionaries = 2*AO.SizeDictionaries;
00460         d = (DICTIONARY **)Malloc1(AO.SizeDictionaries*sizeof(DICTIONARY *),"Dictionaries");
00461         for ( i = 0; i < AO.NumDictionaries; i++ ) d[i] = AO.Dictionaries[i];
00462         if ( AO.Dictionaries != 0 ) M_free(AO.Dictionaries,"Dictionaries");
00463         AO.Dictionaries = d;
00464     }
00465 /*
00466     Now create an empty dictionary.
00467 */
00468     dict = (DICTIONARY *)Malloc1(sizeof(DICTIONARY),"Dictionary");
00469     AO.Dictionaries[AO.NumDictionaries++] = dict;
00470     dict->elements = 0;
00471     dict->name = strDup1(name,"DictionaryName");
00472     dict->sizeelements = 0;
00473     dict->numelements = 0;
00474     dict->numbers = 0;
00475     dict->variables = 0;
00476     dict->characters = 0;
00477     dict->funwith = 0;
00478     dict->gnumelements = 0;
00479     dict->ranges = 0;
00480
00481     return(AO.NumDictionaries);
00482 }
00483
00484 /*
00485     #] AddDictionary :
00486     #[ AddToDictionary :
00487
00488         To be called from #add left:right
00489 */
00490
00491 int AddToDictionary(DICTIONARY *dict,UBYTE *left,UBYTE *right)
00492 {
00493     GETIDENTITY
00494     CBUF *C = cbuf+AC.cbufnum;
00495     WORD *w = AT.WorkPointer;
00496     WORD *OldWork = AT.WorkPointer;
00497     WORD *s, oldnumrhs = C->numrhs, oldnumlhs = C->numlhs;
00498     WORD *ow, *ww, *mm, oldEside, *where = 0, type, number, range[3];
00499     LONG oldcpointer;
00500     int error = 0, sizelhs, sizerhs, i, retcode;
00501     UBYTE *r;
00502     DICTIONARY_ELEMENT *new;
00503     WORD power = (WORD)('^'), times = (WORD)('*');
00504     if ( ( left[0] == '^' && left[1] == 0 )
00505       || ( left[0] == '*' && left[1] == '*' && left[2] == 0 ) ) {
00506         type = DICT_SPECIALCHARACTER;
00507         number = 1;
00508         where = &power;
00509         goto TestDouble;
00510     }
00511     else if ( left[0] == '*' && left[1] == 0 ) {
00512         type = DICT_SPECIALCHARACTER;
00513         number = 1;
00514         where = &times;
00515         goto TestDouble;
00516     }
00517     else if ( left[0] == '(' ) {  /* range of extra symbols */
00518         WORD x1 = 0, x2 = 0;
00519         r = left+1;
00520         while ( FG.cTable[*r] == 1 ) x1 = 10*x1 + *r++ - '0';
00521         if ( *r == ',' ) {
00522             r++;
00523             while ( FG.cTable[*r] == 1 ) x2 = 10*x2 + *r++ - '0';
00524         }
00525         else x2 = x1;
00526         number = 2;
00527         if ( *r != ')' ) {
00528             MesPrint("&Illegal range specification in LHS of %#add instruction.");
00529             return(1);
00530         }
00531         type = DICT_RANGE;
00532         if ( x1 <= 0 || x2 <= 0 || x1 > x2 ) {
00533             MesPrint("&Illegal range in LHS of %#add instruction.");
```

```
00534             return(1);
00535         }
00536         range[0] = x1;
00537         range[1] = x2;
00538         range[2] = 0;
00539         where = range;
00540         goto TestDouble;
00541     }
00542 /*
00543     Translate the left part. Determine type.
00544     We follow the code in CoIdExpression and then veto what we do not like.
00545     Just make sure to pop what needs to be popped in the compiler buffer.
00546 */
00547     AC.ProtoType = w;
00548     *w++ = SUBEXPRESSION;
00549     *w++ = SUBEXPSIZE;
00550     *w++ = C->numrhs+1;
00551     *w++ = 1;
00552     *w++ = AC.cbufnum;
00553     FILLSUB(w)
00554     AC.WildC = w;
00555     AC.NwildC = 0;
00556     AT.WorkPointer = s = w + 4*AM.MaxWildcards + 8;
00557 /*
00558     Now read the LHS
00559 */
00560     oldcpointer = AddLHS(AC.cbufnum) - C->Buffer;
00561
00562     if ( ( retcode = CompileAlgebra(left,LHSIDE,AC.ProtoType) ) < 0 ) { error = 1; }
00563     else AC.ProtoType[2] = retcode;
00564     AT.WorkPointer = s;
00565     if ( AC.NwildC && SortWild(w,AC.NwildC) ) error = 1;
00566
00567     OldWork[1] = AC.WildC-OldWork;
00568     w = AC.WildC;
00569     AT.WorkPointer = w;
00570     s = C->rhs[C->numrhs];
00571 /*
00572     We have the expression in the compiler buffers.
00573     The main level is at lhs[numlhs]
00574     The partial lhs (including ProtoType) is in OldWork (in WorkSpace)
00575     We need to load the result at w after the prototype
00576     Because these sort routines don't use the WorkSpace
00577     there should not be a conflict
00578 */
00579     if ( !error && *s == 0 ) {
00580 IllLeft:MesPrint("&Illegal LHS in dictionary");
00581         AC.lhdollarflag = 0;
00582         return(1);
00583     }
00584     if ( !error && *(s+*s) != 0 ) {
00585         MesPrint("&LHS in dictionary should be one term only");
00586         return(1);
00587     }
00588     if ( error == 0 ) {
00589         if ( NewSort(BHEAD0) || NewSort(BHEAD0) ) {
00590             if ( !error ) error = 1;
00591             return(error);
00592         }
00593         AN.RepPoint = AT.RepCount + 1;
00594         ow = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
00595         mm = s; ww = ow; i = *mm;
00596         while ( --i >= 0 ) {*ww++ = *mm++;} AT.WorkPointer = ww;
00597         AC.lhdollarflag = 0; oldEside = AR.Eside; AR.Eside = LHSIDE;
00598         AR.Cnumlhs = C->numlhs;
00599         if ( Generator(BHEAD ow,C->numlhs) ) {
00600             AR.Eside = oldEside;
00601             LowerSortLevel(); LowerSortLevel(); goto IllLeft;
00602         }
00603         AR.Eside = oldEside;
00604         AT.WorkPointer = w;
00605         if ( EndSort(BHEAD w,0) < 0 ) { LowerSortLevel(); goto IllLeft; }
00606         if ( *w == 0 || *(w+*w) != 0 ) {
00607             MesPrint("&LHS must be one term");
00608             AC.lhdollarflag = 0;
00609             return(1);
00610         }
00611         LowerSortLevel();
00612     }
00613     AT.WorkPointer = w + *w;
00614     AC.DumNum = 0;
00615 /*
00616     Everything is now after OldWork. We can pop the compilerbuffer.
00617     Next test for illegal things like a coefficient
00618     At this point we have:
00619     w = the term of the LHS
00620 */
```

```
00621     C->Pointer = C->Buffer + oldcpointer;
00622     C->numrhs = oldnumrhs;
00623     C->numlhs = oldnumlhs;
00624     AC.lhdollarflag = 0;
00625 /*
00626     Test for undesirables.
00627         1: wildcards
00628         2: sign
00629         3: more than one term
00630         4: composite terms
00631 */
00632     if ( AC.ProtoType[1] != SUBEXPSIZE ) {
00633         MesPrint("& Currently no wildcards allowed in dictionaries.");
00634         return(1);
00635     }
00636     if ( w[w[0]-1] < 0 ) {
00637         MesPrint("& Currently no sign allowed in dictionaries.");
00638         return(1);
00639     }
00640     if ( w[w[0]] != 0 ) {
00641         MesPrint("& More than one term in dictionary element.");
00642         return(1);
00643     }
00644     if ( w[0] == w[w[0]-1]+1 ) {    /* Only coefficient */
00645         WORD *numer, *denom;
00646         WORD nsize, dsize;
00647         nsize = dsize = (w[w[0]-1]-1)/2;
00648         numer = w+1;
00649         denom = numer+nsize;
00650         while ( numer[nsize-1] == 0 ) nsize--;
00651         while ( denom[dsize-1] == 0 ) dsize--;
00652         if ( dsize == 1 && denom[0] == 1 ) {
00653             type = DICT_INTEGERNUMBER;
00654             number = nsize;
00655             where = numer;
00656         }
00657         else {
00658             type = DICT_RATIONALNUMBER;
00659             number = w[0];
00660             where = w;
00661         }
00662     }
00663     else {
00664         s = w + w[0]-1;
00665         if ( s[0] != 3 || s[-1] != 1 || s[-2] != 1 ) {
00666 Compositeness:;
00667             MesPrint("& Currently no composite objects allowed in dictionaries.");
00668             return(1);
00669         }
00670         if ( w[0] != w[2]+4 ) goto Compositeness;
00671         s = w+1;
00672         switch ( *s ) {
00673             case SYMBOL:
00674                 if ( s[1] != 4 || s[3] != 1 ) goto Compositeness;
00675                 type = DICT_SYMBOL;
00676                 number = 1;
00677                 where = s+2;
00678                 break;
00679             case INDEX:
00680                 if ( s[1] != 3 ) goto Compositeness;
00681                 if ( s[2] < 0 ) type = DICT_VECTOR;
00682                 else            type = DICT_INDEX;
00683                 number = 1;
00684                 where = s+2;
00685                 break;
00686             default:
00687                 if ( *s < FUNCTION ) {
00688                     MesPrint("& Illegal object in dictionary.");
00689                     return(1);
00690                 }
00691                 if ( s[1] == FUNHEAD ) {
00692                     type = DICT_FUNCTION;
00693                     number = 1;
00694                     where = s;
00695                     break;
00696                 }
00697                 else {
00698                     type = DICT_FUNCTION_WITH_ARGUMENTS;
00699                     number = s[1];
00700                     where = s;
00701                 }
00702                 break;
00703         }
00704     }
00705 TestDouble:;
00706 /*
00707     Create a new element
```

```
00708 */
00709     if ( dict->numelements >= dict->sizeelements ) {
00710         DICTIONARY_ELEMENT **d;
00711         if ( dict->sizeelements <= 0 ) dict->sizeelements = 10;
00712         else                           dict->sizeelements *= 2;
00713         d = (DICTIONARY_ELEMENT **)Malloc1(
00714             sizeof(DICTIONARY_ELEMENT *)*dict->sizeelements,"Dictionary elements");
00715         for ( i = 0; i < dict->numelements; i++ )
00716             d[i] = dict->elements[i];
00717         if ( dict->elements ) M_free(dict->elements,"Dictionary elements");
00718         dict->elements = d;
00719     }
00720     sizelhs = number+1;
00721     sizerhs = 1; r = right; while ( *r++ ) sizerhs++;
00722     sizerhs = (sizerhs+sizeof(WORD)-1)/sizeof(WORD)+1;
00723     new = (DICTIONARY_ELEMENT *)Malloc1(sizeof(DICTIONARY_ELEMENT)
00724                 +sizeof(WORD)*(sizelhs+sizerhs),"Dictionary element");
00725     new->lhs = (WORD *)(new+1);
00726     new->rhs = new->lhs+sizelhs;
00727     new->type = type;
00728     new->size = number;
00729     for ( i = 0; i < number; i++ ) new->lhs[i] = where[i];
00730     new->lhs[i] = 0;
00731     r = (UBYTE *)(new->rhs);
00732     while ( *right ) {
00733         if ( *right == '\\' && ( right[1] == '`' || right[1] == '\"' ) ) right++;
00734         *r++ = *right++;
00735     }
00736     *r = 0;
00737
00738     dict->elements[dict->numelements++] = new;
00739
00740     switch ( type ) {
00741         case DICT_INTEGERNUMBER:
00742         case DICT_RATIONALNUMBER:
00743             dict->numbers++; break;
00744         case DICT_SYMBOL:
00745         case DICT_VECTOR:
00746         case DICT_INDEX:
00747         case DICT_FUNCTION:
00748             dict->variables++; break;
00749         case DICT_FUNCTION_WITH_ARGUMENTS:
00750             dict->funwith++; break;
00751         case DICT_SPECIALCHARACTER:
00752             dict->characters++; break;
00753         case DICT_RANGE:
00754             dict->ranges++; break;
00755     }
00756
00757     AT.WorkPointer = OldWork;
00758     return(0);
00759 }
00760
00761 /*
00762     #] AddToDictionary :
00763     #[ UseDictionary :
00764 */
00765
00766 int UseDictionary(UBYTE *name,UBYTE *options)
00767 {
00768     int i;
00769     for ( i = 0; i < AO.NumDictionaries; i++ ) {
00770         if ( StrCmp(AO.Dictionaries[i]->name,name) == 0 ) {
00771             AO.CurrentDictionary = i+1;
00772             if ( SetDictionaryOptions(options) < 0 ) {
00773                 AO.CurrentDictionary = 0;
00774                 return(-1);
00775             }
00776             else {  /* Now test whether what is requested is really there? */
00777                 return(0);
00778             }
00779         }
00780     }
00781     MesPrint("@There is no dictionary with the name %s",name);
00782     exit(-1);
00783 }
00784
00785 /*
00786     #] UseDictionary :
00787     #[ SetDictionaryOptions :
00788 */
00789
00790 int SetDictionaryOptions(UBYTE *options)
00791 {
00792     UBYTE *opt, *s, c;
00793     int retval = 0;
00794     s = options;
```

```
00795     AO.CurDictNumbers = DICT_ALLNUMBERS;
00796     AO.CurDictVariables = DICT_DOVARIABLES;
00797     AO.CurDictSpecials = DICT_DOSPECIALS;
00798     AO.CurDictFunWithArgs = DICT_DOFUNWITHARGS;
00799     AO.CurDictNumberWarning = 0;
00800     AO.CurDictNotInFunctions= 0;
00801     AO.CurDictInDollars = DICT_NOTINDOLLARS;
00802     while ( *s ) {
00803         opt = s;
00804         while ( *s && *s != ',' ) s++;
00805         c = *s; *s = 0;
00806         if ( opt[0] == '$' && opt[1] == 0 ) {
00807             AO.CurDictInDollars = DICT_INDOLLARS;
00808         }
00809         else if ( StrICmp(opt,(UBYTE *)"nonumbers") == 0 ) {
00810             AO.CurDictNumbers = DICT_NONUMBERS;
00811         }
00812         else if ( StrICmp(opt,(UBYTE *)"integersonly") == 0 ) {
00813             AO.CurDictNumbers = DICT_INTEGERONLY;
00814         }
00815         else if ( StrICmp(opt,(UBYTE *)"rationalsonly") == 0 ) {
00816             AO.CurDictNumbers = DICT_RATIONALONLY;
00817         }
00818         else if ( StrICmp(opt,(UBYTE *)"allnumbers") == 0 ) {
00819             AO.CurDictNumbers = DICT_ALLNUMBERS;
00820         }
00821         else if ( StrICmp(opt,(UBYTE *)"novariables") == 0 ) {
00822             AO.CurDictVariables = DICT_NOVARIABLES;
00823         }
00824         else if ( StrICmp(opt,(UBYTE *)"numbersonly") == 0 ) {
00825             AO.CurDictNumbers = DICT_ALLNUMBERS;
00826             AO.CurDictVariables = DICT_NOVARIABLES;
00827             AO.CurDictSpecials = DICT_NOSPECIALS;
00828             AO.CurDictFunWithArgs = DICT_NOFUNWITHARGS;
00829         }
00830         else if ( StrICmp(opt,(UBYTE *)"variablesonly") == 0 ) {
00831             AO.CurDictNumbers = DICT_NONUMBERS;
00832             AO.CurDictVariables = DICT_DOVARIABLES;
00833             AO.CurDictSpecials = DICT_NOSPECIALS;
00834             AO.CurDictFunWithArgs = DICT_NOFUNWITHARGS;
00835         }
00836         else if ( StrICmp(opt,(UBYTE *)"nospecials") == 0 ) {
00837             AO.CurDictSpecials = DICT_NOSPECIALS;
00838         }
00839         else if ( StrICmp(opt,(UBYTE *)"specialsonly") == 0 ) {
00840             AO.CurDictNumbers = DICT_NONUMBERS;
00841             AO.CurDictVariables = DICT_NOVARIABLES;
00842             AO.CurDictSpecials = DICT_DOSPECIALS;
00843             AO.CurDictFunWithArgs = DICT_NOFUNWITHARGS;
00844         }
00845         else if ( StrICmp(opt,(UBYTE *)"nofunwithargs") == 0 ) {
00846             AO.CurDictFunWithArgs = DICT_NOFUNWITHARGS;
00847         }
00848         else if ( StrICmp(opt,(UBYTE *)"funwithargsonly") == 0 ) {
00849             AO.CurDictNumbers = DICT_NONUMBERS;
00850             AO.CurDictVariables = DICT_NOVARIABLES;
00851             AO.CurDictSpecials = DICT_NOSPECIALS;
00852             AO.CurDictFunWithArgs = DICT_DOFUNWITHARGS;
00853         }
00854         else if ( StrICmp(opt,(UBYTE *)"warnings") == 0
00855                 || StrICmp(opt,(UBYTE *)"warning") == 0 ) {
00856             AO.CurDictNumberWarning = 1;
00857         }
00858         else if ( StrICmp(opt,(UBYTE *)"nowarnings") == 0
00859                 || StrICmp(opt,(UBYTE *)"nowarning") == 0 ) {
00860             AO.CurDictNumberWarning = 0;
00861         }
00862         else if ( StrICmp(opt,(UBYTE *)"infunctions") == 0 ) {
00863             AO.CurDictNotInFunctions= 0;
00864         }
00865         else if ( StrICmp(opt,(UBYTE *)"notinfunctions") == 0 ) {
00866             AO.CurDictNotInFunctions= 1;
00867         }
00868         else {
00869             MesPrint("@ Unrecognized option in %#SetDictionary: %s",opt);
00870             retval = -1;
00871         }
00872         *s = c;
00873         if ( c == ',' ) s++;
00874     }
00875     return(retval);
00876 }
00877
00878 /*
00879     #] SetDictionaryOptions :
00880     #[ UnSetDictionary :
00881 */
```

```
00882
00883 void UnSetDictionary(void)
00884 {
00885     AO.CurrentDictionary = 0;
00886     AO.CurDictNumbers = -1;
00887     AO.CurDictVariables = -1;
00888     AO.CurDictSpecials = -1;
00889     AO.CurDictFunWithArgs = -1;
00890     AO.CurDictFunWithArgs = -1;
00891     AO.CurDictNumberWarning = -1;
00892     AO.CurDictNotInFunctions= -1;
00893 }
00894
00895 /*
00896     #] UnSetDictionary :
00897     #[ RemoveDictionary :
00898
00899     Mostly needed for .clear
00900 */
00901
00902 void RemoveDictionary(DICTIONARY *dict)
00903 {
00904     int i;
00905     if ( dict == 0 ) return;
00906     for ( i = 0; i < AO.NumDictionaries; i++ ) {
00907         if ( AO.Dictionaries[i] == dict ) {
00908             for (i++; i < AO.NumDictionaries; i++ ) {
00909                 AO.Dictionaries[i-1] = AO.Dictionaries[i];
00910             }
00911             AO.NumDictionaries--;
00912             goto removeit;
00913         }
00914     }
00915     MesPrint("@ Dictionary not found in RemoveDictionary");
00916     exit(-1);
00917 removeit:;
00918     for ( i = 0; i < dict->numelements; i++ )
00919             M_free(dict->elements[i],"Dictionary element");
00920     for ( i = 0; i < dict->numelements; i++ ) dict->elements[i] = 0;
00921     if ( dict->elements ) M_free(dict->elements,"Dictionary elements");
00922     if ( dict->name ) {
00923         M_free(dict->name,"DictionaryName");
00924         dict->name = 0;
00925     }
00926     dict->sizeelements = 0;
00927     dict->numelements = 0;
00928     dict->numbers = 0;
00929     dict->variables = 0;
00930     dict->characters = 0;
00931     dict->funwith = 0;
00932     dict->gnumelements = 0;
00933     dict->ranges = 0;
00934 }
00935
00936 /*
00937     #] RemoveDictionary :
00938     #[ ShrinkDictionary :
00939
00940     To be called after a .store to restore the dictionary to the state
00941     it had at the last .global
00942     We do not make the elements array shorter.
00943 */
00944
00945 void ShrinkDictionary(DICTIONARY *dict)
00946 {
00947     while ( dict->numelements > dict->gnumelements ) {
00948         dict->numelements--;
00949         M_free(dict->elements[dict->numelements],"Dictionary element");
00950         dict->elements[dict->numelements] = 0;
00951     }
00952 }
00953
00954 /*
00955     #] ShrinkDictionary :
00956     #[ DoPreOpenDictionary :
00957 */
00958
00959 int DoPreOpenDictionary(UBYTE *s)
00960 {
00961     UBYTE *name;
00962     int dict;
00963     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
00964     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
00965     while ( *s == ' ' ) s++;
00966
00967     name = s; s = SkipAName(s);
00968     if ( *s != 0 && *s != ';' ) {
```

```
00969         MesPrint("@proper syntax is #opendictionary name");
00970         return(-1);
00971     }
00972     *s = 0;
00973
00974     if ( AP.OpenDictionary > 0 ) {
00975         MesPrint("@you cannot nest #opendictionary instructions");
00976         MesPrint("@dictionary %s is open already",
00977                 AO.Dictionaries[AP.OpenDictionary-1]->name);
00978         return(-1);
00979     }
00980     if ( AO.CurrentDictionary > 0 ) {
00981         MesPrint("@before opening a dictionary you have to first close the selected dictionary");
00982         return(-1);
00983     }
00984 /*
00985     Do we have this dictionary already?
00986 */
00987     dict = FindDictionary(name);
00988     if ( dict == 0 ) dict = AddDictionary(name);
00989     AP.OpenDictionary = dict;
00990     return(0);
00991 }
00992
00993 /*
00994     #] DoPreOpenDictionary :
00995     #[ DoPreCloseDictionary :
00996 */
00997
00998 int DoPreCloseDictionary(UBYTE *s)
00999 {
01000     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
01001     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
01002     while ( *s == ' ' ) s++;
01003
01004     if ( AP.OpenDictionary == 0 && AO.CurrentDictionary == 0 ) {
01005         MesPrint("@you have neither an open, nor a selected dictionary");
01006         return(-1);
01007     }
01008
01009     AP.OpenDictionary = 0;
01010     AO.CurrentDictionary = 0;
01011
01012     AO.CurDictNotInFunctions = 0;
01013
01014     return(0);
01015 }
01016
01017 /*
01018     #] DoPreCloseDictionary :
01019     #[ DoPreUseDictionary :
01020 */
01021
01022 int DoPreUseDictionary(UBYTE *s)
01023 {
01024     UBYTE *options, c, *ss, *sss, *name;
01025     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
01026     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
01027     while ( *s == ' ' ) s++;
01028
01029     if ( AP.OpenDictionary > 0 ) {
01030         MesPrint("@before selecting a dictionary you have to first close the open dictionary");
01031         return(-1);
01032     }
01033
01034     name = s; s = SkipAName(s);
01035     ss = s; while ( *s && *s != '(' ) s++;
01036     c = *ss; *ss = 0;
01037     if ( c == 0 ) {
01038         options = ss;
01039     }
01040     else {
01041         options = s+1; SKIPBRA3(s)
01042         if ( *s != ')' ) {
01043             MesPrint("@Irregular end of %#UseDictionary instruction");
01044             return(-1);
01045         }
01046         sss = s;
01047         s++; while ( *s == ' ' || *s == '\t' || *s == ';' ) s++;
01048         *sss = 0;
01049         if ( *s ) {
01050             MesPrint("@Irregular end of %#UseDictionary instruction");
01051             return(-1);
01052         }
01053     }
01054     return(UseDictionary(name,options));
01055 }
```

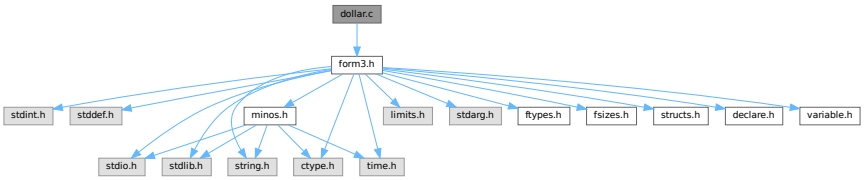
```
01056
01057 /*
01058     #] DoPreUseDictionary :
01059     #[ DoPreAdd :
01060
01061     Syntax:
01062         #add left :right
01063         #add left : "right"
01064     Adds to the currently open dictionary
01065 */
01066
01067 int DoPreAdd(UBYTE *s)
01068 {
01069     UBYTE *left, *right;
01070
01071     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
01072     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
01073     while ( *s == ' ' ) s++;
01074
01075     if ( AP.OpenDictionary == 0 ) {
01076         MesPrint("@there is no open dictionary to add to");
01077         return(-1);
01078     }
01079 /*
01080     Scan to the : and mark the left and right parts.
01081 */
01082     left = s;
01083     while ( *s && *s != ':' ) {
01084         if ( *s == '[' ) { SKIPBRA1(s) s++; }
01085         else if ( *s == '{' ) { SKIPBRA2(s) s++; }
01086         else if ( *s == '(' ) { SKIPBRA3(s) s++; }
01087         else if ( *s == ']' || *s == '}' || *s == ')' ) {
01088             MesPrint("@unmatched brackets in #add instruction");
01089             return(-1);
01090         }
01091         else s++;
01092     }
01093     if ( *s == 0 ) {
01094         MesPrint("@Missing : in #add instruction");
01095         return(-1);
01096     }
01097     *s++ = 0;
01098     right = s;
01099     while ( *s == ' ' || *s == '\t' ) s++;
01100     if ( *s == '"' && s[1] ) {
01101         right = s+1;
01102         s = s+2;
01103         while ( *s ) s++;
01104         while ( s[-1] != '"' ) s--;
01105         if ( s <= right ) {
01106             MesPrint("@Irregular use of double quotes in #add instruction");
01107             return(-1);
01108         }
01109         s[-1] = 0;
01110     }
01111     return(AddToDictionary(AO.Dictionaries[AP.OpenDictionary-1],left,right));
01112 }
01113
01114 /*
01115     #] DoPreAdd :
01116     #[ DictToBytes :
01117 */
01118 /* UNFINISHED_FEATURE_EXCL_START */
01119 LONG DictToBytes(DICTIONARY *dict,UBYTE *buf)
01120 {
01121     int numelements = dict->numelements, sizeelement, i, j, x;
01122     UBYTE *s1, *s2 = buf;
01123     DICTIONARY_ELEMENT *e;
01124 /*
01125     First copy the struct
01126 */
01127     s1 = (UBYTE *)dict; j = sizeof(DICTIONARY);
01128     NCOPY(s2,s1,j)
01129 /*
01130     Now the elements. Put a size indicator in front of each of them.
01131 */
01132     for ( i = 0; i < numelements; i++ ) {
01133         e = dict->elements[i];
01134         sizeelement = sizeof(DICTIONARY_ELEMENT)+(e->size+1)*sizeof(WORD);
01135         s1 = (UBYTE *)e->rhs; x = 0;
01136         while ( *s1 ) { s1++; x++; }
01137         x /= sizeof(WORD);
01138         sizeelement += (x+1) * sizeof(WORD);
01139         s1 = (UBYTE *)(&sizeelement); j = sizeof(WORD); NCOPY(s2,s1,j)
01140         s1 = (UBYTE *)e; j = sizeof(DICTIONARY_ELEMENT); NCOPY(s2,s1,j)
01141         s1 = (UBYTE *)e->lhs; j = (e->size+1)*(sizeof(WORD)); NCOPY(s2,s1,j)
01142         s1 = (UBYTE *)e->rhs; j = (x+1)*(sizeof(WORD)); NCOPY(s2,s1,j)
```

```
01143     }
01144     return(s2-buf);
01145 }
01146 /* UNFINISHED_FEATURE_EXCL_STOP */
01147 /*
01148     #] DictToBytes :
01149     #[ DictFromBytes :
01150 */
01151 /* UNFINISHED_FEATURE_EXCL_START */
01152 DICTIONARY *DictFromBytes(UBYTE *buf)
01153 {
01154     DICTIONARY *dict = Malloc1(sizeof(DICTIONARY),"Dictionary");
01155     UBYTE *s1, *s2;
01156     int i, j, sizeelement;
01157     DICTIONARY_ELEMENT *e;
01158 /*
01159     First read the dictionary itself
01160 */
01161     s1 = buf;
01162     s2 = (UBYTE *)dict; j = sizeof(DICTIONARY); NCOPY(s2,s1,j)
01163 /*
01164     Allocate the elements array:
01165 */
01166     dict->elements = (DICTIONARY_ELEMENT **)Malloc1(
01167             sizeof(DICTIONARY_ELEMENT *)*dict->sizeelements,"dictionary elements");
01168     for ( i = 0; i < dict->numelements; i++ ) {
01169         s2 = (UBYTE *)(&sizeelement); j = sizeof(WORD); NCOPY(s2,s1,j)
01170         e = (DICTIONARY_ELEMENT *)Malloc1(sizeelement*sizeof(UBYTE),"dictionary element");
01171         dict->elements[i] = e;
01172         j = sizeelement; s2 = (UBYTE *)e; NCOPY(s2,s1,j)
01173         e->lhs = (WORD *)(e+1);
01174         e->rhs = e->lhs + e->size+1;
01175     }
01176     return(dict);
01177 }
01178 /* UNFINISHED_FEATURE_EXCL_STOP */
01179 /*
01180     #] DictFromBytes :
01181 */
```

4.27 dollar.c File Reference

`#include "form3.h"`

Include dependency graph for dollar.c:



Macros

- #define STEP2

Functions

- int CatchDollar (int par)
- int AssignDollar (PHEAD WORD *term, WORD level)
- UBYTE * WriteDollarToBuffer (WORD numdollar, WORD par)
- UBYTE * WriteDollarFactorToBuffer (WORD numdollar, WORD numfac, WORD par)
- void AddToDollarBuffer (UBYTE *s)

- void TermAssign (WORD *term)
- int PutTermInDollar (WORD *term, WORD numdollar)
- void WildDollars (PHEAD WORD *term)
- WORD DolToTensor (PHEAD WORD numdollar)
- WORD DolToFunction (PHEAD WORD numdollar)
- WORD DolToVector (PHEAD WORD numdollar)
- WORD DolToNumber (PHEAD WORD numdollar)
- WORD DolToSymbol (PHEAD WORD numdollar)
- WORD DolToIndex (PHEAD WORD numdollar)
- DOLLARS DolToTerms (PHEAD WORD numdollar)
- LONG DolToLong (PHEAD WORD numdollar)
- int ExecInside (UBYTE *s)
- int InsideDollar (PHEAD WORD *ll, WORD level)
- void ExchangeDollars (int num1, int num2)
- LONG TermsInDollar (WORD num)
- LONG SizeOfDollar (WORD num)
- UBYTE * PreIfDollarEval (UBYTE *s, int *value)
- WORD * TranslateExpression (UBYTE *s)
- int IsSetMember (WORD *buffer, WORD numset)
- int IsMultipleOf (WORD *buf1, WORD *buf2)
- int TwoExprCompare (WORD *buf1, WORD *buf2, int oprtr)
- int DollarRaiseLow (UBYTE *name, LONG value)
- WORD EvalDoLoopArg (PHEAD WORD *arg, WORD par)
- WORD TestDoLoop (PHEAD WORD *lhsbuf, WORD level)
- WORD TestEndDoLoop (PHEAD WORD *lhsbuf, WORD level)
- int DollarFactorize (PHEAD WORD numdollar)
- void CleanDollarFactors (DOLLARS d)
- WORD * TakeDollarContent (PHEAD WORD *dollarbuffer, WORD **factor)
- WORD * MakeDollarInteger (PHEAD WORD *bufin, WORD **bufout)
- WORD * MakeDollarMod (PHEAD WORD *buffer, WORD **bufout)
- int GetDolNum (PHEAD WORD *t, WORD *tstop)
- void AddPotModdollar (WORD numdollar)

4.27.1 Detailed Description

The routines that deal with the dollar variables. The name administration is to be found in the file names.c

Definition in file dollar.c.

4.27.2 Macro Definition Documentation

4.27.2.1 STEP2

```
#define STEP2
```

Factors a dollar expression. Notation: d->nfactors becomes nonzero. if the number of factors is one, we leave d->factors zero. Otherwise factors is an array of pointers to the factors. These are pointers of the type FAC-DOLLAR. fd->where pointer to contents in term notation fd->size size of the buffer fd->where points to fd->type DOLNUMBER or DOLTERMS fd->value value if type is DOLNUMBER and it fits in a WORD.

Definition at line 2937 of file dollar.c.

4.27.3 Function Documentation

4.27.3.1 CatchDollar()

```
int CatchDollar (
            int par )
```

Definition at line 52 of file dollar.c.

4.27.3.2 AssignDollar()

```
int AssignDollar (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 207 of file dollar.c.

4.27.3.3 WriteDollarToBuffer()

```
UBYTE * WriteDollarToBuffer (
            WORD numdollar,
            WORD par )
```

Definition at line 581 of file dollar.c.

4.27.3.4 WriteDollarFactorToBuffer()

```
UBYTE * WriteDollarFactorToBuffer (
            WORD numdollar,
            WORD numfac,
            WORD par )
```

Definition at line 672 of file dollar.c.

4.27.3.5 AddToDollarBuffer()

```
void AddToDollarBuffer (
            UBYTE * s )
```

Definition at line 747 of file dollar.c.

4.27.3.6 TermAssign()

```
void TermAssign (
            WORD * term )
```

Definition at line 788 of file dollar.c.

4.27.3.7 PutTermInDollar()

```
int PutTermInDollar (
            WORD * term,
            WORD numdollar )
```

Definition at line 848 of file dollar.c.

4.27.3.8 WildDollars()

```
void WildDollars (
            PHEAD WORD * term )
```

Definition at line 876 of file dollar.c.

4.27.3.9 DolToTensor()

```
WORD DolToTensor (
            PHEAD WORD numdollar )
```

Definition at line 1069 of file dollar.c.

4.27.3.10 DolToFunction()

```
WORD DolToFunction (
            PHEAD WORD numdollar )
```

Definition at line 1130 of file dollar.c.

4.27.3.11 DolToVector()

```
WORD DolToVector (
            PHEAD WORD numdollar )
```

Definition at line 1187 of file dollar.c.

4.27.3.12 DolToNumber()

```
WORD DolToNumber (
            PHEAD WORD numdollar )
```

Definition at line 1251 of file dollar.c.

4.27.3.13 DolToSymbol()

```
WORD DolToSymbol (
            PHEAD WORD numdollar )
```

Definition at line 1310 of file dollar.c.

4.27.3.14 DolToIndex()

```
WORD DolToIndex (
            PHEAD WORD numdollar )
```

Definition at line 1364 of file dollar.c.

4.27.3.15 DolToTerms()

```
DOLLARS DolToTerms (
            PHEAD WORD numdollar )
```

Definition at line 1445 of file dollar.c.

4.27.3.16 DolToLong()

```
LONG DolToLong (
            PHEAD WORD numdollar )
```

Definition at line 1592 of file dollar.c.

4.27.3.17 ExecInside()

```
int ExecInside (
            UBYTE * s )
```

Definition at line 1667 of file dollar.c.

4.27.3.18 InsideDollar()

```
int InsideDollar (
            PHEAD WORD * ll,
            WORD level )
```

Definition at line 1732 of file dollar.c.

4.27.3.19 ExchangeDollars()

```
void ExchangeDollars (
            int num1,
            int num2 )
```

Definition at line 1831 of file dollar.c.

4.27.3.20 TermsInDollar()

```
LONG TermsInDollar (
            WORD num )
```

Definition at line 1849 of file dollar.c.

4.27.3.21 SizeOfDollar()

```
LONG SizeOfDollar (
            WORD num )
```

Definition at line 1898 of file dollar.c.

4.27.3.22 PreIfDollarEval()

```
UBYTE * PreIfDollarEval (
            UBYTE * s,
            int * value )
```

Definition at line 1960 of file dollar.c.

4.27.3.23 TranslateExpression()

```
WORD * TranslateExpression (
            UBYTE * s )
```

Definition at line 2142 of file dollar.c.

4.27.3.24 IsSetMember()

```
int IsSetMember (
            WORD * buffer,
            WORD numset )
```

Definition at line 2199 of file dollar.c.

4.27.3.25 IsMultipleOf()

```
int IsMultipleOf (
            WORD * buf1,
            WORD * buf2 )
```

Definition at line 2380 of file dollar.c.

4.27.3.26 TwoExprCompare()

```
int TwoExprCompare (
            WORD * buf1,
            WORD * buf2,
            int oprtr )
```

Definition at line 2455 of file dollar.c.

4.27.3.27 DollarRaiseLow()

```
int DollarRaiseLow (
            UBYTE * name,
            LONG value )
```

Definition at line 2536 of file dollar.c.

4.27.3.28 EvalDoLoopArg()

```
WORD EvalDoLoopArg (
            PHEAD WORD * arg,
            WORD par )
```

Evaluates one argument of a do loop. Such an argument is constructed from SNUMBERs DOLLAREXPRESSIONs and possibly DOLLAREXPR2s which indicate factors of the preceding dollar. Hence we have SNUMBER,num DOLLAREXPRESSION,numdollar DOLLAREXPRESSION,numdollar,DOLLAREXPR2,numfactor DOLLAREXPRESSION,numdollar,DOLLAREXPR2,numfactor,DOLLAREXPR2,numfactor etc. Because we have a do-loop at every stage we should have a number. The notation in DOLLAREXPR2 is that $>= 0$ is number of yet another dollar and < 0 is -n-1 with n the array element or zero. The return value is the (short) number. The routine works its way through the list in a recursive manner.

Definition at line 2635 of file dollar.c.

References EvalDoLoopArg().

Referenced by EvalDoLoopArg().

Here is the call graph for this function:

4.27.3.29 TestDoLoop()

```
WORD TestDoLoop (
            PHEAD WORD * lhsbuf,
            WORD level )
```

Definition at line 2746 of file dollar.c.

4.27.3.30 TestEndDoLoop()

```
WORD TestEndDoLoop (
            PHEAD WORD * lhsbuf,
            WORD level )
```

Definition at line 2824 of file dollar.c.

4.27.3.31 DollarFactorize()

```
int DollarFactorize (
            PHEAD WORD numdollar )
```

Definition at line 2939 of file dollar.c.

4.27.3.32 CleanDollarFactors()

```
void CleanDollarFactors (
            DOLLARS d )
```

Definition at line 3538 of file dollar.c.

4.27.3.33 TakeDollarContent()

```
WORD * TakeDollarContent (
            PHEAD WORD * dollarbuffer,
            WORD ** factor )
```

Definition at line 3560 of file dollar.c.

4.27.3.34 MakeDollarInteger()

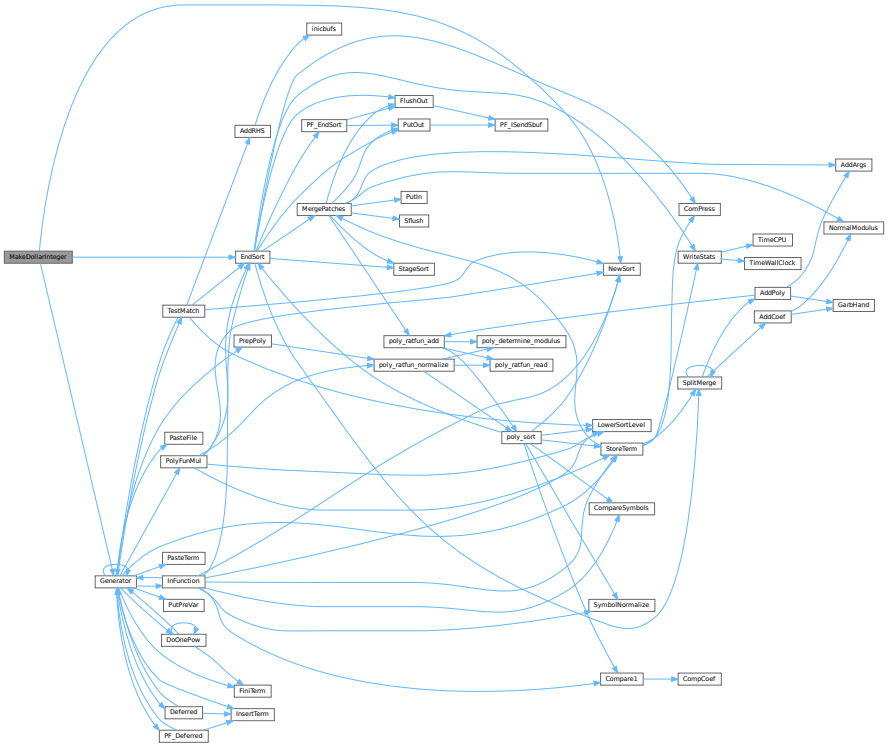
```
WORD * MakeDollarInteger (
            PHEAD WORD * bufin,
            WORD ** bufout )
```

For normalizing everything to integers we have to determine for all elements of this argument the LCM of the denominators and the GCD of the numerators. The input argument is in bufin. The number that comes out is the return value. The normalized argument is in bufout.

Definition at line 3612 of file dollar.c.

References EndSort(), Generator(), and NewSort().

Here is the call graph for this function:



4.27.3.35 MakeDollarMod()

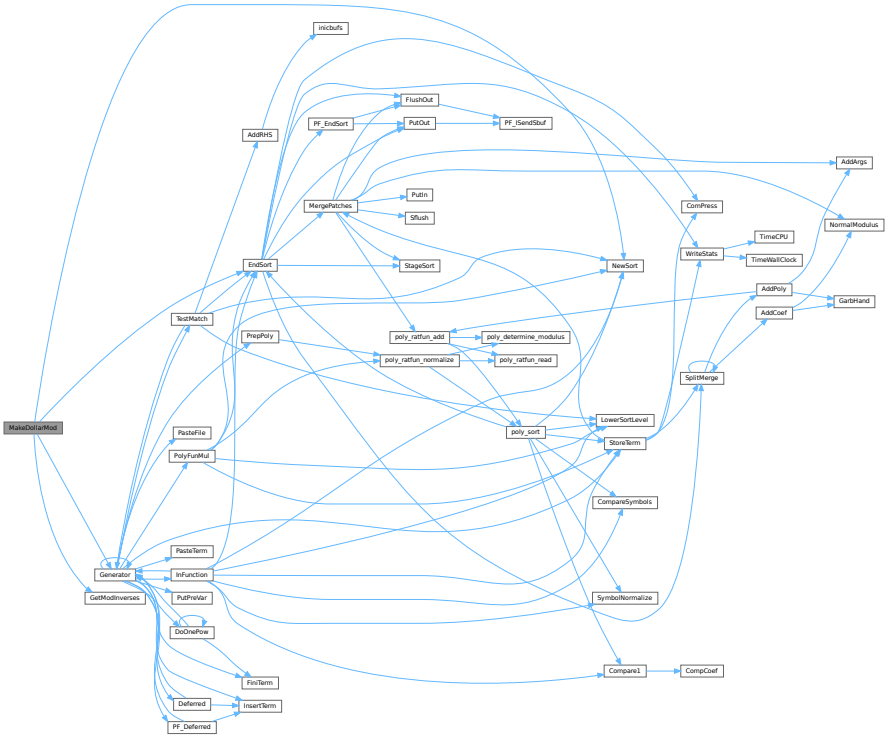
```
WORD * MakeDollarMod (
            PHEAD WORD * buffer,
            WORD ** bufout )
```

Similar to MakeDollarInteger but now with modulus arithmetic using only a one WORD 'prime'. We make the coefficient of the first term in the argument equal to one. Already the coefficients are taken modulus AN.cmod and AN.ncmod == 1

Definition at line 3786 of file dollar.c.

References EndSort(), Generator(), GetModInverses(), and NewSort().

Here is the call graph for this function:



4.27.3.36 GetDolNum()

```
int GetDolNum (
            PHEAD WORD * t,
            WORD * tstop )
```

Definition at line 3831 of file dollar.c.

4.27.3.37 AddPotModdollar()

```
void AddPotModdollar (
            WORD numdollar )
```

Adds a $-variable specified by *numdollar* to the list of potentially modified $-variables unless it has already been included in the list.

Parameters

numdollar	The index of the $-variable to be added.

Definition at line 3944 of file dollar.c.

4.28 dollar.c

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032 /*
00033     #[ Includes :
00034 */
00035
00036 #include "form3.h"
00037
00038 static UBYTE underscore[2] = {'_',0};
00039
00040 /*
00041     #] Includes :
00042     #[ CatchDollar :
00043
00044     Works out a dollar expression during compile type.
00045     Steals it from the buffer and puts it in an assignment.
00046     At the moment we should keep this inside the small buffer.
00047     Later with more sort buffers we can do this better.
00048     Par == 0 : regular assignment
00049     par == -1: after error. Just make zero for now.
00050 */
00051
00052 int CatchDollar(int par)
00053 {
00054     GETIDENTITY
00055     CBUF *C = cbuf + AC.cbufnum;
00056     int error = 0, numterms = 0, numdollar, resetmods = 0;
00057     LONG newsize, retval;
00058     WORD *w, *t, n, nsize, *oldwork = AT.WorkPointer, *dbuffer;
00059     WORD oldncmod = AN.ncmod;
00060     DOLLARS d;
00061     if ( AN.ncmod && ( ( AC.modmode & ALSODOLLARS ) == 0 ) ) AN.ncmod = 0;
00062     if ( AN.ncmod && AN.cmod == 0 ) { SetMods(); resetmods = 1; }
00063
00064     numdollar = C->lhs[C->numlhs][2];
00065
00066     d = Dollars+numdollar;
00067     if ( par == -1 ) {
00068         d->type = DOLUNDEFINED;
00069         cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00070         cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
00071         if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"$-buffer old");
00072         d->size = 0; d->where = &(AM.dollarzero);
00073         cbuf[AM.dbufnum].rhs[numdollar] = d->where;
00074         AN.ncmod = oldncmod;
00075         if ( resetmods ) UnSetMods();
00076         return(0);
00077     }
00078 #ifdef WITHMPI
00079     /*
00080      * The problem here is that only the master can make an assignment
00081      * like #$a=g; where g is an expression: only the master has an access to
00082      * the expression. So, in cases where the RHS contains expression names,
00083      * only the master invokes Generator() and then broadcasts the result to
00084      * the all slaves.
00085      * Broadcasting must be performed immediately; one cannot postpone it
00086      * to the end of the module because the dollar variable is visible
```

```
00087      * in the current module. For the same reason, this should be done
00088      * regardless of on/off parallel status.
00089      * If the RHS does not contain any expression names, it can be processed
00090      * in each slave.
00091      */
00092     if ( PF.me == MASTER || !AC.RhsExprInModuleFlag ) {
00093 #endif
00094
00095     EXCHINOUT
00096
00097     if ( NewSort(BHEAD0) ) { if ( !error ) error = 1; goto onerror; }
00098     if ( NewSort(BHEAD0) ) {
00099         LowerSortLevel();
00100         if ( !error ) error = 1;
00101         goto onerror;
00102     }
00103     AN.RepPoint = AT.RepCount + 1;
00104     w = C->rhs[C->lhs[C->numlhs][5]];
00105     while ( *w ) {
00106         n = *w; t = oldwork;
00107         NCOPY(t,w,n)
00108         AT.WorkPointer = t;
00109         AR.Cnumlhs = C->numlhs;
00110         if ( Generator(BHEAD oldwork,C->numlhs) ) { error = 1; break; }
00111     }
00112     AT.WorkPointer = oldwork;
00113     AN.tryterm = 0; /* for now */
00114     dbuffer = 0;
00115     if ( ( retval = EndSort(BHEAD (WORD *)((void *)(&dbuffer)),2) ) < 0 ) { error = 1; }
00116     LowerSortLevel();
00117     if ( retval <= 1 || dbuffer == 0 ) {
00118         d->type = DOLZERO;
00119         if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"$-buffer old");
00120         d->size = 0; d->where = &(AM.dollarzero);
00121         cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00122         cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
00123         goto docopy2;
00124     }
00125     w = dbuffer;
00126     if ( error == 0 )
00127         while ( *w ) { w += *w; numterms++; }
00128     else
00129         goto onerror;
00130     newsize = (w-dbuffer)+1;
00131 #ifdef WITHMPI
00132     }
00133     if ( AC.RhsExprInModuleFlag )
00134         /* PF_BroadcastPreDollar allocates dbuffer for slaves! */
00135         if ( (error = PF_BroadcastPreDollar(&dbuffer, &newsize, &numterms)) != 0 )
00136             goto onerror;
00137 #endif
00138     if ( newsize < MINALLOC ) newsize = MINALLOC;
00139     newsize = ((newsize+7)/8)*8;
00140     if ( numterms == 0 ) {
00141         d->type = DOLZERO;
00142         goto docopy;
00143     }
00144     else if ( numterms == 1 ) {
00145         t = dbuffer;
00146         n = *t;
00147         nsize = t[n-1];
00148         if ( nsize < 0 ) { nsize = -nsize; }
00149         if ( nsize == (n-1) ) { /* numerical */
00150             nsize = (nsize-1)/2;
00151             w = t + 1 + nsize;
00152             if ( *w != 1 ) goto doterms;
00153             w++; while ( w < ( t + n - 1 ) ) { if ( *w ) break; w++; }
00154             if ( w < ( t + n - 1 ) ) goto doterms;
00155             d->type = DOLNUMBER;
00156             goto docopy;
00157         }
00158         else if ( n == 7 && t[6] == 3 && t[5] == 1 && t[4] == 1
00159             && t[1] == INDEX && t[2] == 3 ) {
00160             d->type = DOLINDEX;
00161             d->index = t[3];
00162             goto docopy;
00163         }
00164         else goto doterms;
00165     }
00166     else {
00167 doterms:;
00168         d->type = DOLTERMS;
00169         cbuf[AM.dbufnum].CanCommu[numdollar] = numcommute(dbuffer,
00170                 &(cbuf[AM.dbufnum].NumTerms[numdollar]));
00171 docopy:;
00172         if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"$-buffer old");
00173         d->size = newsize; d->where = dbuffer;
```

```
00174 docopy2:;
00175         cbuf[AM.dbufnum].rhs[numdollar] = d->where;
00176     }
00177     if ( C->Pointer > C->rhs[C->numrhs] ) C->Pointer = C->rhs[C->numrhs];
00178     C->numlhs--; C->numrhs--;
00179 onerror:
00180 #ifdef WITHMPI
00181     if ( PF.me == MASTER || !AC.RhsExprInModuleFlag )
00182 #endif
00183     BACKINOUT
00184     AN.ncmod = oldncmod;
00185     if ( resetmods ) UnSetMods();
00186     return(error);
00187 }
00188
00189 /*
00190     #] CatchDollar :
00191     #[ AssignDollar :
00192
00193     To be called from Generator. Assigns an expression to a $ variable.
00194     This one is slightly different from CatchDollar.
00195     We have no easy buffer this time.
00196     We will have to hack our way using what we normally use for functions.
00197
00198     Note that in the threaded case we trust the user. That means that
00199     we are not going to recheck whether there is a maximum, minimum or sum.
00200     If the user says it is like that, we treat it like that.
00201     We only check that in this centralized version MODLOCAL isn't used.
00202
00203     In a later stage dtype could be used for actually checking MODMAX
00204     and MODMIN cases.
00205 */
00206
00207 int AssignDollar(PHEAD WORD *term, WORD level)
00208 {
00209     GETBIDENTITY
00210     CBUF *C = cbuf+AM.rbufnum;
00211     int numterms = 0, numdollar = C->lhs[level][2];
00212     LONG newsize;
00213     DOLLARS d = Dollars + numdollar;
00214     WORD *w, *t, n, nsize, *rh = cbuf[C->lhs[level][7]].rhs[C->lhs[level][5]];
00215     WORD *ss, *ww;
00216     WORD olddefer, oldcompress, oldncmod = AN.ncmod;
00217 #ifdef WITHPTHREADS
00218     int nummodopt, dtype = -1, dw;
00219     WORD numvalue;
00220     if ( AN.ncmod && ( ( AC.modmode & ALSODOLLARS ) == 0 ) ) AN.ncmod = 0;
00221     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
00222 /*
00223         Here we come only when the module runs with more than one thread.
00224         This must be a variable with a special module option.
00225         For the multi-threaded version we only allow MODSUM, MODMAX and MODMIN.
00226 */
00227         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
00228             if ( numdollar == ModOptdollars[nummodopt].number ) break;
00229         }
00230         if ( nummodopt >= NumModOptdollars ) {
00231             MLOCK(ErrorMessageLock);
00232             MesPrint("Illegal attempt to change $-variable in multi-threaded module %l",AC.CModule);
00233             MUNLOCK(ErrorMessageLock);
00234             Terminate(-1);
00235         }
00236         dtype = ModOptdollars[nummodopt].type;
00237         if ( DollarLocalCopy(dtype) ) {
00238             d = ModOptdollars[nummodopt].dstruct+AT.identity;
00239         }
00240     }
00241 #endif
00242     DUMMYUSE(term);
00243     w = rh;
00244 /*
00245     First some shortcuts
00246 */
00247     if ( *w == 0 ) {
00248 /*
00249         #[ Thread version : Zero case
00250 */
00251 #ifdef WITHPTHREADS
00252         if ( dtype > 0 ) {
00253             if ( ! DollarLocalCopy(dtype) ) { LOCK(d->pthreadslock); }
00254 NewValIsZero:;
00255             switch ( d->type ) {
00256                 case DOLZERO: goto NoChangeZero;
00257                 case DOLNUMBER:
00258                 case DOLTERMS:
00259                     if ( ( dw = d->where[0] ) > 0 && d->where[dw] != 0 ) {
00260                         break; /* was not a single number. Trust the user */
```

```
00261                     }
00262                     if ( dtype == MODMAX && d->where[dw-1] >= 0 ) goto NoChangeZero;
00263                     if ( dtype == MODMIN && d->where[dw-1] <= 0 ) goto NoChangeZero;
00264                     break;
00265                 default:
00266                     numvalue = DolToNumber(BHEAD numdollar);
00267                     if ( AN.ErrorInDollar != 0 ) break;
00268                     if ( dtype == MODMAX && numvalue >= 0 ) goto NoChangeZero;
00269                     if ( dtype == MODMIN && numvalue <= 0 ) goto NoChangeZero;
00270                     break;
00271             }
00272             d->type = DOLZERO;
00273             d->where[0] = 0;
00274             cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00275             cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
00276 NoChangeZero:;
00277             CleanDollarFactors(d);
00278             if ( ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
00279             AN.ncmod = oldncmod;
00280             return(0);
00281         }
00282 #endif
00283 /*
00284         #] Thread version :
00285 */
00286         d->type = DOLZERO;
00287         d->where[0] = 0;
00288         cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00289         cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
00290         CleanDollarFactors(d);
00291         AN.ncmod = oldncmod;
00292         return(0);
00293     }
00294     else if ( *w == 4 && w[4] == 0 && w[2] == 1 ) {
00295 /*
00296         #[ Thread version : New value is 'single precision'
00297 */
00298 #ifdef WITHPTHREADS
00299         if ( dtype > 0 ) {
00300             if ( ! DollarLocalCopy(dtype) ) { LOCK(d->pthreadslock); }
00301             if ( d->size < MINALLOC ) {
00302                 WORD oldsize, *oldwhere, i;
00303                 oldsize = d->size; oldwhere = d->where;
00304                 d->size = MINALLOC;
00305                 d->where = (WORD *)Malloc1(d->size*sizeof(WORD),"dollar contents");
00306                 cbuf[AM.dbufnum].rhs[numdollar] = d->where;
00307                 if ( oldsize > 0 ) {
00308                     for ( i = 0; i < oldsize; i++ ) d->where[i] = oldwhere[i];
00309                 }
00310                 else d->where[0] = 0;
00311                 if ( oldwhere && oldwhere != &(AM.dollarzero) ) M_free(oldwhere,"dollar contents");
00312             }
00313             switch ( d->type ) {
00314                 case DOLZERO:
00315 HandleDolZero:;
00316                     if ( dtype == MODMAX && w[3] <= 0 ) goto NoChangeOne;
00317                     if ( dtype == MODMIN && w[3] >= 0 ) goto NoChangeOne;
00318                     break;
00319                 case DOLNUMBER:
00320                 case DOLTERMS:
00321                     if ( ( dw = d->where[0] ) > 0 && d->where[dw] != 0 ) {
00322                         break; /* was not a single number. Trust the user */
00323                     }
00324                     if ( dtype == MODMAX && CompCoef(d->where,w) >= 0 ) goto NoChangeOne;
00325                     if ( dtype == MODMIN && CompCoef(d->where,w) <= 0 ) goto NoChangeOne;
00326                     break;
00327                 default:
00328                     {
00329 /*
00330                         Note that we convert the type for the next time around.
00331 */
00332                         WORD extraterm[4];
00333                         numvalue = DolToNumber(BHEAD numdollar);
00334                         if ( AN.ErrorInDollar != 0 ) break;
00335                         if ( numvalue == 0 ) {
00336                             d->type = DOLZERO;
00337                             d->where[0] = 0;
00338                             cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00339                             cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
00340                             goto HandleDolZero;
00341                         }
00342                         d->where[0] = extraterm[0] = 4;
00343                         d->where[1] = extraterm[1] = ABS(numvalue);
00344                         d->where[2] = extraterm[2] = 1;
00345                         d->where[3] = extraterm[3] = numvalue > 0 ? 3 : -3;
00346                         d->where[4] = 0;
00347                         d->type = DOLNUMBER;
```

```
00348                         if ( dtype == MODMAX && CompCoef(extraterm,w) >= 0 ) goto NoChangeOne;
00349                         if ( dtype == MODMIN && CompCoef(extraterm,w) <= 0 ) goto NoChangeOne;
00350                         break;
00351                     }
00352             }
00353             d->where[0] = w[0];
00354             d->where[1] = w[1];
00355             d->where[2] = w[2];
00356             d->where[3] = w[3];
00357             d->where[4] = 0;
00358             d->type = DOLNUMBER;
00359             cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00360             cbuf[AM.dbufnum].NumTerms[numdollar] = 1;
00361 NoChangeOne:;
00362             CleanDollarFactors(d);
00363             if ( ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
00364             AN.ncmod = oldncmod;
00365             return(0);
00366         }
00367 #endif
00368 /*
00369         #] Thread version :
00370 */
00371         if ( d->size < MINALLOC ) {
00372             if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollar contents");
00373             d->size = MINALLOC;
00374             d->where = (WORD *)Malloc1(d->size*sizeof(WORD),"dollar contents");
00375             cbuf[AM.dbufnum].rhs[numdollar] = d->where;
00376         }
00377         d->where[0] = w[0];
00378         d->where[1] = w[1];
00379         d->where[2] = w[2];
00380         d->where[3] = w[3];
00381         d->where[4] = 0;
00382         d->type = DOLNUMBER;
00383         cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00384         cbuf[AM.dbufnum].NumTerms[numdollar] = 1;
00385         CleanDollarFactors(d);
00386         AN.ncmod = oldncmod;
00387         return(0);
00388     }
00389 /*
00390     Now the real evaluation.
00391     We need to lock here before we work out the RHS, in case the RHS itself
00392     depends on the dollar variable.
00393 */
00394 #ifdef WITHPTHREADS
00395     if ( ! DollarLocalCopy(dtype) ) { LOCK(d->pthreadslock); }
00396 #endif
00397     CleanDollarFactors(d);
00398 /*
00399     The following case cannot occur. We treated it already
00400
00401     if ( *w == 0 ) {
00402         ss = 0; numterms = 0; newsize = 0;
00403         olddefer = AR.DeferFlag; AR.DeferFlag = 0;
00404         oldcompress = AR.NoCompress; AR.NoCompress = 1;
00405     }
00406     else
00407 */
00408     {
00409 /*
00410         New value is an expression that has to be evaluated first
00411         This is all generic. It won't foliate due to the sort level
00412 */
00413         if ( NewSort(BHEAD0) ) {
00414             AN.ncmod = oldncmod;
00415             return(1);
00416         }
00417         olddefer = AR.DeferFlag; AR.DeferFlag = 0;
00418         oldcompress = AR.NoCompress; AR.NoCompress = 1;
00419         while ( *w ) {
00420             n = *w; t = ww = AT.WorkPointer;
00421             NCOPY(t,w,n);
00422             AT.WorkPointer = t;
00423             if ( Generator(BHEAD ww,AR.Cnumlhs) ) {
00424                 AT.WorkPointer = ww;
00425                 LowerSortLevel();
00426                 AR.DeferFlag = olddefer;
00427                 AN.ncmod = oldncmod;
00428                 return(1);
00429             }
00430             AT.WorkPointer = ww;
00431         }
00432         AN.tryterm = 0; /* for now */
00433         if ( ( newsize = EndSort(BHEAD (WORD *)((void *)(&ss)),2) ) < 0 ) {
00434             AN.ncmod = oldncmod;
```

```
00435             return(1);
00436         }
00437         numterms = 0; t = ss; while ( *t ) { numterms++; t += *t; }
00438     }
00439
00440     if ( numterms == 0 ) {
00441 /*
00442         the new value evaluates to zero
00443 */
00444 #ifdef WITHPTHREADS
00445         if ( dtype == MODMAX || dtype == MODMIN ) {
00446             if ( ss ) { M_free(ss,"Sort of $"); ss = 0; }
00447             AR.DeferFlag = olddefer; AR.NoCompress = oldcompress;
00448             goto NewValIsZero;
00449         }
00450         else
00451 #endif
00452         {
00453           if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollar contents");
00454           d->where = &(AM.dollarzero);
00455           d->size = 0;
00456           cbuf[AM.dbufnum].rhs[numdollar] = 0;
00457           cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00458           cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
00459           d->type = DOLZERO;
00460         }
00461         if ( ss ) { M_free(ss,"Sort of $"); ss = 0; }
00462     }
00463     else {
00464 /*
00465         #[ Thread version :
00466 */
00467 #ifdef WITHPTHREADS
00468         if ( dtype == MODMAX || dtype == MODMIN ) {
00469             if ( numterms == 1 && ( *ss-1 == ABS(ss[*ss-1]) ) ) { /* is number */
00470               switch ( d->type ) {
00471                 case DOLZERO:
00472 HandleDolZero1:;
00473                     if ( dtype == MODMAX && ss[*ss-1] > 0 ) break;
00474                     if ( dtype == MODMIN && ss[*ss-1] < 0 ) break;
00475                     if ( ss ) { M_free(ss,"Sort of $"); ss = 0; }
00476                     AR.DeferFlag = olddefer; AR.NoCompress = oldcompress;
00477                     goto NoChange;
00478                 case DOLTERMS:
00479                 case DOLNUMBER:
00480                     if ( ( dw = d->where[0] ) > 0 && d->where[dw] != 0 ) break;
00481                     if ( dtype == MODMAX && CompCoef(ss,d->where) > 0 ) break;
00482                     if ( dtype == MODMIN && CompCoef(ss,d->where) < 0 ) break;
00483                     if ( ss ) { M_free(ss,"Sort of $"); ss = 0; }
00484                     AR.DeferFlag = olddefer; AR.NoCompress = oldcompress;
00485                     goto NoChange;
00486                 default: {
00487                     WORD extraterm[4];
00488                     numvalue = DolToNumber(BHEAD numdollar);
00489                     if ( AN.ErrorInDollar != 0 ) break;
00490                     if ( numvalue == 0 ) {
00491                         d->type = DOLZERO;
00492                         d->where[0] = 0;
00493                         cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00494                         cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
00495                         goto HandleDolZero1;
00496                     }
00497                     d->where[0] = extraterm[0] = 4;
00498                     d->where[1] = extraterm[1] = ABS(numvalue);
00499                     d->where[2] = extraterm[2] = 1;
00500                     d->where[3] = extraterm[3] = numvalue > 0 ? 3 : -3;
00501                     d->where[4] = 0;
00502                     d->type = DOLNUMBER;
00503                     if ( dtype == MODMAX && CompCoef(ss,extraterm) > 0 ) break;
00504                     if ( dtype == MODMIN && CompCoef(ss,extraterm) < 0 ) break;
00505                     if ( ss ) { M_free(ss,"Sort of $"); ss = 0; }
00506                     AR.DeferFlag = olddefer; AR.NoCompress = oldcompress;
00507                     goto NoChange;
00508                 }
00509               }
00510             }
00511             else {
00512                 if ( ss ) { M_free(ss,"Sort of $"); ss = 0; }
00513                 AR.DeferFlag = olddefer; AR.NoCompress = oldcompress;
00514                 goto NoChange;
00515             }
00516         }
00517 #endif
00518 /*
00519         #] Thread version :
00520 */
00521         d->type = DOLTERMS;
```

```
00522         if ( d->where && d->where != &(AM.dollarzero) ) { M_free(d->where,"dollar contents"); d->where
      = 0; }
00523         d->size = newsize + 1;
00524         d->where = ss;
00525         cbuf[AM.dbufnum].rhs[numdollar] = w = d->where;
00526     }
00527     AR.DeferFlag = olddefer; AR.NoCompress = oldcompress;
00528 /*
00529     Now find the special cases
00530 */
00531     if ( numterms == 0 ) {
00532         d->type = DOLZERO;
00533     }
00534     else if ( numterms == 1 ) {
00535         t = d->where;
00536         n = *t;
00537         nsize = t[n-1];
00538         if ( nsize < 0 ) { nsize = -nsize; }
00539         if ( nsize == (n-1) ) {
00540             nsize = (nsize-1)/2;
00541             w = t + 1 + nsize;
00542             if ( *w == 1 ) {
00543                 w++; while ( w < ( t + n - 1 ) ) { if ( *w ) break; w++; }
00544                 if ( w >= ( t + n - 1 ) ) d->type = DOLNUMBER;
00545             }
00546         }
00547         else if ( n == 7 && t[6] == 3 && t[5] == 1 && t[4] == 1
00548             && t[1] == INDEX && t[2] == 3 ) {
00549             d->type = DOLINDEX;
00550             d->index = t[3];
00551         }
00552     }
00553     if ( d->type == DOLTERMS ) {
00554         cbuf[AM.dbufnum].CanCommu[numdollar] = numcommute(d->where,
00555             &(cbuf[AM.dbufnum].NumTerms[numdollar]));
00556     }
00557     else {
00558         cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00559         cbuf[AM.dbufnum].NumTerms[numdollar] = 1;
00560     }
00561 #ifdef WITHPTHREADS
00562 NoChange:;
00563     if ( ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
00564 #endif
00565     AN.ncmod = oldncmod;
00566     return(0);
00567 }
00568
00569 /*
00570     #] AssignDollar :
00571     #[ WriteDollarToBuffer :
00572
00573     Takes the numbered dollar expression and writes it to output.
00574     We catch however the output in a buffer and return its address.
00575     This routine is needed when we need a text representation of
00576     a dollar expression like for the construction `$name' in the preprocessor.
00577     If par==0 we leave the current printing mode.
00578     If par==1 we insist on normal mode
00579 */
00580
00581 UBYTE *WriteDollarToBuffer(WORD numdollar, WORD par)
00582 {
00583     DOLLARS d = Dollars+numdollar;
00584     UBYTE *s, *oldcurbufwrt = AO.CurBufWrt;
00585     WORD *t, lbrac = 0, first = 0, arg[2], oldOutputMode = AC.OutputMode;
00586     WORD oldinfbrack = AO.InFbrack;
00587     int error = 0;
00588     int dict = AO.CurrentDictionary;
00589
00590     AO.DollarOutSizeBuffer = 32;
00591     AO.DollarOutBuffer = (UBYTE *)Malloc1(AO.DollarOutSizeBuffer,"DollarOutBuffer");
00592     AO.DollarInOutBuffer = 1;
00593     AO.PrintType = 1;
00594     AO.InFbrack = 0;
00595     s = AO.DollarOutBuffer;
00596     *s = 0;
00597     if ( par > 0 && AO.CurDictInDollars == 0 ) {
00598         AC.OutputMode = NORMALFORMAT;
00599         AO.CurrentDictionary = 0;
00600     }
00601     else {
00602         AO.CurBufWrt = (UBYTE *)underscore;
00603     }
00604     AO.OutInBuffer = 1;
00605     switch ( d->type ) {
00606         case DOLARGUMENT:
00607             WriteArgument(d->where);
```

```
00608             break;
00609         case DOLSUBTERM:
00610             WriteSubTerm(d->where,1);
00611             break;
00612         case DOLNUMBER:
00613         case DOLTERMS:
00614             t = d->where;
00615             while ( *t ) {
00616                 if ( WriteTerm(t,&lbrac,first,PRINTON,0) ) {
00617                     error = 1; break;
00618                 }
00619                 t += *t;
00620             }
00621             break;
00622         case DOLWILDARGS:
00623             t = d->where+1;
00624             while ( *t ) {
00625                 WriteArgument(t);
00626                 NEXTARG(t)
00627                 if ( *t ) TokenToLine((UBYTE *)(","));
00628             }
00629             break;
00630         case DOLINDEX:
00631             arg[0] = -INDEX; arg[1] = d->index;
00632             WriteArgument(arg);
00633             break;
00634         case DOLZERO:
00635             *s++ = '0'; *s = 0;
00636             AO.DollarInOutBuffer = 1;
00637             break;
00638         case DOLUNDEFINED:
00639             *s = 0;
00640             AO.DollarInOutBuffer = 1;
00641             break;
00642     }
00643     AC.OutputMode = oldOutputMode;
00644     AO.OutInBuffer = 0;
00645     AO.InFbrack = oldinfbrack;
00646     AO.CurBufWrt = oldcurbufwrt;
00647     AO.CurrentDictionary = dict;
00648     if ( error ) {
00649         MLOCK(ErrorMessageLock);
00650         MesPrint("&Illegal dollar object for writing");
00651         MUNLOCK(ErrorMessageLock);
00652         M_free(AO.DollarOutBuffer,"DollarOutBuffer");
00653         AO.DollarOutBuffer = 0;
00654         AO.DollarOutSizeBuffer = 0;
00655         return(0);
00656     }
00657     return(AO.DollarOutBuffer);
00658 }
00659
00660 /*
00661     #] WriteDollarToBuffer :
00662     #[ WriteDollarFactorToBuffer :
00663
00664     Takes the numbered dollar expression and writes it to output.
00665     We catch however the output in a buffer and return its address.
00666     This routine is needed when we need a text representation of
00667     a dollar expression like for the construction `$name' in the preprocessor.
00668     If par==0 we leave the current printing mode.
00669     If par==1 we insist on normal mode
00670 */
00671
00672 UBYTE *WriteDollarFactorToBuffer(WORD numdollar, WORD numfac, WORD par)
00673 {
00674     DOLLARS d = Dollars+numdollar;
00675     UBYTE *s, *oldcurbufwrt = AO.CurBufWrt;
00676     WORD *t, lbrac = 0, first = 0, n[5], oldOutputMode = AC.OutputMode;
00677     WORD oldinfbrack = AO.InFbrack;
00678     int error = 0;
00679     int dict = AO.CurrentDictionary;
00680
00681     if ( numfac > d->nfactors || numfac < 0 ) {
00682         MLOCK(ErrorMessageLock);
00683         MesPrint("&Illegal factor number for this dollar variable: %d",numfac);
00684         MesPrint("&There are %d factors",d->nfactors);
00685         MUNLOCK(ErrorMessageLock);
00686         return(0);
00687     }
00688
00689     AO.DollarOutSizeBuffer = 32;
00690     AO.DollarOutBuffer = (UBYTE *)Malloc1(AO.DollarOutSizeBuffer,"DollarOutBuffer");
00691     AO.DollarInOutBuffer = 1;
00692     AO.PrintType = 1;
00693     AO.InFbrack = 0;
00694     s = AO.DollarOutBuffer;
```

```
00695     *s = 0;
00696     if ( par > 0 ) {
00697         AC.OutputMode = NORMALFORMAT;
00698         AO.CurrentDictionary = 0;
00699     }
00700     else {
00701         AO.CurBufWrt = (UBYTE *)underscore;
00702     }
00703     AO.OutInBuffer = 1;
00704     if ( numfac == 0 ) {    /* write the number d->nfactors */
00705         n[0] = 4; n[1] = d->nfactors; n[2] = 1; n[3] = 3; n[4] = 0; t = n;
00706     }
00707     else if ( numfac ==  1 && d->factors == 0 ) {   /* Here d->factors is zero and d->where is fine */
00708         t = d->where;
00709     }
00710     else if ( d->factors[numfac-1].where == 0 ) {   /* write the value */
00711         if ( d->factors[numfac-1].value < 0 ) {
00712             n[0] = 4; n[1] = -d->factors[numfac-1].value; n[2] = 1; n[3] = -3; n[4] = 0; t = n;
00713         }
00714         else {
00715             n[0] = 4; n[1] = d->factors[numfac-1].value; n[2] = 1; n[3] = 3; n[4] = 0; t = n;
00716         }
00717     }
00718     else { t = d->factors[numfac-1].where; }
00719     while ( *t ) {
00720         if ( WriteTerm(t,&lbrac,first,PRINTON,0) ) {
00721             error = 1; break;
00722         }
00723         t += *t;
00724     }
00725     AC.OutputMode = oldOutputMode;
00726     AO.OutInBuffer = 0;
00727     AO.InFbrack = oldinfbrack;
00728     AO.CurBufWrt = oldcurbufwrt;
00729     AO.CurrentDictionary = dict;
00730     if ( error ) {
00731         MLOCK(ErrorMessageLock);
00732         MesPrint("&Illegal dollar object for writing");
00733         MUNLOCK(ErrorMessageLock);
00734         M_free(AO.DollarOutBuffer,"DollarOutBuffer");
00735         AO.DollarOutBuffer = 0;
00736         AO.DollarOutSizeBuffer = 0;
00737         return(0);
00738     }
00739     return(AO.DollarOutBuffer);
00740 }
00741
00742 /*
00743     #] WriteDollarFactorToBuffer :
00744     #[ AddToDollarBuffer :
00745 */
00746
00747 void AddToDollarBuffer(UBYTE *s)
00748 {
00749     int i;
00750     UBYTE *t = s, *u, *newdob;
00751     LONG j;
00752     while ( *t ) { t++; }
00753     i = t - s;
00754     while ( i + AO.DollarInOutBuffer >= AO.DollarOutSizeBuffer ) {
00755         j = AO.DollarInOutBuffer;
00756         AO.DollarOutSizeBuffer *= 2;
00757         t = AO.DollarOutBuffer;
00758         newdob = (UBYTE *)Malloc1(AO.DollarOutSizeBuffer,"DollarOutBuffer");
00759         u = newdob;
00760         while ( --j >= 0 ) *u++ = *t++;
00761         M_free(AO.DollarOutBuffer,"DollarOutBuffer");
00762         AO.DollarOutBuffer = newdob;
00763     }
00764     t = AO.DollarOutBuffer + AO.DollarInOutBuffer-1;
00765     while ( t == AO.DollarOutBuffer && ( *s == '+' || *s == ' ' ) ) s++;
00766     i = 0;
00767     if ( AO.CurrentDictionary == 0 ) {
00768         while ( *s ) {
00769             if ( *s == ' ' ) { s++; continue; }
00770             *t++ = *s++; i++;
00771         }
00772     }
00773     else {
00774         while ( *s ) { *t++ = *s++; i++; }
00775     }
00776     *t = 0;
00777     AO.DollarInOutBuffer += i;
00778 }
00779
00780 /*
00781     #] AddToDollarBuffer :
```

```
00782     #[ TermAssign :
00783
00784     This routine is called from a piece of code in Normalize that has been
00785     commented out.
00786 */
00787
00788 void TermAssign(WORD *term)
00789 {
00790     DOLLARS d;
00791     WORD *t, *tstop, *astop, *w, *m;
00792     WORD i, newsize;
00793     for (;;) {
00794         astop = term + *term;
00795         tstop = astop - ABS(astop[-1]);
00796         t = term + 1;
00797         while ( t < tstop ) {
00798             if ( *t == AM.termfunnum && t[1] == FUNHEAD+2
00799             && t[FUNHEAD] == -DOLLAREXPRESSION ) {
00800                 d = Dollars + t[FUNHEAD+1];
00801                 newsize = *term - FUNHEAD - 1;
00802                 if ( newsize < MINALLOC ) newsize = MINALLOC;
00803                 newsize = ((newsize+7)/8)*8;
00804                 if ( d->size > 2*newsize && d->size > 1000 ) {
00805                     if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollar
      contents");
00806                     d->size = 0;
00807                     d->where = &(AM.dollarzero);
00808                 }
00809                 if ( d->size < newsize ) {
00810                     if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollar
      contents");
00811                     d->size = newsize;
00812                     d->where = (WORD *)Malloc1(newsize*sizeof(WORD),"dollar contents");
00813                 }
00814                 cbuf[AM.dbufnum].rhs[t[FUNHEAD+1]] = w = d->where;
00815                 m = term;
00816                 while ( m < t ) *w++ = *m++;
00817                 m += t[1];
00818                 while ( m < tstop ) {
00819                     if ( *m == AM.termfunnum && m[1] == FUNHEAD+2
00820                     && m[FUNHEAD] == -DOLLAREXPRESSION ) { m += m[1]; }
00821                     else {
00822                         i = m[1];
00823                         while ( --i >= 0 ) *w++ = *m++;
00824                     }
00825                 }
00826                 while ( m < astop ) *w++ = *m++;
00827                 *(d->where) = w - d->where;
00828                 *w = 0;
00829                 d->type = DOLTERMS;
00830                 w = t; m = t + t[1];
00831                 while ( m < astop ) *w++ = *m++;
00832                 *term = w - term;
00833                 break;
00834             }
00835             t += t[1];
00836         }
00837         if ( t >= tstop ) return;
00838     }
00839 }
00840
00841 /*
00842     #] TermAssign :
00843     #[ PutTermInDollar :
00844
00845     We assume here that the dollar is local.
00846 */
00847
00848 int PutTermInDollar(WORD *term, WORD numdollar)
00849 {
00850     DOLLARS d = Dollars+numdollar;
00851     WORD i;
00852     if ( term == 0 || *term == 0 ) {
00853         d->type = DOLZERO;
00854         return(0);
00855     }
00856     if ( d->size < *term || d->size > 2*term[0] || d->where == 0 ) {
00857         if ( d->size > 0 && d->where ) {
00858             M_free(d->where,"dollar contents");
00859         }
00860         d->where = Malloc1((term[0]+1)*sizeof(WORD),"dollar contents");
00861         d->size = term[0]+1;
00862     }
00863     d->type = DOLTERMS;
00864     for ( i = 0; i < term[0]; i++ ) d->where[i] = term[i];
00865     d->where[i] = 0;
00866     return(0);
```

```
00867 }
00868
00869 /*
00870     #] PutTermInDollar :
00871     #[ WildDollars :
00872
00873     Note that we cannot upload wildcards into dollar variables when WITHPTHREADS.
00874 */
00875
00876 void WildDollars(PHEAD WORD *term)
00877 {
00878     GETBIDENTITY
00879     DOLLARS d;
00880     WORD *m, *t, *w, *ww, *orig = 0, *wildvalue, *wildstop;
00881     int numdollar;
00882     LONG weneed, i;
00883 #ifdef WITHPTHREADS
00884     int dtype = -1;
00885 #endif
00886     if ( term == 0 ) {
00887         m = wildvalue = AN.WildValue;
00888         wildstop = AN.WildStop;
00889     }
00890     else {
00891         ww = term + *term; ww -= ABS(ww[-1]); w = term+1;
00892         while ( w < ww && *w != SUBEXPRESSION ) w += w[1];
00893         if ( w >= ww ) return;
00894         wildstop = w + w[1];
00895         w += SUBEXPSIZE;
00896         wildvalue = m = w;
00897     }
00898     while ( m < wildstop ) {
00899         if ( *m != LOADDOLLAR ) { m += m[1]; continue; }
00900         t = m - 4;
00901         while ( *t == LOADDOLLAR || *t == FROMSET || *t == SETTONUM ) t -= 4;
00902         if ( t < wildvalue ) {
00903 /* INTERNAL_ERROR_EXCL_START */
00904             MLOCK(ErrorMessageLock);
00905             MesPrint("!>Serious bug in wildcard prototype. Found in WildDollars");
00906             MUNLOCK(ErrorMessageLock);
00907             Terminate(-1);
00908 /* INTERNAL_ERROR_EXCL_STOP */
00909         }
00910         numdollar = m[2];
00911         d = Dollars + numdollar;
00912 #ifdef WITHPTHREADS
00913         {
00914             int nummodopt;
00915             dtype = -1;
00916             if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
00917                 for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
00918                     if ( numdollar == ModOptdollars[nummodopt].number ) break;
00919                 }
00920                 if ( nummodopt < NumModOptdollars ) {
00921                     dtype = ModOptdollars[nummodopt].type;
00922                     if ( DollarLocalCopy(dtype) ) {
00923                         d = ModOptdollars[nummodopt].dstruct+AT.identity;
00924                     }
00925                     else {
00926                         MLOCK(ErrorMessageLock);
00927                         MesPrint("&Illegal attempt to use $-variable %s in module %l",
00928                             DOLLARNAME(Dollars,numdollar),AC.CModule);
00929                         MUNLOCK(ErrorMessageLock);
00930                         Terminate(-1);
00931                     }
00932                 }
00933             }
00934         }
00935 #endif
00936 /*
00937         The value of this wildcard goes into our $-variable
00938         First compute the space we need.
00939 */
00940         switch ( *t ) {
00941             case SYMTONUM:
00942                 weneed = 5;
00943                 break;
00944             case SYMTOSYM:
00945                 weneed = 9;
00946                 break;
00947             case SYMTOSUB:
00948             case VECTOSUB:
00949             case INDTOSUB:
00950                 orig = cbuf[AT.ebufnum].rhs[t[3]];
00951                 w = orig; while ( *w ) w += *w;
00952                 weneed = w - orig + 1;
00953                 break;
```

```
00954             case VECTOMIN:
00955             case VECTOVEC:
00956             case INDTOIND:
00957                 weneed = 8;
00958                 break;
00959             case FUNTOFUN:
00960                 weneed = FUNHEAD+5;
00961                 break;
00962             case ARGTOARG:
00963                 orig = cbuf[AT.ebufnum].rhs[t[3]];
00964                 if ( *orig > 0 ) weneed = *orig+2;
00965                 else {
00966                     w = orig+1; while ( *w ) { NEXTARG(w) }
00967                     weneed = w - orig + 1;
00968                 }
00969                 break;
00970             default:
00971                 weneed = MINALLOC;
00972                 break;
00973         }
00974         if ( weneed < MINALLOC ) weneed = MINALLOC;
00975         weneed = ((weneed+7)/8)*8;
00976         if ( d->size > 2*weneed && d->size > 1000 ) {
00977             if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollarspace");
00978             d->where = &(AM.dollarzero);
00979             d->size = 0;
00980         }
00981         if ( d->size < weneed ) {
00982             if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollarspace");
00983             d->where = (WORD *)Malloc1(weneed*sizeof(WORD),"dollarspace");
00984             d->size = weneed;
00985         }
00986 /*
00987         It is not clear what the following code does for TFORM
00988
00989         if ( dtype != MODLOCAL ) {
00990 */
00991             cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00992             cbuf[AM.dbufnum].NumTerms[numdollar] = 1;
00993 /*          cbuf[AM.dbufnum].rhs[numdollar] = d->where; */
00994             cbuf[AM.dbufnum].rhs[numdollar] = (WORD *)(1);
00995 /*
00996         }
00997         Now load up the value of the wildcard in compiler buffer format
00998 */
00999         w = d->where;
01000         d->type = DOLTERMS;
01001         switch ( *t ) {
01002             case SYMTONUM:
01003                 d->where[0] = 4; d->where[2] = 1;
01004                 if ( t[3] >= 0 ) { d->where[1] = t[3]; d->where[3] = 3; }
01005                 else { d->where[1] = -t[3]; d->where[3] = -3; }
01006                 if ( t[3] == 0 ) { d->type = DOLZERO; d->where[0] = 0; }
01007                 else { d->type = DOLNUMBER; d->where[4] = 0; }
01008                 break;
01009             case SYMTOSYM:
01010                 *w++ = 8;
01011                 *w++ = SYMBOL;
01012                 *w++ = 4;
01013                 *w++ = t[3];
01014                 *w++ = 1;
01015                 *w++ = 1;
01016                 *w++ = 1;
01017                 *w++ = 3;
01018                 *w = 0;
01019                 break;
01020             case SYMTOSUB:
01021             case VECTOSUB:
01022             case INDTOSUB:
01023                 while ( *orig ) {
01024                     i = *orig; while ( --i >= 0 ) *w++ = *orig++;
01025                 }
01026                 *w = 0;
01027 /*
01028                 And then we have to fix up CanCommu
01029 */
01030                 break;
01031             case VECTOMIN:
01032                 *w++ = 7; *w++ = INDEX; *w++ = 3; *w++ = t[3];
01033                 *w++ = 1; *w++ = 1; *w++ = -3; *w = 0;
01034                 break;
01035             case VECTOVEC:
01036                 *w++ = 7; *w++ = INDEX; *w++ = 3; *w++ = t[3];
01037                 *w++ = 1; *w++ = 1; *w++ = 3; *w = 0;
01038                 break;
01039             case INDTOIND:
01040                 d->type = DOLINDEX; d->index = t[3]; *w = 0;
```

```
01041                 break;
01042             case FUNTOFUN:
01043                 *w++ = FUNHEAD+4; *w++ = t[3]; *w++ = FUNHEAD;
01044                 FILLFUN(w)
01045                 *w++ = 1; *w++ = 1; *w++ = 3; *w = 0;
01046                 break;
01047             case ARGTOARG:
01048                 if ( *orig > 0 ) ww = orig + *orig + 1;
01049                 else {
01050                     ww = orig+1; while ( *ww ) { NEXTARG(ww) }
01051                 }
01052                 while ( orig < ww ) *w++ = *orig++;
01053                 *w = 0;
01054                 d->type = DOLWILDARGS;
01055                 break;
01056             default:
01057                 d->type = DOLUNDEFINED;
01058                 break;
01059         }
01060         m += m[1];
01061     }
01062 }
01063
01064 /*
01065     #] WildDollars :
01066     #[ DolToTensor :    with LOCK
01067 */
01068
01069 WORD DolToTensor(PHEAD WORD numdollar)
01070 {
01071     GETBIDENTITY
01072     DOLLARS d = Dollars + numdollar;
01073     WORD retval;
01074 #ifdef WITHPTHREADS
01075     int nummodopt, dtype = -1;
01076     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01077         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01078             if ( numdollar == ModOptdollars[nummodopt].number ) break;
01079         }
01080         if ( nummodopt < NumModOptdollars ) {
01081             dtype = ModOptdollars[nummodopt].type;
01082             if ( DollarLocalCopy(dtype) ) {
01083                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
01084             }
01085             else {
01086                 LOCK(d->pthreadslock);
01087             }
01088         }
01089     }
01090 #endif
01091     AN.ErrorInDollar = 0;
01092     if ( d->type == DOLTERMS && d->where[0] == FUNHEAD+4 &&
01093     d->where[FUNHEAD+4] == 0 && d->where[FUNHEAD+3] == 3 &&
01094     d->where[FUNHEAD+2] == 1 && d->where[FUNHEAD+1] == 1 &&
01095     d->where[1] >= FUNCTION && d->where[1] < FUNCTION+WILDOFFSET
01096     && functions[d->where[1]-FUNCTION].spec >= TENSORFUNCTION ) {
01097         retval = d->where[1];
01098     }
01099     else if ( d->type == DOLARGUMENT &&
01100     d->where[0] <= -FUNCTION && d->where[0] > -FUNCTION-WILDOFFSET
01101     && functions[-d->where[0]-FUNCTION].spec >= TENSORFUNCTION ) {
01102         retval = -d->where[0];
01103     }
01104     else if ( d->type == DOLWILDARGS && d->where[0] == 0
01105     && d->where[1] <= -FUNCTION && d->where[1] > -FUNCTION-WILDOFFSET
01106     && d->where[2] == 0
01107     && functions[-d->where[1]-FUNCTION].spec >= TENSORFUNCTION ) {
01108         retval = -d->where[1];
01109     }
01110     else if ( d->type == DOLSUBTERM &&
01111     d->where[0] >= FUNCTION && d->where[0] < FUNCTION+WILDOFFSET
01112     && functions[d->where[0]-FUNCTION].spec >= TENSORFUNCTION ) {
01113         retval = d->where[0];
01114     }
01115     else {
01116         AN.ErrorInDollar = 1;
01117         retval = 0;
01118     }
01119 #ifdef WITHPTHREADS
01120     if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
01121 #endif
01122     return(retval);
01123 }
01124
01125 /*
01126     #] DolToTensor :
01127     #[ DolToFunction :  with LOCK
```

```
01128 */
01129
01130 WORD DolToFunction(PHEAD WORD numdollar)
01131 {
01132     GETBIDENTITY
01133     DOLLARS d = Dollars + numdollar;
01134     WORD retval;
01135 #ifdef WITHPTHREADS
01136     int nummodopt, dtype = -1;
01137     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01138         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01139             if ( numdollar == ModOptdollars[nummodopt].number ) break;
01140         }
01141         if ( nummodopt < NumModOptdollars ) {
01142             dtype = ModOptdollars[nummodopt].type;
01143             if ( DollarLocalCopy(dtype) ) {
01144                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
01145             }
01146             else {
01147                 LOCK(d->pthreadslock);
01148             }
01149         }
01150     }
01151 #endif
01152     AN.ErrorInDollar = 0;
01153     if ( d->type == DOLTERMS && d->where[0] == FUNHEAD+4 &&
01154     d->where[FUNHEAD+4] == 0 && d->where[FUNHEAD+3] == 3 &&
01155     d->where[FUNHEAD+2] == 1 && d->where[FUNHEAD+1] == 1 &&
01156     d->where[1] >= FUNCTION && d->where[1] < FUNCTION+WILDOFFSET ) {
01157         retval = d->where[1];
01158     }
01159     else if ( d->type == DOLARGUMENT &&
01160     d->where[0] <= -FUNCTION && d->where[0] > -FUNCTION-WILDOFFSET ) {
01161         retval = -d->where[0];
01162     }
01163     else if ( d->type == DOLWILDARGS && d->where[0] == 0
01164     && d->where[1] <= -FUNCTION && d->where[1] > -FUNCTION-WILDOFFSET
01165     && d->where[2] == 0 ) {
01166         retval = -d->where[1];
01167     }
01168     else if ( d->type == DOLSUBTERM &&
01169     d->where[0] >= FUNCTION && d->where[0] < FUNCTION+WILDOFFSET ) {
01170         retval = d->where[0];
01171     }
01172     else {
01173         AN.ErrorInDollar = 1;
01174         retval = 0;
01175     }
01176 #ifdef WITHPTHREADS
01177     if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
01178 #endif
01179     return(retval);
01180 }
01181
01182 /*
01183     #] DolToFunction :
01184     #[ DolToVector :    with LOCK
01185 */
01186
01187 WORD DolToVector(PHEAD WORD numdollar)
01188 {
01189     GETBIDENTITY
01190     DOLLARS d = Dollars + numdollar;
01191     WORD retval;
01192 #ifdef WITHPTHREADS
01193     int nummodopt, dtype = -1;
01194     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01195         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01196             if ( numdollar == ModOptdollars[nummodopt].number ) break;
01197         }
01198         if ( nummodopt < NumModOptdollars ) {
01199             dtype = ModOptdollars[nummodopt].type;
01200             if ( DollarLocalCopy(dtype) ) {
01201                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
01202             }
01203             else {
01204                 LOCK(d->pthreadslock);
01205             }
01206         }
01207     }
01208 #endif
01209     AN.ErrorInDollar = 0;
01210     if ( d->type == DOLINDEX && d->index < 0 ) {
01211         retval = d->index;
01212     }
01213     else if ( d->type == DOLARGUMENT && ( d->where[0] == -VECTOR
01214     || d->where[0] == -MINVECTOR ) ) {
```

```
01215         retval = d->where[1];
01216     }
01217     else if ( d->type == DOLSUBTERM && d->where[0] == INDEX
01218     && d->where[1] == 3 && d->where[2] < 0 ) {
01219         retval = d->where[2];
01220     }
01221     else if ( d->type == DOLTERMS && d->where[0] == 7 &&
01222     d->where[7] == 0 && d->where[6] == 3 &&
01223     d->where[5] == 1 && d->where[4] == 1 &&
01224     d->where[1] >= INDEX && d->where[3] < 0 ) {
01225         retval = d->where[3];
01226     }
01227     else if ( d->type == DOLWILDARGS && d->where[0] == 0
01228     && ( d->where[1] == -VECTOR || d->where[1] == -MINVECTOR )
01229     && d->where[3] == 0 ) {
01230         retval = d->where[2];
01231     }
01232     else if ( d->type == DOLWILDARGS && d->where[0] == 1
01233     && d->where[1] < 0 ) {
01234         retval = d->where[1];
01235     }
01236     else {
01237         AN.ErrorInDollar = 1;
01238         retval = 0;
01239     }
01240 #ifdef WITHPTHREADS
01241     if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
01242 #endif
01243     return(retval);
01244 }
01245
01246 /*
01247     #] DolToVector :
01248     #[ DolToNumber :
01249 */
01250
01251 WORD DolToNumber(PHEAD WORD numdollar)
01252 {
01253     GETBIDENTITY
01254     DOLLARS d = Dollars + numdollar;
01255 #ifdef WITHPTHREADS
01256     int nummodopt, dtype = -1;
01257     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01258         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01259             if ( numdollar == ModOptdollars[nummodopt].number ) break;
01260         }
01261         if ( nummodopt < NumModOptdollars ) {
01262             dtype = ModOptdollars[nummodopt].type;
01263             if ( DollarLocalCopy(dtype) ) {
01264                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
01265             }
01266         }
01267     }
01268 #endif
01269     AN.ErrorInDollar = 0;
01270     if ( ( d->type == DOLTERMS || d->type == DOLNUMBER )
01271      && d->where[0] == 4 &&
01272     d->where[4] == 0 && ( d->where[3] == 3 || d->where[3] == -3 )
01273      && d->where[2] == 1 && ( d->where[1] & TOPBITONLY ) == 0 ) {
01274         if ( d->where[3] > 0 ) return(d->where[1]);
01275         else return(-d->where[1]);
01276     }
01277     else if ( d->type == DOLARGUMENT && d->where[0] == -SNUMBER ) {
01278         return(d->where[1]);
01279     }
01280     else if ( d->type == DOLARGUMENT && d->where[0] == -INDEX
01281     && d->where[1] >= 0 && d->where[1] < AM.OffsetIndex ) {
01282         return(d->where[1]);
01283     }
01284     else if ( d->type == DOLZERO ) return(0);
01285     else if ( d->type == DOLWILDARGS && d->where[0] == 0
01286     && d->where[1] == -SNUMBER && d->where[3] == 0 ) {
01287         return(d->where[2]);
01288     }
01289     else if ( d->type == DOLINDEX && d->index >= 0 && d->index < AM.OffsetIndex ) {
01290         return(d->index);
01291     }
01292     else if ( d->type == DOLWILDARGS && d->where[0] == 1
01293     && d->where[1] >= 0 && d->where[1] < AM.OffsetIndex ) {
01294         return(d->where[1]);
01295     }
01296     else if ( d->type == DOLWILDARGS && d->where[0] == 0
01297     && d->where[1] == -INDEX && d->where[3] == 0 && d->where[2] >= 0
01298     && d->where[2] < AM.OffsetIndex ) {
01299         return(d->where[2]);
01300     }
01301     AN.ErrorInDollar = 1;
```

```
01302     return(0);
01303 }
01304
01305 /*
01306     #] DolToNumber :
01307     #[ DolToSymbol :    with LOCK
01308 */
01309
01310 WORD DolToSymbol(PHEAD WORD numdollar)
01311 {
01312     GETBIDENTITY
01313     DOLLARS d = Dollars + numdollar;
01314     WORD retval;
01315 #ifdef WITHPTHREADS
01316     int nummodopt, dtype = -1;
01317     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01318         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01319             if ( numdollar == ModOptdollars[nummodopt].number ) break;
01320         }
01321         if ( nummodopt < NumModOptdollars ) {
01322             dtype = ModOptdollars[nummodopt].type;
01323             if ( DollarLocalCopy(dtype) ) {
01324                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
01325             }
01326             else {
01327                 LOCK(d->pthreadslock);
01328             }
01329         }
01330     }
01331 #endif
01332     AN.ErrorInDollar = 0;
01333     if ( d->type == DOLTERMS && d->where[0] == 8 &&
01334     d->where[8] == 0 && d->where[7] == 3 && d->where[6] == 1
01335      && d->where[5] == 1 && d->where[4] == 1 && d->where[1] == SYMBOL ) {
01336         retval = d->where[3];
01337     }
01338     else if ( d->type == DOLARGUMENT && d->where[0] == -SYMBOL ) {
01339         retval = d->where[1];
01340     }
01341     else if ( d->type == DOLSUBTERM && d->where[0] == SYMBOL
01342     && d->where[1] == 4 && d->where[3] == 1 ) {
01343         retval = d->where[2];
01344     }
01345     else if ( d->type == DOLWILDARGS && d->where[0] == 0
01346     && d->where[1] == -SYMBOL && d->where[3] == 0 ) {
01347         retval = d->where[2];
01348     }
01349     else {
01350         AN.ErrorInDollar = 1;
01351         retval = -1;
01352     }
01353 #ifdef WITHPTHREADS
01354     if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
01355 #endif
01356     return(retval);
01357 }
01358
01359 /*
01360     #] DolToSymbol :
01361     #[ DolToIndex :     with LOCK
01362 */
01363
01364 WORD DolToIndex(PHEAD WORD numdollar)
01365 {
01366     GETBIDENTITY
01367     DOLLARS d = Dollars + numdollar;
01368     WORD retval;
01369 #ifdef WITHPTHREADS
01370     int nummodopt, dtype = -1;
01371     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01372         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01373             if ( numdollar == ModOptdollars[nummodopt].number ) break;
01374         }
01375         if ( nummodopt < NumModOptdollars ) {
01376             dtype = ModOptdollars[nummodopt].type;
01377             if ( DollarLocalCopy(dtype) ) {
01378                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
01379             }
01380             else {
01381                 LOCK(d->pthreadslock);
01382             }
01383         }
01384     }
01385 #endif
01386     AN.ErrorInDollar = 0;
01387     if ( d->type == DOLTERMS && d->where[0] == 7 &&
01388     d->where[7] == 0 && d->where[6] == 3 && d->where[5] == 1
```

```
01389      && d->where[4] == 1 && d->where[1] == INDEX && d->where[3] >= 0 ) {
01390         retval = d->where[3];
01391     }
01392     else if ( d->type == DOLARGUMENT && d->where[0] == -SNUMBER
01393     && d->where[1] >= 0 && d->where[1] < AM.OffsetIndex ) {
01394         retval = d->where[1];
01395     }
01396     else if ( d->type == DOLARGUMENT && d->where[0] == -INDEX
01397     && d->where[1] >= 0 ) {
01398         retval = d->where[1];
01399     }
01400     else if ( d->type == DOLZERO ) return(0);
01401     else if ( d->type == DOLWILDARGS && d->where[0] == 0
01402     && d->where[1] == -SNUMBER && d->where[3] == 0 && d->where[2] >= 0
01403     && d->where[2] < AM.OffsetIndex ) {
01404         retval = d->where[2];
01405     }
01406     else if ( d->type == DOLINDEX && d->index >= 0 ) {
01407         retval = d->index;
01408     }
01409     else if ( d->type == DOLNUMBER && d->where[0] == 4 && d->where[2] == 1
01410     && d->where[3] == 3 && d->where[4] == 0 && d->where[1] < AM.OffsetIndex ) {
01411         retval = d->where[1];
01412     }
01413     else if ( d->type == DOLWILDARGS && d->where[0] == 1
01414     && d->where[1] >= 0 ) {
01415         retval = d->where[1];
01416     }
01417     else if ( d->type == DOLSUBTERM && d->where[0] == INDEX
01418     && d->where[1] == 3 && d->where[2] >= 0 ) {
01419         retval = d->where[2];
01420     }
01421     else if ( d->type == DOLWILDARGS && d->where[0] == 0
01422     && d->where[1] == -INDEX && d->where[3] == 0 && d->where[2] >= 0 ) {
01423         retval = d->where[2];
01424     }
01425     else {
01426         AN.ErrorInDollar = 1;
01427         retval = 0;
01428     }
01429 #ifdef WITHPTHREADS
01430     if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
01431 #endif
01432     return(retval);
01433 }
01434
01435 /*
01436     #] DolToIndex :
01437     #[ DolToTerms :
01438
01439     Returns a struct of type DOLLARS which contains a copy of the
01440     original dollar variable, provided it can be expressed in terms of
01441     an expression (type = DOLTERMS). Otherwise it returns zero.
01442     The dollar is expressed in terms in the buffer "where"
01443 */
01444
01445 DOLLARS DolToTerms(PHEAD WORD numdollar)
01446 {
01447     GETBIDENTITY
01448     LONG size;
01449     DOLLARS d = Dollars + numdollar, newd;
01450     WORD *t, *w, i;
01451 #ifdef WITHPTHREADS
01452     int nummodopt, dtype = -1;
01453     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01454         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01455             if ( numdollar == ModOptdollars[nummodopt].number ) break;
01456         }
01457         if ( nummodopt < NumModOptdollars ) {
01458             dtype = ModOptdollars[nummodopt].type;
01459             if ( DollarLocalCopy(dtype) ) {
01460                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
01461             }
01462         }
01463     }
01464 #endif
01465     AN.ErrorInDollar = 0;
01466     switch ( d->type ) {
01467         case DOLARGUMENT:
01468             t = d->where;
01469             if ( t[0] < 0 ) {
01470 ShortArgument:
01471                 w = AT.WorkPointer;
01472                 if ( t[0] <= -FUNCTION ) {
01473                     *w++ = FUNHEAD+4; *w++ = -t[0];
01474                     *w++ = FUNHEAD; FILLFUN(w)
01475                     *w++ = 1; *w++ = 1; *w++ = 3;
```

```
01476                 }
01477                 else if ( t[0] == -SYMBOL ) {
01478                     *w++ = 8; *w++ = SYMBOL; *w++ = 4; *w++ = t[1];
01479                     *w++ = 1; *w++ = 1; *w++ = 1; *w++ = 3;
01480                 }
01481                 else if ( t[0] == -VECTOR || t[0] == -INDEX ) {
01482                     *w++ = 7; *w++ = INDEX; *w++ = 3; *w++ = t[1];
01483                     *w++ = 1; *w++ = 1; *w++ = 3;
01484                 }
01485                 else if ( t[0] == -MINVECTOR ) {
01486                     *w++ = 7; *w++ = INDEX; *w++ = 3; *w++ = t[1];
01487                     *w++ = 1; *w++ = 1; *w++ = -3;
01488                 }
01489                 else if ( t[0] == -SNUMBER ) {
01490                     *w++ = 4;
01491                     if ( t[1] < 0 ) {
01492                         *w++ = -t[1]; *w++ = 1; *w++ = -3;
01493                     }
01494                     else {
01495                         *w++ = t[1]; *w++ = 1; *w++ = 3;
01496                     }
01497                 }
01498                 *w = 0; size = w - AT.WorkPointer;
01499                 w = AT.WorkPointer;
01500                 break;
01501             }
01502             /* fall through */
01503         case DOLNUMBER:
01504         case DOLTERMS:
01505             t = d->where;
01506             while ( *t ) t += *t;
01507             size = t - d->where;
01508             w = d->where;
01509             break;
01510         case DOLSUBTERM:
01511             w = AT.WorkPointer;
01512             size = d->where[1];
01513             *w++ = size+4; t = d->where; NCOPY(w,t,size)
01514             *w++ = 1; *w++ = 1; *w++ = 3;
01515             w = AT.WorkPointer; size = d->where[1]+4;
01516             break;
01517         case DOLINDEX:
01518             w = AT.WorkPointer;
01519             *w++ = 7; *w++ = INDEX; *w++ = 3; *w++ = d->index;
01520             *w++ = 1; *w++ = 1; *w++ = 3; *w = 0;
01521             w = AT.WorkPointer; size = 7;
01522             break;
01523         case DOLWILDARGS:
01524 /*
01525             In some cases we can make a copy
01526 */
01527             t = d->where+1;
01528             if ( *t == 0 ) return(0);
01529             NEXTARG(t);
01530             if ( *t ) { /* More than one argument in here */
01531                 MLOCK(ErrorMessageLock);
01532                 MesPrint("Trying to convert a $ with an argument field into an expression");
01533                 MUNLOCK(ErrorMessageLock);
01534                 Terminate(-1);
01535             }
01536 /*
01537             Now we have a single argument
01538 */
01539             t = d->where+1;
01540             if ( *t < 0 ) goto ShortArgument;
01541             size = *t - ARGHEAD;
01542             w = t + ARGHEAD;
01543             break;
01544         case DOLUNDEFINED:
01545             MLOCK(ErrorMessageLock);
01546             MesPrint("Trying to use an undefined $ in an expression");
01547             MUNLOCK(ErrorMessageLock);
01548             Terminate(-1);
01549             /* fall through */
01550         case DOLZERO:
01551             if ( d->where ) { d->where[0] = 0; }
01552             else d->where = &(AM.dollarzero);
01553             size = 0;
01554             w = d->where;
01555             break;
01556         default:
01557             return(0);
01558     }
01559     newd = (DOLLARS)Malloc1(sizeof(struct DoLlArS)+(size+1)*sizeof(WORD),
01560                 "Copy of dollar variable");
01561     t = (WORD *)(newd+1);
01562     newd->where = t;
```

```
01563     newd->name = d->name;
01564     newd->node = d->node;
01565     newd->type = DOLTERMS;
01566     newd->size = size;
01567     newd->numdummies = d->numdummies;
01568 #ifdef WITHPTHREADS
01569     INIRECLOCK(newd->pthreadslock);
01570 #endif
01571     size++;
01572     NCOPY(t,w,size);
01573     newd->nfactors = d->nfactors;
01574     if ( d->nfactors > 1 ) {
01575         newd->factors = (FACDOLLAR *)Malloc1(d->nfactors*sizeof(FACDOLLAR),"Dollar factors");
01576         for ( i = 0; i < d->nfactors; i++ ) {
01577             newd->factors[i].where = 0;
01578             newd->factors[i].size = 0;
01579             newd->factors[i].type = DOLUNDEFINED;
01580             newd->factors[i].value = d->factors[i].value;
01581         }
01582     }
01583     else { newd->factors = 0; }
01584     return(newd);
01585 }
01586
01587 /*
01588     #] DolToTerms :
01589     #[ DolToLong :
01590 */
01591
01592 LONG DolToLong(PHEAD WORD numdollar)
01593 {
01594     GETBIDENTITY
01595     DOLLARS d = Dollars + numdollar;
01596     LONG x;
01597 #ifdef WITHPTHREADS
01598     int nummodopt, dtype = -1;
01599     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01600         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01601             if ( numdollar == ModOptdollars[nummodopt].number ) break;
01602         }
01603         if ( nummodopt < NumModOptdollars ) {
01604             dtype = ModOptdollars[nummodopt].type;
01605             if ( DollarLocalCopy(dtype) ) {
01606                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
01607             }
01608         }
01609     }
01610 #endif
01611     AN.ErrorInDollar = 0;
01612     if ( ( d->type == DOLTERMS || d->type == DOLNUMBER )
01613      && d->where[0] == 4 &&
01614     d->where[4] == 0 && ( d->where[3] == 3 || d->where[3] == -3 )
01615      && d->where[2] == 1 && ( d->where[1] & TOPBITONLY ) == 0 ) {
01616         x = d->where[1];
01617         if ( d->where[3] > 0 ) return(x);
01618         else return(-x);
01619     }
01620     else if ( ( d->type == DOLTERMS || d->type == DOLNUMBER )
01621      && d->where[0] == 6 &&
01622     d->where[6] == 0 && ( d->where[5] == 5 || d->where[5] == -5 )
01623      && d->where[3] == 1 && d->where[4] == 1 && ( d->where[2] & TOPBITONLY ) == 0 ) {
01624         x = d->where[1] + ( (LONG)(d->where[2]) « BITSINWORD );
01625         if ( d->where[5] > 0 ) return(x);
01626         else return(-x);
01627     }
01628     else if ( d->type == DOLARGUMENT && d->where[0] == -SNUMBER ) {
01629         x = d->where[1];
01630         return(x);
01631     }
01632     else if ( d->type == DOLARGUMENT && d->where[0] == -INDEX
01633     && d->where[1] >= 0 && d->where[1] < AM.OffsetIndex ) {
01634         x = d->where[1];
01635         return(x);
01636     }
01637     else if ( d->type == DOLZERO ) return(0);
01638     else if ( d->type == DOLWILDARGS && d->where[0] == 0
01639     && d->where[1] == -SNUMBER && d->where[3] == 0 ) {
01640         x = d->where[2];
01641         return(x);
01642     }
01643     else if ( d->type == DOLINDEX && d->index >= 0 && d->index < AM.OffsetIndex ) {
01644         x = d->index;
01645         return(x);
01646     }
01647     else if ( d->type == DOLWILDARGS && d->where[0] == 1
01648     && d->where[1] >= 0 && d->where[1] < AM.OffsetIndex ) {
01649         x = d->where[1];
```

```
01650         return(x);
01651     }
01652     else if ( d->type == DOLWILDARGS && d->where[0] == 0
01653     && d->where[1] == -INDEX && d->where[3] == 0 && d->where[2] >= 0
01654     && d->where[2] < AM.OffsetIndex ) {
01655         x = d->where[2];
01656         return(x);
01657     }
01658     AN.ErrorInDollar = 1;
01659     return(0);
01660 }
01661
01662 /*
01663     #] DolToLong :
01664     #[ ExecInside :
01665 */
01666
01667 int ExecInside(UBYTE *s)
01668 {
01669     GETIDENTITY
01670     UBYTE *t, c;
01671     WORD *w, number;
01672     int error = 0;
01673     w = AT.WorkPointer;
01674     if ( AC.insidelevel >= MAXNEST ) {
01675         MLOCK(ErrorMessageLock);
01676         MesPrint("@Nesting of inside statements more than %d levels",(WORD)MAXNEST);
01677         MUNLOCK(ErrorMessageLock);
01678         return(-1);
01679     }
01680     AC.insidesumcheck[AC.insidelevel] = NestingChecksum();
01681     AC.insidestack[AC.insidelevel] = cbuf[AC.cbufnum].Pointer
01682                                    - cbuf[AC.cbufnum].Buffer + 2;
01683     AC.insidelevel++;
01684     *w++ = TYPEINSIDE;
01685     w++; w++;
01686     for(;;) {   /* Look for a (comma separated) list of dollar variables */
01687         while ( *s == ',' ) s++;
01688         if ( *s == 0 ) break;
01689         if ( *s == '$' ) {
01690             s++; t = s;
01691             if ( FG.cTable[*s] != 0 ) {
01692                 MLOCK(ErrorMessageLock);
01693                 MesPrint("Illegal name for $ variable: %s",s-1);
01694                 MUNLOCK(ErrorMessageLock);
01695                 goto skipdol;
01696             }
01697             while ( FG.cTable[*s] == 0 || FG.cTable[*s] == 1 ) s++;
01698             c = *s; *s = 0;
01699             if ( ( number = GetDollar(t) ) < 0 ) {
01700                 number = AddDollar(t,0,0,0);
01701             }
01702             *s = c;
01703             *w++ = number;
01704             AddPotModdollar(number);
01705         }
01706         else {
01707             MLOCK(ErrorMessageLock);
01708             MesPrint("&Illegal object in Inside statement");
01709             MUNLOCK(ErrorMessageLock);
01710 skipdol:    error = 1;
01711             while ( *s && *s != ',' && s[1] != '$' ) s++;
01712             if ( *s == 0 ) break;
01713         }
01714     }
01715     AT.WorkPointer[1] = w - AT.WorkPointer;
01716     AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
01717     return(error);
01718 }
01719
01720 /*
01721     #] ExecInside :
01722     #[ InsideDollar :
01723
01724     Execution part of Inside $a;
01725     We have to take the variables one by one and then
01726     convert them into proper terms and call Generator for the proper levels.
01727     The conversion copies the whole dollar into a new buffer, making us
01728     insensitive to redefinitions of $a inside the Inside.
01729     In the end we sort and redefine $a.
01730 */
01731
01732 int InsideDollar(PHEAD WORD *ll, WORD level)
01733 {
01734     GETBIDENTITY
01735     int numvar = (int)(ll[1]-3), j, error = 0;
01736     WORD numdol, *oldcterm, *oldwork = AT.WorkPointer, olddefer, *r, *m;
```

```
01737     WORD oldnumlhs, *dbuffer;
01738     DOLLARS d, newd;
01739     oldcterm = AN.cTerm; AN.cTerm = 0;
01740     oldnumlhs = AR.Cnumlhs; AR.Cnumlhs = ll[2];
01741     ll += 3;
01742     olddefer = AR.DeferFlag;
01743     AR.DeferFlag = 0;
01744     while ( --numvar >= 0 ) {
01745         numdol = *ll++;
01746         d = Dollars + numdol;
01747         {
01748 #ifdef WITHPTHREADS
01749         int nummodopt, dtype = -1;
01750         if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01751             for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01752                 if ( numdol == ModOptdollars[nummodopt].number ) break;
01753             }
01754             if ( nummodopt < NumModOptdollars ) {
01755                 dtype = ModOptdollars[nummodopt].type;
01756                 if ( DollarLocalCopy(dtype) ) {
01757                     d = ModOptdollars[nummodopt].dstruct+AT.identity;
01758                 }
01759                 else {
01760                     LOCK(d->pthreadslock);
01761                 }
01762             }
01763         }
01764 #endif
01765         newd = DolToTerms(BHEAD numdol);
01766         if ( newd == 0 ) {
01767             continue;
01768         }
01769         if ( newd->where[0] == 0 ) {
01770             // DolToTerms potentially allocates memory. Free it.
01771             // The free below is inside the while loop.
01772             if ( newd->factors ) M_free(newd->factors,"Dollar factors");
01773             M_free(newd,"Copy of dollar variable");
01774             continue;
01775         }
01776         r = newd->where;
01777         NewSort(BHEAD0);
01778         while ( *r ) {  /* Sum over the terms */
01779             m = AT.WorkPointer;
01780             j = *r;
01781             while ( --j >= 0 ) *m++ = *r++;
01782             AT.WorkPointer = m;
01783 /*
01784             What to do with dummy indices?
01785 */
01786             if ( Generator(BHEAD oldwork,level) ) {
01787                 LowerSortLevel();
01788                 error = -1; goto idcall;
01789             }
01790             AT.WorkPointer = oldwork;
01791         }
01792         AN.tryterm = 0; /* for now */
01793         if ( EndSort(BHEAD (WORD *)((void *)(&dbuffer)),2) < 0 ) { error = 1; break; }
01794         if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"old buffer of dollar");
01795         d->where = dbuffer;
01796         if ( dbuffer == 0 || *dbuffer == 0 ) {
01797             d->type = DOLZERO;
01798             if ( dbuffer ) M_free(dbuffer,"buffer of dollar");
01799             d->where = &(AM.dollarzero); d->size = 0;
01800         }
01801         else {
01802             d->type = DOLTERMS;
01803             r = d->where; while ( *r ) r += *r;
01804             d->size = (r-d->where)+1;
01805         }
01806 /*      cbuf[AM.dbufnum].rhs[numdol] = d->where; */
01807         cbuf[AM.dbufnum].rhs[numdol] = (WORD *)(1);
01808 /*
01809         Now we have a little cleaning up to do
01810 */
01811 #ifdef WITHPTHREADS
01812         if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
01813 #endif
01814         if ( newd->factors ) M_free(newd->factors,"Dollar factors");
01815         M_free(newd,"Copy of dollar variable");
01816         }
01817     }
01818 idcall:;
01819     AR.Cnumlhs = oldnumlhs;
01820     AR.DeferFlag = olddefer;
01821     AN.cTerm = oldcterm;
01822     AT.WorkPointer = oldwork;
01823     return(error);
```

```
01824 }
01825
01826 /*
01827     #] InsideDollar :
01828     #[ ExchangeDollars :
01829 */
01830
01831 void ExchangeDollars(int num1, int num2)
01832 {
01833     DOLLARS d1, d2;
01834     WORD node1, node2;
01835     LONG nam;
01836     d1 = Dollars + num1; node1 = d1->node;
01837     d2 = Dollars + num2; node2 = d2->node;
01838     nam = d1->name; d1->name = d2->name; d2->name = nam;
01839     d1->node = node2; d2->node = node1;
01840     AC.dollarnames->namenode[node1].number = num2;
01841     AC.dollarnames->namenode[node2].number = num1;
01842 }
01843
01844 /*
01845     #] ExchangeDollars :
01846     #[ TermsInDollar :
01847 */
01848
01849 LONG TermsInDollar(WORD num)
01850 {
01851     GETIDENTITY
01852     DOLLARS d = Dollars + num;
01853     WORD *t;
01854     LONG n;
01855 #ifdef WITHPTHREADS
01856     int nummodopt, dtype = -1;
01857     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01858         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01859             if ( num == ModOptdollars[nummodopt].number ) break;
01860         }
01861         if ( nummodopt < NumModOptdollars ) {
01862             dtype = ModOptdollars[nummodopt].type;
01863             if ( DollarLocalCopy(dtype) ) {
01864                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
01865             }
01866             else {
01867                 LOCK(d->pthreadslock);
01868             }
01869         }
01870     }
01871 #endif
01872     if ( d->type == DOLTERMS ) {
01873         n = 0;
01874         t = d->where;
01875         while ( *t ) { t += *t; n++; }
01876     }
01877     else if ( d->type == DOLWILDARGS ) {
01878         n = 0;
01879         if ( d->where[0] == 0 ) {
01880             t = d->where+1;
01881             while ( *t != 0 ) { NEXTARG(t); n++; }
01882         }
01883         else if ( d->where[0] == 1 ) n = 1;
01884     }
01885     else if ( d->type == DOLZERO ) n = 0;
01886     else n = 1;
01887 #ifdef WITHPTHREADS
01888     if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
01889 #endif
01890     return(n);
01891 }
01892
01893 /*
01894     #] TermsInDollar :
01895     #[ SizeOfDollar :
01896 */
01897
01898 LONG SizeOfDollar(WORD num)
01899 {
01900     GETIDENTITY
01901     DOLLARS d = Dollars + num;
01902     WORD *t;
01903     LONG n;
01904 #ifdef WITHPTHREADS
01905     int nummodopt, dtype = -1;
01906     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01907         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01908             if ( num == ModOptdollars[nummodopt].number ) break;
01909         }
01910         if ( nummodopt < NumModOptdollars ) {
```

```
01911             dtype = ModOptdollars[nummodopt].type;
01912             if ( DollarLocalCopy(dtype) ) {
01913                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
01914             }
01915             else {
01916                 LOCK(d->pthreadslock);
01917             }
01918         }
01919     }
01920 #endif
01921     if ( d->type == DOLTERMS ) {
01922         t = d->where;
01923         while ( *t ) t += *t;
01924         t++;
01925         n = (LONG)(t - d->where);
01926     }
01927     else if ( d->type == DOLWILDARGS ) {
01928         n = 0;
01929         if ( d->where[0] == 0 ) {
01930             t = d->where+1;
01931             while ( *t != 0 ) { NEXTARG(t); n++; }
01932             t++;
01933             n = (LONG)(t - d->where);
01934         }
01935         else if ( d->where[0] == 1 ) n = 1;
01936     }
01937     else if ( d->type == DOLZERO ) n = 0;
01938     else n = 1;
01939 #ifdef WITHPTHREADS
01940     if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
01941 #endif
01942     return(n);
01943 }
01944
01945 /*
01946     #] SizeOfDollar :
01947     #[ PreIfDollarEval :
01948
01949     Routine is invoked in #if etc after $( is encountered.
01950     $(expr1 operator expr2) makes compares between expressions,
01951     $(expr1 operator _keyword) makes compares between expressions,
01952     interpreted as expressions. We are here mainly looking at $variables.
01953     First we look for the operator:
01954         >, <, ==, >=, <=, != : < means that it comes before.
01955     _keywords can be:
01956         _set(setname)    (does the expr belong to the set (only with == or !=))
01957         _productof(expr)
01958 */
01959
01960 UBYTE *PreIfDollarEval(UBYTE *s, int *value)
01961 {
01962     GETIDENTITY
01963     UBYTE *s1,*s2,*s3,*s4,*s5,*t,c,c1,c2,c3;
01964     int oprtr, type;
01965     WORD *buf1 = 0, *buf2 = 0, numset, *oldwork = AT.WorkPointer;
01966     EXCHINOUT
01967 /*
01968     Find the three composing objects (epxression, operator, expression or keyw
01969 */
01970     while ( *s == ' ' || *s == '\t' || *s == '\n' || *s == '\r' ) s++;
01971     s1 = t = s;
01972     while ( *t != '=' && *t != '!' && *t != '>' && *t != '<' ) {
01973         if ( *t == '[' ) { SKIPBRA1(t) }
01974         else if ( *t == '{' ) { SKIPBRA2(t) }
01975         else if ( *t == '(' ) { SKIPBRA3(t) }
01976         else if ( *t == ']' || *t == '}' || *t == ')' ) {
01977             MLOCK(ErrorMessageLock);
01978             MesPrint("@Improper bracketting in #if");
01979             MUNLOCK(ErrorMessageLock);
01980             goto onerror;
01981         }
01982         t++;
01983     }
01984     s2 = t;
01985     while ( *t == '=' || *t == '!' || *t == '>' || *t == '<' ) t++;
01986     s3 = t;
01987     while ( *t && *t != ')' ) {
01988         if ( *t == '[' ) { SKIPBRA1(t) }
01989         else if ( *t == '{' ) { SKIPBRA2(t) }
01990         else if ( *t == '(' ) { SKIPBRA3(t) }
01991         else if ( *t == ']' || *t == '}' ) {
01992             MLOCK(ErrorMessageLock);
01993             MesPrint("@Improper brackets in #if");
01994             MUNLOCK(ErrorMessageLock);
01995             goto onerror;
01996         }
01997         t++;
```

```
01998     }
01999     if ( *t == 0 ) {
02000         MLOCK(ErrorMessageLock);
02001         MesPrint("@Missing ) to match $( in #if");
02002         MUNLOCK(ErrorMessageLock);
02003         goto onerror;
02004     }
02005     s4 = t; c2 = *s4; *s4 = 0;
02006     if ( s2+2 < s3 || s2 == s3 ) {
02007 IllOp:;
02008         MLOCK(ErrorMessageLock);
02009         MesPrint("@Illegal operator in $( option of #if");
02010         MUNLOCK(ErrorMessageLock);
02011         goto onerror;
02012     }
02013     if ( s2+1 == s3 ) {
02014         if ( *s2 == '=' ) oprtr = EQUAL;
02015         else if ( *s2 == '>' ) oprtr = GREATER;
02016         else if ( *s2 == '<' ) oprtr = LESS;
02017         else goto IllOp;
02018     }
02019     else if ( *s2 == '!' && s2[1] == '=' ) oprtr = NOTEQUAL;
02020     else if ( *s2 == '=' && s2[1] == '=' ) oprtr = EQUAL;
02021     else if ( *s2 == '<' && s2[1] == '=' ) oprtr = LESSEQUAL;
02022     else if ( *s2 == '>' && s2[1] == '=' ) oprtr = GREATEREQUAL;
02023     else goto IllOp;
02024     c1 = *s2; *s2 = 0;
02025 /*
02026     The two expressions are now zero terminated
02027     Look for the special keywords
02028 */
02029     while ( *s3 == ' ' || *s3 == '\t' || *s3 == '\n' || *s3 == '\r' ) s3++;
02030     t = s3;
02031     while ( chartype[*t] == 0 ) t++;
02032     if ( *t == '_' ) {
02033         t++; c = *t; *t = 0;
02034         if ( StrICmp(s3,(UBYTE *)"set_") == 0 ) {
02035             if ( oprtr != EQUAL && oprtr != NOTEQUAL ) {
02036 ImpOp:;
02037                 MLOCK(ErrorMessageLock);
02038                 MesPrint("@Improper operator for special keyword in $( ) option");
02039                 MUNLOCK(ErrorMessageLock);
02040                 goto onerror;
02041             }
02042             type = 1;
02043         }
02044         else if ( StrICmp(s3,(UBYTE *)"multipleof_") == 0 ) {
02045             if ( oprtr != EQUAL && oprtr != NOTEQUAL ) goto ImpOp;
02046             type = 2;
02047         }
02048 /*
02049         else if ( StrICmp(s3,(UBYTE *)"productof_") == 0 ) {
02050             if ( oprtr != EQUAL && oprtr != NOTEQUAL ) goto ImpOp;
02051             type = 3;
02052         }
02053 */
02054         else type = 0;
02055     }
02056     else { type = 0; c = *t; }
02057     if ( type > 0 ) {
02058         *t++ = c; s3 = t; s5 = s4-1;
02059         while ( *s5 != ')' ) {
02060             if ( *s5 == ' ' || *s5 == '\t' || *s5 == '\n' || *s5 == '\r' ) s5--;
02061             else {
02062                 MLOCK(ErrorMessageLock);
02063                 MesPrint("@Improper use of special keyword in $( ) option");
02064                 MUNLOCK(ErrorMessageLock);
02065                 goto onerror;
02066             }
02067         }
02068         c3 = *s5; *s5 = 0;
02069     }
02070     else { c3 = c2; s5 = s4; }
02071 /*
02072     Expand the first expression.
02073 */
02074     if ( ( buf1 = TranslateExpression(s1) ) == 0 ) {
02075         AT.WorkPointer = oldwork;
02076         goto onerror;
02077     }
02078     if ( type == 1 ) {  /* determine the set */
02079         if ( *s3 == '{' ) {
02080             t = s3+1;
02081             SKIPBRA2(s3)
02082             numset = DoTempSet(t,s3);
02083             s3++;
02084             if ( numset < 0 ) {
```

```
02085 noset:;
02086                 MLOCK(ErrorMessageLock);
02087                 MesPrint("@Argument of set_ is not a valid set");
02088                 MUNLOCK(ErrorMessageLock);
02089                 goto onerror;
02090             }
02091         }
02092         else {
02093             t = s3;
02094             while ( FG.cTable[*s3] == 0 || FG.cTable[*s3] == 1
02095                 || *s3 == '_' ) s3++;
02096             c = *s3; *s3 = 0;
02097             if ( GetName(AC.varnames,t,&numset,NOAUTO) != CSET ) {
02098                 *s3 = c; goto noset;
02099             }
02100             *s3 = c;
02101         }
02102         while ( *s3 == ' ' || *s3 == '\t' || *s3 == '\n' || *s3 == '\r' ) s3++;
02103         if ( s3 != s5 ) goto noset;
02104         *value = IsSetMember(buf1,numset);
02105         if ( oprtr == NOTEQUAL ) *value ^= 1;
02106     }
02107     else {
02108         if ( ( buf2 = TranslateExpression(s3) ) == 0 ) goto onerror;
02109     }
02110     if ( type == 0 ) {
02111         *value = TwoExprCompare(buf1,buf2,oprtr);
02112     }
02113     else if ( type == 2 ) {
02114         *value = IsMultipleOf(buf1,buf2);
02115         if ( oprtr == NOTEQUAL ) *value ^= 1;
02116     }
02117 /*
02118     else if ( type == 3 ) {
02119         *value = IsProductOf(buf1,buf2);
02120         if ( oprtr == NOTEQUAL ) *value ^= 1;
02121     }
02122 */
02123     if ( buf1 ) M_free(buf1,"Buffer in $()");
02124     if ( buf2 ) M_free(buf2,"Buffer in $()");
02125     *s5 = c3; *s4++ = c2; *s2 = c1;
02126     AT.WorkPointer = oldwork;
02127     BACKINOUT
02128     return(s4);
02129 onerror:
02130     if ( buf1 ) M_free(buf1,"Buffer in $()");
02131     if ( buf2 ) M_free(buf2,"Buffer in $()");
02132     AT.WorkPointer = oldwork;
02133     BACKINOUT
02134     return(0);
02135 }
02136
02137 /*
02138     #] PreIfDollarEval :
02139     #[ TranslateExpression :
02140 */
02141
02142 WORD *TranslateExpression(UBYTE *s)
02143 {
02144     GETIDENTITY
02145     CBUF *C = cbuf+AC.cbufnum;
02146     WORD oldnumrhs = C->numrhs;
02147     LONG oldcpointer = C->Pointer - C->Buffer;
02148     WORD *w = AT.WorkPointer;
02149     WORD retcode, oldEside;
02150     WORD *outbuffer;
02151     *w++ = SUBEXPSIZE + 4;
02152     AC.ProtoType = w;
02153     *w++ = SUBEXPRESSION;
02154     *w++ = SUBEXPSIZE;
02155     *w++ = C->numrhs+1;
02156     *w++ = 1;
02157     *w++ = AC.cbufnum;
02158     FILLSUB(w)
02159     *w++ = 1; *w++ = 1; *w++ = 3; *w++ = 0;
02160     AT.WorkPointer = w;
02161     if ( ( retcode = CompileAlgebra(s,RHSIDE,AC.ProtoType) ) < 0 ) {
02162         MLOCK(ErrorMessageLock);
02163         MesPrint("@Error translating first expression in $( ) option");
02164         MUNLOCK(ErrorMessageLock);
02165         return(0);
02166     }
02167     else { AC.ProtoType[2] = retcode; }
02168 /*
02169     Evaluate this expression
02170 */
02171     if ( NewSort(BHEAD0) || NewSort(BHEAD0) ) { return(0); }
```

```
02172     AN.RepPoint = AT.RepCount + 1;
02173     oldEside = AR.Eside; AR.Eside = RHSIDE;
02174     AR.Cnumlhs = C->numlhs;
02175     if ( Generator(BHEAD AC.ProtoType-1,C->numlhs) ) {
02176         AR.Eside = oldEside;
02177         LowerSortLevel(); LowerSortLevel(); return(0);
02178     }
02179     AR.Eside = oldEside;
02180     AT.WorkPointer = w;
02181     AN.tryterm = 0; /* for now */
02182     if ( EndSort(BHEAD (WORD *)((void *)(&outbuffer)),2) < 0 ) { LowerSortLevel(); return(0); }
02183     LowerSortLevel();
02184     C->Pointer = C->Buffer + oldcpointer;
02185     C->numrhs = oldnumrhs;
02186     AT.WorkPointer = AC.ProtoType - 1;
02187     return(outbuffer);
02188 }
02189
02190 /*
02191     #] TranslateExpression :
02192     #[ IsSetMember :
02193
02194     Checks whether the expression in the buffer can be seen as an element
02195     of the given set.
02196     For the special sets: if more than one term: no match!!!
02197 */
02198
02199 int IsSetMember(WORD *buffer, WORD numset)
02200 {
02201     WORD *t = buffer, *tt, num, csize, num1;
02202     WORD bufterm[4];
02203     int i, j, type;
02204     if ( numset < AM.NumFixedSets ) {
02205         if ( t[*t] != 0 ) return(0);    /* More than one term */
02206         if ( *t == 0 ) {
02207             if ( numset == POS0_ || numset == NEG0_ || numset == EVEN_
02208             || numset == Z_ || numset == Q_ ) return(1);
02209             else return(0);
02210         }
02211         if ( numset == SYMBOL_ ) {
02212             if ( *t == 8 && t[1] == SYMBOL && t[7] == 3 && t[6] == 1
02213             && t[5] == 1 && t[4] == 1 ) return(1);
02214             else return(0);
02215         }
02216         if ( numset == INDEX_ ) {
02217             if ( *t == 7 && t[1] == INDEX && t[6] == 3 && t[5] == 1
02218             && t[4] == 1 && t[3] > 0 ) return(1);
02219             if ( *t == 4 && t[3] == 3 && t[2] == 1 && t[1] < AM.OffsetIndex)
02220                 return(1);
02221             return(0);
02222         }
02223         if ( numset == FIXED_ ) {
02224             if ( *t == 7 && t[1] == INDEX && t[6] == 3 && t[5] == 1
02225             && t[4] == 1 && t[3] > 0 && t[3] < AM.OffsetIndex ) return(1);
02226             if ( *t == 4 && t[3] == 3 && t[2] == 1 && t[1] < AM.OffsetIndex)
02227                 return(1);
02228             return(0);
02229         }
02230         if ( numset == DUMMYINDEX_ ) {
02231             if ( *t == 7 && t[1] == INDEX && t[6] == 3 && t[5] == 1
02232             && t[4] == 1 && t[3] >= AM.IndDum && t[3] < AM.IndDum+MAXDUMMIES ) return(1);
02233             if ( *t == 4 && t[3] == 3 && t[2] == 1
02234                  && t[1] >= AM.IndDum && t[1] < AM.IndDum+MAXDUMMIES ) return(1);
02235             return(0);
02236         }
02237         if ( numset == VECTOR_ ) {
02238             if ( *t == 7 && t[1] == INDEX && t[6] == 3 && t[5] == 1
02239             && t[4] == 1 && t[3] < (AM.OffsetVector+WILDOFFSET) && t[3] >= AM.OffsetVector )
      return(1);
02240             return(0);
02241         }
02242         tt = t + *t - 1;
02243         if ( ABS(tt[0]) != *t-1 ) return(0);
02244         if ( numset == Q_ ) return(1);
02245         if ( numset == POS_ || numset == POS0_ ) return(tt[0]>0);
02246         else if ( numset == NEG_ || numset == NEG0_ ) return(tt[0]<0);
02247         i = (ABS(tt[0])-1)/2;
02248         tt -= i;
02249         if ( tt[0] != 1 ) return(0);
02250         for ( j = 1; j < i; j++ ) { if ( tt[j] != 0 ) return(0); }
02251         if ( numset == Z_ ) return(1);
02252         if ( numset == ODD_ ) return(t[1]&1);
02253         if ( numset == EVEN_ ) return(1-(t[1]&1));
02254         return(0);
02255     }
02256     if ( t[*t] != 0 ) return(0);    /* More than one term */
02257     type = Sets[numset].type;
```

```
02258     switch ( type ) {
02259         case CSYMBOL:
02260             if ( t[0] == 8 && t[1] == SYMBOL && t[7] == 3 && t[6] == 1
02261             && t[5] == 1 && t[4] == 1 ) {
02262                 num = t[3];
02263             }
02264             else if ( t[0] == 4 && t[2] == 1 && t[1] <= MAXPOWER ) {
02265                 num = t[1];
02266                 if ( t[3] < 0 ) num = -num;
02267                 num += 2*MAXPOWER;
02268             }
02269             else return(0);
02270             break;
02271         case CVECTOR:
02272             if ( t[0] == 7 && t[1] == INDEX && t[6] == 3 && t[5] == 1
02273             && t[4] == 1 && t[3] < 0 ) {
02274                 num = t[3];
02275             }
02276             else return(0);
02277             break;
02278         case CINDEX:
02279             if ( t[0] == 7 && t[1] == INDEX && t[6] == 3 && t[5] == 1
02280             && t[4] == 1 && t[3] > 0 ) {
02281                 num = t[3];
02282             }
02283             else if ( t[0] == 4 && t[3] == 3 && t[2] == 1 && t[1] < AM.OffsetIndex ) {
02284                 num = t[1];
02285             }
02286             else return(0);
02287             break;
02288         case CFUNCTION:
02289             if ( t[0] == 4+FUNHEAD && t[3+FUNHEAD] == 3 && t[2+FUNHEAD] == 1
02290             && t[1+FUNHEAD] == 1 && t[1] >= FUNCTION ) {
02291                 num = t[1];
02292             }
02293             else return(0);
02294             break;
02295         case CNUMBER:
02296             if ( t[0] == 4 && t[2] == 1 && t[1] <= AM.OffsetIndex && t[3] == 3 ) {
02297                 num = t[1];
02298             }
02299             else return(0);
02300             break;
02301         case CRANGE:
02302             csize = t[t[0]-1];
02303             csize = ABS(csize);
02304             if ( csize != t[0]-1 ) return(0);
02305             if ( Sets[numset].first < 3*MAXPOWER ) {
02306                 num1 = num = Sets[numset].first;
02307                 if ( num >= MAXPOWER ) num -= 2*MAXPOWER;
02308                 if ( num == 0 ) {
02309                     if ( num1 < MAXPOWER ) {
02310                         if ( t[t[0]-1] >= 0 ) return(0);
02311                     }
02312                     else if ( t[t[0]-1] > 0 ) return(0);
02313                 }
02314                 else {
02315                     bufterm[0] = 4; bufterm[1] = ABS(num);
02316                     bufterm[2] = 1;
02317                     if ( num < 0 ) bufterm[3] = -3;
02318                     else bufterm[3] = 3;
02319                     num = CompCoef(t,bufterm);
02320                     if ( num1 < MAXPOWER ) {
02321                         if ( num >= 0 ) return(0);
02322                     }
02323                     else if ( num > 0 ) return(0);
02324                 }
02325             }
02326             if ( Sets[numset].last > -3*MAXPOWER ) {
02327                 num1 = num = Sets[numset].last;
02328                 if ( num <= -MAXPOWER ) num += 2*MAXPOWER;
02329                 if ( num == 0 ) {
02330                     if ( num1 > -MAXPOWER ) {
02331                         if ( t[t[0]-1] <= 0 ) return(0);
02332                     }
02333                     else if ( t[t[0]-1] < 0 ) return(0);
02334                 }
02335                 else {
02336                     bufterm[0] = 4; bufterm[1] = ABS(num);
02337                     bufterm[2] = 1;
02338                     if ( num < 0 ) bufterm[3] = -3;
02339                     else bufterm[3] = 3;
02340                     num = CompCoef(t,bufterm);
02341                     if ( num1 > -MAXPOWER ) {
02342                         if ( num <= 0 ) return(0);
02343                     }
02344                     else if ( num < 0 ) return(0);
```

```
02345                 }
02346             }
02347             return(1);
02348             break;
02349         default: return(0);
02350     }
02351     t  = SetElements + Sets[numset].first;
02352     tt = SetElements + Sets[numset].last;
02353     do {
02354         if ( num == *t ) return(1);
02355         t++;
02356     } while ( t < tt );
02357     return(0);
02358 }
02359
02360 /*
02361     #] IsSetMember :
02362     #[ IsProductOf :
02363
02364     Checks whether the expression in buf1 is a single term multiple of
02365     the expression in buf2.
02366
02367 int IsProductOf(WORD *buf1, WORD *buf2)
02368 {
02369     return(0);
02370 }
02371
02372
02373     #] IsProductOf :
02374     #[ IsMultipleOf :
02375
02376     Checks whether the expression in buf1 is a numerical multiple of
02377     the expression in buf2.
02378 */
02379
02380 int IsMultipleOf(WORD *buf1, WORD *buf2)
02381 {
02382     GETIDENTITY
02383     LONG num1, num2;
02384     WORD *t1, *t2, *m1, *m2, *r1, *r2, nc1, nc2, ni1, ni2;
02385     UWORD *IfScrat1, *IfScrat2;
02386     int i, j;
02387     if ( *buf1 == 0 && *buf2 == 0 ) return(1);
02388 /*
02389     First count terms
02390 */
02391     t1 = buf1; t2 = buf2; num1 = 0; num2 = 0;
02392     while ( *t1 ) { t1 += *t1; num1++; }
02393     while ( *t2 ) { t2 += *t2; num2++; }
02394     if ( num1 != num2 ) return(0);
02395 /*
02396     Test similarity of terms. Difference up to a number.
02397 */
02398     t1 = buf1; t2 = buf2;
02399     while ( *t1 ) {
02400         m1 = t1+1; m2 = t2+1; t1 += *t1; t2 += *t2;
02401         r1 = t1 - ABS(t1[-1]); r2 = t2 - ABS(t2[-1]);
02402         if ( r1-m1 != r2-m2 ) return(0);
02403         while ( m1 < r1 ) {
02404             if ( *m1 != *m2 ) return(0);
02405             m1++; m2++;
02406         }
02407     }
02408 /*
02409     Now we have to test the constant factor
02410 */
02411     IfScrat1 = (UWORD *)(TermMalloc("IsMultipleOf")); IfScrat2 = (UWORD
      *)(TermMalloc("IsMultipleOf"));
02412     t1 = buf1; t2 = buf2;
02413     t1 += *t1; t2 += *t2;
02414     if ( *t1 == 0 && *t2 == 0 ) return(1);
02415     r1 = t1 - ABS(t1[-1]); r2 = t2 - ABS(t2[-1]);
02416     nc1 = REDLENG(t1[-1]); nc2 = REDLENG(t2[-1]);
02417     if ( DivRat(BHEAD (UWORD *)r1,nc1,(UWORD *)r2,nc2,IfScrat1,&ni1) ) {
02418         MLOCK(ErrorMessageLock);
02419         MesPrint("@Called from MultipleOf in $( )");
02420         MUNLOCK(ErrorMessageLock);
02421         TermFree(IfScrat1,"IsMultipleOf"); TermFree(IfScrat2,"IsMultipleOf");
02422         Terminate(-1);
02423     }
02424     while ( *t1 ) {
02425         t1 += *t1; t2 += *t2;
02426         r1 = t1 - ABS(t1[-1]); r2 = t2 - ABS(t2[-1]);
02427         nc1 = REDLENG(t1[-1]); nc2 = REDLENG(t2[-1]);
02428         if ( DivRat(BHEAD (UWORD *)r1,nc1,(UWORD *)r2,nc2,IfScrat2,&ni2) ) {
02429             MLOCK(ErrorMessageLock);
02430             MesPrint("@Called from MultipleOf in $( )");
```

```
02431             MUNLOCK(ErrorMessageLock);
02432             TermFree(IfScrat1,"IsMultipleOf"); TermFree(IfScrat2,"IsMultipleOf");
02433             Terminate(-1);
02434         }
02435         if ( ni1 != ni2 ) return(0);
02436         i = 2*ABS(ni1);
02437         for ( j = 0; j < i; j++ ) {
02438             if ( IfScrat1[j] != IfScrat2[j] ) {
02439                 TermFree(IfScrat1,"IsMultipleOf"); TermFree(IfScrat2,"IsMultipleOf");
02440                 return(0);
02441             }
02442         }
02443     }
02444     TermFree(IfScrat1,"IsMultipleOf"); TermFree(IfScrat2,"IsMultipleOf");
02445     return(1);
02446 }
02447
02448 /*
02449     #] IsMultipleOf :
02450     #[ TwoExprCompare :
02451
02452     Compares the expressions in buf1 and buf2 according to oprtr
02453 */
02454
02455 int TwoExprCompare(WORD *buf1, WORD *buf2, int oprtr)
02456 {
02457     GETIDENTITY
02458     WORD *t1, *t2, cond;
02459     t1 = buf1; t2 = buf2;
02460     while ( *t1 && *t2 ) {
02461         cond = CompareTerms(BHEAD t1,t2,1);
02462         if ( cond != 0 ) {
02463             if ( cond > 0 ) { /* t1 comes first */
02464                 switch ( oprtr ) {  /* t1 is less */
02465                     case EQUAL: return(0);
02466                     case NOTEQUAL: return(1);
02467                     case GREATEREQUAL: return(0);
02468                     case GREATER: return(0);
02469                     case LESS: return(1);
02470                     case LESSEQUAL: return(1);
02471                 }
02472             }
02473             else {
02474                 switch ( oprtr ) {
02475                     case EQUAL: return(0);
02476                     case NOTEQUAL: return(1);
02477                     case GREATEREQUAL: return(1);
02478                     case GREATER: return(1);
02479                     case LESS: return(0);
02480                     case LESSEQUAL: return(0);
02481                 }
02482             }
02483         }
02484         t1 += *t1; t2 += *t2;
02485     }
02486     if ( *t1 == *t2 ) { /* They are equal */
02487         switch ( oprtr ) {
02488             case EQUAL: return(1);
02489             case NOTEQUAL: return(0);
02490             case GREATEREQUAL: return(1);
02491             case GREATER: return(0);
02492             case LESS: return(0);
02493             case LESSEQUAL: return(1);
02494         }
02495     }
02496     else if ( *t1 ) {  /* t1 is greater */
02497         switch ( oprtr ) {
02498             case EQUAL: return(0);
02499             case NOTEQUAL: return(1);
02500             case GREATEREQUAL: return(1);
02501             case GREATER: return(1);
02502             case LESS: return(0);
02503             case LESSEQUAL: return(0);
02504         }
02505     }
02506     else {
02507         switch ( oprtr ) {  /* t1 is less */
02508             case EQUAL: return(0);
02509             case NOTEQUAL: return(1);
02510             case GREATEREQUAL: return(0);
02511             case GREATER: return(0);
02512             case LESS: return(1);
02513             case LESSEQUAL: return(1);
02514         }
02515     }
02516 /* INTERNAL_ERROR_EXCL_START */
02517     MLOCK(ErrorMessageLock);
```

```
02518     MesPrint("!>Internal problems with operator in $( )");
02519     MUNLOCK(ErrorMessageLock);
02520     Terminate(-1);
02521 /* INTERNAL_ERROR_EXCL_STOP */
02522     return(0);
02523 }
02524
02525 /*
02526     #] TwoExprCompare :
02527     #[ DollarRaiseLow :
02528
02529     Raises or lowers the numerical value of a dollar variable
02530     Not to be used in parallel.
02531 */
02532
02533 static UWORD *dscrat = 0;
02534 static WORD ndscrat;
02535
02536 int DollarRaiseLow(UBYTE *name, LONG value)
02537 {
02538     GETIDENTITY
02539     int num;
02540     DOLLARS d;
02541     int sgn = 1;
02542     WORD lnum[4], nnum, *t1, *t2, i;
02543     UBYTE *s, c;
02544     s = name; while ( *s ) s++;
02545     if ( s[-1] == '-' && s[-2] == '-' && s > name+2 ) s -= 2;
02546     else if ( s[-1] == '+' && s[-2] == '+' && s > name+2 ) s -= 2;
02547     c = *s; *s = 0;
02548     num = GetDollar(name);
02549     *s = c;
02550     d = Dollars + num;
02551     if ( value < 0 ) { value = -value; sgn = -1; }
02552     if ( d->type == DOLZERO ) {
02553         if ( d->where ) M_free(d->where,"DollarRaiseLow");
02554         d->size = MINALLOC;
02555         d->where = (WORD *)Malloc1(d->size*sizeof(WORD),"DollarRaiseLow");
02556         if ( ( value & AWORDMASK ) != 0 ) {
02557             d->where[0] = 6; d->where[1] = value » BITSINWORD;
02558             d->where[2] = (WORD)value; d->where[3] = 1; d->where[4] = 0;
02559             d->where[5] = 5*sgn; d->where[6] = 0;
02560             d->type = DOLTERMS;
02561         }
02562         else {
02563             d->where[0] = 4; d->where[1] = (WORD)value; d->where[2] = 1;
02564             d->where[3] = 3*sgn; d->where[4] = 0;
02565             d->type = DOLNUMBER;
02566         }
02567     }
02568     else if ( d->type == DOLNUMBER || ( d->type == DOLTERMS
02569     && d->where[d->where[0]] == 0
02570     && d->where[0] == ABS(d->where[d->where[0]-1])+1 ) ) {
02571         if ( ( value & AWORDMASK ) != 0 ) {
02572             lnum[0] = value » BITSINWORD;
02573             lnum[1] = (WORD)value; lnum[2] = 1; lnum[3] = 0;
02574             nnum = 2*sgn;
02575         }
02576         else {
02577             lnum[0] = (WORD)value; lnum[1] = 1; nnum = sgn;
02578         }
02579         i = d->where[d->where[0]-1];
02580         i = REDLENG(i);
02581         if ( dscrat == 0 ) {
02582             dscrat = (UWORD *)Malloc1((AM.MaxTal+2)*sizeof(UWORD),"DollarRaiseLow");
02583         }
02584         if ( AddRat(BHEAD (UWORD *)(d->where+1),i,
02585             (UWORD *)lnum,nnum,dscrat,&ndscrat) ) {
02586                 MLOCK(ErrorMessageLock);
02587                 MesCall("DollarRaiseLow");
02588                 MUNLOCK(ErrorMessageLock);
02589                 Terminate(-1);
02590         }
02591         ndscrat = INCLENG(ndscrat);
02592         i = ABS(ndscrat);
02593         if ( i == 0 ) {
02594             M_free(d->where,"DollarRaiseLow");
02595             d->where = 0;
02596             d->type = DOLZERO;
02597             d->size = 0;
02598             return(0);
02599         }
02600         if ( i+2 > d->size ) {
02601             M_free(d->where,"DollarRaiseLow");
02602             d->size = i+2;
02603             if ( d->size < MINALLOC ) d->size = MINALLOC;
02604             d->size = ((d->size+7)/8)*8;
```

```
02605             d->where = (WORD *)Malloc1(d->size*sizeof(WORD),"DollarRaiseLow");
02606         }
02607         t1 = d->where; *t1++ = i+1; t2 = (WORD *)dscrat;
02608         while ( --i > 0 ) *t1++ = *t2++;
02609         *t1++ = ndscrat; *t1 = 0;
02610         d->type = DOLTERMS;
02611     }
02612     return(0);
02613 }
02614
02615 /*
02616     #] DollarRaiseLow :
02617     #[ EvalDoLoopArg :
02618 */
02635 WORD EvalDoLoopArg(PHEAD WORD *arg, WORD par)
02636 {
02637     WORD num, type, *td;
02638     DOLLARS d;
02639     if ( *arg == SNUMBER ) return(arg[1]);
02640     if ( *arg == DOLLAREXPR2 && arg[1] < 0 ) return(-arg[1]-1);
02641     d = Dollars + arg[1];
02642 #ifdef WITHPTHREADS
02643     {
02644         int nummodopt, dtype = -1;
02645         if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
02646             for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
02647                 if ( arg[1] == ModOptdollars[nummodopt].number ) break;
02648             }
02649             if ( nummodopt < NumModOptdollars ) {
02650                 dtype = ModOptdollars[nummodopt].type;
02651                 if ( DollarLocalCopy(dtype) ) {
02652                     d = ModOptdollars[nummodopt].dstruct+AT.identity;
02653                 }
02654             }
02655         }
02656     }
02657 #endif
02658     if ( *arg == DOLLAREXPRESSION ) {
02659         if ( arg[2] != DOLLAREXPR2 ) {  /* end of chain */
02660 endofchain:
02661             type = d->type;
02662             if ( type == DOLZERO ) {}
02663             else if ( type == DOLNUMBER ) {
02664                 td = d->where;
02665                 if ( ( td[0] != 4 ) || ( (td[1]&SPECMASK) != 0 ) || ( td[2] != 1 ) ) {
02666                     MLOCK(ErrorMessageLock);
02667                     if ( par == -1 ) {
02668                         MesPrint("$-variable is not a short number in print statement");
02669                     }
02670                     else {
02671                         MesPrint("$-variable is not a short number in do loop");
02672                     }
02673                     MUNLOCK(ErrorMessageLock);
02674                     Terminate(-1);
02675                 }
02676                 return( td[3] > 0 ? td[1]: -td[1] );
02677             }
02678             else {
02679                 MLOCK(ErrorMessageLock);
02680                 if ( par == -1 ) {
02681                     MesPrint("$-variable is not a number in print statement");
02682                 }
02683                 else {
02684                     MesPrint("$-variable is not a number in do loop");
02685                 }
02686                 MUNLOCK(ErrorMessageLock);
02687                 Terminate(-1);
02688             }
02689             return(0);
02690         }
02691         num = EvalDoLoopArg(BHEAD arg+2,par);
02692     }
02693     else if ( *arg == DOLLAREXPR2 ) {
02694         if ( arg[1] < 0 ) { num = -arg[1]-1; }
02695         else if ( arg[2] != DOLLAREXPR2 && par == -1 ) {
02696             goto endofchain;
02697         }
02698         else              { num = EvalDoLoopArg(BHEAD arg+2,par); }
02699     }
02700     else {
02701         MLOCK(ErrorMessageLock);
02702         if ( par == -1 ) {
02703             MesPrint("Invalid $-variable in print statement");
02704         }
02705         else {
02706             MesPrint("Invalid $-variable in do loop");
02707         }
```

```
02708         MUNLOCK(ErrorMessageLock);
02709         Terminate(-1);
02710         return(0);
02711     }
02712     if ( num == 0 ) return(d->nfactors);
02713     if ( num > d->nfactors || num < 1 ) {
02714         MLOCK(ErrorMessageLock);
02715         if ( par == -1 ) {
02716             MesPrint("Not a valid factor number for $-variable in print statement");
02717         }
02718         else {
02719             MesPrint("Not a valid factor number for $-variable in do loop");
02720         }
02721         MUNLOCK(ErrorMessageLock);
02722         Terminate(-1);
02723         return(0);
02724     }
02725     if ( d->factors[num].type == DOLNUMBER )
02726         return(d->factors[num].value);
02727     else {  /* If correct, type can only be DOLNUMBER or DOLTERMS */
02728         MLOCK(ErrorMessageLock);
02729         if ( par == -1 ) {
02730             MesPrint("$-variable in print statement is not a number");
02731         }
02732         else {
02733             MesPrint("$-variable in do loop is not a number");
02734         }
02735         MUNLOCK(ErrorMessageLock);
02736         Terminate(-1);
02737         return(0);
02738     }
02739 }
02740
02741 /*
02742     #] EvalDoLoopArg :
02743     #[ TestDoLoop :
02744 */
02745
02746 WORD TestDoLoop(PHEAD WORD *lhsbuf, WORD level)
02747 {
02748     GETBIDENTITY
02749     WORD start,finish,incr;
02750     WORD *h;
02751     DOLLARS d;
02752     h = lhsbuf + 4; /* address of the start value */
02753     start = EvalDoLoopArg(BHEAD h,0);
02754     while ( ( *h == DOLLAREXPRESSION || *h == DOLLAREXPR2 )
02755         && ( h[2] == DOLLAREXPR2 ) ) h += 2;
02756     h += 2;
02757     finish = EvalDoLoopArg(BHEAD h,0);
02758     while ( ( *h == DOLLAREXPRESSION || *h == DOLLAREXPR2 )
02759         && ( h[2] == DOLLAREXPR2 ) ) h += 2;
02760     h += 2;
02761     incr = EvalDoLoopArg(BHEAD h,0);
02762
02763     if ( ( finish == start ) || ( finish > start && incr > 0 )
02764     || ( finish < start && incr < 0 ) ) {}
02765     else { level = lhsbuf[3]; } /* skips the loop */
02766 /*
02767     Put start in the dollar variable indicated by lhsbuf[2]
02768 */
02769     d = Dollars + lhsbuf[2];
02770 #ifdef WITHPTHREADS
02771     {
02772         int nummodopt, dtype = -1;
02773         if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
02774             for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
02775                 if ( lhsbuf[2] == ModOptdollars[nummodopt].number ) break;
02776             }
02777             if ( nummodopt < NumModOptdollars ) {
02778                 dtype = ModOptdollars[nummodopt].type;
02779                 if ( DollarLocalCopy(dtype) ) {
02780                     d = ModOptdollars[nummodopt].dstruct+AT.identity;
02781                 }
02782             }
02783         }
02784     }
02785 #endif
02786
02787     if ( d->size < MINALLOC ) {
02788         if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollar contents");
02789         d->size = MINALLOC;
02790         d->where = (WORD *)Malloc1(d->size*sizeof(WORD),"dollar contents");
02791     }
02792     if ( start > 0 ) {
02793         d->where[0] = 4;
02794         d->where[1] = start;
```

```
02795         d->where[2] = 1;
02796         d->where[3] = 3;
02797         d->where[4] = 0;
02798         d->type = DOLNUMBER;
02799     }
02800     else if ( start < 0 ) {
02801         d->where[0] = 4;
02802         d->where[1] = -start;
02803         d->where[2] = 1;
02804         d->where[3] = -3;
02805         d->where[4] = 0;
02806         d->type = DOLNUMBER;
02807     }
02808     else
02809         d->type = DOLZERO;
02810 
02811     if ( d == Dollars + lhsbuf[2] ) {
02812         cbuf[AM.dbufnum].CanCommu[lhsbuf[2]] = 0;
02813         cbuf[AM.dbufnum].NumTerms[lhsbuf[2]] = 1;
02814         cbuf[AM.dbufnum].rhs[lhsbuf[2]] = d->where;
02815     }
02816     return(level);
02817 }
02818 
02819 /*
02820     #] TestDoLoop :
02821     #[ TestEndDoLoop :
02822 */
02823 
02824 WORD TestEndDoLoop(PHEAD WORD *lhsbuf, WORD level)
02825 {
02826     GETBIDENTITY
02827     WORD start,finish,incr,value;
02828     WORD *h;
02829     DOLLARS d;
02830     h = lhsbuf + 4; /* address of the start value */
02831     start = EvalDoLoopArg(BHEAD h,0);
02832     while ( ( *h == DOLLAREXPRESSION || *h == DOLLAREXPR2 )
02833         && ( h[2] == DOLLAREXPR2 ) ) h += 2;
02834     h += 2;
02835     finish = EvalDoLoopArg(BHEAD h,0);
02836     while ( ( *h == DOLLAREXPRESSION || *h == DOLLAREXPR2 )
02837         && ( h[2] == DOLLAREXPR2 ) ) h += 2;
02838     h += 2;
02839     incr = EvalDoLoopArg(BHEAD h,0);
02840 
02841     if ( ( finish == start ) || ( finish > start && incr > 0 )
02842     || ( finish < start && incr < 0 ) ) {}
02843     else { level = lhsbuf[3]; } /* skips the loop */
02844 /*
02845     Put start in the dollar variable indicated by lhsbuf[2]
02846 */
02847     d = Dollars + lhsbuf[2];
02848 #ifdef WITHPTHREADS
02849     {
02850         int nummodopt, dtype = -1;
02851         if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
02852             for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
02853                 if ( lhsbuf[2] == ModOptdollars[nummodopt].number ) break;
02854             }
02855             if ( nummodopt < NumModOptdollars ) {
02856                 dtype = ModOptdollars[nummodopt].type;
02857                 if ( DollarLocalCopy(dtype) ) {
02858                     d = ModOptdollars[nummodopt].dstruct+AT.identity;
02859                 }
02860             }
02861         }
02862     }
02863 #endif
02864 /*
02865     Get the value
02866 */
02867     if ( d->type == DOLZERO ) {
02868         value = 0;
02869     }
02870     else if ( ( d->type == DOLNUMBER || d->type == DOLTERMS )
02871     && ( d->where[4] == 0 ) && ( d->where[0] == 4 )
02872     && ( d->where[1] > 0 ) && ( d->where[2] == 1 ) ) {
02873         value = ( d->where[3] < 0 ) ? -d->where[1]: d->where[1];
02874     }
02875     else {
02876         MLOCK(ErrorMessageLock);
02877         MesPrint("Wrong type of object in do loop parameter");
02878         MUNLOCK(ErrorMessageLock);
02879         Terminate(-1);
02880         return(level);
02881     }
```

```
02882     value += incr;
02883     if ( ( finish > start && value <= finish ) ||
02884          ( finish < start && value >= finish ) ||
02885          ( finish == start && value == finish ) ) {}
02886     else level = lhsbuf[3];
02887
02888     if ( d->size < MINALLOC ) {
02889         if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollar contents");
02890         d->size = MINALLOC;
02891         d->where = (WORD *)Malloc1(d->size*sizeof(WORD),"dollar contents");
02892     }
02893     if ( value > 0 ) {
02894         d->where[0] = 4;
02895         d->where[1] = value;
02896         d->where[2] = 1;
02897         d->where[3] = 3;
02898         d->where[4] = 0;
02899         d->type = DOLNUMBER;
02900     }
02901     else if ( start < 0 ) {
02902         d->where[0] = 4;
02903         d->where[1] = -value;
02904         d->where[2] = 1;
02905         d->where[3] = -3;
02906         d->where[4] = 0;
02907         d->type = DOLNUMBER;
02908     }
02909     else
02910         d->type = DOLZERO;
02911
02912     if ( d == Dollars + lhsbuf[2] ) {
02913         cbuf[AM.dbufnum].CanCommu[lhsbuf[2]] = 0;
02914         cbuf[AM.dbufnum].NumTerms[lhsbuf[2]] = 1;
02915         cbuf[AM.dbufnum].rhs[lhsbuf[2]] = d->where;
02916     }
02917     return(level);
02918 }
02919
02920 /*
02921     #] TestEndDoLoop :
02922     #[ DollarFactorize :
02923 */
02936 /* #define STEP2 */
02937 #define STEP2
02938
02939 int DollarFactorize(PHEAD WORD numdollar)
02940 {
02941     GETBIDENTITY
02942     DOLLARS d = Dollars + numdollar;
02943     CBUF *C, *CC;
02944     WORD *oldworkpointer;
02945     WORD *buf1, *t, *term, *buf1content, *buf2, *termextra;
02946     WORD *buf3, *argextra;
02947 #ifdef STEP2
02948     WORD *tstop, pow, *r;
02949 #endif
02950     int i, j, jj, action = 0, sign = 1;
02951     LONG insize, ii;
02952     WORD startebuf = cbuf[AT.ebufnum].numrhs;
02953     WORD nfactors, factorsincontent, extrafactor = 0;
02954     WORD oldsorttype = AR.SortType;
02955
02956 #ifdef WITHPTHREADS
02957     int nummodopt, dtype;
02958     dtype = -1;
02959     if ( AS.MultiThreaded ) {
02960         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
02961             if ( numdollar == ModOptdollars[nummodopt].number ) break;
02962         }
02963         if ( nummodopt < NumModOptdollars ) {
02964             dtype = ModOptdollars[nummodopt].type;
02965             if ( DollarLocalCopy(dtype) ) {
02966                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
02967             }
02968             else {
02969                 LOCK(d->pthreadslock);
02970             }
02971         }
02972     }
02973 #endif
02974     CleanDollarFactors(d);
02975 #ifdef WITHPTHREADS
02976     if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
02977 #endif
02978     if ( d->type != DOLTERMS ) {    /* only one term */
02979         if ( d->type != DOLZERO ) d->nfactors = 1;
02980         return(0);
```

```
02981     }
02982     if ( d->where[d->where[0]] == 0 ) { /* only one term. easy */
02983     }
02984 /*
02985     Here should come the code for the factorization
02986     We copied the routine ArgFactorize in argument.c and changed the
02987     memory management completely. For the actual factorization it
02988     calls WORD *DoFactorizeDollar(PHEAD WORD *expr) which allocates
02989     space for the answer. Notation:
02990      term,...,term,0,term,...,term,0,term,...,term,0,0
02991
02992         #[ Step 1: sort the terms properly and/or make copy  --> buf1,insize
02993 */
02994     term = d->where;
02995     AR.SortType = SORTHIGHFIRST;
02996     if ( oldsorttype != AR.SortType ) {
02997         NewSort(BHEAD0);
02998         while ( *term ) {
02999             t = term + *term;
03000             if ( AN.ncmod != 0 ) {
03001                 if ( AN.ncmod != 1 || ( (WORD)AN.cmod[0] < 0 ) ) {
03002                     AR.SortType = oldsorttype;
03003                     MLOCK(ErrorMessageLock);
03004                     MesPrint("Factorization modulus a number, greater than a WORD not implemented.");
03005                     MUNLOCK(ErrorMessageLock);
03006                     Terminate(-1);
03007                 }
03008                 if ( Modulus(term) ) {
03009                     AR.SortType = oldsorttype;
03010                     MLOCK(ErrorMessageLock);
03011                     MesCall("DollarFactorize");
03012                     MUNLOCK(ErrorMessageLock);
03013                     Terminate(-1);
03014                 }
03015                 if ( !*term) { term = t; continue; }
03016             }
03017             StoreTerm(BHEAD term);
03018             term = t;
03019         }
03020         AN.tryterm = 0; /* for now */
03021         EndSort(BHEAD (WORD *)((void *)(&buf1)),2);
03022         t = buf1; while ( *t ) t += *t;
03023         insize = t - buf1;
03024     }
03025     else {
03026         t = term; while ( *t ) t += *t;
03027         ii = insize = t - term;
03028         buf1 = (WORD *)Malloc1((insize+1)*sizeof(WORD),"DollarFactorize-1");
03029         t = buf1;
03030         NCOPY(t,term,ii);
03031         *t++ = 0;
03032     }
03033 /*
03034         #] Step 1:
03035         #[ Step 2: take out the 'content'.
03036 */
03037 #ifdef STEP2
03038     buf1content = TermMalloc("DollarContent");
03039     AN.tryterm = -1;
03040     if ( ( buf2 = TakeContent(BHEAD buf1,buf1content) ) == 0 ) {
03041         AN.tryterm = 0;
03042         TermFree(buf1content,"DollarContent");
03043         M_free(buf1,"DollarFactorize-1");
03044         AR.SortType = oldsorttype;
03045         MLOCK(ErrorMessageLock);
03046         MesCall("DollarFactorize");
03047         MUNLOCK(ErrorMessageLock);
03048         Terminate(-1);
03049         return(1);
03050     }
03051     else if ( ( buf1content[0] == 4 ) && ( buf1content[1] == 1 ) &&
03052               ( buf1content[2] == 1 ) && ( buf1content[3] == 3 ) ) { /* Nothing happened */
03053         AN.tryterm = 0;
03054         if ( buf2 != buf1 ) {
03055             M_free(buf2,"DollarFactorize-2");
03056             buf2 = buf1;
03057         }
03058         factorsincontent = 0;
03059     }
03060     else {
03061 /*
03062         The way we took out objects is rather brutish. We have to normalize
03063 */
03064         AN.tryterm = 0;
03065         if ( buf2 != buf1 ) M_free(buf1,"DollarFactorize-1");
03066         buf1 = buf2;
03067         t = buf1; while ( *t ) t += *t;
```

```
03068         insize = t - buf1;
03069 /*
03070         Now analyse how many factors there are in the content
03071 */
03072         factorsincontent = 0;
03073         term = buf1content;
03074         tstop = term + *term;
03075         if ( tstop[-1] < 0 ) factorsincontent++;
03076         if ( ABS(tstop[-1]) == 3 && tstop[-2] == 1 && tstop[-3] == 1 ) {
03077             tstop -= ABS(tstop[-1]);
03078         }
03079         else {
03080             factorsincontent++;
03081             tstop -= ABS(tstop[-1]);
03082         }
03083         term++;
03084         while ( term < tstop ) {
03085             switch ( *term ) {
03086                 case SYMBOL:
03087                     t = term+2; i = (term[1]-2)/2;
03088                     while ( i > 0 ) {
03089                         factorsincontent += ABS(t[1]);
03090                         i--; t += 2;
03091                     }
03092                     break;
03093                 case DOTPRODUCT:
03094                     t = term+2; i = (term[1]-2)/3;
03095                     while ( i > 0 ) {
03096                         factorsincontent += ABS(t[2]);
03097                         i--; t += 3;
03098                     }
03099                     break;
03100                 case VECTOR:
03101                 case DELTA:
03102                     factorsincontent += (term[1]-2)/2;
03103                     break;
03104                 case INDEX:
03105                     factorsincontent += term[1]-2;
03106                     break;
03107                 default:
03108                     if ( *term >= FUNCTION ) factorsincontent++;
03109                     break;
03110             }
03111             term += term[1];
03112         }
03113     }
03114 #else
03115     factorsincontent = 0;
03116     buf1content = 0;
03117 #endif
03118 /*
03119         #] Step 2: take out the 'content'.
03120         #[ Step 3: ConvertToPoly
03121                 if there are objects that are not SYMBOLs,
03122                 invoke ConvertToPoly
03123                 We keep the original in buf1 in case there are no factors
03124 */
03125     t = buf1;
03126     while ( *t ) {
03127         if ( ( t[1] != SYMBOL ) && ( *t != (ABS(t[*t-1])+1) ) ) {
03128             action = 1; break;
03129         }
03130         t += *t;
03131     }
03132     if ( DetCommu(buf1) > 1 ) {
03133         MesPrint("Cannot factorize a $-expression with more than one noncommuting object");
03134         AR.SortType = oldsorttype;
03135         M_free(buf1,"DollarFactorize-2");
03136         if ( buf1content ) TermFree(buf1content,"DollarContent");
03137         MesCall("DollarFactorize");
03138         Terminate(-1);
03139         return(-1);
03140     }
03141     if ( action ) {
03142         t = buf1;
03143         termextra = AT.WorkPointer;
03144         NewSort(BHEAD0);
03145         NewSort(BHEAD0);
03146         while ( *t ) {
03147             if ( LocalConvertToPoly(BHEAD t,termextra,startebuf,0) < 0 ) {
03148 getout:
03149                 AR.SortType = oldsorttype;
03150                 M_free(buf1,"DollarFactorize-2");
03151                 if ( buf1content ) TermFree(buf1content,"DollarContent");
03152                 MesCall("DollarFactorize");
03153                 Terminate(-1);
03154                 return(-1);
```

```
03155             }
03156             StoreTerm(BHEAD termextra);
03157             t += *t;
03158         }
03159         AN.tryterm = 0; /* for now */
03160         if ( EndSort(BHEAD (WORD *)((void *)(&buf2)),2) < 0 ) { goto getout; }
03161         LowerSortLevel();
03162         t = buf2; while ( *t > 0 ) t += *t;
03163     }
03164     else {
03165         buf2 = buf1;
03166     }
03167 /*
03168         #] Step 3: ConvertToPoly
03169         #[ Step 4: Now the hard work.
03170 */
03171     if ( ( buf3 = poly_factorize_dollar(BHEAD buf2) ) == 0 ) {
03172         MesCall("DollarFactorize");
03173         AR.SortType = oldsorttype;
03174         if ( buf2 != buf1 && buf2 ) M_free(buf2,"DollarFactorize-3");
03175         M_free(buf1,"DollarFactorize-3");
03176         if ( buf1content ) TermFree(buf1content,"DollarContent");
03177         Terminate(-1);
03178         return(-1);
03179     }
03180     if ( buf2 != buf1 && buf2 ) {
03181         M_free(buf2,"DollarFactorize-3");
03182         buf2 = 0;
03183     }
03184     term = buf3;
03185     AR.SortType = oldsorttype;
03186 /*
03187     Count the factors and strip a factor -1
03188 */
03189     nfactors = 0;
03190     while ( *term ) {
03191 #ifdef STEP2
03192         if ( *term == 4 && term[4] == 0 && term[3] == -3 && term[2] == 1
03193             && term[1] == 1 ) {
03194             WORD *tt1, *tt2, *ttstop;
03195             sign = -sign;
03196             tt1 = term; tt2 = term + *term + 1;
03197             ttstop = tt2;
03198             while ( *ttstop ) {
03199                 while ( *ttstop ) ttstop += *ttstop;
03200                 ttstop++;
03201             }
03202             while ( tt2 < ttstop ) *tt1++ = *tt2++;
03203             *tt1 = 0;
03204             factorsincontent++;
03205             extrafactor++;
03206         }
03207         else
03208 #endif
03209         {
03210             term += *term;
03211             while ( *term ) { term += *term; }
03212             nfactors++; term++;
03213         }
03214     }
03215 /*
03216     We have now:
03217         buf1: the original before ConvertToPoly for if only one factor
03218         buf3: the factored expression with nfactors factors
03219
03220         #] Step 4:
03221         #[ Step 5: ConvertFromPoly
03222                 If ConvertToPoly was used, use now ConvertFromPoly
03223                 Be careful: there should be more than one factor now.
03224 */
03225 #ifdef WITHPTHREADS
03226     if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { LOCK(d->pthreadslock); }
03227 #endif
03228     if ( nfactors ==  1 && extrafactor == 0 ) { /* we can use the buf1 contents */
03229         if ( factorsincontent == 0 ) {
03230             d->nfactors = 1;
03231 #ifdef WITHPTHREADS
03232             if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
03233 #endif
03234 /*
03235             We used here (before 3-sep-2015) the original and did not make
03236             provisions for having a factors struct, figuring that all info
03237             is identical to the full dollar. This makes things too
03238             complicated at later stages.
03239 */
03240             d->factors = (FACDOLLAR *)Malloc1(sizeof(FACDOLLAR),"factors in dollar");
03241             term = buf1; while ( *term ) term += *term;
```

```
03242             d->factors[0].size = i = term - buf1;
03243             d->factors[0].where = t = (WORD *)Malloc1(sizeof(WORD)*(i+1),"DollarFactorize-5");
03244             term = buf1; NCOPY(t,term,i); *t = 0;
03245             AR.SortType = oldsorttype;
03246             M_free(buf3,"DollarFactorize-4");
03247             if ( buf2 != buf1 && buf2 ) M_free(buf2,"DollarFactorize-4");
03248             M_free(buf1,"DollarFactorize-4");
03249             if ( buf1content ) TermFree(buf1content,"DollarContent");
03250             return(0);
03251         }
03252         else {
03253             d->factors = (FACDOLLAR *)Malloc1(sizeof(FACDOLLAR)*(nfactors+factorsincontent),"factors
      in dollar");
03254             term = buf1; while ( *term ) term += *term;
03255             d->factors[0].size = i = term - buf1;
03256             d->factors[0].where = t = (WORD *)Malloc1(sizeof(WORD)*(i+1),"DollarFactorize-5");
03257             term = buf1; NCOPY(t,term,i); *t = 0;
03258             M_free(buf3,"DollarFactorize-4");
03259             buf3 = 0;
03260             if ( buf2 != buf1 && buf2 ) {
03261                 M_free(buf2,"DollarFactorize-4");
03262                 buf2 = 0;
03263             }
03264         }
03265     }
03266     else if ( action ) {
03267         C = cbuf+AC.cbufnum;
03268         CC = cbuf+AT.ebufnum;
03269         oldworkpointer = AT.WorkPointer;
03270         d->factors = (FACDOLLAR *)Malloc1(sizeof(FACDOLLAR)*(nfactors+factorsincontent),"factors in
      dollar");
03271         term = buf3;
03272         for ( i = 0; i < nfactors; i++ ) {
03273             argextra = AT.WorkPointer;
03274             NewSort(BHEAD0);
03275             NewSort(BHEAD0);
03276             while ( *term ) {
03277                 if ( ConvertFromPoly(BHEAD term,argextra,numxsymbol,CC->numrhs-startebuf+numxsymbol
03278                 ,startebuf-numxsymbol,1) <= 0 ) {
03279                     LowerSortLevel();
03280 getout2:            AR.SortType = oldsorttype;
03281                     M_free(d->factors,"factors in dollar");
03282                     d->factors = 0;
03283 #ifdef WITHPTHREADS
03284                     if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
03285 #endif
03286                     M_free(buf3,"DollarFactorize-4");
03287                     if ( buf2 != buf1 && buf2 ) M_free(buf2,"DollarFactorize-4");
03288                     M_free(buf1,"DollarFactorize-4");
03289                     if ( buf1content ) TermFree(buf1content,"DollarContent");
03290                     return(-3);
03291                 }
03292                 AT.WorkPointer = argextra + *argextra;
03293 /*
03294                 ConvertFromPoly leaves terms with subexpressions. Hence:
03295 */
03296                 if ( Generator(BHEAD argextra,C->numlhs+1) ) {
03297                     goto getout2;
03298                 }
03299                 term += *term;
03300             }
03301             term++;
03302             AT.WorkPointer = oldworkpointer;
03303             AN.tryterm = 0; /* for now */
03304             EndSort(BHEAD (WORD *)((void *)(&(d->factors[i].where))),2);
03305             LowerSortLevel();
03306             d->factors[i].type = DOLTERMS;
03307             t = d->factors[i].where;
03308             while ( *t ) t += *t;
03309             d->factors[i].size = t - d->factors[i].where;
03310         }
03311         CC->numrhs = startebuf;
03312     }
03313     else {
03314         C = cbuf+AC.cbufnum;
03315         oldworkpointer = AT.WorkPointer;
03316         d->factors = (FACDOLLAR *)Malloc1(sizeof(FACDOLLAR)*(nfactors+factorsincontent),"factors in
      dollar");
03317         term = buf3;
03318         for ( i = 0; i < nfactors; i++ ) {
03319             NewSort(BHEAD0);
03320             while ( *term ) {
03321                 argextra = oldworkpointer;
03322                 j = *term;
03323                 NCOPY(argextra,term,j)
03324                 AT.WorkPointer = argextra;
03325                 if ( Generator(BHEAD oldworkpointer,C->numlhs+1) ) {
```

```
03326                       goto getout2;
03327                   }
03328               }
03329               term++;
03330               AT.WorkPointer = oldworkpointer;
03331               AN.tryterm = 0; /* for now */
03332               EndSort(BHEAD (WORD *)((void *)(&(d->factors[i].where))),2);
03333               d->factors[i].type = DOLTERMS;
03334               t = d->factors[i].where;
03335               while ( *t ) t += *t;
03336               d->factors[i].size = t - d->factors[i].where;
03337           }
03338       }
03339       d->nfactors = nfactors + factorsincontent;
03340   /*
03341           #] Step 5: ConvertFromPoly
03342           #[ Step 6: The factors of the content
03343   */
03344       if ( buf3 ) M_free(buf3,"DollarFactorize-5");
03345       if ( buf2 != buf1 && buf2 ) M_free(buf2,"DollarFactorize-5");
03346       M_free(buf1,"DollarFactorize-5");
03347       j = nfactors;
03348   #ifdef STEP2
03349       term = buf1content;
03350       tstop = term + *term;
03351       if ( tstop[-1] < 0 ) { tstop[-1] = -tstop[-1]; sign = -sign; }
03352       tstop -= tstop[-1];
03353       term++;
03354       while ( term < tstop ) {
03355           switch ( *term ) {
03356               case SYMBOL:
03357                   t = term+2; i = (term[1]-2)/2;
03358                   while ( i > 0 ) {
03359                       if ( t[1] < 0 ) { t[1] = -t[1]; pow = -1; }
03360                       else { pow = 1; }
03361                       for ( jj = 0; jj < t[1]; jj++ ) {
03362                           r = d->factors[j].where = (WORD *)Malloc1(9*sizeof(WORD),"factor");
03363                           r[0] = 8; r[1] = SYMBOL; r[2] = 4; r[3] = *t; r[4] = pow;
03364                           r[5] = 1; r[6] = 1; r[7] = 3; r[8] = 0;
03365                           d->factors[j].type = DOLTERMS;
03366                           d->factors[j].size = 8;
03367                           j++;
03368                       }
03369                       i--; t += 2;
03370                   }
03371                   break;
03372               case DOTPRODUCT:
03373                   t = term+2; i = (term[1]-2)/3;
03374                   while ( i > 0 ) {
03375                       if ( t[2] < 0 ) { t[2] = -t[2]; pow = -1; }
03376                       else { pow = 1; }
03377                       for ( jj = 0; jj < t[2]; jj++ ) {
03378                           r = d->factors[j].where = (WORD *)Malloc1(10*sizeof(WORD),"factor");
03379                           r[0] = 9; r[1] = DOTPRODUCT; r[2] = 5; r[3] = t[0]; r[4] = t[1];
03380                           r[5] = pow; r[6] = 1; r[7] = 1; r[8] = 3; r[9] = 0;
03381                           d->factors[j].type = DOLTERMS;
03382                           d->factors[j].size = 9;
03383                           j++;
03384                       }
03385                       i--; t += 3;
03386                   }
03387                   break;
03388               case VECTOR:
03389               case DELTA:
03390                   t = term+2; i = (term[1]-2)/2;
03391                   while ( i > 0 ) {
03392                       for ( jj = 0; jj < t[1]; jj++ ) {
03393                           r = d->factors[j].where = (WORD *)Malloc1(9*sizeof(WORD),"factor");
03394                           r[0] = 8; r[1] = *term; r[2] = 4; r[3] = *t; r[4] = t[1];
03395                           r[5] = 1; r[6] = 1; r[7] = 3; r[8] = 0;
03396                           d->factors[j].type = DOLTERMS;
03397                           d->factors[j].size = 8;
03398                           j++;
03399                       }
03400                       i--; t += 2;
03401                   }
03402                   break;
03403               case INDEX:
03404                   t = term+2; i = term[1]-2;
03405                   while ( i > 0 ) {
03406                       for ( jj = 0; jj < t[1]; jj++ ) {
03407                           r = d->factors[j].where = (WORD *)Malloc1(8*sizeof(WORD),"factor");
03408                           r[0] = 7; r[1] = *term; r[2] = 3; r[3] = *t;
03409                           r[4] = 1; r[5] = 1; r[6] = 3; r[7] = 0;
03410                           d->factors[j].type = DOLTERMS;
03411                           d->factors[j].size = 7;
03412                           j++;
```

```
03413                     }
03414                     i--; t++;
03415                 }
03416                 break;
03417             default:
03418                 if ( *term >= FUNCTION ) {
03419                     r = d->factors[j].where = (WORD *)Malloc1((term[1]+5)*sizeof(WORD),"factor");
03420                     *r++ = d->factors[j].size = term[1]+4;
03421                     for ( jj = 0; jj < t[1]; jj++ ) *r++ = term[jj];
03422                     *r++ = 1; *r++ = 1; *r++ = 3; *r = 0;
03423                     j++;
03424                 }
03425                 break;
03426         }
03427         term += term[1];
03428     }
03429 #endif
03430 /*
03431         #] Step 6:
03432         #[ Step 7: Numerical factors
03433 */
03434 #ifdef STEP2
03435     term = buf1content;
03436     tstop = term + *term;
03437     if ( tstop[-1] == 3 && tstop[-2] == 1 && tstop[-3] == 1 ) {}
03438     else if ( tstop[-1] == 3 && tstop[-2] == 1 && (UWORD)(tstop[-3]) <= MAXPOSITIVE ) {
03439         d->factors[j].where = 0;
03440         d->factors[j].size  = 0;
03441         d->factors[j].type  = DOLNUMBER;
03442         d->factors[j].value = sign*tstop[-3];
03443         sign = 1;
03444         j++;
03445     }
03446     else {
03447         d->factors[j].where = r = (WORD *)Malloc1((tstop[-1]+2)*sizeof(WORD),"numfactor");
03448         d->factors[j].size  = tstop[-1]+1;
03449         d->factors[j].type  = DOLTERMS;
03450         d->factors[j].value = 0;
03451         i = tstop[-1];
03452         t = tstop - i;
03453         *r++ = tstop[-1]+1;
03454         NCOPY(r,t,i);
03455         *r = 0;
03456         if ( sign < 0 ) {
03457             r = d->factors[j].where;
03458             while ( *r ) {
03459                 r += *r; r[-1] = -r[-1];
03460             }
03461             sign = 1;
03462         }
03463         j++;
03464     }
03465 #endif
03466     if ( sign < 0 ) { /* Note that this guy should come first */
03467         for ( jj = j; jj > 0; jj-- ) {
03468             d->factors[jj] = d->factors[jj-1];
03469         }
03470         d->factors[0].where = 0;
03471         d->factors[0].size  = 0;
03472         d->factors[0].type  = DOLNUMBER;
03473         d->factors[0].value = -1;
03474         j++;
03475     }
03476     d->nfactors = j;
03477     if ( buf1content ) TermFree(buf1content,"DollarContent");
03478 /*
03479         #] Step 7:
03480         #[ Step 8: Sorting the factors
03481
03482     There are d->nfactors factors. Look which ones have a 'where'
03483     Sort them by bubble sort
03484 */
03485     if ( d->nfactors > 1 ) {
03486         WORD ***fac, j1, j2, k, ret, *s1, *s2, *s3;
03487         LONG **facsize, x;
03488         facsize = (LONG **)Malloc1((sizeof(WORD **)+sizeof(LONG *))*d->nfactors,"SortDollarFactors");
03489         fac = (WORD ***)(facsize+d->nfactors);
03490         k = 0;
03491         for ( j = 0; j < d->nfactors; j++ ) {
03492             if ( d->factors[j].where ) {
03493                 fac[k] = &(d->factors[j].where);
03494                 facsize[k] = &(d->factors[j].size);
03495                 k++;
03496             }
03497         }
03498         if ( k > 1 ) {
03499             for ( j = 1; j < k; j++ ) { /* bubble sort */
```

```
03500                 j1 = j; j2 = j1-1;
03501 nextj1:;
03502                 s1 = *(fac[j1]); s2 = *(fac[j2]);
03503                 while ( *s1 && *s2 ) {
03504                     if ( ( ret = CompareTerms(BHEAD s2, s1, (WORD)2) ) == 0 ) {
03505                         s1 += *s1; s2 += *s2;
03506                     }
03507                     else if ( ret > 0 ) goto nextj;
03508                     else {
03509 exch:
03510                         s3 = *(fac[j1]); *(fac[j1]) = *(fac[j2]); *(fac[j2]) = s3;
03511                         x = *(facsize[j1]); *(facsize[j1]) = *(facsize[j2]); *(facsize[j2]) = x;
03512                         j1--; j2--;
03513                         if ( j1 > 0 ) goto nextj1;
03514                         goto nextj;
03515                     }
03516                 }
03517                 if ( *s1 ) goto nextj;
03518                 if ( *s2 ) goto exch;
03519 nextj:;
03520             }
03521         }
03522         M_free(facsize,"SortDollarFactors");
03523     }
03524 /*
03525         #] Step 8:
03526 */
03527 #ifdef WITHPTHREADS
03528     if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
03529 #endif
03530     return(0);
03531 }
03532
03533 /*
03534     #] DollarFactorize :
03535     #[ CleanDollarFactors :
03536 */
03537
03538 void CleanDollarFactors(DOLLARS d)
03539 {
03540     int i;
03541     if ( d->nfactors >= 1 ) {
03542         for ( i = 0; i < d->nfactors; i++ ) {
03543             if ( d->factors )
03544                 if ( d->factors[i].where )
03545                     M_free(d->factors[i].where,"dollar factors");
03546         }
03547     }
03548     if ( d->factors ) {
03549         M_free(d->factors,"dollar factors");
03550         d->factors = 0;
03551     }
03552     d->nfactors = 0;
03553 }
03554
03555 /*
03556     #] CleanDollarFactors :
03557     #[ TakeDollarContent :
03558 */
03559
03560 WORD *TakeDollarContent(PHEAD WORD *dollarbuffer, WORD **factor)
03561 {
03562     WORD *remain, *t;
03563     int pow;
03564 /*
03565     We force the sign of the first term to be positive.
03566 */
03567     t = dollarbuffer; pow = 1;
03568     t += *t;
03569     if ( t[-1] < 0 ) {
03570         pow = 0;
03571         t[-1] = -t[-1];
03572         while ( *t ) {
03573             t += *t; t[-1] = -t[-1];
03574         }
03575     }
03576 /*
03577     Now the GCD of the numerators and the LCM of the denominators:
03578 */
03579     if ( AN.cmod != 0 ) {
03580         if ( ( *factor = MakeDollarMod(BHEAD dollarbuffer,&remain) ) == 0 ) {
03581             Terminate(-1);
03582         }
03583         if ( pow == 0 ) {
03584             (*factor)[**factor-1] = -(*factor)[**factor-1];
03585             (*factor)[**factor-1] += AN.cmod[0];
03586         }
```

```
03587     }
03588     else {
03589         if ( ( *factor = MakeDollarInteger(BHEAD dollarbuffer,&remain) ) == 0 ) {
03590             Terminate(-1);
03591         }
03592         if ( pow == 0 ) {
03593             (*factor)[**factor-1] = -(*factor)[**factor-1];
03594         }
03595     }
03596     return(remain);
03597 }
03598
03599 /*
03600     #] TakeDollarContent :
03601     #[ MakeDollarInteger :
03602 */
03612 WORD *MakeDollarInteger(PHEAD WORD *bufin,WORD **bufout)
03613 {
03614     GETBIDENTITY
03615     UWORD *GCDbuffer, *GCDbuffer2, *LCMbuffer, *LCMb, *LCMc;
03616     WORD *r, *r1, *r2, *r3, *rnext, i, k, j, *oldworkpointer, *factor;
03617     WORD kGCD, kLCM, kGCD2, kkLCM, jLCM, jGCD;
03618     CBUF *C = cbuf+AC.cbufnum;
03619
03620     GCDbuffer = NumberMalloc("MakeDollarInteger");
03621     GCDbuffer2 = NumberMalloc("MakeDollarInteger");
03622     LCMbuffer = NumberMalloc("MakeDollarInteger");
03623     LCMb = NumberMalloc("MakeDollarInteger");
03624     LCMc = NumberMalloc("MakeDollarInteger");
03625     r = bufin;
03626 /*
03627     First take the first term to load up the LCM and the GCD
03628 */
03629     r2 = r + *r;
03630     j = r2[-1];
03631     r3 = r2 - ABS(j);
03632     k = REDLENG(j);
03633     if ( k < 0 ) k = -k;
03634     while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
03635     for ( kGCD = 0; kGCD < k; kGCD++ ) GCDbuffer[kGCD] = r3[kGCD];
03636     k = REDLENG(j);
03637     if ( k < 0 ) k = -k;
03638     r3 += k;
03639     while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
03640     for ( kLCM = 0; kLCM < k; kLCM++ ) LCMbuffer[kLCM] = r3[kLCM];
03641     r1 = r2;
03642 /*
03643     Now go through the rest of the terms in this argument.
03644 */
03645     while ( *r1 ) {
03646         r2 = r1 + *r1;
03647         j = r2[-1];
03648         r3 = r2 - ABS(j);
03649         k = REDLENG(j);
03650         if ( k < 0 ) k = -k;
03651         while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
03652         if ( ( ( GCDbuffer[0] == 1 ) && ( kGCD == 1 ) ) ) {
03653 /*
03654             GCD is already 1
03655 */
03656         }
03657         else if ( ( ( k != 1 ) || ( r3[0] != 1 ) ) ) {
03658             if ( GcdLong(BHEAD GCDbuffer,kGCD,(UWORD *)r3,k,GCDbuffer2,&kGCD2) ) {
03659                 goto MakeDollarIntegerErr;
03660             }
03661             kGCD = kGCD2;
03662             for ( i = 0; i < kGCD; i++ ) GCDbuffer[i] = GCDbuffer2[i];
03663         }
03664         else {
03665             kGCD = 1; GCDbuffer[0] = 1;
03666         }
03667         k = REDLENG(j);
03668         if ( k < 0 ) k = -k;
03669         r3 += k;
03670         while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
03671         if ( ( ( LCMbuffer[0] == 1 ) && ( kLCM == 1 ) ) ) {
03672             for ( kLCM = 0; kLCM < k; kLCM++ )
03673                 LCMbuffer[kLCM] = r3[kLCM];
03674         }
03675         else if ( ( k != 1 ) || ( r3[0] != 1 ) ) {
03676             if ( GcdLong(BHEAD LCMbuffer,kLCM,(UWORD *)r3,k,LCMb,&kkLCM) ) {
03677                 goto MakeDollarIntegerErr;
03678             }
03679             DivLong((UWORD *)r3,k,LCMb,kkLCM,LCMb,&kkLCM,LCMc,&jLCM);
03680             MulLong(LCMbuffer,kLCM,LCMb,kkLCM,LCMc,&jLCM);
03681             for ( kLCM = 0; kLCM < jLCM; kLCM++ )
03682                 LCMbuffer[kLCM] = LCMc[kLCM];
```

```
03683         }
03684         else {} /* LCM doesn't change */
03685         r1 = r2;
03686     }
03687 /*
03688     Now put the factor together: GCD/LCM
03689 */
03690     r3 = (WORD *)(GCDbuffer);
03691     if ( kGCD == kLCM ) {
03692         for ( jGCD = 0; jGCD < kGCD; jGCD++ )
03693             r3[jGCD+kGCD] = LCMbuffer[jGCD];
03694         k = kGCD;
03695     }
03696     else if ( kGCD > kLCM ) {
03697         for ( jGCD = 0; jGCD < kLCM; jGCD++ )
03698             r3[jGCD+kGCD] = LCMbuffer[jGCD];
03699         for ( jGCD = kLCM; jGCD < kGCD; jGCD++ )
03700             r3[jGCD+kGCD] = 0;
03701         k = kGCD;
03702     }
03703     else {
03704         for ( jGCD = kGCD; jGCD < kLCM; jGCD++ )
03705             r3[jGCD] = 0;
03706         for ( jGCD = 0; jGCD < kLCM; jGCD++ )
03707             r3[jGCD+kLCM] = LCMbuffer[jGCD];
03708         k = kLCM;
03709     }
03710     j = 2*k+1;
03711 /*
03712     Now we have to write this to factor
03713 */
03714     factor = r1 = (WORD *)Malloc1((j+2)*sizeof(WORD),"MakeDollarInteger");
03715     *r1++ = j+1; r2 = r3;
03716     for ( i = 0; i < k; i++ ) { *r1++ = *r2++; *r1++ = *r2++; }
03717     *r1++ = j;
03718     *r1 = 0;
03719 /*
03720     Next we have to take the factor out from the argument.
03721     This cannot be done in location, because the denominator stuff can make
03722     coefficients longer.
03723
03724     We do this via a sort because the things may be jumbled any way and we
03725     do not know in advance how much space we need.
03726 */
03727     NewSort(BHEAD0);
03728     r = bufin;
03729     oldworkpointer = AT.WorkPointer;
03730     while ( *r ) {
03731         rnext = r + *r;
03732         j = ABS(rnext[-1]);
03733         r3 = rnext - j;
03734         r2 = oldworkpointer;
03735         while ( r < r3 ) *r2++ = *r++;
03736         j = (j-1)/2;    /* reduced length. Remember, k is the other red length */
03737         if ( DivRat(BHEAD (UWORD *)r3,j,GCDbuffer,k,(UWORD *)r2,&i) ) {
03738             goto MakeDollarIntegerErr;
03739         }
03740         i = 2*i+1;
03741         r2 = r2 + i;
03742         if ( rnext[-1] < 0 ) r2[-1] = -i;
03743         else                 r2[-1] =  i;
03744         *oldworkpointer = r2-oldworkpointer;
03745         AT.WorkPointer = r2;
03746         if ( Generator(BHEAD oldworkpointer,C->numlhs) ) {
03747             goto MakeDollarIntegerErr;
03748         }
03749         r = rnext;
03750     }
03751     AT.WorkPointer = oldworkpointer;
03752     AN.tryterm = 0; /* for now */
03753     EndSort(BHEAD (WORD *)bufout,2);
03754 /*
03755     Cleanup
03756 */
03757     NumberFree(LCMc,"MakeDollarInteger");
03758     NumberFree(LCMb,"MakeDollarInteger");
03759     NumberFree(LCMbuffer,"MakeDollarInteger");
03760     NumberFree(GCDbuffer2,"MakeDollarInteger");
03761     NumberFree(GCDbuffer,"MakeDollarInteger");
03762     return(factor);
03763
03764 MakeDollarIntegerErr:
03765     NumberFree(LCMc,"MakeDollarInteger");
03766     NumberFree(LCMb,"MakeDollarInteger");
03767     NumberFree(LCMbuffer,"MakeDollarInteger");
03768     NumberFree(GCDbuffer2,"MakeDollarInteger");
03769     NumberFree(GCDbuffer,"MakeDollarInteger");
```

```
03770     MesCall("MakeDollarInteger");
03771     Terminate(-1);
03772     return(0);
03773 }
03774
03775 /*
03776     #] MakeDollarInteger :
03777     #[ MakeDollarMod :
03778 */
03786 WORD *MakeDollarMod(PHEAD WORD *buffer, WORD **bufout)
03787 {
03788     GETBIDENTITY
03789     WORD *r, *r1, x, xx, ix, ip;
03790     WORD *factor, *oldworkpointer;
03791     int i;
03792     CBUF *C = cbuf+AC.cbufnum;
03793     r = buffer;
03794     x = r[*r-3];
03795     if ( r[*r-1] < 0 ) x += AN.cmod[0];
03796     if ( GetModInverses(x,(WORD)(AN.cmod[0]),&ix,&ip) ) {
03797         Terminate(-1);
03798     }
03799     factor = (WORD *)Malloc1(5*sizeof(WORD),"MakeDollarMod");
03800     factor[0] = 4; factor[1] = x; factor[2] = 1; factor[3] = 3; factor[4] = 0;
03801 /*
03802     Now we have to multiply all coefficients by ix.
03803     This does not make things longer, but we should keep to the conventions
03804     of MakeDollarInteger.
03805 */
03806     NewSort(BHEAD0);
03807     r = buffer;
03808     oldworkpointer = AT.WorkPointer;
03809     while ( *r ) {
03810         r1 = oldworkpointer; i = *r;
03811         NCOPY(r1,r,i);
03812         xx = r1[-3]; if ( r1[-1] < 0 ) xx += AN.cmod[0];
03813         r1[-1] = (WORD)((((LONG)xx)*ix) % AN.cmod[0]);
03814         *r1 = 0; AT.WorkPointer = r1;
03815         if ( Generator(BHEAD oldworkpointer,C->numlhs) ) {
03816             Terminate(-1);
03817         }
03818     }
03819     AT.WorkPointer = oldworkpointer;
03820     AN.tryterm = 0; /* for now */
03821     EndSort(BHEAD (WORD *)bufout,2);
03822     return(factor);
03823 }
03824 /*
03825     #] MakeDollarMod :
03826     #[ GetDolNum :
03827
03828     Evaluates a chain of DOLLAREXPR2 into a number
03829 */
03830
03831 int GetDolNum(PHEAD WORD *t, WORD *tstop)
03832 {
03833     DOLLARS d;
03834     WORD num, *w;
03835     if ( t+3 < tstop && t[3] == DOLLAREXPR2 ) {
03836         d = Dollars + t[2];
03837 #ifdef WITHPTHREADS
03838         {
03839             int nummodopt, dtype;
03840             dtype = -1;
03841             if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
03842                 for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
03843                     if ( t[2] == ModOptdollars[nummodopt].number ) break;
03844                 }
03845                 if ( nummodopt < NumModOptdollars ) {
03846                     dtype = ModOptdollars[nummodopt].type;
03847                     if ( DollarLocalCopy(dtype) ) {
03848                         d = ModOptdollars[nummodopt].dstruct+AT.identity;
03849                     }
03850                     else {
03851                         MLOCK(ErrorMessageLock);
03852                         MesPrint("&Illegal attempt to use $-variable %s in module %l",
03853                             DOLLARNAME(Dollars,t[2]),AC.CModule);
03854                         MUNLOCK(ErrorMessageLock);
03855                         Terminate(-1);
03856                     }
03857                 }
03858             }
03859         }
03860 #endif
03861         if ( d->factors == 0 ) {
03862             MLOCK(ErrorMessageLock);
03863             MesPrint("Attempt to use a factor of an unfactored $-variable");
```

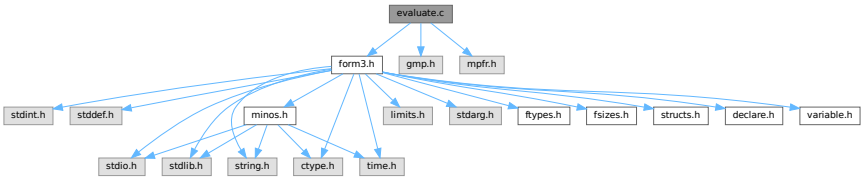
```
03864             MUNLOCK(ErrorMessageLock);
03865             Terminate(-1);
03866         }
03867         num = GetDolNum(BHEAD t+t[1],tstop);
03868         if ( num == 0 ) return(d->nfactors);
03869         if ( num > d->nfactors ) {
03870             MLOCK(ErrorMessageLock);
03871             MesPrint("Attempt to use an nonexisting factor %d of a $-variable",num);
03872             MUNLOCK(ErrorMessageLock);
03873             Terminate(-1);
03874         }
03875         w = d->factors[num-1].where;
03876         if ( w == 0 ) return(d->factors[num-1].value);
03877         if ( w[0] == 4 && w[4] == 0 && w[3] == 3 && w[2] == 1 && w[1] > 0
03878         && w[1] < MAXPOSITIVE ) return(w[1]);
03879         else {
03880             MLOCK(ErrorMessageLock);
03881             MesPrint("Illegal type of factor number of a $-variable");
03882             MUNLOCK(ErrorMessageLock);
03883             Terminate(-1);
03884         }
03885     }
03886     else if ( t[2] < 0 ) {
03887         return(-t[2]-1);
03888     }
03889     else {
03890         d = Dollars + t[2];
03891 #ifdef WITHPTHREADS
03892         {
03893             int nummodopt, dtype;
03894             dtype = -1;
03895             if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
03896                 for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
03897                     if ( t[2] == ModOptdollars[nummodopt].number ) break;
03898                 }
03899                 if ( nummodopt < NumModOptdollars ) {
03900                     dtype = ModOptdollars[nummodopt].type;
03901                     if ( DollarLocalCopy(dtype) ) {
03902                         d = ModOptdollars[nummodopt].dstruct+AT.identity;
03903                     }
03904                     else {
03905                         MLOCK(ErrorMessageLock);
03906                         MesPrint("&Illegal attempt to use $-variable %s in module %l",
03907                             DOLLARNAME(Dollars,t[2]),AC.CModule);
03908                         MUNLOCK(ErrorMessageLock);
03909                         Terminate(-1);
03910                     }
03911                 }
03912             }
03913         }
03914 #endif
03915         if ( d->type == DOLZERO ) return(0);
03916         if ( d->type == DOLTERMS || d->type == DOLNUMBER ) {
03917             if ( d->where[0] == 4 && d->where[4] == 0 && d->where[3] == 3
03918                 && d->where[2] == 1 && d->where[1] > 0
03919                 && d->where[1] < MAXPOSITIVE ) return(d->where[1]);
03920             MLOCK(ErrorMessageLock);
03921             MesPrint("Attempt to use an nonexisting factor of a $-variable");
03922             MUNLOCK(ErrorMessageLock);
03923             Terminate(-1);
03924         }
03925         MLOCK(ErrorMessageLock);
03926         MesPrint("Illegal type of factor number of a $-variable");
03927         MUNLOCK(ErrorMessageLock);
03928         Terminate(-1);
03929     }
03930     return(0);
03931 }
03932
03933 /*
03934     #] GetDolNum :
03935     #[ AddPotModdollar :
03936 */
03937
03944 void AddPotModdollar(WORD numdollar)
03945 {
03946     int i, n = NumPotModdollars;
03947     for ( i = 0; i < n; i++ ) {
03948         if ( numdollar == PotModdollars[i] ) break;
03949     }
03950     if ( i >= n ) {
03951         *(WORD *)FromList(&AC.PotModDolList) = numdollar;
03952     }
03953 }
03954
03955 /*
03956     #] AddPotModdollar :
```

```
03957 */
```

4.29 evaluate.c File Reference

```
#include "form3.h"
#include <gmp.h>
#include <mpfr.h>
```

Include dependency graph for evaluate.c:



Macros

- #define EXTRAPRECISION 4
- #define RND MPFR_RNDN
- #define auxr1 (((mpfr_t *)(AT.auxr_))[0])
- #define auxr2 (((mpfr_t *)(AT.auxr_))[1])
- #define auxr3 (((mpfr_t *)(AT.auxr_))[2])
- #define auxr4 (((mpfr_t *)(AT.auxr_))[3])
- #define auxr5 (((mpfr_t *)(AT.auxr_))[4])

Functions

- int PackFloat (WORD *, mpf_t)
- int UnpackFloat (mpf_t, WORD *)
- void RatToFloat (mpf_t, UWORD *, int)
- void FormtoZ (mpz_t, UWORD *, WORD)
- void IntegerToFloatr (mpfr_t result, UWORD *formlong, int longsize)
- void RatToFloatr (mpfr_t result, UWORD *formrat, int ratsize)
- void SetfFloatPrecision (LONG prec)
- void ClearfFloat (void)
- int GetFloatArgument (PHEAD mpfr_t f_out, WORD *fun, int par)
- int GetPiArgument (PHEAD WORD *arg)
- int EvaluateFun (PHEAD WORD *term, WORD level, WORD *pars)

Variables

- mpf_t ln2

4.29.1 Detailed Description

Evaluation of functions for the floating-point system, by interfacing with MPFR.

Definition in file evaluate.c.

4.29.2 Macro Definition Documentation

4.29.2.1 EXTRAPRECISION

```
#define EXTRAPRECISION 4
```

Definition at line 39 of file evaluate.c.

4.29.2.2 RND

```
#define RND MPFR_RNDN
```

Definition at line 41 of file evaluate.c.

4.29.2.3 auxr1

```
#define auxr1 (((mpfr_t *)(AT.auxr_))[0])
```

Definition at line 43 of file evaluate.c.

4.29.2.4 auxr2

```
#define auxr2 (((mpfr_t *)(AT.auxr_))[1])
```

Definition at line 44 of file evaluate.c.

4.29.2.5 auxr3

```
#define auxr3 (((mpfr_t *)(AT.auxr_))[2])
```

Definition at line 45 of file evaluate.c.

4.29.2.6 auxr4

```
#define auxr4 (((mpfr_t *)(AT.auxr_))[3])
```

Definition at line 46 of file evaluate.c.

4.29.2.7 auxr5

```
#define auxr5 (((mpfr_t *)(AT.auxr_))[4])
```

Definition at line 47 of file evaluate.c.

4.29.3 Function Documentation

4.29.3.1 PackFloat()

```
int PackFloat (
            WORD * fun,
            mpf_t infloat )
```

Definition at line 271 of file float.c.

4.29.3.2 UnpackFloat()

```
int UnpackFloat (
            mpf_t outfloat,
            WORD * fun )
```

Definition at line 366 of file float.c.

4.29.3.3 RatToFloat()

```
void RatToFloat (
            mpf_t result,
            UWORD * formrat,
            int ratsize )
```

Definition at line 624 of file float.c.

4.29.3.4 FormtoZ()

```
void FormtoZ (
            mpz_t z,
            UWORD * a,
            WORD na )
```

Definition at line 515 of file float.c.

4.29.3.5 IntegerToFloatr()

```
void IntegerToFloatr (
            mpfr_t result,
            UWORD * formlong,
            int longsize )
```

Definition at line 65 of file evaluate.c.

4.29.3.6 RatToFloatr()

```
void RatToFloatr (
            mpfr_t result,
            UWORD * formrat,
            int ratsize )
```

Definition at line 81 of file evaluate.c.

4.29.3.7 SetfFloatPrecision()

```
void SetfFloatPrecision (
            LONG prec )
```

Definition at line 108 of file evaluate.c.

4.29.3.8 ClearfFloat()

```
void ClearfFloat (
            void  )
```

Definition at line 141 of file evaluate.c.

4.29.3.9 GetFloatArgument()

```
int GetFloatArgument (
            PHEAD mpfr_t f_out,
            WORD * fun,
            int par )
```

Definition at line 180 of file evaluate.c.

4.29.3.10 GetPiArgument()

```
int GetPiArgument (
            PHEAD WORD * arg )
```

Definition at line 277 of file evaluate.c.

4.29.3.11 EvaluateFun()

```
int EvaluateFun (
            PHEAD WORD * term,
            WORD level,
            WORD * pars )
```

Definition at line 370 of file evaluate.c.

4.29.4 Variable Documentation

4.29.4.1 ln2

```
mpf_t ln2
```

Definition at line 49 of file evaluate.c.

4.30 evaluate.c

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
00026  *
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
00031 /*
00032     #[ includes :
00033 */
00034
00035 #include "form3.h"
00036 #include <gmp.h>
00037 #include <mpfr.h>
00038
00039 #define EXTRAPRECISION 4
00040
00041 #define RND MPFR_RNDN
00042
00043 #define auxr1 (((mpfr_t *)(AT.auxr_))[0])
00044 #define auxr2 (((mpfr_t *)(AT.auxr_))[1])
00045 #define auxr3 (((mpfr_t *)(AT.auxr_))[2])
00046 #define auxr4 (((mpfr_t *)(AT.auxr_))[3])
00047 #define auxr5 (((mpfr_t *)(AT.auxr_))[4])
00048
00049 mpf_t ln2;
00050
00051 int PackFloat(WORD *,mpf_t);
00052 int UnpackFloat(mpf_t,WORD *);
00053 void RatToFloat(mpf_t, UWORD *, int);
00054 void FormtoZ(mpz_t,UWORD *,WORD);
00055
00056 /*
00057     #] includes :
00058     #[ mpfr_ :
00059         #[ IntegerToFloatr :
00060
00061         Converts a Form long integer to a mpfr_ float of default size.
00062         We assume that sizeof(unsigned long int) = 2*sizeof(UWORD).
00063 */
00064
00065 void IntegerToFloatr(mpfr_t result, UWORD *formlong, int longsize)
00066 {
00067     mpz_t z;
00068     mpz_init(z);
00069     FormtoZ(z,formlong,longsize);
```

```
00070     mpfr_set_z(result,z,RND);
00071     mpz_clear(z);
00072 }
00073
00074 /*
00075         #] IntegerToFloatr :
00076         #[ RatToFloatr :
00077
00078         Converts a Form rational to a gmp float of default size.
00079 */
00080
00081 void RatToFloatr(mpfr_t result, UWORD *formrat, int ratsize)
00082 {
00083     GETIDENTITY
00084     int nnum, nden;
00085     UWORD *num, *den;
00086     int sgn = 0;
00087     if ( ratsize < 0 ) { ratsize = -ratsize; sgn = 1; }
00088     nnum = nden = (ratsize-1)/2;
00089     num = formrat; den = formrat+nnum;
00090     while ( nnum > 1 && num[nnum-1] == 0 ) { nnum--; }
00091     while ( nden > 1 && den[nden-1] == 0 ) { nden--; }
00092     IntegerToFloatr(auxr4,num,nnum);
00093     IntegerToFloatr(auxr5,den,nden);
00094     mpfr_div(result,auxr4,auxr5,RND);
00095     if ( sgn > 0 ) mpfr_neg(result,result,RND);
00096 }
00097
00098 /*
00099         #] RatToFloatr :
00100         #[ SetfFloatPrecision :
00101
00102     The AC.DefaultPrecision is in bits.
00103     prec is in limbs.
00104     fprec is NOT in bits for mpfr_ which is slightly less than in mpf_
00105     We also set the auxr1,auxr2,auxr3,auxr4,auxr5 variables.
00106 */
00107
00108 void SetfFloatPrecision(LONG prec)
00109 {
00110 /*
00111     mpfr_prec_t fprec = prec * mp_bits_per_limb - EXTRAPRECISION;
00112 */
00113     mpfr_prec_t fprec = prec + 1;
00114     mpfr_set_default_prec(fprec);
00115 #ifdef WITHPTHREADS
00116     int totnum = AM.totalnumberofthreads, id;
00117     mpfr_t *a;
00118   #ifdef WITHSORTBOTS
00119     totnum = MaX(2*AM.totalnumberofthreads-3,AM.totalnumberofthreads);
00120   #endif
00121     for ( id = 0; id < totnum; id++ ) {
00122         AB[id]->T.auxr_ = (void *)Malloc1(sizeof(mpfr_t)*5,"AB[id]->T.auxr_");
00123         a = (mpfr_t *)AB[id]->T.auxr_;
00124 /*
00125         We work here with a[0] etc because the aux1 etc contain B which
00126         in the current routine would be AB[0] only
00127 */
00128         mpfr_inits2(fprec,a[0],a[1],a[2],a[3],a[4],(mpfr_ptr)0);
00129     }
00130 #else
00131     AT.auxr_ = (void *)Malloc1(sizeof(mpfr_t)*5,"AT.auxr_");
00132     mpfr_inits2(fprec,auxr1,auxr2,auxr3,auxr4,auxr5,(mpfr_ptr)0);
00133 #endif
00134 }
00135
00136 /*
00137         #] SetfFloatPrecision :
00138         #[ ClearfFloat :
00139 */
00140
00141 void ClearfFloat(void)
00142 {
00143 #ifdef WITHPTHREADS
00144     int totnum = AM.totalnumberofthreads, id;
00145     mpfr_t *a;
00146   #ifdef WITHSORTBOTS
00147     totnum = MaX(2*AM.totalnumberofthreads-3,AM.totalnumberofthreads);
00148   #endif
00149     if ( AB[0]->T.auxr_ ) {
00150         for ( id = 0; id < totnum; id++ ) {
00151             a = (mpfr_t *)AB[id]->T.auxr_;
00152             mpfr_clears(a[0],a[1],a[2],a[3],a[4],(mpfr_ptr)0);
00153             M_free(AB[id]->T.auxr_,"AB[id]->T.auxr_");
00154             AB[id]->T.auxr_ = 0;
00155         }
00156     }
```

```
00157 #else
00158     if ( AT.auxr_ ) {
00159         mpfr_clears(auxr1,auxr2,auxr3,auxr4,auxr5,(mpfr_ptr)0);
00160         M_free(AT.auxr_,"AT.auxr_");
00161         AT.auxr_ = 0;
00162     }
00163 #endif
00164 }
00165
00166 /*
00167         #] ClearfFloat :
00168         #[ GetFloatArgument :
00169
00170         Convert an argument to an mpfr float if possible.
00171         Return value: 0: was converted successfully.
00172                       -1: could not convert.
00173         Note: arguments with more than one term should be evaluated first
00174         inside an Argument environment. If there is still more than one
00175         term remaining we get the code: "could not convert".
00176         par tells which argument we need. If it is negative it should also
00177         be the last argument.
00178 */
00179
00180 int GetFloatArgument(PHEAD mpfr_t f_out,WORD *fun,int par)
00181 {
00182     WORD *term, *tn, *t, *tstop, first, ncoef, *arg, *argp, abspar;
00183     arg = fun+FUNHEAD;
00184     abspar = ABS(par);
00185     while ( arg < fun+fun[1] && abspar > 1 ) { abspar--; NEXTARG(arg) }
00186     if ( arg >= fun+fun[1] || abspar!= 1 ) return(-1);
00187     if ( par < 0 ) {
00188         argp = arg; NEXTARG(argp); if ( argp != fun+fun[1] ) return(-1);
00189     }
00190     if ( *arg < 0 ) {
00191         if ( *arg == -SNUMBER ) {
00192             mpfr_set_si(f_out,(LONG)(arg[1]),RND);
00193         }
00194         else if ( *arg == -SYMBOL && ( arg[1] == PISYMBOL ) ) {
00195             mpfr_const_pi(f_out,RND);
00196         }
00197         else if ( *arg == -INDEX && arg[1] >= 0 && arg[1] < AM.OffsetIndex ) {
00198             mpfr_set_ui(f_out,(ULONG)(arg[1]),RND);
00199         }
00200         else { return(-1); }
00201         return(0);
00202     }
00203     else if ( arg[0] != arg[ARGHEAD]+ARGHEAD ) {    /* more than one term */
00204         return(-1);
00205     }
00206     term = arg+ARGHEAD;
00207     tn = term+*term;
00208     tstop = tn-ABS(tn[-1]);
00209     t = term+1;
00210     first = 1;
00211     while ( t <= tstop ) {
00212         if ( t == tstop ) { /* Fraction */
00213             if ( first ) RatToFloatr(f_out,(UWORD *)tstop,tn[-1]);
00214             else {
00215                 RatToFloatr(auxr4,(UWORD *)tstop,tn[-1]);
00216                 mpfr_mul(f_out,f_out,auxr4,RND);
00217             }
00218             return(0);
00219         }
00220         else if ( *t == FLOATFUN ) {
00221             if ( t+t[1] != tstop ) return(-1);
00222             if ( UnpackFloat(aux5,t) < 0 ) return(-1);
00223             if ( first ) {
00224                 mpfr_set_f(f_out,aux5,RND);
00225                 first = 0;
00226             }
00227             else {
00228                 mpfr_set_f(auxr4,aux5,RND);
00229                 mpfr_mul(f_out,f_out,auxr4,RND);
00230             }
00231             first = 0;
00232             if ( tn[-1] < 0 ) { /* change sign */
00233                 ncoef = -tn[-1];
00234                 mpfr_neg(f_out,f_out,RND);
00235             }
00236             else ncoef = tn[-1];
00237             if ( ncoef == 3 && tn[-2] == 1 && tn[-3] == 1 ) return(0);
00238 /*
00239             If the argument was properly normalized we are not supposed to come here.
00240 */
00241 /* INTERNAL_ERROR_EXCL_START */
00242             MLOCK(ErrorMessageLock);
00243             MesPrint("!>Unnormalized argument in GetFloatArgument: %a",*term,term);
```

```
00244             MUNLOCK(ErrorMessageLock);
00245             Terminate(-1);
00246 /* INTERNAL_ERROR_EXCL_STOP */
00247         }
00248         else if ( t[0] == SYMBOL && t[1] == 4 && t[2] == PISYMBOL && t[3] == 1 ) {
00249             if ( first ) {
00250                 mpfr_const_pi(f_out,RND);
00251                 first = 0;
00252             }
00253             else {
00254                 mpfr_const_pi(auxr5,RND);
00255                 mpfr_mul(f_out,f_out,auxr5,RND);
00256             }
00257         }
00258         else {  /* This we do not / cannot do */
00259             return(-1);
00260         }
00261         t += t[1];
00262     }
00263     return(0);
00264 }
00265
00266 /*
00267         #] GetFloatArgument :
00268         #[ GetPiArgument :
00269
00270         Tests for sin,cos,tan whether the argument is a simple
00271         multiple of pi or can be reduced to such.
00272         Return value: -1: the answer is no.
00273         0-23: we have 0, 1/12*pi_,...,23/12*pi_
00274         With success most values can be worked out by simple means.
00275 */
00276
00277 int GetPiArgument(PHEAD WORD *arg)
00278 {
00279     UWORD *coef, *co, *tco, twelve, *numer, *denom, *c, *d;
00280     WORD i, ii, i2, iflip, nc, nd, *t, *tn, *tstop;
00281     int rem;
00282 /*
00283     One: determine whether there is a rational coefficient and a pi_:
00284 */
00285     if ( *arg == -SNUMBER && arg[1] == 0 ) return(0);
00286     if ( *arg == -SYMBOL && arg[1] == PISYMBOL ) return(12);
00287     if ( *arg < 0 ) return(-1);
00288     if ( arg[ARGHEAD] != *arg-ARGHEAD ) return(-1);
00289     t = arg+ARGHEAD+1;
00290     tn = arg+*arg; tstop = tn - ABS(tn[-1]);
00291     if ( *t != SYMBOL || t[1] != 4 || t[2] != PISYMBOL || t[3] != 1
00292         || t+t[1] != tstop ) return(-1);
00293 /*
00294     The denominator must be a divisor of 12
00295     1: copy the coefficient
00296 */
00297     co = coef = NumberMalloc("GetPiArgument");
00298     tco = (UWORD *)tstop;
00299     i = tn[-1]; if ( i < 0 ) { i = -i; iflip = 1; } else iflip = 0;
00300     ii = (i-1)/2;
00301     twelve = 12;
00302     NCOPY(co,tco,i);
00303     co = coef;
00304     Mully(BHEAD co,&ii,&twelve,1);
00305 /*
00306     Now the denominator should be 1
00307 */
00308     i = ii; i2 = i-1;
00309     numer = co; denom = numer + i;
00310     if ( i > 1 ) {
00311         while ( i2 > 0 ) {
00312             if ( denom[i2--] != 0 ) {
00313                 NumberFree(co,"GetPiArgument");
00314                 return(-1);
00315             }
00316         }
00317     }
00318     if ( *denom != 1 ) {
00319         NumberFree(co,"GetPiArgument");
00320         return(-1);
00321     }
00322 /*
00323     Now we need the numerator modulus 24.
00324 */
00325     if ( i == 1 ) {
00326         rem = *numer % 24;
00327     }
00328     else {
00329         c = NumberMalloc("GetPiArgument");
00330         d = NumberMalloc("GetPiArgument");
```

```
00331         twelve *= 2;
00332         DivLong(numer,i,&twelve,1,c,&nc,d,&nd);
00333         rem = *d % 24;
00334         NumberFree(d,"GetPiArgument");
00335         NumberFree(c,"GetPiArgument");
00336     }
00337     NumberFree(co,"GetPiArgument");
00338     if ( iflip && rem != 0 ) rem = 24-rem;
00339     return(rem);
00340 }
00341
00342 /*
00343         #] GetPiArgument :
00344         #[ EvaluateFun :
00345
00346         What we need to do is:
00347         1: look for a function to be treated.
00348         2: make sure its argument is treatable.
00349         3: call the proper mpfr_ function.
00350         4: accumulate the result.
00351         For some functions we have to insert 'smart' shortcuts as is
00352         the case with sin_(pi_) or sin_(pi_/6) of sqrt(4/9) etc.
00353         Otherwise we may have to insert a value for pi_ first.
00354         There are several types of arguments:
00355         a: (short) integers.
00356         b: rationals.
00357         c: floats.
00358         We accumulate the result(s) in auxr2. The argument comes in aux5
00359         and a result in auxr3 which then gets multiplied into auxr2.
00360         In the end we have to combine auxr2 with whatever coefficient
00361         existed already.
00362
00363         When the float system is started we need for aux only 3 variables
00364         per thread. auxr1-auxr3. This should be done separately.
00365         The main problem is the conversion of mpfr_t to float_ and/or mpf_t
00366         and back.
00367 */
00368
00369
00370 int EvaluateFun(PHEAD WORD *term, WORD level, WORD *pars)
00371 {
00372     WORD *t, *tstop, *tt, *tnext, *newterm, i;
00373     WORD *oldworkpointer = AT.WorkPointer, nsize, nsgn, nsgn2;
00374     int retval = 0, first = 1, pimul;
00375
00376     tstop = term + *term; tstop -= ABS(tstop[-1]);
00377     if ( AT.WorkPointer < term+*term ) AT.WorkPointer = term + *term;
00378     t = term+1;
00379     mpfr_set_ui(auxr2,1L,RND);
00380     while ( t < tstop ) {
00381         if ( pars[2] == *t ) {  /* have to do this one if possible */
00382 TestArgument:
00383 /*
00384             There must be a single argument, except for the AGM or atan2 functions
00385 */
00386             tnext = t+t[1]; tt = t+FUNHEAD; NEXTARG(tt);
00387             if( *t == SYMBOL) {
00388                 for ( WORD* ti = t+2; ti < t+t[1]; ti+=2 ) {
00389                     if( ( *ti == PISYMBOL || *ti == EESYMBOL  || *ti == EMSYMBOL )
00390                      && ( pars[2] == ALLFUNCTIONS || pars[3] == *ti ) ) {
00391                         if ( *ti == PISYMBOL )
00392                             mpfr_const_pi(auxr3,RND);
00393                         else if ( *ti == EESYMBOL ) {
00394                             mpfr_set_ui(auxr3,1,RND);
00395                             mpfr_exp(auxr3,auxr3,RND);
00396                         }
00397                         else if ( *ti == EMSYMBOL )
00398                             mpfr_const_euler(auxr3,RND);
00399                         if ( ti[1] != 1 )
00400                             mpfr_pow_si(auxr3,auxr3,ti[1],RND);
00401                         mpfr_mul(auxr2,auxr2,auxr3,RND);
00402                         ti[1] = 0;
00403                         first = 0;
00404                     }
00405                 }
00406                 goto nextfun;
00407             }
00408             if ( tt != tnext && *t != AGMFUNCTION && *t != ATAN2FUNCTION) goto nextfun;
00409             if ( *t == SINFUNCTION ) {
00410                 pimul = GetPiArgument(BHEAD t+FUNHEAD);
00411                 if ( pimul >= 0 && pimul < 24 ) {
00412                     if ( pimul == 0 || pimul == 12 ) goto getout;
00413                     if ( pimul > 12 ) { pimul = 24-pimul; nsgn = -1; }
00414                     else nsgn = 1;
00415                     if ( pimul > 6 ) pimul = 12-pimul;
00416                     switch ( pimul ) {
00417                         case 1:
```

```
00418 label1:
00419                               mpfr_sqrt_ui(auxr3,3L,RND);
00420                               mpfr_sub_ui(auxr3,auxr3,1L,RND);
00421                               mpfr_sqrt_ui(auxr1,2L,RND);
00422                               mpfr_div_ui(auxr1,auxr1,4L,RND);
00423                               mpfr_mul(auxr3,auxr3,auxr1,RND);
00424                               if ( nsgn < 0 ) mpfr_neg(auxr3,auxr3,RND);
00425                               mpfr_mul(auxr2,auxr2,auxr3,RND);
00426                               break;
00427                           case 2:
00428 label2:
00429                               mpfr_div_ui(auxr2,auxr2,2L,RND);
00430                               if ( nsgn < 0 ) mpfr_neg(auxr2,auxr2,RND);
00431                               break;
00432                           case 3:
00433 label3:
00434                               mpfr_sqrt_ui(auxr3,2L,RND);
00435                               mpfr_div_ui(auxr3,auxr3,2L,RND);
00436                               if ( nsgn < 0 ) mpfr_neg(auxr3,auxr3,RND);
00437                               mpfr_mul(auxr2,auxr2,auxr3,RND);
00438                               break;
00439                           case 4:
00440 label4:
00441                               mpfr_sqrt_ui(auxr3,3L,RND);
00442                               mpfr_div_ui(auxr3,auxr3,2L,RND);
00443                               if ( nsgn < 0 ) mpfr_neg(auxr3,auxr3,RND);
00444                               mpfr_mul(auxr2,auxr2,auxr3,RND);
00445                               break;
00446                           case 5:
00447 label5:
00448                               mpfr_sqrt_ui(auxr3,3L,RND);
00449                               mpfr_add_ui(auxr3,auxr3,1L,RND);
00450                               mpfr_sqrt_ui(auxr1,2L,RND);
00451                               mpfr_div_ui(auxr1,auxr1,4L,RND);
00452                               mpfr_mul(auxr3,auxr3,auxr1,RND);
00453                               if ( nsgn < 0 ) mpfr_neg(auxr3,auxr3,RND);
00454                               mpfr_mul(auxr2,auxr2,auxr3,RND);
00455                               break;
00456                           case 6:
00457 label6:
00458                               if ( nsgn < 0 ) mpfr_neg(auxr2,auxr2,RND);
00459                               break;
00460                       }
00461                       *t = 0; first = 0;
00462                       goto nextfun;
00463                   }
00464               }
00465               else if ( *t == COSFUNCTION ) {
00466                   pimul = GetPiArgument(BHEAD t+FUNHEAD);
00467                   if ( pimul >= 0 && pimul < 24 ) {
00468                       if ( pimul > 12 ) pimul = 24-pimul;
00469                       if ( pimul > 6 ) { pimul = 12-pimul; nsgn = -1; }
00470                       else nsgn = 1;
00471                       if ( pimul == 6 ) goto getout;
00472                       switch ( pimul ) {
00473                           case 0: goto label6;
00474                           case 1: goto label5;
00475                           case 2: goto label4;
00476                           case 3: goto label3;
00477                           case 4: goto label2;
00478                           case 5: goto label1;
00479                       }
00480                   }
00481               }
00482               else if ( *t == TANFUNCTION ) {
00483                   pimul = GetPiArgument(BHEAD t+FUNHEAD);
00484                   if ( pimul >= 0 && pimul < 24 ) {
00485                       if ( pimul == 6 || pimul == 18 ) goto nextfun;
00486                       if ( pimul == 0 || pimul == 12 ) goto getout;
00487                       if ( pimul > 12 ) { pimul = 24-pimul; nsgn = -1; }
00488                       else nsgn = 1;
00489                       if ( pimul > 6 ) { pimul = 12-pimul; nsgn = -nsgn; }
00490                       switch ( pimul ) {
00491                           case 1:
00492                               mpfr_sqrt_ui(auxr3,3L,RND);
00493                               mpfr_sub_ui(auxr3,auxr3,2L,RND);
00494                               if ( nsgn > 0 ) mpfr_neg(auxr3,auxr3,RND);
00495                               mpfr_mul(auxr2,auxr2,auxr3,RND);
00496                               break;
00497                           case 2:
00498                               mpfr_sqrt_ui(auxr3,3L,RND);
00499                               mpfr_div_ui(auxr3,auxr3,3L,RND);
00500                               if ( nsgn < 0 ) mpfr_neg(auxr3,auxr3,RND);
00501                               mpfr_mul(auxr2,auxr2,auxr3,RND);
00502                               break;
00503                           case 3:
00504                               break;
```

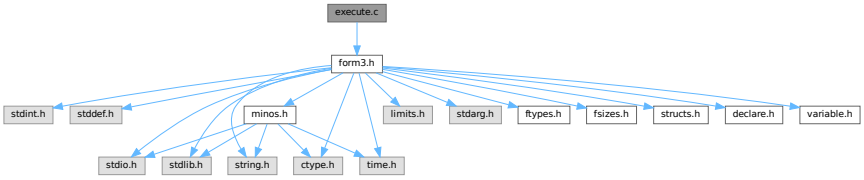
```
00505                         case 4:
00506                             mpfr_sqrt_ui(auxr3,3L,RND);
00507                             if ( nsgn < 0 ) mpfr_neg(auxr3,auxr3,RND);
00508                             mpfr_mul(auxr2,auxr2,auxr3,RND);
00509                             break;
00510                         case 5:
00511                             mpfr_sqrt_ui(auxr3,3L,RND);
00512                             mpfr_add_ui(auxr3,auxr3,2L,RND);
00513                             if ( nsgn < 0 ) mpfr_neg(auxr3,auxr3,RND);
00514                             mpfr_mul(auxr2,auxr2,auxr3,RND);
00515                             break;
00516                     }
00517                     *t = 0; first = 0;
00518                     goto nextfun;
00519                 }
00520             }
00521
00522             if ( *t == AGMFUNCTION || *t == ATAN2FUNCTION ) {
00523                 if ( GetFloatArgument(BHEAD auxr1,t,1) < 0 ) goto nextfun;
00524                 if ( GetFloatArgument(BHEAD auxr3,t,-2) < 0 ) goto nextfun;
00525             }
00526             else if ( GetFloatArgument(BHEAD auxr1,t,-1) < 0 ) goto nextfun;
00527             nsgn = mpfr_sgn(auxr1);
00528
00529             switch ( *t ) {
00530                 case SQRTFUNCTION:
00531                     if ( nsgn < 0 ) goto nextfun;
00532                     if ( nsgn == 0 ) goto getout;
00533                     else mpfr_sqrt(auxr3,auxr1,RND);
00534                     mpfr_mul(auxr2,auxr2,auxr3,RND);
00535                 break;
00536                 case LNFUNCTION:
00537                     if ( nsgn <= 0 ) goto nextfun;
00538                     if ( mpfr_cmp_ui(auxr1,1L) == 0 ) goto getout;
00539                     else mpfr_log(auxr3,auxr1,RND);
00540                     mpfr_mul(auxr2,auxr2,auxr3,RND);
00541                 break;
00542                 case EXPFUNCTION:
00543                     mpfr_exp(auxr3,auxr1,RND);
00544                     mpfr_mul(auxr2,auxr2,auxr3,RND);
00545                 break;
00546                 case LI2FUNCTION: /* should be between -1 and +1 */
00547                     if ( nsgn == 0 ) goto getout;
00548                     mpfr_abs(auxr3,auxr1,RND);
00549                     if ( mpfr_cmp_ui(auxr3,1L) > 0 ) goto nextfun;
00550                     mpfr_li2(auxr3,auxr1,RND);
00551                     mpfr_mul(auxr2,auxr2,auxr3,RND);
00552                 break;
00553                 case GAMMAFUN:
00554 /*
00555                     We cannot do this when the argument is a non-positive integer
00556 */
00557                     if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] <= 0 ) goto nextfun;
00558                     if ( t[FUNHEAD] == t[1]-FUNHEAD && ABS(t[t[1]-1]) ==
00559                         t[FUNHEAD]-ARGHEAD-1 && t[t[1]-1] < 0 ) {
00560                         nsize = (-t[t[1]-1]-1)/2;
00561                         if ( t[t[1]-1-nsize] == 1 ) {
00562                             for ( i = 1; i < nsize; i++ ) {
00563                                 if ( t[t[1]-1-nsize+i] != 0 ) break;
00564                             }
00565                             if ( i >= nsize ) goto nextfun;
00566                         }
00567                     }
00568                     mpfr_gamma(auxr3,auxr1,RND);
00569                     mpfr_mul(auxr2,auxr2,auxr3,RND);
00570                 break;
00571                 case AGMFUNCTION:
00572                     nsgn = mpfr_sgn(auxr1);
00573                     nsgn2 = mpfr_sgn(auxr3);
00574                     if ( nsgn == 0 || nsgn2 == 0) goto getout;
00575                     if ( nsgn < 0 || nsgn2 < 0 ) goto nextfun;
00576                     mpfr_agm(auxr3,auxr1,auxr3,RND);
00577                     mpfr_mul(auxr2,auxr2,auxr3,RND);
00578                 break;
00579                 case SINHFUNCTION:
00580                     if ( nsgn == 0 ) goto getout;
00581                     mpfr_sinh(auxr3,auxr1,RND);
00582                     mpfr_mul(auxr2,auxr2,auxr3,RND);
00583                 break;
00584                 case COSHFUNCTION:
00585                     mpfr_cosh(auxr3,auxr1,RND);
00586                     mpfr_mul(auxr2,auxr2,auxr3,RND);
00587                 break;
00588                 case TANHFUNCTION:
00589                     if ( nsgn == 0 ) goto getout;
00590                     mpfr_tanh(auxr3,auxr1,RND);
00591                     mpfr_mul(auxr2,auxr2,auxr3,RND);
```

```
00592                 break;
00593                 case ASINHFUNCTION:
00594                     if ( nsgn == 0 ) goto getout;
00595                     mpfr_asinh(auxr3,auxr1,RND);
00596                     mpfr_mul(auxr2,auxr2,auxr3,RND);
00597                 break;
00598                 case ACOSHFUNCTION:
00599                     if ( mpfr_cmp_ui(auxr1,1L) < 0 ) goto nextfun;
00600                     if ( mpfr_cmp_ui(auxr1,1L) == 0 ) goto getout;
00601                     mpfr_acosh(auxr3,auxr1,RND);
00602                     mpfr_mul(auxr2,auxr2,auxr3,RND);
00603                 break;
00604                 case ATANHFUNCTION:
00605                     if ( nsgn == 0 ) goto getout;
00606                     mpfr_abs(auxr3,auxr1,RND);
00607                     if ( mpfr_cmp_ui(auxr3,1L) >= 0 ) goto nextfun;
00608                     mpfr_atanh(auxr3,auxr1,RND);
00609                     mpfr_mul(auxr2,auxr2,auxr3,RND);
00610                 break;
00611                 case ASINFUNCTION:
00612                     if ( nsgn == 0 ) goto getout;
00613                     mpfr_abs(auxr3,auxr1,RND);
00614                     if ( mpfr_cmp_ui(auxr3,1L) > 0 ) goto nextfun;
00615                     mpfr_asin(auxr3,auxr1,RND);
00616                     mpfr_mul(auxr2,auxr2,auxr3,RND);
00617                 break;
00618                 case ACOSFUNCTION:
00619                     if ( mpfr_cmp_ui(auxr1,1L) == 0 ) goto getout;
00620                     mpfr_abs(auxr3,auxr1,RND);
00621                     if ( mpfr_cmp_ui(auxr3,1L) > 0 ) goto nextfun;
00622                     mpfr_acos(auxr3,auxr1,RND);
00623                     mpfr_mul(auxr2,auxr2,auxr3,RND);
00624                 break;
00625                 case ATANFUNCTION:
00626                     if ( nsgn == 0 ) goto getout;
00627                     mpfr_atan(auxr3,auxr1,RND);
00628                     mpfr_mul(auxr2,auxr2,auxr3,RND);
00629                 break;
00630                 case ATAN2FUNCTION:
00631                     nsgn = mpfr_sgn(auxr1);
00632                     nsgn2 = mpfr_sgn(auxr3);
00633                     // We follow the conventions of mpfr here:
00634                     if ( nsgn == 0 && nsgn2 >= 0) goto getout;
00635                     mpfr_atan2(auxr3,auxr1,auxr3,RND);
00636                     mpfr_mul(auxr2,auxr2,auxr3,RND);
00637                 break;
00638                 case SINFUNCTION:
00639                     mpfr_sin(auxr3,auxr1,RND);
00640                     mpfr_mul(auxr2,auxr2,auxr3,RND);
00641                 break;
00642                 case COSFUNCTION:
00643                     mpfr_cos(auxr3,auxr1,RND);
00644                     mpfr_mul(auxr2,auxr2,auxr3,RND);
00645                 break;
00646                 case TANFUNCTION:
00647                     mpfr_tan(auxr3,auxr1,RND);
00648                     mpfr_mul(auxr2,auxr2,auxr3,RND);
00649                 break;
00650                 default:
00651                     goto nextfun;
00652                 break;
00653             }
00654             first = 0;
00655             *t = 0;
00656             goto nextfun;
00657         }
00658         else if ( pars[2] == ALLFUNCTIONS ) {
00659 /*
00660             Now we have to test whether this is one of our functions
00661 */
00662             if ( t[1] == FUNHEAD ) goto nextfun;
00663             switch ( *t ) {
00664                 case SQRTFUNCTION:
00665                 case LNFUNCTION:
00666                 case LI2FUNCTION:
00667                 case LINFUNCTION:
00668                 case EXPFUNCTION:
00669                 case ASINFUNCTION:
00670                 case ACOSFUNCTION:
00671                 case ATANFUNCTION:
00672                 case ATAN2FUNCTION:
00673                 case SINHFUNCTION:
00674                 case COSHFUNCTION:
00675                 case TANHFUNCTION:
00676                 case ASINHFUNCTION:
00677                 case ACOSHFUNCTION:
00678                 case ATANHFUNCTION:
```

```
00679                 case SINFUNCTION:
00680                 case COSFUNCTION:
00681                 case TANFUNCTION:
00682                 case AGMFUNCTION:
00683                 case SYMBOL:
00684                     goto TestArgument;
00685                 case MZV:
00686                 case EULER:
00687                 case MZVHALF:
00688                 default:
00689                     goto nextfun;
00690             }
00691         }
00692         else goto nextfun;
00693 nextfun:
00694         t += t[1];
00695     }
00696     if ( first == 1 ) return(Generator(BHEAD term,level));
00697     mpfr_get_f(aux4,auxr2,RND);
00698 /*
00699     Step 3:
00700     Now the regular coefficient, if it is not 1/1.
00701     We have two cases: size +- 3, or bigger.
00702 */
00703
00704     nsize = term[*term-1];
00705     if ( nsize < 0 ) { nsize = -nsize; nsgn = -1; }
00706     else nsgn = 1;
00707     if ( aux4->_mp_size < 0 ) {
00708         aux4->_mp_size = -aux4->_mp_size;
00709         nsgn = -nsgn;
00710     }
00711
00712     if ( nsize == 3 ) {
00713         if ( tstop[0] != 1 ) {
00714             mpf_mul_ui(aux4,aux4,(ULONG)((UWORD)tstop[0]));
00715         }
00716         if ( tstop[1] != 1 ) {
00717             mpf_div_ui(aux4,aux4,(ULONG)((UWORD)tstop[1]));
00718         }
00719     }
00720     else {
00721         RatToFloat(aux5,(UWORD *)tstop,nsize);
00722         mpf_mul(aux4,aux4,aux5);
00723     }
00724 /*
00725     Now we have to locate possible other float_ functions.
00726     Note possible incompatibilities between the mpf and mpfr formats.
00727 */
00728     t = term+1;
00729     while ( t < tstop ) {
00730         if ( *t == FLOATFUN ) {
00731             UnpackFloat(aux5,t);
00732             mpf_mul(aux4,aux4,aux5);
00733         }
00734         t += t[1];
00735     }
00736 /*
00737     Now we should compose the new term in the WorkSpace.
00738 */
00739     t = term+1;
00740     newterm = AT.WorkPointer;
00741     tt = newterm+1;
00742     while ( t < tstop ) {
00743         if ( *t == 0 || *t == FLOATFUN ) t += t[1];
00744         else {
00745             i = t[1]; NCOPY(tt,t,i);
00746         }
00747     }
00748     PackFloat(tt,aux4);
00749     tt += tt[1];
00750     *tt++ = 1; *tt++ = 1; *tt++ = 3*nsgn;
00751     *newterm = tt-newterm;
00752     AT.WorkPointer = tt;
00753     retval = Generator(BHEAD newterm,level);
00754 getout:
00755     AT.WorkPointer = oldworkpointer;
00756     return(retval);
00757 }
00758
00759 /*
00760         #] EvaluateFun :
00761     #] mpfr_ :
00762 */
```

4.31 execute.c File Reference

```
#include "form3.h"
```
Include dependency graph for execute.c:



Macros

- #define CURRENTBRACKET 1
- #define BRACKETCURRENTEXPR 2
- #define BRACKETOTHEREXPR 3
- #define NOBRACKETACTIVE 4

Functions

- int CleanExpr (WORD par)
- int PopVariables (void)
- void MakeGlobal (void)
- void TestDrop (void)
- void PutInVflags (WORD nexpr)
- int DoExecute (WORD par, WORD skip)
- int PutBracket (PHEAD WORD ∗termin)
- void SpecialCleanup (PHEAD0)
- void SetMods (void)
- void UnSetMods (void)
- void ExchangeExpressions (int num1, int num2)
- int GetFirstBracket (WORD ∗term, int num)
- int GetFirstTerm (WORD ∗term, int num, int pre)
- int GetContent (WORD ∗content, int num)
- int CleanupTerm (WORD ∗term)
- WORD ContentMerge (PHEAD WORD ∗content, WORD ∗term)
- LONG TermsInExpression (WORD num)
- LONG SizeOfExpression (WORD num)
- void UpdatePositions (void)
- LONG CountTerms1 (PHEAD0)
- LONG TermsInBracket (PHEAD WORD ∗term, WORD level)

4.31.1 Detailed Description

The routines that start the execution phase of a module. It also contains the routines for placing the bracket subterm.

Definition in file execute.c.

4.31.2 Macro Definition Documentation

4.31.2.1 CURRENTBRACKET

```
#define CURRENTBRACKET 1
```

Definition at line 2672 of file execute.c.

4.31.2.2 BRACKETCURRENTEXPR

```
#define BRACKETCURRENTEXPR 2
```

Definition at line 2673 of file execute.c.

4.31.2.3 BRACKETOTHEREXPR

```
#define BRACKETOTHEREXPR 3
```

Definition at line 2674 of file execute.c.

4.31.2.4 NOBRACKETACTIVE

```
#define NOBRACKETACTIVE 4
```

Definition at line 2675 of file execute.c.

4.31.3 Function Documentation

4.31.3.1 CleanExpr()

```
int CleanExpr (
            WORD par )
```

Definition at line 47 of file execute.c.

4.31.3.2 PopVariables()

```
int PopVariables (
            void  )
```

Definition at line 211 of file execute.c.

4.31.3.3 MakeGlobal()

```
void MakeGlobal (
            void  )
```

Definition at line 383 of file execute.c.

4.31.3.4 TestDrop()

```
void TestDrop (
            void  )
```

Definition at line 484 of file execute.c.

4.31.3.5 PutInVflags()

```
void PutInVflags (
            WORD nexpr )
```

Definition at line 562 of file execute.c.

4.31.3.6 DoExecute()

```
int DoExecute (
            WORD par,
            WORD skip )
```

Definition at line 616 of file execute.c.

4.31.3.7 PutBracket()

```
int PutBracket (
            PHEAD WORD * termin )
```

Definition at line 1214 of file execute.c.

4.31.3.8 SpecialCleanup()

```
void SpecialCleanup (
            PHEAD0  )
```

Definition at line 1728 of file execute.c.

4.31.3.9 SetMods()

```
void SetMods (
            void  )
```

Definition at line 1742 of file execute.c.

4.31.3.10 UnSetMods()

```
void UnSetMods (
            void  )
```

Definition at line 1760 of file execute.c.

4.31.3.11 ExchangeExpressions()

```
void ExchangeExpressions (
            int num1,
            int num2 )
```

Definition at line 1775 of file execute.c.

4.31.3.12 GetFirstBracket()

```
int GetFirstBracket (
            WORD * term,
            int num )
```

Definition at line 1853 of file execute.c.

4.31.3.13 GetFirstTerm()

```
int GetFirstTerm (
            WORD * term,
            int num,
            int pre )
```

Gets the first term of an expression. Routine should be thread safe.

Parameters

out	*term*	Pointer to the buffer where the term should be written.
in	*num*	The expression number from which to get the term.
in	*pre*	Denotes a pre-processor call (1) or not (0). Used to determine the correct source file for the expression.

Returns

First WORD of term, a negative value denotes an error.

Definition at line 1972 of file execute.c.

References FiLe::handle, VaRrEnUm::lo, and ReNuMbEr::symb.

4.31.3.14 GetContent()

```
int GetContent (
            WORD * content,
            int num )
```

Definition at line 2080 of file execute.c.

4.31.3.15 CleanupTerm()

```
int CleanupTerm (
            WORD * term )
```

Definition at line 2197 of file execute.c.

4.31.3.16 ContentMerge()

```
WORD ContentMerge (
            PHEAD WORD * content,
            WORD * term )
```

Definition at line 2221 of file execute.c.

4.31.3.17 TermsInExpression()

```
LONG TermsInExpression (
            WORD num )
```

Definition at line 2485 of file execute.c.

4.31.3.18 SizeOfExpression()

```
LONG SizeOfExpression (
            WORD num )
```

Definition at line 2497 of file execute.c.

4.31.3.19 UpdatePositions()

```
void UpdatePositions (
            void  )
```

Definition at line 2509 of file execute.c.

4.31.3.20 CountTerms1()

```
LONG CountTerms1 (
            PHEAD0  )
```

Definition at line 2569 of file execute.c.

4.31.3.21 TermsInBracket()

```
LONG TermsInBracket (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 2677 of file execute.c.

4.32 execute.c

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032 /*
00033     #[ Includes : execute.c
00034 */
00035
00036 #include "form3.h"
00037
00038 /*
00039     #] Includes :
00040     #[ DoExecute :
00041         #[ CleanExpr :
00042
00043         par == 1 after .store or .clear
00044         par == 0 after .sort
00045 */
00046
00047 int CleanExpr(WORD par)
00048 {
00049     GETIDENTITY
00050     WORD j, n, i;
00051     POSITION length;
00052     EXPRESSIONS e_in, e_out, e;
00053     int numhid = 0;
00054     NAMENODE *node;
00055     n = NumExpressions;
00056     j = 0;
00057     e_in = e_out = Expressions;
00058     if ( n > 0 ) { do {
00059         e_in->vflags &= ~( TOBEFACTORED | TOBEUNFACTORED );
00060         if ( par ) {
00061             if ( e_in->renumlists ) {
00062                 if ( e_in->renumlists != AN.dummyrenumlist )
00063                         M_free(e_in->renumlists,"Renumber-lists");
00064                 e_in->renumlists = 0;
00065             }
00066             if ( e_in->renum ) {
00067                 M_free(e_in->renum,"Renumber"); e_in->renum = 0;
00068             }
00069         }
00070         if ( e_in->status == HIDDENLEXPRESSION
00071         || e_in->status == HIDDENGEXPRESSION ) numhid++;
00072         switch ( e_in->status ) {
00073             case SPECTATOREXPRESSION:
00074             case LOCALEXPRESSION:
00075             case HIDDENLEXPRESSION:
00076                 if ( par ) {
00077                     AC.exprnames->namenode[e_in->node].type = CDELETE;
00078                     AC.DidClean = 1;
00079                     if ( e_in->status != HIDDENLEXPRESSION )
00080                         ClearBracketIndex(e_in-Expressions);
00081                     break;
00082                 }
00083                 /* fall through */
00084             case GLOBALEXPRESSION:
00085             case HIDDENGEXPRESSION:
00086                 if ( par ) {
```

```
00087 #ifdef WITHMPI
00088                     /*
00089                      * Broadcast the global expression from the master to the all workers.
00090                      */
00091                     if ( PF_BroadcastExpr(e_in, e_in->status == HIDDENGEXPRESSION ? AR.hidefile :
      AR.outfile) ) return -1;
00092                     if ( PF.me == MASTER ) {
00093 #endif
00094                     e = e_in;
00095                     i = n-1;
00096                     while ( --i >= 0 ) {
00097                         e++;
00098                         if ( e_in->status == HIDDENGEXPRESSION ) {
00099                             if ( e->status == HIDDENGEXPRESSION
00100                             || e->status == HIDDENLEXPRESSION ) break;
00101                         }
00102                         else {
00103                             if ( e->status == GLOBALEXPRESSION
00104                             || e->status == LOCALEXPRESSION ) break;
00105                         }
00106                     }
00107 #ifdef WITHMPI
00108                     }
00109                     else {
00110                         /*
00111                          * On the slaves, the broadcast expression is sitting at the end of the file.
00112                          */
00113                         e = e_in;
00114                         i = -1;
00115                     }
00116 #endif
00117                     if ( i >= 0 ) {
00118                         DIFPOS(length,e->onfile,e_in->onfile);
00119                     }
00120                     else {
00121                         FILEHANDLE *f = e_in->status == HIDDENGEXPRESSION ? AR.hidefile : AR.outfile;
00122                         if ( f->handle < 0 ) {
00123                             SETBASELENGTH(length,TOLONG(f->POfull)
00124                              - TOLONG(f->PObuffer)
00125                              - BASEPOSITION(e_in->onfile));
00126                         }
00127                         else {
00128                             SeekFile(f->handle,&(f->filesize),SEEK_SET);
00129                             DIFPOS(length,f->filesize,e_in->onfile);
00130                         }
00131                     }
00132                     if ( ToStorage(e_in,&length) ) {
00133                         return(MesCall("CleanExpr"));
00134                     }
00135                     e_in->status = STOREDEXPRESSION;
00136                     if ( e_in->status != HIDDENGEXPRESSION )
00137                         ClearBracketIndex(e_in-Expressions);
00138                 }
00139                 /* fall through */
00140             case SKIPLEXPRESSION:
00141             case DROPLEXPRESSION:
00142             case DROPHLEXPRESSION:
00143             case DROPGEXPRESSION:
00144             case DROPHGEXPRESSION:
00145             case STOREDEXPRESSION:
00146             case DROPSPECTATOREXPRESSION:
00147                 if ( e_out != e_in ) {
00148                     node = AC.exprnames->namenode + e_in->node;
00149                     node->number = e_out - Expressions;
00150 
00151                     e_out->onfile = e_in->onfile;
00152                     e_out->size = e_in->size;
00153                     e_out->printflag = 0;
00154                     if ( par ) e_out->status = STOREDEXPRESSION;
00155                     else e_out->status = e_in->status;
00156                     e_out->name = e_in->name;
00157                     e_out->node = e_in->node;
00158                     e_out->renum = e_in->renum;
00159                     e_out->renumlists = e_in->renumlists;
00160                     e_out->counter = e_in->counter;
00161                     e_out->hidelevel = e_in->hidelevel;
00162                     e_out->inmem = e_in->inmem;
00163                     e_out->bracketinfo = e_in->bracketinfo;
00164                     e_out->newbracketinfo = e_in->newbracketinfo;
00165                     e_out->numdummies = e_in->numdummies;
00166                     e_out->numfactors = e_in->numfactors;
00167                     e_out->vflags = e_in->vflags;
00168                     e_out->uflags = e_in->uflags;
00169                     e_out->sizeprototype = e_in->sizeprototype;
00170                 }
00171 #ifdef PARALLELCODE
00172                 e_out->partodo = 0;
```

```
00173 #endif
00174                 e_out++;
00175                 j++;
00176                 break;
00177             case DROPPEDEXPRESSION:
00178                 break;
00179             default:
00180                 AC.exprnames->namenode[e_in->node].type = CDELETE;
00181                 AC.DidClean = 1;
00182                 break;
00183         }
00184         e_in++;
00185     } while ( --n > 0 ); }
00186     UpdateMaxSize();
00187     NumExpressions = j;
00188     if ( numhid == 0 && AR.hidefile->PObuffer ) {
00189         if ( AR.hidefile->handle >= 0 ) {
00190             CloseFile(AR.hidefile->handle);
00191             remove(AR.hidefile->name);
00192             AR.hidefile->handle = -1;
00193         }
00194         AR.hidefile->POfull =
00195         AR.hidefile->POfill = AR.hidefile->PObuffer;
00196         PUTZERO(AR.hidefile->POposition);
00197     }
00198     FlushSpectators();
00199     return(0);
00200 }
00201
00202 /*
00203         #] CleanExpr :
00204         #[ PopVariables :
00205
00206     Pops the local variables from the tables.
00207     The Expressions are reprocessed and their tables are compactified.
00208
00209 */
00210
00211 int PopVariables(void)
00212 {
00213     GETIDENTITY
00214     WORD i, j;
00215     int retval;
00216     UBYTE *s;
00217
00218     retval = CleanExpr(1);
00219     ResetVariables(1);
00220
00221     if ( AC.DidClean ) CompactifyTree(AC.exprnames,EXPRNAMES);
00222
00223     AC.CodesFlag = AM.gCodesFlag;
00224     AC.NamesFlag = AM.gNamesFlag;
00225     AC.StatsFlag = AM.gStatsFlag;
00226 #ifdef WITHFLOAT
00227     AC.MaxWeight = AM.gMaxWeight;
00228     AC.DefaultPrecision = AM.gDefaultPrecision;
00229 #endif
00230     AC.OldFactArgFlag = AM.gOldFactArgFlag;
00231     AC.TokensWriteFlag = AM.gTokensWriteFlag;
00232     AC.extrasymbols = AM.gextrasymbols;
00233     if ( AC.extrasym ) { M_free(AC.extrasym,"extrasym"); AC.extrasym = 0; }
00234     i = 1; s = AM.gextrasym; while ( *s ) { s++; i++; }
00235     AC.extrasym = (UBYTE *)Malloc1(i*sizeof(UBYTE),"extrasym");
00236     for ( j = 0; j < i; j++ ) AC.extrasym[j] = AM.gextrasym[j];
00237     AO.NoSpacesInNumbers = AM.gNoSpacesInNumbers;
00238     AO.IndentSpace = AM.gIndentSpace;
00239     AC.lUnitTrace = AM.gUnitTrace;
00240     AC.lDefDim = AM.gDefDim;
00241     AC.lDefDim4 = AM.gDefDim4;
00242     if ( AC.halfmod ) {
00243         if ( AC.ncmod == AM.gncmod && AC.modmode == AM.gmodmode ) {
00244             j = ABS(AC.ncmod);
00245             while ( --j >= 0 ) {
00246                 if ( AC.cmod[j] != AM.gcmod[j] ) break;
00247             }
00248             if ( j >= 0 ) {
00249                 M_free(AC.halfmod,"halfmod");
00250                 AC.halfmod = 0; AC.nhalfmod = 0;
00251             }
00252         }
00253         else {
00254             M_free(AC.halfmod,"halfmod");
00255             AC.halfmod = 0; AC.nhalfmod = 0;
00256         }
00257     }
00258     if ( AC.modinverses ) {
00259         if ( AC.ncmod == AM.gncmod && AC.modmode == AM.gmodmode ) {
```

```
00260             j = ABS(AC.ncmod);
00261             while ( --j >= 0 ) {
00262                 if ( AC.cmod[j] != AM.gcmod[j] ) break;
00263             }
00264             if ( j >= 0 ) {
00265                 M_free(AC.modinverses,"modinverses");
00266                 AC.modinverses = 0;
00267             }
00268         }
00269         else {
00270             M_free(AC.modinverses,"modinverses");
00271             AC.modinverses = 0;
00272         }
00273     }
00274     AN.ncmod = AC.ncmod = AM.gncmod;
00275     AC.npowmod = AM.gnpowmod;
00276     AC.modmode = AM.gmodmode;
00277     if ( ( ( AC.modmode & INVERSETABLE ) != 0 ) && ( AC.modinverses == 0 ) )
00278             MakeInverses();
00279     AC.funpowers = AM.gfunpowers;
00280     AC.lPolyFun = AM.gPolyFun;
00281     AC.lPolyFunInv = AM.gPolyFunInv;
00282     AC.lPolyFunType = AM.gPolyFunType;
00283     AC.lPolyFunExp = AM.gPolyFunExp;
00284     AR.PolyFunVar = AC.lPolyFunVar = AM.gPolyFunVar;
00285     AC.lPolyFunPow = AM.gPolyFunPow;
00286     AC.parallelflag = AM.gparallelflag;
00287     AC.ProcessBucketSize = AC.mProcessBucketSize = AM.gProcessBucketSize;
00288     AC.properorderflag = AM.gproperorderflag;
00289     AC.ThreadBucketSize = AM.gThreadBucketSize;
00290     AC.ThreadStats = AM.gThreadStats;
00291     AC.FinalStats = AM.gFinalStats;
00292     AC.OldGCDflag = AM.gOldGCDflag;
00293     AC.WTimeStatsFlag = AM.gWTimeStatsFlag;
00294     AC.ThreadsFlag = AM.gThreadsFlag;
00295     AC.ThreadBalancing = AM.gThreadBalancing;
00296     AC.ThreadSortFileSynch = AM.gThreadSortFileSynch;
00297     AC.ProcessStats = AM.gProcessStats;
00298     AC.OldParallelStats = AM.gOldParallelStats;
00299     AC.IsFortran90 = AM.gIsFortran90;
00300     AC.SizeCommuteInSet = AM.gSizeCommuteInSet;
00301     PruneExtraSymbols(AM.gnumextrasym);
00302
00303     if ( AC.Fortran90Kind ) {
00304         M_free(AC.Fortran90Kind,"Fortran90 Kind");
00305         AC.Fortran90Kind = 0;
00306     }
00307     if ( AM.gFortran90Kind ) {
00308         AC.Fortran90Kind = strDup1(AM.gFortran90Kind,"Fortran90 Kind");
00309     }
00310     if ( AC.ThreadsFlag && AM.totalnumberofthreads > 1 ) AS.MultiThreaded = 1;
00311     {
00312         UWORD *p, *m;
00313         p = AM.gcmod;
00314         m = AC.cmod;
00315         j = ABS(AC.ncmod);
00316         NCOPY(m,p,j);
00317         p = AM.gpowmod;
00318         m = AC.powmod;
00319         j = AC.npowmod;
00320         NCOPY(m,p,j);
00321         if ( AC.DirtPow ) {
00322             if ( MakeModTable() ) {
00323                 MesPrint("===No printing in powers of generator");
00324             }
00325             AC.DirtPow = 0;
00326         }
00327     }
00328     {
00329         WORD *p, *m;
00330         p = AM.gUniTrace;
00331         m = AC.lUniTrace;
00332         j = 4;
00333         NCOPY(m,p,j);
00334     }
00335     AC.Cnumpows = AM.gCnumpows;
00336     AC.OutputMode = AM.gOutputMode;
00337     AC.OutputSpaces = AM.gOutputSpaces;
00338     AC.OutNumberType = AM.gOutNumberType;
00339     AR.SortType = AC.SortType = AM.gSortType;
00340     AC.ShortStatsMax = AM.gShortStatsMax;
00341 /*
00342     Now we have to clean up the commutation properties
00343 */
00344     for ( i = 0; i < NumFunctions; i++ ) functions[i].flags &= ~COULDCOMMUTE;
00345     if ( AC.CommuteInSet ) {
00346         WORD *g, *gg;
```

```
00347         g = AC.CommuteInSet;
00348         while ( *g ) {
00349             gg = g+1; g += *g;
00350             while ( gg < g ) {
00351                 if ( *gg <= GAMMASEVEN && *gg >= GAMMA ) {
00352                     functions[GAMMA-FUNCTION].flags |= COULDCOMMUTE;
00353                     functions[GAMMAI-FUNCTION].flags |= COULDCOMMUTE;
00354                     functions[GAMMAFIVE-FUNCTION].flags |= COULDCOMMUTE;
00355                     functions[GAMMASIX-FUNCTION].flags |= COULDCOMMUTE;
00356                     functions[GAMMASEVEN-FUNCTION].flags |= COULDCOMMUTE;
00357                 }
00358                 else {
00359                     functions[*gg-FUNCTION].flags |= COULDCOMMUTE;
00360                 }
00361             }
00362         }
00363     }
00364 /*
00365     Clean up the dictionaries.
00366 */
00367     for ( i = AO.NumDictionaries-1; i >= AO.gNumDictionaries; i-- ) {
00368         RemoveDictionary(AO.Dictionaries[i]);
00369         M_free(AO.Dictionaries[i],"Dictionary");
00370     }
00371     for( ; i >= 0; i-- ) {
00372         ShrinkDictionary(AO.Dictionaries[i]);
00373     }
00374     AO.NumDictionaries = AO.gNumDictionaries;
00375     return(retval);
00376 }
00377
00378 /*
00379         #] PopVariables :
00380         #[ MakeGlobal :
00381 */
00382
00383 void MakeGlobal(void)
00384 {
00385     WORD i, j, *pp, *mm;
00386     UWORD *p, *m;
00387     UBYTE *s;
00388     Globalize(0);
00389
00390     AM.gCodesFlag = AC.CodesFlag;
00391     AM.gNamesFlag = AC.NamesFlag;
00392     AM.gStatsFlag = AC.StatsFlag;
00393 #ifdef WITHFLOAT
00394     AM.gMaxWeight = AC.MaxWeight;
00395     AM.gDefaultPrecision = AC.DefaultPrecision;
00396 #endif
00397     AM.gOldFactArgFlag = AC.OldFactArgFlag;
00398     AM.gextrasymbols = AC.extrasymbols;
00399     if ( AM.gextrasym ) { M_free(AM.gextrasym,"extrasym"); AM.gextrasym = 0; }
00400     i = 1; s = AC.extrasym; while ( *s ) { s++; i++; }
00401     AM.gextrasym = (UBYTE *)Malloc1(i*sizeof(UBYTE),"extrasym");
00402     for ( j = 0; j < i; j++ ) AM.gextrasym[j] = AC.extrasym[j];
00403     AM.gTokensWriteFlag= AC.TokensWriteFlag;
00404     AM.gNoSpacesInNumbers = AO.NoSpacesInNumbers;
00405     AM.gIndentSpace = AO.IndentSpace;
00406     AM.gUnitTrace = AC.lUnitTrace;
00407     AM.gDefDim = AC.lDefDim;
00408     AM.gDefDim4 = AC.lDefDim4;
00409     AM.gncmod = AC.ncmod;
00410     AM.gnpowmod = AC.npowmod;
00411     AM.gmodmode = AC.modmode;
00412     AM.gCnumpows = AC.Cnumpows;
00413     AM.gOutputMode = AC.OutputMode;
00414     AM.gOutputSpaces = AC.OutputSpaces;
00415     AM.gOutNumberType = AC.OutNumberType;
00416     AM.gfunpowers = AC.funpowers;
00417     AM.gPolyFun = AC.lPolyFun;
00418     AM.gPolyFunInv = AC.lPolyFunInv;
00419     AM.gPolyFunType = AC.lPolyFunType;
00420     AM.gPolyFunExp = AC.lPolyFunExp;
00421     AM.gPolyFunVar = AC.lPolyFunVar;
00422     AM.gPolyFunPow = AC.lPolyFunPow;
00423     AM.gparallelflag = AC.parallelflag;
00424     AM.gProcessBucketSize = AC.ProcessBucketSize;
00425     AM.gproperorderflag = AC.properorderflag;
00426     AM.gThreadBucketSize = AC.ThreadBucketSize;
00427     AM.gThreadStats = AC.ThreadStats;
00428     AM.gFinalStats = AC.FinalStats;
00429     AM.gOldGCDflag = AC.OldGCDflag;
00430     AM.gWTimeStatsFlag = AC.WTimeStatsFlag;
00431     AM.gThreadsFlag = AC.ThreadsFlag;
00432     AM.gThreadBalancing = AC.ThreadBalancing;
00433     AM.gThreadSortFileSynch = AC.ThreadSortFileSynch;
```

```
00434     AM.gProcessStats = AC.ProcessStats;
00435     AM.gOldParallelStats = AC.OldParallelStats;
00436     AM.gIsFortran90 = AC.IsFortran90;
00437     AM.gSizeCommuteInSet = AC.SizeCommuteInSet;
00438     AM.gnumextrasym = (cbuf+AM.sbufnum)->numrhs;
00439     if ( AM.gFortran90Kind ) {
00440         M_free(AM.gFortran90Kind,"Fortran 90 Kind");
00441         AM.gFortran90Kind = 0;
00442     }
00443     if ( AC.Fortran90Kind ) {
00444         AM.gFortran90Kind = strDup1(AC.Fortran90Kind,"Fortran 90 Kind");
00445     }
00446     p = AM.gcmod;
00447     m = AC.cmod;
00448     i = ABS(AC.ncmod);
00449     NCOPY(p,m,i);
00450     p = AM.gpowmod;
00451     m = AC.powmod;
00452     i = AC.npowmod;
00453     NCOPY(p,m,i);
00454     pp = AM.gUniTrace;
00455     mm = AC.lUniTrace;
00456     i = 4;
00457     NCOPY(pp,mm,i);
00458     AM.gSortType = AC.SortType;
00459     AM.gShortStatsMax = AC.ShortStatsMax;
00460
00461     if ( AO.CurrentDictionary > 0 || AP.OpenDictionary > 0 ) {
00462         Warning("You cannot have an open or selected dictionary at a .global. Dictionary closed.");
00463         AP.OpenDictionary = 0;
00464         AO.CurrentDictionary = 0;
00465     }
00466
00467     AO.gNumDictionaries = AO.NumDictionaries;
00468     for ( i = 0; i < AO.NumDictionaries; i++ ) {
00469         AO.Dictionaries[i]->gnumelements = AO.Dictionaries[i]->numelements;
00470     }
00471     if ( AM.NumSpectatorFiles > 0 ) {
00472         for ( i = 0; i < AM.SizeForSpectatorFiles; i++ ) {
00473             if ( AM.SpectatorFiles[i].name != 0 )
00474                     AM.SpectatorFiles[i].flags |= GLOBALSPECTATORFLAG;
00475         }
00476     }
00477 }
00478
00479 /*
00480         #] MakeGlobal :
00481         #[ TestDrop :
00482 */
00483
00484 void TestDrop(void)
00485 {
00486     EXPRESSIONS e;
00487     WORD j;
00488     for ( j = 0, e = Expressions; j < NumExpressions; j++, e++ ) {
00489         switch ( e->status ) {
00490             case SKIPLEXPRESSION:
00491                 e->status = LOCALEXPRESSION;
00492                 break;
00493             case UNHIDELEXPRESSION:
00494                 e->status = LOCALEXPRESSION;
00495                 ClearBracketIndex(j);
00496                 e->bracketinfo = e->newbracketinfo; e->newbracketinfo = 0;
00497                 break;
00498             case HIDELEXPRESSION:
00499                 e->status = HIDDENLEXPRESSION;
00500                 break;
00501             case SKIPGEXPRESSION:
00502                 e->status = GLOBALEXPRESSION;
00503                 break;
00504             case UNHIDEGEXPRESSION:
00505                 e->status = GLOBALEXPRESSION;
00506                 ClearBracketIndex(j);
00507                 e->bracketinfo = e->newbracketinfo; e->newbracketinfo = 0;
00508                 break;
00509             case HIDEGEXPRESSION:
00510                 e->status = HIDDENGEXPRESSION;
00511                 break;
00512             case DROPLEXPRESSION:
00513             case DROPGEXPRESSION:
00514             case DROPHLEXPRESSION:
00515             case DROPHGEXPRESSION:
00516             case DROPSPECTATOREXPRESSION:
00517                 e->status = DROPPEDEXPRESSION;
00518                 ClearBracketIndex(j);
00519                 e->bracketinfo = e->newbracketinfo; e->newbracketinfo = 0;
00520                 if ( e->replace >= 0 ) {
```

```
00521                     Expressions[e->replace].replace = REGULAREXPRESSION;
00522                     AC.exprnames->namenode[e->node].number = e->replace;
00523                     e->replace = REGULAREXPRESSION;
00524                 }
00525                 else {
00526                     AC.exprnames->namenode[e->node].type = CDELETE;
00527                     AC.DidClean = 1;
00528                 }
00529                 break;
00530             case LOCALEXPRESSION:
00531             case GLOBALEXPRESSION:
00532                 ClearBracketIndex(j);
00533                 e->bracketinfo = e->newbracketinfo; e->newbracketinfo = 0;
00534                 break;
00535             case HIDDENLEXPRESSION:
00536             case HIDDENGEXPRESSION:
00537                 break;
00538             case INTOHIDELEXPRESSION:
00539                 ClearBracketIndex(j);
00540                 e->bracketinfo = e->newbracketinfo; e->newbracketinfo = 0;
00541                 e->status = HIDDENLEXPRESSION;
00542                 break;
00543             case INTOHIDEGEXPRESSION:
00544                 ClearBracketIndex(j);
00545                 e->bracketinfo = e->newbracketinfo; e->newbracketinfo = 0;
00546                 e->status = HIDDENGEXPRESSION;
00547                 break;
00548             default:
00549                 ClearBracketIndex(j);
00550                 e->bracketinfo = 0;
00551                 break;
00552         }
00553         if ( e->replace == NEWLYDEFINEDEXPRESSION ) e->replace = REGULAREXPRESSION;
00554     }
00555 }
00556
00557 /*
00558         #] TestDrop :
00559         #[ PutInVflags :
00560 */
00561
00562 void PutInVflags(WORD nexpr)
00563 {
00564     EXPRESSIONS e = Expressions + nexpr;
00565     POSITION *old;
00566     WORD *oldw;
00567     int i;
00568 restart:;
00569     if ( AS.OldOnFile == 0 ) {
00570         AS.NumOldOnFile = 20;
00571         AS.OldOnFile = (POSITION *)Malloc1(AS.NumOldOnFile*sizeof(POSITION),"file pointers");
00572     }
00573     else if ( nexpr >= AS.NumOldOnFile ) {
00574         old = AS.OldOnFile;
00575         AS.OldOnFile = (POSITION *)Malloc1(2*AS.NumOldOnFile*sizeof(POSITION),"file pointers");
00576         for ( i = 0; i < AS.NumOldOnFile; i++ ) AS.OldOnFile[i] = old[i];
00577         AS.NumOldOnFile = 2*AS.NumOldOnFile;
00578         M_free(old,"process file pointers");
00579     }
00580     if ( AS.OldNumFactors == 0 ) {
00581         AS.NumOldNumFactors = 20;
00582         AS.OldNumFactors = (WORD *)Malloc1(AS.NumOldNumFactors*sizeof(WORD),"numfactors pointers");
00583         AS.Oldvflags = (WORD *)Malloc1(AS.NumOldNumFactors*sizeof(WORD),"vflags pointers");
00584         AS.Olduflags = (WORD *)Malloc1(AS.NumOldNumFactors*sizeof(WORD),"uflags pointers");
00585     }
00586     else if ( nexpr >= AS.NumOldNumFactors ) {
00587         oldw = AS.OldNumFactors;
00588         AS.OldNumFactors = (WORD *)Malloc1(2*AS.NumOldNumFactors*sizeof(WORD),"numfactors pointers");
00589         for ( i = 0; i < AS.NumOldNumFactors; i++ ) AS.OldNumFactors[i] = oldw[i];
00590         M_free(oldw,"numfactors pointers");
00591         oldw = AS.Oldvflags;
00592         AS.Oldvflags = (WORD *)Malloc1(2*AS.NumOldNumFactors*sizeof(WORD),"vflags pointers");
00593         for ( i = 0; i < AS.NumOldNumFactors; i++ ) AS.Oldvflags[i] = oldw[i];
00594         M_free(oldw,"vflags pointers");
00595         oldw = AS.Olduflags;
00596         AS.Olduflags = (WORD *)Malloc1(2*AS.NumOldNumFactors*sizeof(WORD),"uflags pointers");
00597         for ( i = 0; i < AS.NumOldNumFactors; i++ ) AS.Olduflags[i] = oldw[i];
00598         M_free(oldw,"uflags pointers");
00599         AS.NumOldNumFactors = 2*AS.NumOldNumFactors;
00600     }
00601 /*
00602     The next is needed when we Load a .sav file with lots of expressions.
00603 */
00604     if ( nexpr >= AS.NumOldOnFile || nexpr >= AS.NumOldNumFactors ) goto restart;
00605     AS.OldOnFile[nexpr] = e->onfile;
00606     AS.OldNumFactors[nexpr] = e->numfactors;
00607     AS.Oldvflags[nexpr] = e->vflags;
```

```
00608     AS.Olduflags[nexpr] = e->uflags;
00609 }
00610
00611 /*
00612         #] PutInVflags :
00613         #[ DoExecute :
00614 */
00615
00616 int DoExecute(WORD par, WORD skip)
00617 {
00618     GETIDENTITY
00619     int RetCode = 0;
00620     int i, oldmultithreaded = AS.MultiThreaded;
00621 #ifdef PARALLELCODE
00622     int j;
00623 #endif
00624
00625     SpecialCleanup(BHEAD0);
00626     if ( skip ) goto skipexec;
00627     if ( AC.IfLevel > 0 ) {
00628         MesPrint(" %d endif statement(s) missing",AC.IfLevel);
00629         RetCode = 1;
00630     }
00631     if ( AC.WhileLevel > 0 ) {
00632         MesPrint(" %d endwhile statement(s) missing",AC.WhileLevel);
00633         RetCode = 1;
00634     }
00635     if ( AC.arglevel > 0 ) {
00636         MesPrint(" %d endargument statement(s) missing",AC.arglevel);
00637         RetCode = 1;
00638     }
00639     if ( AC.termlevel > 0 ) {
00640         MesPrint(" %d endterm statement(s) missing",AC.termlevel);
00641         RetCode = 1;
00642     }
00643     if ( AC.insidelevel > 0 ) {
00644         MesPrint(" %d endinside statement(s) missing",AC.insidelevel);
00645         RetCode = 1;
00646     }
00647     if ( AC.inexprlevel > 0 ) {
00648         MesPrint(" %d endinexpression statement(s) missing",AC.inexprlevel);
00649         RetCode = 1;
00650     }
00651     if ( AC.NumLabels > 0 ) {
00652         for ( i = 0; i < AC.NumLabels; i++ ) {
00653             if ( AC.Labels[i] < 0 ) {
00654                 MesPrint(" -->Label %s missing",AC.LabelNames[i]);
00655                 RetCode = 1;
00656             }
00657         }
00658     }
00659     if ( AC.SwitchLevel > 0 ) {
00660         MesPrint(" %d endswitch statement(s) missing",AC.SwitchLevel);
00661         RetCode = 1;
00662     }
00663     if ( AC.dolooplevel > 0 ) {
00664         MesPrint(" %d enddo statement(s) missing",AC.dolooplevel);
00665         RetCode = 1;
00666     }
00667     if ( AP.OpenDictionary > 0 ) {
00668         MesPrint(" Dictionary %s has not been closed.",
00669             AO.Dictionaries[AP.OpenDictionary-1]->name);
00670         AP.OpenDictionary = 0;
00671         RetCode = 1;
00672     }
00673     if ( RetCode ) return(RetCode);
00674     AR.Cnumlhs = cbuf[AM.rbufnum].numlhs;
00675
00676     if ( ( AS.ExecMode = par ) == GLOBALMODULE ) AS.ExecMode = 0;
00677 #ifdef PARALLELCODE
00678 /*
00679         Now check whether we have either the regular parallel flag or the
00680         mparallel flag set.
00681         Next check whether any of the expressions has partodo set.
00682         If any of the above we need to check what the dollar status is.
00683 */
00684     AC.partodoflag = -1;
00685     if ( NumPotModdollars >= 0 ) {
00686         for ( i = 0; i < NumExpressions; i++ ) {
00687             if ( Expressions[i].partodo ) { AC.partodoflag = 1; break; }
00688         }
00689     }
00690 #ifdef WITHMPI
00691     if ( AC.partodoflag > 0 && PF.numtasks < 3 ) {
00692         AC.partodoflag = 0;
00693     }
00694 #endif
```

```
00695     if ( AC.partodoflag > 0 || ( NumPotModdollars > 0 && AC.mparallelflag == PARALLELFLAG ) ) {
00696       if ( NumPotModdollars > NumModOptdollars ) {
00697         AC.mparallelflag |= NOPARALLEL_DOLLAR;
00698 #ifdef WITHPTHREADS
00699         AS.MultiThreaded = 0;
00700 #endif
00701         AC.partodoflag = 0;
00702       }
00703       else {
00704         for ( i = 0; i < NumPotModdollars; i++ ) {
00705           for ( j = 0; j < NumModOptdollars; j++ )
00706             if ( PotModdollars[i] == ModOptdollars[j].number ) break;
00707           if ( j >= NumModOptdollars ) {
00708             AC.mparallelflag |= NOPARALLEL_DOLLAR;
00709 #ifdef WITHPTHREADS
00710             AS.MultiThreaded = 0;
00711 #endif
00712             AC.partodoflag = 0;
00713             break;
00714           }
00715           switch ( ModOptdollars[j].type ) {
00716             case MODSUM:
00717             case MODMAX:
00718             case MODMIN:
00719             case MODLOCAL:
00720                 break;
00721             default:
00722                 AC.mparallelflag |= NOPARALLEL_DOLLAR;
00723                 AS.MultiThreaded = 0;
00724                 AC.partodoflag = 0;
00725                 break;
00726           }
00727         }
00728       }
00729     }
00730     else if ( ( AC.mparallelflag & NOPARALLEL_USER ) != 0 ) {
00731 #ifdef WITHPTHREADS
00732         AS.MultiThreaded = 0;
00733 #endif
00734         AC.partodoflag = 0;
00735     }
00736     if ( AC.partodoflag == 0 ) {
00737         for ( i = 0; i < NumExpressions; i++ ) {
00738             Expressions[i].partodo = 0;
00739         }
00740     }
00741     else if ( AC.partodoflag == -1 ) {
00742         AC.partodoflag = 0;
00743     }
00744 #endif
00745 #ifdef WITHMPI
00746     /*
00747      * Check RHS expressions.
00748      */
00749     if ( AC.RhsExprInModuleFlag && (AC.mparallelflag == PARALLELFLAG || AC.partodoflag) ) {
00750         if (PF.rhsInParallel) {
00751             PF.mkSlaveInfile=1;
00752             if(PF.me != MASTER){
00753                 PF.slavebuf.PObuffer=(WORD *)Malloc1(AM.ScratSize*sizeof(WORD),"PF inbuf");
00754                 PF.slavebuf.POsize=AM.ScratSize*sizeof(WORD);
00755                 PF.slavebuf.POfull = PF.slavebuf.POfill = PF.slavebuf.PObuffer;
00756                 PF.slavebuf.POstop= PF.slavebuf.PObuffer+AM.ScratSize;
00757                 PUTZERO(PF.slavebuf.POposition);
00758             }/*if(PF.me != MASTER)*/
00759         }
00760         else {
00761             AC.mparallelflag |= NOPARALLEL_RHS;
00762             AC.partodoflag = 0;
00763             for ( i = 0; i < NumExpressions; i++ ) {
00764                 Expressions[i].partodo = 0;
00765             }
00766         }
00767     }
00768     /*
00769      * Set $-variables with MODSUM to zero on the slaves.
00770      */
00771     if ( (AC.mparallelflag == PARALLELFLAG || AC.partodoflag) && PF.me != MASTER ) {
00772         for ( i = 0; i < NumModOptdollars; i++ ) {
00773             if ( ModOptdollars[i].type == MODSUM ) {
00774                 DOLLARS d = Dollars + ModOptdollars[i].number;
00775                 d->type = DOLZERO;
00776                 if ( d->where && d->where != &AM.dollarzero ) M_free(d->where, "old content of
      dollar");
00777                 d->where = &AM.dollarzero;
00778                 d->size = 0;
00779                 CleanDollarFactors(d);
00780             }
```

```
00781         }
00782     }
00783 #endif
00784     AR.SortType = AC.SortType;
00785 #ifdef WITHMPI
00786     if ( PF.me == MASTER )
00787 #endif
00788     {
00789         if ( AC.SetupFlag ) WriteSetup();
00790         if ( AC.NamesFlag || AC.CodesFlag ) WriteLists();
00791     }
00792     if ( par == GLOBALMODULE ) MakeGlobal();
00793     if ( RevertScratch() ) return(-1);
00794     if ( AC.ncmod ) SetMods();
00795 /*
00796     Warn if the module has to run in sequential mode due to some problems.
00797 */
00798 #ifdef WITHMPI
00799     if ( PF.me == MASTER )
00800 #endif
00801     {
00802         if ( !AC.ThreadsFlag || AC.mparallelflag & NOPARALLEL_USER ) {
00803             /* The user switched off the parallel execution explicitly. */
00804         }
00805         else if ( AC.mparallelflag & NOPARALLEL_DOLLAR ) {
00806             if ( AC.WarnFlag >= 1 ) {  /* Warning */
00807                 int i, j, k, n;
00808                 UBYTE *s, *s1;
00809                 s = strDup1((UBYTE *)"","NOPARALLEL_DOLLAR s");
00810                 n = 0;
00811                 j = NumPotModdollars;
00812                 for ( i = 0; i < j; i++ ) {
00813                     for ( k = 0; k < NumModOptdollars; k++ )
00814                         if ( ModOptdollars[k].number == PotModdollars[i] ) break;
00815                     if ( k >= NumModOptdollars ) {
00816                         /* global $-variable */
00817                         if ( n > 0 )
00818                             s = AddToString(s,(UBYTE *)", ",0);
00819                         s = AddToString(s,(UBYTE *)"$",0);
00820                         s = AddToString(s,DOLLARNAME(Dollars,PotModdollars[i]),0);
00821                         n++;
00822                     }
00823                 }
00824                 s1 = strDup1((UBYTE *)"This module is forced to run in sequential mode due to
      $-variable","NOPARALLEL_DOLLAR s1");
00825                 if ( n != 1 )
00826                     s1 = AddToString(s1,(UBYTE *)"s",0);
00827                 s1 = AddToString(s1,(UBYTE *)": ",0);
00828                 s1 = AddToString(s1,s,0);
00829                 Warning((char *)s1);
00830                 M_free(s,"NOPARALLEL_DOLLAR s");
00831                 M_free(s1,"NOPARALLEL_DOLLAR s1");
00832             }
00833         }
00834         else if ( AC.mparallelflag & NOPARALLEL_RHS ) {
00835             HighWarning("This module is forced to run in sequential mode due to RHS expression
      names");
00836         }
00837         else if ( AC.mparallelflag & NOPARALLEL_CONVPOLY ) {
00838             HighWarning("This module is forced to run in sequential mode due to conversion to extra
      symbols");
00839         }
00840         else if ( AC.mparallelflag & NOPARALLEL_SPECTATOR ) {
00841             HighWarning("This module is forced to run in sequential mode due to
      tospectator/copyspectator");
00842         }
00843         else if ( AC.mparallelflag & NOPARALLEL_TBLDOLLAR ) {
00844             HighWarning("This module is forced to run in sequential mode due to $-variable assignments
      in tables");
00845         }
00846         else if ( AC.mparallelflag & NOPARALLEL_NPROC ) {
00847             HighWarning("This module is forced to run in sequential mode because there is only one
      processor");
00848         }
00849     }
00850 /*
00851     Now the actual execution
00852 */
00853 #ifdef WITHMPI
00854     /*
00855      * Turn on AS.printflag to print runtime errors occurring on slaves.
00856      */
00857     AS.printflag = 1;
00858 #endif
00859     if ( AP.preError == 0 && ( Processor() || WriteAll() ) ) RetCode = -1;
00860 #ifdef WITHMPI
00861     AS.printflag = 0;
```

```
00862 #endif
00863 /*
00864     That was it. Next is cleanup.
00865 */
00866     if ( AC.ncmod ) UnSetMods();
00867     AS.MultiThreaded = oldmultithreaded;
00868     TableReset();
00869
00870 /*[28sep2005 mt]:*/
00871 #ifdef WITHMPI
00872     /* Combine and then broadcast modified dollar variables. */
00873     if ( NumPotModdollars > 0 ) {
00874         RetCode = PF_CollectModifiedDollars();
00875         if ( RetCode ) return RetCode;
00876         RetCode = PF_BroadcastModifiedDollars();
00877         if ( RetCode ) return RetCode;
00878     }
00879     /* Broadcast the list of objects converted to symbols in AM.sbufnum. */
00880     if ( AC.topolynomialflag & TOPOLYNOMIALFLAG ) {
00881         RetCode = PF_BroadcastCBuf(AM.sbufnum);
00882         if ( RetCode ) return RetCode;
00883     }
00884     /*
00885      * Broadcast AR.expflags, which may be used on the slaves in the next module
00886      * via ZERO_ or UNCHANGED_. It also broadcasts several flags of each expression.
00887      */
00888     RetCode = PF_BroadcastExpFlags();
00889     if ( RetCode ) return RetCode;
00890     /*
00891      * Clean the hide file on the slaves, which was used for RHS expressions
00892      * broadcast from the master at the beginning of the module.
00893      */
00894     if ( PF.me != MASTER && AR.hidefile->PObuffer ) {
00895         if ( AR.hidefile->handle >= 0 ) {
00896             CloseFile(AR.hidefile->handle);
00897             AR.hidefile->handle = -1;
00898             remove(AR.hidefile->name);
00899         }
00900         AR.hidefile->POfull = AR.hidefile->POfill = AR.hidefile->PObuffer;
00901         PUTZERO(AR.hidefile->POposition);
00902     }
00903 #endif
00904 #ifdef WITHPTHREADS
00905     for ( j = 0; j < NumModOptdollars; j++ ) {
00906         if ( ModOptdollars[j].dstruct ) {
00907
00908             //Here we must collect global maximum or minimum dollar values, for
00909             //MODMAX and MODMIN cases, and put them in the global dollar variable.
00910             //Then we clean up.
00911
00912             if ( ModOptdollars[j].type == MODMAX || ModOptdollars[j].type == MODMIN ) {
00913                 const WORD globalnumber = ModOptdollars[j].number;
00914                 const DOLLARS gd = Dollars + globalnumber;
00915
00916                 // If the global dollar is DOLZERO, convert it into DOLNUMBER.
00917                 // Note that parts of the code rely on the trailing zero.
00918                 if ( gd->type == DOLZERO ) {
00919                     gd->type = DOLNUMBER;
00920                     if ( ! gd->where || gd->where == &(AM.dollarzero) ) {
00921                         gd->size = MINALLOC;
00922                         gd->where = (WORD*)Malloc1(gd->size*sizeof(WORD), "dollar contents");
00923                     }
00924                     gd->where[0] = 4;
00925                     gd->where[1] = 0;
00926                     gd->where[2] = 1;
00927                     gd->where[3] = 3;
00928                     gd->where[4] = 0;
00929                 }
00930
00931                 for ( i = 0; i < AM.totalnumberofthreads; i++ ) {
00932                     const DOLLARS ld = &(ModOptdollars[j].dstruct[i]);
00933                     // This can happen when a thread doesn't obtain a dollar value,
00934                     // for instance if it did not process any terms.
00935                     if ( ld->type == DOLUNDEFINED ) continue;
00936
00937                     if ( ld->type != DOLZERO && ld->type != DOLNUMBER ) {
00938                         MLOCK(ErrorMessageLock);
00939                         MesPrint("Illegal dollar variable type in MODMIN/MODMAX case: %d", ld->type);
00940                         MUNLOCK(ErrorMessageLock);
00941                         Terminate(-1);
00942                     }
00943
00944                     // If the thread value is DOLZERO, convert it to DOLNUMBER:
00945                     if ( ld->type == DOLZERO ) {
00946                         ld->type = DOLNUMBER;
00947                         if ( ! ld->where || ld->where == &(AM.dollarzero) ) {
00948                             ld->size = MINALLOC;
```

```
00949                           ld->where = (WORD*)Malloc1(ld->size*sizeof(WORD), "dollar contents");
00950                       }
00951                       ld->where[0] = 4;
00952                       ld->where[1] = 0;
00953                       ld->where[2] = 1;
00954                       ld->where[3] = 3;
00955                       ld->where[4] = 0;
00956                   }
00957
00958                   // If the global dollar is DOLUNDEFINED, just take the thread value. This comes
00959                   // after the above "continue" if the local dollar is DOLUNDEFINED: thus, if the
00960                   // global and all local dollars are DOLUNDEFINED, the final global dollar will
00961                   // still be DOLUNDEFINED.
00962                   if ( gd->type == DOLUNDEFINED ) {
00963                       gd->type = ld->type;
00964                       if ( ! gd->where || gd->where == &(AM.dollarzero) ) {
00965                           gd->size = MINALLOC;
00966                           gd->where = (WORD*)Malloc1(gd->size*sizeof(WORD), "dollar contents");
00967                       }
00968                       for ( int v = 0; v < 5; v++ ) {
00969                           gd->where[v] = ld->where[v];
00970                       }
00971                       continue;
00972                   }
00973
00974                   // MODMIN and MODMAX are supposed to work only for "short integers". Thus
00975                   // the term data should be "4 N 1 +-3". Use CompCoef to make the comparison:
00976                   const WORD cmp = CompCoef(gd->where, ld->where);
00977                   if ( ( ModOptdollars[j].type == MODMAX && cmp < 0 ) ||
00978                        ( ModOptdollars[j].type == MODMIN && cmp > 0 ) ) {
00979                       // Update the global value:
00980                       for ( int v = 0; v < 5; v++ ) {
00981                           gd->where[v] = ld->where[v];
00982                       }
00983                       if ( gd->where[4] != 0 ) {
00984                           // The loop above should have put a trailing zero after the coeff
00985                           // size. If not, there is probably a bug somewhere else...
00986                           MLOCK(ErrorMessageLock);
00987                           MesPrint("Missing trailing zero in MODMIN/MODMAX global dollar %d",
00988                               globalnumber);
00989                           MUNLOCK(ErrorMessageLock);
00990                           Terminate(-1);
00991                       }
00992                   }
00993               }
00994
00995               // If the global dollar is 0, convert it into DOLZERO:
00996               if ( gd->where[1] == 0 ) {
00997                   gd->type = DOLZERO;
00998                   if ( gd->where && gd->where != &(AM.dollarzero) ) {
00999                       M_free(gd->where, "dollar contents");
01000                   }
01001                   gd->size = 0;
01002                   gd->where = &(AM.dollarzero);
01003               }
01004           }
01005
01006           // Now clean up the thread-local variables
01007           for ( i = 0; i < AM.totalnumberofthreads; i++ ) {
01008               if ( ModOptdollars[j].dstruct[i].size > 0 ) {
01009                   CleanDollarFactors(&(ModOptdollars[j].dstruct[i]));
01010                   if ( ModOptdollars[j].dstruct[i].where
01011                       && ModOptdollars[j].dstruct[i].where != &(AM.dollarzero) ) {
01012                       M_free(ModOptdollars[j].dstruct[i].where,"Local dollar value");
01013                   }
01014               }
01015           }
01016 /*
01017           Now clean up the whole array.
01018 */
01019           M_free(ModOptdollars[j].dstruct,"Local DOLLARS");
01020           ModOptdollars[j].dstruct = 0;
01021       }
01022   }
01023 #endif
01024 /*:[28sep2005 mt]*/
01025
01026 /*
01027     @@@@@@@@@@@@@@@
01028     Now follows the code to invalidate caches for all objects in the
01029     PotModdollars. There are NumPotModdollars of them and PotModdollars
01030     is an array of WORD.
01031 */
01032 /*
01033     Cleanup:
01034 */
01035 #ifdef JV_IS_WRONG
```

```
01036 /*
01037     Giving back this memory gives way too much activity with Malloc1
01038     Better to keep it and just put the number of used objects to zero (JV)
01039     If you put the lijst equal to NULL, please also make maxnum = 0
01040 */
01041     if ( ModOptdollars ) M_free(ModOptdollars, "ModOptdollars pointer");
01042     if ( PotModdollars ) M_free(PotModdollars, "PotModdollars pointer");
01043 
01044     /* ModOptdollars changed to AC.ModOptDolList.lijst because AIX C compiler complained. MF
      30/07/2003. */
01045     AC.ModOptDolList.lijst = NULL;
01046     /* PotModdollars changed to AC.PotModDolList.lijst because AIX C compiler complained. MF
      30/07/2003. */
01047     AC.PotModDolList.lijst = NULL;
01048 #endif
01049     NumPotModdollars = 0;
01050     NumModOptdollars = 0;
01051 
01052 skipexec:
01053 /*
01054     Clean up the switch information.
01055     We keep the switch array and heap.
01056 */
01057 if ( AC.SwitchInArray > 0 ) {
01058     for ( i = 0; i < AC.SwitchInArray; i++ ) {
01059         SWITCH *sw = AC.SwitchArray + i;
01060         if ( sw->table ) M_free(sw->table,"Switch table");
01061         sw->table = 0;
01062         sw->defaultcase.ncase = 0;
01063         sw->defaultcase.value = 0;
01064         sw->defaultcase.compbuffer = 0;
01065         sw->endswitch.ncase = 0;
01066         sw->endswitch.value = 0;
01067         sw->endswitch.compbuffer = 0;
01068         sw->typetable = 0;
01069         sw->maxcase = 0;
01070         sw->mincase = 0;
01071         sw->numcases = 0;
01072         sw->tablesize = 0;
01073         sw->caseoffset = 0;
01074         sw->iflevel = 0;
01075         sw->whilelevel = 0;
01076         sw->nestingsum = 0;
01077     }
01078     AC.SwitchInArray = 0;
01079     AC.SwitchLevel = 0;
01080 }
01081 #ifdef PARALLELCODE
01082     AC.numpfirstnum = 0;
01083 #endif
01084     AC.DidClean = 0;
01085     AC.PolyRatFunChanged = 0;
01086     TestDrop();
01087     if ( par == STOREMODULE || par == CLEARMODULE ) {
01088         ClearOptimize();
01089         if ( par == STOREMODULE && PopVariables() ) RetCode = -1;
01090         if ( AR.infile->handle >= 0 ) {
01091             CloseFile(AR.infile->handle);
01092             remove(AR.infile->name);
01093             AR.infile->handle = -1;
01094         }
01095         AR.infile->POfill = AR.infile->PObuffer;
01096         PUTZERO(AR.infile->POposition);
01097         AR.infile->POfull = AR.infile->PObuffer;
01098         if ( AR.outfile->handle >= 0 ) {
01099             CloseFile(AR.outfile->handle);
01100             remove(AR.outfile->name);
01101             AR.outfile->handle = -1;
01102         }
01103         AR.outfile->POfull =
01104         AR.outfile->POfill = AR.outfile->PObuffer;
01105         PUTZERO(AR.outfile->POposition);
01106         if ( AR.hidefile->handle >= 0 ) {
01107             CloseFile(AR.hidefile->handle);
01108             remove(AR.hidefile->name);
01109             AR.hidefile->handle = -1;
01110         }
01111         AR.hidefile->POfull =
01112         AR.hidefile->POfill = AR.hidefile->PObuffer;
01113         PUTZERO(AR.hidefile->POposition);
01114         AC.HideLevel = 0;
01115         if ( par == CLEARMODULE ) {
01116             if ( DeleteStore(0) < 0 ) {
01117 /* INTERNAL_ERROR_EXCL_START */
01118                 MesPrint("!>Cannot restart the storage file");
01119                 RetCode = -1;
01120 /* INTERNAL_ERROR_EXCL_STOP */
```

```
01121             }
01122             else RetCode = 0;
01123             CleanUp(1);
01124             ResetVariables(2);
01125             AM.gProcessBucketSize = AM.hProcessBucketSize;
01126             AM.gparallelflag = PARALLELFLAG;
01127             AM.gnumextrasym = AM.ggnumextrasym;
01128             PruneExtraSymbols(AM.ggnumextrasym);
01129             IniVars();
01130         }
01131         ClearSpectators(par);
01132     }
01133     else {
01134         if ( CleanExpr(0) ) RetCode = -1;
01135         if ( AC.DidClean ) CompactifyTree(AC.exprnames,EXPRNAMES);
01136         ResetVariables(0);
01137         CleanUpSort(-1);
01138     }
01139     clearcbuf(AC.cbufnum);
01140     if ( AC.MultiBracketBuf != 0 ) {
01141       for ( i = 0; i < MAXMULTIBRACKETLEVELS; i++ ) {
01142         if ( AC.MultiBracketBuf[i] ) {
01143             M_free(AC.MultiBracketBuf[i],"bracket buffer i");
01144             AC.MultiBracketBuf[i] = 0;
01145         }
01146       }
01147       AC.MultiBracketLevels = 0;
01148       M_free(AC.MultiBracketBuf,"multi bracket buffer");
01149       AC.MultiBracketBuf = 0;
01150     }
01151
01152     if ( AC.SortReallocateFlag ) {
01153         /* Reallocate the sort buffers to reduce resident set usage */
01154         /* AT.SS is the same as AT.S0 here */
01155         SORTING* S = AT.S0;
01156         M_free(S->lBuffer, "SortReallocate lBuffer+sBuffer");
01157         S->lBuffer = Malloc1(sizeof(*(S->lBuffer))*(S->LargeSize+S->SmallEsize), "SortReallocate
      lBuffer+sBuffer");
01158         S->lTop = S->lBuffer+S->LargeSize;
01159         S->sBuffer = S->lTop;
01160         if ( S->LargeSize == 0 ) { S->lBuffer = 0; S->lTop = 0; }
01161         S->sTop = S->sBuffer + S->SmallSize;
01162         S->sTop2 = S->sBuffer + S->SmallEsize;
01163         S->sHalf = S->sBuffer + (LONG)((S->SmallSize+S->SmallEsize)>>1);
01164
01165 #ifdef WITHPTHREADS
01166         /* We have to re-set the pointers into master lBuffer in the SortBlocks */
01167         UpdateSortBlocks(AM.totalnumberofthreads-1);
01168
01169         /* The SortBots do not have a real sort buffer to reallocate. */
01170         /* AB[0] has been reallocated above already. */
01171         for ( i = 1; i < AM.totalnumberofthreads; i++ ) {
01172             SORTING* S = AB[i]->T.S0;
01173             M_free(S->lBuffer, "SortReallocate lBuffer+sBuffer");
01174             S->lBuffer = Malloc1(sizeof(*(S->lBuffer))*(S->LargeSize+S->SmallEsize), "SortReallocate
      lBuffer+sBuffer");
01175             S->lTop = S->lBuffer+S->LargeSize;
01176             S->sBuffer = S->lTop;
01177             if ( S->LargeSize == 0 ) { S->lBuffer = 0; S->lTop = 0; }
01178             S->sTop = S->sBuffer + S->SmallSize;
01179             S->sTop2 = S->sBuffer + S->SmallEsize;
01180             S->sHalf = S->sBuffer + (LONG)((S->SmallSize+S->SmallEsize)>>1);
01181         }
01182 #endif
01183     }
01184     if ( AC.SortReallocateFlag == 2 ) {
01185         /* The Flag was set for a single module by the preprocessor #sortreallocate,
01186            so turn it off again. */
01187         AC.SortReallocateFlag = 0;
01188     }
01189
01190     return(RetCode);
01191 }
01192
01193 /*
01194         #] DoExecute :
01195         #[ PutBracket :
01196
01197     Routine uses the bracket info to split a term into two pieces:
01198     1: the part outside the bracket, and
01199     2: the part inside the bracket.
01200     These parts are separated by a subterm of type HAAKJE.
01201     This subterm looks like: HAAKJE,3,level
01202     The level is used for nestings of brackets. The print routines
01203     cannot handle this yet (31-Mar-1988).
01204
01205     The Bracket selector is in AT.BrackBuf in the form of a regular term,
```

```
01206     but without coefficient.
01207     When AR.BracketOn < 0 we have a socalled antibracket. The main effect
01208     is an exchange of the inner and outer part and where the coefficient goes.
01209
01210     Routine recoded to facilitate b p1,p2; etc for dotproducts and tensors
01211     15-oct-1991
01212 */
01213
01214 int PutBracket(PHEAD WORD *termin)
01215 {
01216     GETBIDENTITY
01217     WORD *t, *t1, *b, i, j, *lastfun;
01218     WORD *t2, *s1, *s2;
01219     WORD *bStop, *bb, *bf, *tStop;
01220     WORD *term1,*term2, *m1, *m2, *tStopa;
01221     WORD *bbb = 0, *bind, *binst = 0, bwild = 0, *bss = 0, *bns = 0, bset = 0;
01222     term1 = AT.WorkPointer+1;
01223     term2 = (WORD *)(((UBYTE *)(term1)) + AM.MaxTer);
01224     if ( ( (WORD *)(((UBYTE *)(term2)) + AM.MaxTer) ) > AT.WorkTop ) {
01225         MesWork();
01226     }
01227     if ( AR.BracketOn < 0 ) {
01228         t2 = term1; t1 = term2;     /* AntiBracket */
01229     }
01230     else {
01231         t1 = term1; t2 = term2;     /* Regular bracket */
01232     }
01233     b = AT.BrackBuf; bStop = b+*b; b++;
01234     while ( b < bStop ) {
01235         if ( *b == INDEX ) { bwild = 1; bbb = b+2; binst = b + b[1]; }
01236         if ( *b == SETSET ) { bset = 1; bss = b+2; bns = b + b[1]; }
01237         b += b[1];
01238     }
01239
01240     t = termin; tStopa = t + *t; i = *(t + *t -1); i = ABS(i);
01241     if ( AR.PolyFun && AT.PolyAct ) tStop = termin + AT.PolyAct;
01242 #ifdef WITHFLOAT
01243     else if ( AT.FloatPos ) tStop = termin + AT.FloatPos;
01244 #endif
01245     else tStop = tStopa - i;
01246     t++;
01247     if ( AR.BracketOn < 0 ) {
01248         lastfun = 0;
01249         while ( t < tStop && *t >= FUNCTION
01250             && functions[*t-FUNCTION].commute ) {
01251             b = AT.BrackBuf+1;
01252             while ( b < bStop ) {
01253                 if ( *b == *t ) {
01254                     lastfun = t;
01255                     while ( t < tStop && *t >= FUNCTION
01256                         && functions[*t-FUNCTION].commute ) t += t[1];
01257                     goto NextNcom1;
01258                 }
01259                 b += b[1];
01260             }
01261             if ( bset ) {
01262                 b = bss;
01263                 while ( b < bns ) {
01264                     if ( b[1] == CFUNCTION ) { /* Set of functions */
01265                         SETS set = Sets+b[0]; WORD i;
01266                         for ( i = set->first; i < set->last; i++ ) {
01267                             if ( SetElements[i] == *t ) {
01268                                 lastfun = t;
01269                                 while ( t < tStop && *t >= FUNCTION
01270                                     && functions[*t-FUNCTION].commute ) t += t[1];
01271                                 goto NextNcom1;
01272                             }
01273                         }
01274                     }
01275                     b += 2;
01276                 }
01277             }
01278             if ( bwild && *t >= FUNCTION && functions[*t-FUNCTION].spec ) {
01279                 s1 = t + t[1];
01280                 s2 = t + FUNHEAD;
01281                 while ( s2 < s1 ) {
01282                     bind = bbb;
01283                     while ( bind < binst ) {
01284                         if ( *bind == *s2 ) {
01285                             lastfun = t;
01286                             while ( t < tStop && *t >= FUNCTION
01287                                 && functions[*t-FUNCTION].commute ) t += t[1];
01288                             goto NextNcom1;
01289                         }
01290                         bind++;
01291                     }
01292                     s2++;
```

```
01293                 }
01294             }
01295             t += t[1];
01296         }
01297 NextNcom1:
01298         s1 = termin + 1;
01299         if ( lastfun ) {
01300             while ( s1 < lastfun ) *t2++ = *s1++;
01301             while ( s1 < t ) *t1++ = *s1++;
01302         }
01303         else {
01304             while ( s1 < t ) *t2++ = *s1++;
01305         }
01306
01307     }
01308     else {
01309         lastfun = t;
01310         while ( t < tStop && *t >= FUNCTION
01311             && functions[*t-FUNCTION].commute ) {
01312             b = AT.BrackBuf+1;
01313             while ( b < bStop ) {
01314                 if ( *b == *t ) { lastfun = t + t[1]; goto NextNcom; }
01315                 b += b[1];
01316             }
01317             if ( bset ) {
01318                 b = bss;
01319                 while ( b < bns ) {
01320                     if ( b[1] == CFUNCTION ) { /* Set of functions */
01321                         SETS set = Sets+b[0]; WORD i;
01322                         for ( i = set->first; i < set->last; i++ ) {
01323                             if ( SetElements[i] == *t ) {
01324                                 lastfun = t + t[1];
01325                                 goto NextNcom;
01326                             }
01327                         }
01328                     }
01329                     b += 2;
01330                 }
01331             }
01332             if ( bwild && *t >= FUNCTION && functions[*t-FUNCTION].spec ) {
01333                 s1 = t + t[1];
01334                 s2 = t + FUNHEAD;
01335                 while ( s2 < s1 ) {
01336                     bind = bbb;
01337                     while ( bind < binst ) {
01338                         if ( *bind == *s2 ) { lastfun = t + t[1]; goto NextNcom; }
01339                         bind++;
01340                     }
01341                     s2++;
01342                 }
01343             }
01344 NextNcom:
01345             t += t[1];
01346         }
01347         s1 = termin + 1;
01348         while ( s1 < lastfun ) *t1++ = *s1++;
01349         while ( s1 < t ) *t2++ = *s1++;
01350     }
01351 /*
01352     Now we have only commuting functions left. Move the b pointer to them.
01353 */
01354     b = AT.BrackBuf + 1;
01355     while ( b < bStop && *b >= FUNCTION
01356         && ( *b < FUNCTION || functions[*b-FUNCTION].commute ) ) {
01357         b += b[1];
01358     }
01359     bf = b;
01360
01361     while ( t < tStop && ( bf < bStop || bwild || bset ) ) {
01362         b = bf;
01363         while ( b < bStop && *b != *t ) { b += b[1]; }
01364         i = t[1];
01365         if ( *t >= FUNCTION ) { /* We are in function territory */
01366             if ( b < bStop && *b == *t ) goto FunBrac;
01367             if ( bset ) {
01368                 b = bss;
01369                 while ( b < bns ) {
01370                     if ( b[1] == CFUNCTION ) { /* Set of functions */
01371                         SETS set = Sets+b[0]; WORD i;
01372                         for ( i = set->first; i < set->last; i++ ) {
01373                             if ( SetElements[i] == *t ) goto FunBrac;
01374                         }
01375                     }
01376                     b += 2;
01377                 }
01378             }
01379             if ( bwild && *t >= FUNCTION && functions[*t-FUNCTION].spec ) {
```

```
01380                 s1 = t + t[1];
01381                 s2 = t + FUNHEAD;
01382                 while ( s2 < s1 ) {
01383                     bind = bbb;
01384                     while ( bind < binst ) {
01385                         if ( *bind == *s2 ) goto FunBrac;
01386                         bind++;
01387                     }
01388                     s2++;
01389                 }
01390             }
01391             NCOPY(t2,t,i);
01392             continue;
01393 FunBrac:    NCOPY(t1,t,i);
01394             continue;
01395         }
01396 /*
01397     We have left: DELTA, INDEX, VECTOR, DOTPRODUCT, SYMBOL
01398 */
01399         if ( *t == DELTA ) {
01400             if ( b < bStop && *b == DELTA ) {
01401                 b += b[1];
01402                 NCOPY(t1,t,i);
01403             }
01404             else { NCOPY(t2,t,i); }
01405         }
01406         else if ( *t == INDEX ) {
01407             if ( bwild ) {
01408                 m1 = t1; m2 = t2;
01409                 *t1++ = *t; t1++; *t2++ = *t; t2++;
01410                 bind = bbb;
01411                 j = t[1] -2;
01412                 t += 2;
01413                 while ( --j >= 0 ) {
01414                     while ( *bind < *t && bind < binst ) bind++;
01415                     if ( *bind == *t && bind < binst ) {
01416                         *t1++ = *t++;
01417                     }
01418                     else if ( bset ) {
01419                         WORD *b3 = bss;
01420                         while ( b3 < bns ) {
01421                             if ( b3[1] == CVECTOR ) {
01422                                 SETS set = Sets+b3[0]; WORD i;
01423                                 for ( i = set->first; i < set->last; i++ ) {
01424                                     if ( SetElements[i] == *t ) {
01425                                         *t1++ = *t++;
01426                                         goto nextind;
01427                                     }
01428                                 }
01429                             }
01430                             b3 += 2;
01431                         }
01432                         *t2++ = *t++;
01433                     }
01434                     else *t2++ = *t++;
01435 nextind:;
01436                 }
01437                 m1[1] = WORDDIF(t1,m1);
01438                 if ( m1[1] == 2 ) t1 = m1;
01439                 m2[1] = WORDDIF(t2,m2);
01440                 if ( m2[1] == 2 ) t2 = m2;
01441             }
01442             else if ( bset ) {
01443                 m1 = t1; m2 = t2;
01444                 *t1++ = *t; t1++; *t2++ = *t; t2++;
01445                 j = t[1] -2;
01446                 t += 2;
01447                 while ( --j >= 0 ) {
01448                     WORD *b3 = bss;
01449                     while ( b3 < bns ) {
01450                         if ( b3[1] == CVECTOR ) {
01451                             SETS set = Sets+b3[0]; WORD i;
01452                             for ( i = set->first; i < set->last; i++ ) {
01453                                 if ( SetElements[i] == *t ) {
01454                                     *t1++ = *t++;
01455                                     goto nextind2;
01456                                 }
01457                             }
01458                         }
01459                         b3 += 2;
01460                     }
01461                     *t2++ = *t++;
01462 nextind2:;
01463                 }
01464                 m1[1] = WORDDIF(t1,m1);
01465                 if ( m1[1] == 2 ) t1 = m1;
01466                 m2[1] = WORDDIF(t2,m2);
```

```
01467                 if ( m2[1] == 2 ) t2 = m2;
01468             }
01469             else {
01470                 NCOPY(t2,t,i);
01471             }
01472         }
01473         else if ( *t == VECTOR ) {
01474             if ( ( b < bStop && *b == VECTOR ) || bwild ) {
01475                 if ( b < bStop && *b == VECTOR ) {
01476                     bb = b + b[1]; b += 2;
01477                 }
01478                 else bb = b;
01479                 j = t[1] - 2;
01480                 m1 = t1; m2 = t2; *t1++ = *t; *t2++ = *t; t1++; t2++; t += 2;
01481                 while ( j > 0 ) {
01482                     j -= 2;
01483                     while ( b < bb && ( *b < *t ||
01484                     ( *b == *t && b[1] < t[1] ) ) ) b += 2;
01485                     if ( b < bb && ( *t == *b && t[1] == b[1] ) ) {
01486                         *t1++ = *t++; *t1++ = *t++; goto nextvec;
01487                     }
01488                     else if ( bwild ) {
01489                         bind = bbb;
01490                         while ( bind < binst ) {
01491                             if ( *t == *bind || t[1] == *bind ) {
01492                                 *t1++ = *t++; *t1++ = *t++;
01493                                 goto nextvec;
01494                             }
01495                             bind++;
01496                         }
01497                     }
01498                     if ( bset ) {
01499                         WORD *b3 = bss;
01500                         while ( b3 < bns ) {
01501                             if ( b3[1] == CVECTOR ) {
01502                                 SETS set = Sets+b3[0]; WORD i;
01503                                 for ( i = set->first; i < set->last; i++ ) {
01504                                     if ( SetElements[i] == *t ) {
01505                                         *t1++ = *t++; *t1++ = *t++;
01506                                         goto nextvec;
01507                                     }
01508                                 }
01509                             }
01510                             b3 += 2;
01511                         }
01512                     }
01513                     *t2++ = *t++; *t2++ = *t++;
01514 nextvec:;
01515                 }
01516                 m1[1] = WORDDIF(t1,m1);
01517                 if ( m1[1] == 2 ) t1 = m1;
01518                 m2[1] = WORDDIF(t2,m2);
01519                 if ( m2[1] == 2 ) t2 = m2;
01520             }
01521             else if ( bset ) {
01522                 m1 = t1; *t1++ = *t; t1++;
01523                 m2 = t2; *t2++ = *t; t2++;
01524                 s2 = t + i; t += 2;
01525                 while ( t < s2 ) {
01526                     WORD *b3 = bss;
01527                     while ( b3 < bns ) {
01528                         if ( b3[1] == CVECTOR ) {
01529                             SETS set = Sets+b3[0]; WORD i;
01530                             for ( i = set->first; i < set->last; i++ ) {
01531                                 if ( SetElements[i] == *t ) {
01532                                     *t1++ = *t++; *t1++ = *t++;
01533                                     goto nextvec2;
01534                                 }
01535                             }
01536                         }
01537                         b3 += 2;
01538                     }
01539                     *t2++ = *t++; *t2++ = *t++;
01540 nextvec2:;
01541                 }
01542                 m1[1] = WORDDIF(t1,m1);
01543                 if ( m1[1] == 2 ) t1 = m1;
01544                 m2[1] = WORDDIF(t2,m2);
01545                 if ( m2[1] == 2 ) t2 = m2;
01546             }
01547             else {
01548                 NCOPY(t2,t,i);
01549             }
01550         }
01551         else if ( *t == DOTPRODUCT ) {
01552             if ( ( b < bStop && *b == *t ) || bwild ) {
01553                 m1 = t1; *t1++ = *t; t1++;
```

```
01554                 m2 = t2; *t2++ = *t; t2++;
01555                 if ( b >= bStop || *b != *t ) { bb = b; s1 = b; }
01556                 else {
01557                     s1 = b + b[1]; bb = b + 2;
01558                 }
01559                 s2 = t + i; t += 2;
01560                 while ( t < s2 && ( bb < s1 || bwild || bset ) ) {
01561                     while ( bb < s1 && ( *bb < *t ||
01562                     ( *bb == *t && bb[1] < t[1] ) ) ) bb += 3;
01563                     if ( bb < s1 && *bb == *t && bb[1] == t[1] ) {
01564                         *t1++ = *t++; *t1++ = *t++; *t1++ = *t++; bb += 3;
01565                         goto nextdot;
01566                     }
01567                     else if ( bwild ) {
01568                         bind = bbb;
01569                         while ( bind < binst ) {
01570                             if ( *bind == *t || *bind == t[1] ) {
01571                                 *t1++ = *t++; *t1++ = *t++; *t1++ = *t++;
01572                                 goto nextdot;
01573                             }
01574                             bind++;
01575                         }
01576                     }
01577                     if ( bset ) {
01578                         WORD *b3 = bss;
01579                         while ( b3 < bns ) {
01580                             if ( b3[1] == CVECTOR ) {
01581                                 SETS set = Sets+b3[0]; WORD i;
01582                                 for ( i = set->first; i < set->last; i++ ) {
01583                                     if ( SetElements[i] == *t || SetElements[i] == t[1] ) {
01584                                         *t1++ = *t++; *t1++ = *t++; *t1++ = *t++;
01585                                         goto nextdot;
01586                                     }
01587                                 }
01588                             }
01589                             b3 += 2;
01590                         }
01591                     }
01592                     *t2++ = *t++; *t2++ = *t++; *t2++ = *t++;
01593 nextdot:;
01594                 }
01595                 while ( t < s2 ) *t2++ = *t++;
01596                 m1[1] = WORDDIF(t1,m1);
01597                 if ( m1[1] == 2 ) t1 = m1;
01598                 m2[1] = WORDDIF(t2,m2);
01599                 if ( m2[1] == 2 ) t2 = m2;
01600             }
01601             else if ( bset ) {
01602                 m1 = t1; *t1++ = *t; t1++;
01603                 m2 = t2; *t2++ = *t; t2++;
01604                 s2 = t + i; t += 2;
01605                 while ( t < s2 ) {
01606                     WORD *b3 = bss;
01607                     while ( b3 < bns ) {
01608                         if ( b3[1] == CVECTOR ) {
01609                             SETS set = Sets+b3[0]; WORD i;
01610                             for ( i = set->first; i < set->last; i++ ) {
01611                                 if ( SetElements[i] == *t || SetElements[i] == t[1] ) {
01612                                     *t1++ = *t++; *t1++ = *t++; *t1++ = *t++;
01613                                     goto nextdot2;
01614                                 }
01615                             }
01616                         }
01617                         b3 += 2;
01618                     }
01619                     *t2++ = *t++; *t2++ = *t++; *t2++ = *t++;
01620 nextdot2:;
01621                 }
01622                 m1[1] = WORDDIF(t1,m1);
01623                 if ( m1[1] == 2 ) t1 = m1;
01624                 m2[1] = WORDDIF(t2,m2);
01625                 if ( m2[1] == 2 ) t2 = m2;
01626             }
01627             else { NCOPY(t2,t,i); }
01628         }
01629         else if ( *t == SYMBOL ) {
01630             if ( b < bStop && *b == *t ) {
01631                 m1 = t1; *t1++ = *t; t1++;
01632                 m2 = t2; *t2++ = *t; t2++;
01633                 s1 = b + b[1]; bb = b+2;
01634                 s2 = t + i; t += 2;
01635                 while ( bb < s1 && t < s2 ) {
01636                     while ( bb < s1 && *bb < *t ) bb += 2;
01637                     if ( bb >= s1 ) {
01638                         if ( bset ) goto TrySymbolSet;
01639                         break;
01640                     }
```

```
01641                     if ( *bb == *t ) { *t1++ = *t++; *t1++ = *t++; }
01642                     else if ( bset ) {
01643                         WORD *bbb;
01644 TrySymbolSet:
01645                         bbb = bss;
01646                         while ( bbb < bns ) {
01647                             if ( bbb[1] == CSYMBOL ) { /* Set of symbols */
01648                                 SETS set = Sets+bbb[0]; WORD i;
01649                                 for ( i = set->first; i < set->last; i++ ) {
01650                                     if ( SetElements[i] == *t ) {
01651                                         *t1++ = *t++; *t1++ = *t++;
01652                                         goto NextSymbol;
01653                                     }
01654                                 }
01655                             }
01656                             bbb += 2;
01657                         }
01658                         *t2++ = *t++; *t2++ = *t++;
01659                     }
01660                     else { *t2++ = *t++; *t2++ = *t++; }
01661 NextSymbol:;
01662                 }
01663                 while ( t < s2 ) *t2++ = *t++;
01664                 m1[1] = WORDDIF(t1,m1);
01665                 if ( m1[1] == 2 ) t1 = m1;
01666                 m2[1] = WORDDIF(t2,m2);
01667                 if ( m2[1] == 2 ) t2 = m2;
01668             }
01669             else if ( bset ) {
01670                 WORD *bbb;
01671                 m1 = t1; *t1++ = *t; t1++;
01672                 m2 = t2; *t2++ = *t; t2++;
01673                 s2 = t + i; t += 2;
01674                 while ( t < s2 ) {
01675                     bbb = bss;
01676                     while ( bbb < bns ) {
01677                         if ( bbb[1] == CSYMBOL ) { /* Set of symbols */
01678                             SETS set = Sets+bbb[0]; WORD i;
01679                             for ( i = set->first; i < set->last; i++ ) {
01680                                 if ( SetElements[i] == *t ) {
01681                                     *t1++ = *t++; *t1++ = *t++;
01682                                     goto NextSymbol2;
01683                                 }
01684                             }
01685                         }
01686                         bbb += 2;
01687                     }
01688                     *t2++ = *t++; *t2++ = *t++;
01689 NextSymbol2:;
01690                 }
01691                 m1[1] = WORDDIF(t1,m1);
01692                 if ( m1[1] == 2 ) t1 = m1;
01693                 m2[1] = WORDDIF(t2,m2);
01694                 if ( m2[1] == 2 ) t2 = m2;
01695             }
01696             else { NCOPY(t2,t,i); }
01697         }
01698         else {
01699             NCOPY(t2,t,i);
01700         }
01701     }
01702     if ( ( i = WORDDIF(tStop,t) ) > 0 ) NCOPY(t2,t,i);
01703     if ( AR.BracketOn < 0 ) {
01704         s1 = t1; t1 = t2; t2 = s1;
01705     }
01706     do { *t2++ = *t++; } while ( t < (WORD *)tStopa );
01707     t = AT.WorkPointer;
01708     i = WORDDIF(t1,term1);
01709     *t++ = 4 + i + WORDDIF(t2,term2);
01710     t += i;
01711     *t++ = HAAKJE;
01712     *t++ = 3;
01713     *t++ = 0;           /* This feature won't be used for a while */
01714     i = WORDDIF(t2,term2);
01715     t1 = term2;
01716     if ( i > 0 ) NCOPY(t,t1,i);
01717
01718     AT.WorkPointer = t;
01719
01720     return(0);
01721 }
01722
01723 /*
01724         #] PutBracket :
01725         #[ SpecialCleanup :
01726 */
01727
```

```
01728 void SpecialCleanup(PHEAD0)
01729 {
01730     GETBIDENTITY
01731     if ( AT.previousEfactor ) M_free(AT.previousEfactor,"Efactor cache");
01732     AT.previousEfactor = 0;
01733 }
01734
01735 /*
01736         #] SpecialCleanup :
01737         #[ SetMods :
01738 */
01739
01740 #ifndef WITHPTHREADS
01741
01742 void SetMods(void)
01743 {
01744     int i, n;
01745     if ( AN.cmod != 0 ) M_free(AN.cmod,"AN.cmod");
01746     n = ABS(AN.ncmod);
01747     AN.cmod = (UWORD *)Malloc1(sizeof(WORD)*n,"AN.cmod");
01748     for ( i = 0; i < n; i++ ) AN.cmod[i] = AC.cmod[i];
01749 }
01750
01751 #endif
01752
01753 /*
01754         #] SetMods :
01755         #[ UnSetMods :
01756 */
01757
01758 #ifndef WITHPTHREADS
01759
01760 void UnSetMods(void)
01761 {
01762     if ( AN.cmod != 0 ) M_free(AN.cmod,"AN.cmod");
01763     AN.cmod = 0;
01764 }
01765
01766 #endif
01767
01768 /*
01769         #] UnSetMods :
01770     #] DoExecute :
01771     #[ Expressions :
01772         #[ ExchangeExpressions :
01773 */
01774
01775 void ExchangeExpressions(int num1, int num2)
01776 {
01777     GETIDENTITY
01778     WORD node1, node2, namesize, TMproto[SUBEXPSIZE];
01779     INDEXENTRY *ind;
01780     EXPRESSIONS e1, e2;
01781     LONG a;
01782     SBYTE *s1, *s2;
01783     int i;
01784     e1 = Expressions + num1;
01785     e2 = Expressions + num2;
01786     node1 = e1->node;
01787     node2 = e2->node;
01788     AC.exprnames->namenode[node1].number = num2;
01789     AC.exprnames->namenode[node2].number = num1;
01790     a = e1->name; e1->name = e2->name; e2->name = a;
01791     namesize = e1->namesize; e1->namesize = e2->namesize; e2->namesize = namesize;
01792     e1->node = node2;
01793     e2->node = node1;
01794     if ( e1->status == STOREDEXPRESSION ) {
01795 /*
01796         Find the name in the index and replace by the new name
01797 */
01798         TMproto[0] = EXPRESSION;
01799         TMproto[1] = SUBEXPSIZE;
01800         TMproto[2] = num1;
01801         TMproto[3] = 1;
01802         { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
01803         AT.TMaddr = TMproto;
01804         ind = FindInIndex(num1,&AR.StoreData,0,0);
01805         s1 = (SBYTE *)(AC.exprnames->namebuffer+e1->name);
01806         i = e1->namesize;
01807         s2 = ind->name;
01808         NCOPY(s2,s1,i);
01809         *s2 = 0;
01810         SeekFile(AR.StoreData.Handle,&(e1->onfile),SEEK_SET);
01811         if ( WriteFile(AR.StoreData.Handle,(UBYTE *)ind,
01812         (LONG)(sizeof(INDEXENTRY))) != sizeof(INDEXENTRY) ) {
01813 /* INTERNAL_ERROR_EXCL_START */
01814             MesPrint("!>File error while exchanging expressions");
```

```
01815             Terminate(-1);
01816 /* INTERNAL_ERROR_EXCL_STOP */
01817         }
01818         FlushFile(AR.StoreData.Handle);
01819     }
01820     if ( e2->status == STOREDEXPRESSION ) {
01821 /*
01822         Find the name in the index and replace by the new name
01823 */
01824         TMproto[0] = EXPRESSION;
01825         TMproto[1] = SUBEXPSIZE;
01826         TMproto[2] = num2;
01827         TMproto[3] = 1;
01828         { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
01829         AT.TMaddr = TMproto;
01830         ind = FindInIndex(num1,&AR.StoreData,0,0);
01831         s1 = (SBYTE *)(AC.exprnames->namebuffer+e2->name);
01832         i = e2->namesize;
01833         s2 = ind->name;
01834         NCOPY(s2,s1,i);
01835         *s2 = 0;
01836         SeekFile(AR.StoreData.Handle,&(e2->onfile),SEEK_SET);
01837         if ( WriteFile(AR.StoreData.Handle,(UBYTE *)ind,
01838         (LONG)(sizeof(INDEXENTRY))) != sizeof(INDEXENTRY) ) {
01839 /* INTERNAL_ERROR_EXCL_START */
01840             MesPrint("!>File error while exchanging expressions");
01841             Terminate(-1);
01842 /* INTERNAL_ERROR_EXCL_STOP */
01843         }
01844         FlushFile(AR.StoreData.Handle);
01845     }
01846 }
01847
01848 /*
01849         #] ExchangeExpressions :
01850         #[ GetFirstBracket :
01851 */
01852
01853 int GetFirstBracket(WORD *term, int num)
01854 {
01855 /*
01856         Gets the first bracket of the expression 'num'
01857         Puts it in term. If no brackets the answer is one.
01858         Routine should be thread-safe
01859 */
01860     GETIDENTITY
01861     POSITION position, oldposition;
01862     RENUMBER renumber;
01863     FILEHANDLE *fi;
01864     WORD type, *oldcomppointer, oldonefile, numword;
01865     WORD *t, *tstop;
01866
01867     oldcomppointer = AR.CompressPointer;
01868     type = Expressions[num].status;
01869     if ( type == STOREDEXPRESSION ) {
01870         WORD TMproto[SUBEXPSIZE];
01871         TMproto[0] = EXPRESSION;
01872         TMproto[1] = SUBEXPSIZE;
01873         TMproto[2] = num;
01874         TMproto[3] = 1;
01875         { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
01876         AT.TMaddr = TMproto;
01877         PUTZERO(position);
01878         if ( ( renumber = GetTable(num,&position,0) ) == 0 ) {
01879             MesCall("GetFirstBracket");
01880             SETERROR(-1)
01881         }
01882         if ( GetFromStore(term,&position,renumber,&numword,num) < 0 ) {
01883             MesCall("GetFirstBracket");
01884             SETERROR(-1)
01885         }
01886 /*
01887 #ifdef WITHPTHREADS
01888 */
01889         if ( renumber->symb.lo != AN.dummyrenumlist )
01890             M_free(renumber->symb.lo,"VarSpace");
01891         M_free(renumber,"Renumber");
01892 /*
01893 #endif
01894 */
01895     }
01896     else {          /* Active expression */
01897         oldonefile = AR.GetOneFile;
01898         if ( type == HIDDENLEXPRESSION || type == HIDDENGEXPRESSION ) {
01899             AR.GetOneFile = 2; fi = AR.hidefile;
01900         }
01901         else {
```

```
01902             AR.GetOneFile = 0; fi = AR.infile;
01903         }
01904         if ( fi->handle >= 0 ) {
01905             PUTZERO(oldposition);
01906 /*
01907             SeekFile(fi->handle,&oldposition,SEEK_CUR);
01908 */
01909             }
01910         else {
01911             SETBASEPOSITION(oldposition,fi->POfill-fi->PObuffer);
01912         }
01913         position = AS.OldOnFile[num];
01914         if ( GetOneTerm(BHEAD term,fi,&position,1) < 0
01915         || ( GetOneTerm(BHEAD term,fi,&position,1) < 0 ) ) {
01916             MLOCK(ErrorMessageLock);
01917             MesCall("GetFirstBracket");
01918             MUNLOCK(ErrorMessageLock);
01919             SETERROR(-1)
01920         }
01921         if ( fi->handle >= 0 ) {
01922 /*
01923             SeekFile(fi->handle,&oldposition,SEEK_SET);
01924             if ( ISNEGPOS(oldposition) ) {
01925                 MLOCK(ErrorMessageLock);
01926                 MesPrint("File error");
01927                 MUNLOCK(ErrorMessageLock);
01928                 SETERROR(-1)
01929             }
01930 */
01931         }
01932         else {
01933             fi->POfill = fi->PObuffer+BASEPOSITION(oldposition);
01934         }
01935         AR.GetOneFile = oldonefile;
01936     }
01937     AR.CompressPointer = oldcomppointer;
01938     if ( *term ) {
01939         tstop = term + *term; tstop -= ABS(tstop[-1]);
01940         t = term + 1;
01941         while ( t < tstop ) {
01942             if ( *t == HAAKJE ) break;
01943             t += t[1];
01944         }
01945         if ( t >= tstop ) {
01946             term[0] = 4; term[1] = 1; term[2] = 1; term[3] = 3;
01947         }
01948         else {
01949             *t++ = 1; *t++ = 1; *t++ = 3; *term = t - term;
01950         }
01951     }
01952     else {
01953         term[0] = 4; term[1] = 1; term[2] = 1; term[3] = 3;
01954     }
01955     return(*term);
01956 }
01957
01958 /*
01959         #] GetFirstBracket :
01960         #[ GetFirstTerm :
01961 */
01962
01972 int GetFirstTerm(WORD *term, int num, int pre)
01973 {
01974     GETIDENTITY
01975     POSITION position, oldposition;
01976     RENUMBER renumber;
01977     FILEHANDLE *fi;
01978     WORD type, *oldcomppointer, oldonefile, numword;
01979
01980     oldcomppointer = AR.CompressPointer;
01981     type = Expressions[num].status;
01982     if ( type == STOREDEXPRESSION ) {
01983         WORD TMproto[SUBEXPSIZE];
01984         TMproto[0] = EXPRESSION;
01985         TMproto[1] = SUBEXPSIZE;
01986         TMproto[2] = num;
01987         TMproto[3] = 1;
01988         { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
01989         AT.TMaddr = TMproto;
01990         PUTZERO(position);
01991         if ( ( renumber = GetTable(num,&position,0) ) == 0 ) {
01992             MesCall("GetFirstTerm");
01993             SETERROR(-1)
01994         }
01995         if ( GetFromStore(term,&position,renumber,&numword,num) < 0 ) {
01996             MesCall("GetFirstTerm");
01997             SETERROR(-1)
```

```
01998         }
01999 /*
02000 #ifdef WITHPTHREADS
02001 */
02002         if ( renumber->symb.lo != AN.dummyrenumlist )
02003             M_free(renumber->symb.lo,"VarSpace");
02004         M_free(renumber,"Renumber");
02005 /*
02006 #endif
02007 */
02008     }
02009     else {          /* Active expression */
02010         oldonefile = AR.GetOneFile;
02011         if ( type == HIDDENLEXPRESSION || type == HIDDENGEXPRESSION ) {
02012             AR.GetOneFile = 2; fi = AR.hidefile;
02013         }
02014         else {
02015             AR.GetOneFile = 0;
02016             if ( Expressions[num].replace == NEWLYDEFINEDEXPRESSION ) {
02017                 /* During execution, if the expression has already been processed it
02018                    will be in the outfile. If it has not, the usage is illegal according
02019                    to the manual, though no error is given. */
02020                 if ( pre == 0 ) { fi = AR.outfile; }
02021                 /* During preprocessing, the expression certainly has not been processed
02022                    yet. Print an error and terminate. */
02023                 else {
02024                     MesPrint("&isnumerical: expression is not yet defined!");
02025                     SETERROR(-1);
02026                 }
02027             }
02028             else {
02029                 /* During execution, we should use the definition as stored at the end
02030                    of the previous module. This is in the infile. */
02031                 if ( pre == 0 ) { fi = AR.infile; }
02032                 /* During preprocessing, this function is called before the RevertScratch
02033                    at the beginning of this module's execution. Thus the expressions are
02034                    in the outfile of the previous module. */
02035                 else { fi = AR.outfile; }
02036             }
02037         }
02038         if ( fi->handle >= 0 ) {
02039             PUTZERO(oldposition);
02040 /*
02041             SeekFile(fi->handle,&oldposition,SEEK_CUR);
02042 */
02043             }
02044         else {
02045             SETBASEPOSITION(oldposition,fi->POfill-fi->PObuffer);
02046         }
02047         position = AS.OldOnFile[num];
02048         if ( GetOneTerm(BHEAD term,fi,&position,1) < 0
02049         || ( GetOneTerm(BHEAD term,fi,&position,1) < 0 ) ) {
02050             MLOCK(ErrorMessageLock);
02051             MesCall("GetFirstTerm");
02052             MUNLOCK(ErrorMessageLock);
02053             SETERROR(-1)
02054         }
02055         if ( fi->handle >= 0 ) {
02056 /*
02057             SeekFile(fi->handle,&oldposition,SEEK_SET);
02058             if ( ISNEGPOS(oldposition) ) {
02059                 MLOCK(ErrorMessageLock);
02060                 MesPrint("File error");
02061                 MUNLOCK(ErrorMessageLock);
02062                 SETERROR(-1)
02063             }
02064 */
02065         }
02066         else {
02067             fi->POfill = fi->PObuffer+BASEPOSITION(oldposition);
02068         }
02069         AR.GetOneFile = oldonefile;
02070     }
02071     AR.CompressPointer = oldcomppointer;
02072     return(*term);
02073 }
02074
02075 /*
02076         #] GetFirstTerm :
02077         #[ GetContent :
02078 */
02079
02080 int GetContent(WORD *content, int num)
02081 {
02082 /*
02083         Gets the content of the expression 'num'
02084         Puts it in content.
```

```
02085         Routine should be thread-safe
02086         The content is defined as the term that will make the expression 'num'
02087         with integer coefficients, no GCD and all common factors taken out,
02088         all negative powers removed when we divide the expression by this
02089         content.
02090 */
02091     GETIDENTITY
02092     POSITION position, oldposition;
02093     RENUMBER renumber;
02094     FILEHANDLE *fi;
02095     WORD type, *oldcomppointer, oldonefile, numword, *term, i;
02096     WORD *cbuffer = TermMalloc("GetContent");
02097     WORD *oldworkpointer = AT.WorkPointer;
02098 
02099     oldcomppointer = AR.CompressPointer;
02100     type = Expressions[num].status;
02101     if ( type == STOREDEXPRESSION ) {
02102         WORD TMproto[SUBEXPSIZE];
02103         TMproto[0] = EXPRESSION;
02104         TMproto[1] = SUBEXPSIZE;
02105         TMproto[2] = num;
02106         TMproto[3] = 1;
02107         { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
02108         AT.TMaddr = TMproto;
02109         PUTZERO(position);
02110         if ( ( renumber = GetTable(num,&position,0) ) == 0 ) goto CalledFrom;
02111         if ( GetFromStore(cbuffer,&position,renumber,&numword,num) < 0 ) goto CalledFrom;
02112         for(;;) {
02113             term = oldworkpointer;
02114             AR.CompressPointer = oldcomppointer;
02115             if ( GetFromStore(term,&position,renumber,&numword,num) < 0 ) goto CalledFrom;
02116             if ( *term == 0 ) break;
02117 /*
02118             'merge' the two terms
02119 */
02120             if ( ContentMerge(BHEAD cbuffer,term) < 0 ) goto CalledFrom;
02121         }
02122 /*
02123 #ifdef WITHPTHREADS
02124 */
02125         if ( renumber->symb.lo != AN.dummyrenumlist )
02126             M_free(renumber->symb.lo,"VarSpace");
02127         M_free(renumber,"Renumber");
02128 /*
02129 #endif
02130 */
02131     }
02132     else {          /* Active expression */
02133         oldonefile = AR.GetOneFile;
02134         if ( type == HIDDENLEXPRESSION || type == HIDDENGEXPRESSION ) {
02135             AR.GetOneFile = 2; fi = AR.hidefile;
02136         }
02137         else {
02138             AR.GetOneFile = 0;
02139             if ( Expressions[num].replace == NEWLYDEFINEDEXPRESSION )
02140                  fi = AR.outfile;
02141             else fi = AR.infile;
02142         }
02143         if ( fi->handle >= 0 ) {
02144             PUTZERO(oldposition);
02145 /*
02146             SeekFile(fi->handle,&oldposition,SEEK_CUR);
02147 */
02148             }
02149         else {
02150             SETBASEPOSITION(oldposition,fi->POfill-fi->PObuffer);
02151         }
02152         position = AS.OldOnFile[num];
02153         if ( GetOneTerm(BHEAD cbuffer,fi,&position,1) < 0 ) goto CalledFrom;
02154         AR.CompressPointer = oldcomppointer;
02155         if ( GetOneTerm(BHEAD cbuffer,fi,&position,1) < 0 ) goto CalledFrom;
02156 /*
02157         Now go through the terms. For each term we have to test whether
02158         what is in cbuffer is also in that term. If not, we have to remove
02159         it from cbuffer. Additionally we have to accumulate the GCD of the
02160         numerators and the LCM of the denominators. This is all done in the
02161         routine ContentMerge.
02162 */
02163         for(;;) {
02164             term = oldworkpointer;
02165             AR.CompressPointer = oldcomppointer;
02166             if ( GetOneTerm(BHEAD term,fi,&position,1) < 0 ) goto CalledFrom;
02167             if ( *term == 0 ) break;
02168 /*
02169             'merge' the two terms
02170 */
02171             if ( ContentMerge(BHEAD cbuffer,term) < 0 ) goto CalledFrom;
```

```
02172         }
02173         if ( fi->handle < 0 ) {
02174             fi->POfill = fi->PObuffer+BASEPOSITION(oldposition);
02175         }
02176         AR.GetOneFile = oldonefile;
02177     }
02178     AR.CompressPointer = oldcomppointer;
02179     for ( i = 0; i < *cbuffer; i++ ) content[i] = cbuffer[i];
02180     TermFree(cbuffer,"GetContent");
02181     AT.WorkPointer = oldworkpointer;
02182     return(*content);
02183 CalledFrom:
02184     MLOCK(ErrorMessageLock);
02185     MesCall("GetContent");
02186     MUNLOCK(ErrorMessageLock);
02187     SETERROR(-1)
02188 }
02189
02190 /*
02191         #] GetContent :
02192         #[ CleanupTerm :
02193
02194         Removes noncommuting objects from the term
02195 */
02196
02197 int CleanupTerm(WORD *term)
02198 {
02199     WORD *tstop, *t, *tfill, *tt;
02200     GETSTOP(term,tstop);
02201     t = term+1;
02202     while ( t < tstop ) {
02203         if ( *t >= FUNCTION && ( functions[*t-FUNCTION].commute || *t == DENOMINATOR ) ) {
02204             tfill = t; tt = t + t[1]; tstop = term + *term;
02205             while ( tt < tstop ) *tfill++ = *tt++;
02206             *term = tfill - term;
02207             tstop -= ABS(tfill[-1]);
02208         }
02209         else {
02210             t += t[1];
02211         }
02212     }
02213     return(0);
02214 }
02215
02216 /*
02217         #] CleanupTerm :
02218         #[ ContentMerge :
02219 */
02220
02221 WORD ContentMerge(PHEAD WORD *content, WORD *term)
02222 {
02223     GETBIDENTITY
02224     WORD *cstop, csize, crsize, sign = 1, numsize, densize, i, tnsize, tdsize;
02225     UWORD *num, *den, *tnum, *tden;
02226     WORD *outfill, *outb = TermMalloc("ContentMerge"), *ct;
02227     WORD *t, *tstop, tsize, trsize, *told;
02228     WORD *t1, *t2, *c1, *c2, i1, i2, *out1;
02229     WORD didsymbol = 0, diddotp = 0, tfirst;
02230     cstop = content + *content;
02231     csize = cstop[-1];
02232     if ( csize < 0 ) { sign = -sign; csize = -csize; }
02233     cstop -= csize;
02234     numsize = densize = crsize = (csize-1)/2;
02235     num = NumberMalloc("ContentMerge");
02236     den = NumberMalloc("ContentMerge");
02237     for ( i = 0; i < numsize; i++ ) num[i] = (UWORD)(cstop[i]);
02238     for ( i = 0; i < densize; i++ ) den[i] = (UWORD)(cstop[i+crsize]);
02239     while ( num[numsize-1] == 0 ) numsize--;
02240     while ( den[densize-1] == 0 ) densize--;
02241 /*
02242     First we do the coefficient
02243 */
02244     tstop = term + *term;
02245     tsize = tstop[-1];
02246     if ( tsize < 0 ) tsize = -tsize;
02247 /*  else { sign = 1; } */
02248     tstop = tstop - tsize;
02249     tnsize = tdsize = trsize = (tsize-1)/2;
02250     tnum = (UWORD *)tstop; tden = (UWORD *)(tstop + trsize);
02251     while ( tnum[tnsize-1] == 0 ) tnsize--;
02252     while ( tden[tdsize-1] == 0 ) tdsize--;
02253     GcdLong(BHEAD num, numsize, tnum, tnsize, num, &numsize);
02254     if ( LcmLong(BHEAD den, densize, tden, tdsize, den, &densize) ) goto CalledFrom;
02255     outfill = outb + 1;
02256     ct = content + 1;
02257     t = term + 1;
02258     while ( ct < cstop ) {
```

```
02259         switch ( *ct ) {
02260             case SYMBOL:
02261                 didsymbol = 1;
02262                 t = term+1;
02263                 while ( t < tstop && *t != *ct ) t += t[1];
02264                 if ( t >= tstop ) break;
02265                 t1 = t+2; t2 = t+t[1];
02266                 c1 = ct+2; c2 = ct+ct[1];
02267                 out1 = outfill; *outfill++ = *ct; outfill++;
02268                 while ( c1 < c2 && t1 < t2 ) {
02269                     if ( *c1 == *t1 ) {
02270                         if ( t1[1] <= c1[1] ) {
02271                             *outfill++ = *t1++; *outfill++ = *t1++;
02272                             c1 += 2;
02273                         }
02274                         else {
02275                             *outfill++ = *c1++; *outfill++ = *c1++;
02276                             t1 += 2;
02277                         }
02278                     }
02279                     else if ( *c1 < *t1 ) {
02280                         if ( c1[1] < 0 ) {
02281                             *outfill++ = *c1++; *outfill++ = *c1++;
02282                         }
02283                         else { c1 += 2; }
02284                     }
02285                     else {
02286                         if ( t1[1] < 0 ) {
02287                             *outfill++ = *t1++; *outfill++ = *t1++;
02288                         }
02289                         else t1 += 2;
02290                     }
02291                 }
02292                 while ( c1 < c2 ) {
02293                     if ( c1[1] < 0 ) { *outfill++ = c1[0]; *outfill++ = c1[1]; }
02294                     c1 += 2;
02295                 }
02296                 while ( t1 < t2 ) {
02297                     if ( t1[1] < 0 ) { *outfill++ = t1[0]; *outfill++ = t1[1]; }
02298                     t1 += 2;
02299                 }
02300                 out1[1] = outfill - out1;
02301                 if ( out1[1] == 2 ) outfill = out1;
02302                 break;
02303             case DOTPRODUCT:
02304                 diddotp = 1;
02305                 t = term+1;
02306                 while ( t < tstop && *t != *ct ) t += t[1];
02307                 if ( t >= tstop ) break;
02308                 t1 = t+2; t2 = t+t[1];
02309                 c1 = ct+2; c2 = ct+ct[1];
02310                 out1 = outfill; *outfill++ = *ct; outfill++;
02311                 while ( c1 < c2 && t1 < t2 ) {
02312                     if ( *c1 == *t1 && c1[1] == t1[1] ) {
02313                         if ( t1[2] <= c1[2] ) {
02314                             *outfill++ = *t1++; *outfill++ = *t1++; *outfill++ = *t1++;
02315                             c1 += 3;
02316                         }
02317                         else {
02318                             *outfill++ = *c1++; *outfill++ = *c1++; *outfill++ = *c1++;
02319                             t1 += 3;
02320                         }
02321                     }
02322                     else if ( *c1 < *t1 || ( *c1 == *t1 && c1[1] < t1[1] ) ) {
02323                         if ( c1[2] < 0 ) {
02324                             *outfill++ = *c1++; *outfill++ = *c1++; *outfill++ = *c1++;
02325                         }
02326                         else { c1 += 3; }
02327                     }
02328                     else {
02329                         if ( t1[2] < 0 ) {
02330                             *outfill++ = *t1++; *outfill++ = *t1++; *outfill++ = *t1++;
02331                         }
02332                         else t1 += 3;
02333                     }
02334                 }
02335                 while ( c1 < c2 ) {
02336                     if ( c1[2] < 0 ) { *outfill++ = c1[0]; *outfill++ = c1[1]; *outfill++ = c1[1]; }
02337                     c1 += 3;
02338                 }
02339                 while ( t1 < t2 ) {
02340                     if ( t1[2] < 0 ) { *outfill++ = t1[0]; *outfill++ = t1[1]; *outfill++ = t1[1]; }
02341                     t1 += 3;
02342                 }
02343                 out1[1] = outfill - out1;
02344                 if ( out1[1] == 2 ) outfill = out1;
02345                 break;
```

```
02346             case INDEX:
02347                 t = term+1;
02348                 while ( t < tstop && *t != *ct ) t += t[1];
02349                 if ( t >= tstop ) break;
02350                 t1 = t+2; t2 = t+t[1];
02351                 c1 = ct+2; c2 = ct+ct[1];
02352                 out1 = outfill; *outfill++ = *ct; outfill++;
02353                 while ( c1 < c2 && t1 < t2 ) {
02354                     if ( *c1 == *t1 ) {
02355                         *outfill++ = *c1++;
02356                         t1 += 1;
02357                     }
02358                     else if ( *c1 < *t1 ) { c1 += 1; }
02359                     else { t1 += 1; }
02360                 }
02361                 out1[1] = outfill - out1;
02362                 if ( out1[1] == 2 ) outfill = out1;
02363                 break;
02364             case VECTOR:
02365             case DELTA:
02366                 t = term+1;
02367                 while ( t < tstop && *t != *ct ) t += t[1];
02368                 if ( t >= tstop ) break;
02369                 t1 = t+2; t2 = t+t[1];
02370                 c1 = ct+2; c2 = ct+ct[1];
02371                 out1 = outfill; *outfill++ = *ct; outfill++;
02372                 while ( c1 < c2 && t1 < t2 ) {
02373                     if ( *c1 == *t1 && c1[1] && t1[1] ) {
02374                         *outfill++ = *c1++; *outfill++ = *c1++;
02375                         t1 += 2;
02376                     }
02377                     else if ( *c1 < *t1 || ( *c1 == *t1 && c1[1] < t1[1] ) ) {
02378                         c1 += 2;
02379                     }
02380                     else {
02381                         t1 += 2;
02382                     }
02383                 }
02384                 out1[1] = outfill - out1;
02385                 if ( out1[1] == 2 ) outfill = out1;
02386                 break;
02387             case GAMMA:
02388             default:            /* Functions */
02389                 told = t;
02390                 t = term+1;
02391                 while ( t < tstop ) {
02392                     if ( *t != *ct ) { t += t[1]; continue; }
02393                     if ( ct[1] != t[1] ) { t += t[1]; continue; }
02394                     if ( ct[2] != t[2] ) { t += t[1]; continue; }
02395                     t1 = t; t2 = ct; i1 = t1[1]; i2 = t2[1];
02396                     while ( i1 > 0 ) {
02397                         if ( *t1 != *t2 ) break;
02398                         t1++; t2++; i1--;
02399                     }
02400                     if ( i1 != 0 ) { t += t[1]; continue; }
02401                     t1 = t;
02402                     for ( i = 0; i < i2; i++ ) { *outfill++ = *t++; }
02403 /*
02404                     Mark as 'used'. The flags must be different!
02405 */
02406                     t1[2] |= SUBTERMUSED1;
02407                     ct[2] |= SUBTERMUSED2;
02408                     t = told;
02409                     break;
02410                 }
02411                 break;
02412         }
02413         ct += ct[1];
02414     }
02415     if ( diddotp == 0 ) {
02416         t = term+1; while ( t < tstop && *t != DOTPRODUCT ) t += t[1];
02417         if ( t < tstop ) { /* now we need the negative powers */
02418             tfirst = 1; told = outfill;
02419             for ( i = 2; i < t[1]; i += 3 ) {
02420                 if ( t[i+2] < 0 ) {
02421                     if ( tfirst ) { *outfill++ = DOTPRODUCT; *outfill++ = 0; tfirst = 0; }
02422                     *outfill++ = t[i]; *outfill++ = t[i+1]; *outfill++ = t[i+2];
02423                 }
02424             }
02425             if ( outfill > told ) told[1] = outfill-told;
02426         }
02427     }
02428     if ( didsymbol == 0 ) {
02429         t = term+1; while ( t < tstop && *t != SYMBOL ) t += t[1];
02430         if ( t < tstop ) { /* now we need the negative powers */
02431             tfirst = 1; told = outfill;
02432             for ( i = 2; i < t[1]; i += 2 ) {
```

```
02433                 if ( t[i+1] < 0 ) {
02434                     if ( tfirst ) { *outfill++ = SYMBOL; *outfill++ = 0; tfirst = 0; }
02435                     *outfill++ = t[i]; *outfill++ = t[i+1];
02436                 }
02437             }
02438             if ( outfill > told ) told[1] = outfill-told;
02439         }
02440     }
02441 /*
02442     Now put the coefficient back.
02443 */
02444     if ( numsize < densize ) {
02445         for ( i = numsize; i < densize; i++ ) num[i] = 0;
02446         numsize = densize;
02447     }
02448     else if ( densize < numsize ) {
02449         for ( i = densize; i < numsize; i++ ) den[i] = 0;
02450         densize = numsize;
02451     }
02452     for ( i = 0; i < numsize; i++ ) *outfill++ = num[i];
02453     for ( i = 0; i < densize; i++ ) *outfill++ = den[i];
02454     csize = numsize+densize+1;
02455     if ( sign < 0 ) csize = -csize;
02456     *outfill++ = csize;
02457     *outb = outfill-outb;
02458     NumberFree(den,"ContentMerge");
02459     NumberFree(num,"ContentMerge");
02460     for ( i = 0; i < *outb; i++ ) content[i] = outb[i];
02461     TermFree(outb,"ContentMerge");
02462 /*
02463     Now we have to 'restore' the term to its original.
02464     We do not restore the content, because if anything was used the
02465     new content overwrites the old. 6-mar-2018 JV
02466 */
02467     t = term + 1;
02468     while ( t < tstop ) {
02469         if ( *t >= FUNCTION ) t[2] &= ~SUBTERMUSED1;
02470         t += t[1];
02471     }
02472     return(*content);
02473 CalledFrom:
02474     MLOCK(ErrorMessageLock);
02475     MesCall("GetContent");
02476     MUNLOCK(ErrorMessageLock);
02477     SETERROR(-1)
02478 }
02479
02480 /*
02481         #] ContentMerge :
02482         #[ TermsInExpression :
02483 */
02484
02485 LONG TermsInExpression(WORD num)
02486 {
02487     LONG x = Expressions[num].counter;
02488     if ( x >= 0 ) return(x);
02489     return(-1);
02490 }
02491
02492 /*
02493         #] TermsInExpression :
02494         #[ SizeOfExpression :
02495 */
02496
02497 LONG SizeOfExpression(WORD num)
02498 {
02499     LONG x = (LONG)(DIVPOS(Expressions[num].size,sizeof(WORD)));
02500     if ( x >= 0 ) return(x);
02501     return(-1);
02502 }
02503
02504 /*
02505         #] SizeOfExpression :
02506         #[ UpdatePositions :
02507 */
02508
02509 void UpdatePositions(void)
02510 {
02511     EXPRESSIONS e = Expressions;
02512     POSITION *old;
02513     WORD *oldw;
02514     int i;
02515     if ( NumExpressions > 0 &&
02516          ( AS.OldOnFile == 0 || AS.NumOldOnFile < NumExpressions ) ) {
02517         if ( AS.OldOnFile ) {
02518             old = AS.OldOnFile;
02519             AS.OldOnFile = (POSITION *)Malloc1(NumExpressions*sizeof(POSITION),"file pointers");
```

```
02520             for ( i = 0; i < AS.NumOldOnFile; i++ ) AS.OldOnFile[i] = old[i];
02521             AS.NumOldOnFile = NumExpressions;
02522             M_free(old,"process file pointers");
02523         }
02524         else {
02525             AS.OldOnFile = (POSITION *)Malloc1(NumExpressions*sizeof(POSITION),"file pointers");
02526             AS.NumOldOnFile = NumExpressions;
02527         }
02528     }
02529     if ( NumExpressions > 0 &&
02530          ( AS.OldNumFactors == 0 || AS.NumOldNumFactors < NumExpressions ) ) {
02531         if ( AS.OldNumFactors ) {
02532             oldw = AS.OldNumFactors;
02533             AS.OldNumFactors = (WORD *)Malloc1(NumExpressions*sizeof(WORD),"numfactors pointers");
02534             for ( i = 0; i < AS.NumOldNumFactors; i++ ) AS.OldNumFactors[i] = oldw[i];
02535             M_free(oldw,"numfactors pointers");
02536             oldw = AS.Oldvflags;
02537             AS.Oldvflags = (WORD *)Malloc1(NumExpressions*sizeof(WORD),"vflags pointers");
02538             for ( i = 0; i < AS.NumOldNumFactors; i++ ) AS.Oldvflags[i] = oldw[i];
02539             M_free(oldw,"vflags pointers");
02540             oldw = AS.Olduflags;
02541             AS.Olduflags = (WORD *)Malloc1(NumExpressions*sizeof(WORD),"uflags pointers");
02542             for ( i = 0; i < AS.NumOldNumFactors; i++ ) AS.Olduflags[i] = oldw[i];
02543             M_free(oldw,"uflags pointers");
02544             AS.NumOldNumFactors = NumExpressions;
02545         }
02546         else {
02547             AS.OldNumFactors = (WORD *)Malloc1(NumExpressions*sizeof(WORD),"numfactors pointers");
02548             AS.Oldvflags = (WORD *)Malloc1(NumExpressions*sizeof(WORD),"vflags pointers");
02549             AS.Olduflags = (WORD *)Malloc1(NumExpressions*sizeof(WORD),"uflags pointers");
02550             AS.NumOldNumFactors = NumExpressions;
02551         }
02552     }
02553     for ( i = 0; i < NumExpressions; i++ ) {
02554         AS.OldOnFile[i] = e[i].onfile;
02555         AS.OldNumFactors[i] = e[i].numfactors;
02556         AS.Oldvflags[i] = e[i].vflags;
02557         AS.Olduflags[i] = e[i].uflags;
02558     }
02559 }
02560
02561 /*
02562         #] UpdatePositions :
02563         #[ CountTerms1 :        LONG CountTerms1()
02564
02565         Counts the terms in the current deferred bracket
02566         Is mainly an adaptation of the routine Deferred in proces.c
02567 */
02568
02569 LONG CountTerms1(PHEAD0)
02570 {
02571     GETBIDENTITY
02572     POSITION oldposition, startposition;
02573     WORD *t, *m, *mstop, decr, i, *oldwork, retval;
02574     WORD *oldipointer = AR.CompressPointer;
02575     WORD oldGetOneFile = AR.GetOneFile, olddeferflag = AR.DeferFlag;
02576     LONG numterms = 0;
02577     AR.GetOneFile = 1;
02578     oldwork = AT.WorkPointer;
02579     AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
02580     AR.DeferFlag = 0;
02581     startposition = AR.DefPosition;
02582 /*
02583         Store old position
02584 */
02585     if ( AR.infile->handle >= 0 ) {
02586         PUTZERO(oldposition);
02587 /*
02588         SeekFile(AR.infile->handle,&oldposition,SEEK_CUR);
02589 */
02590     }
02591     else {
02592         SETBASEPOSITION(oldposition,AR.infile->POfill-AR.infile->PObuffer);
02593         AR.infile->POfill = (WORD *)((UBYTE *)(AR.infile->PObuffer)
02594                     +BASEPOSITION(startposition));
02595     }
02596 /*
02597         Look in the CompressBuffer where the bracket contents start
02598 */
02599     t = m = AR.CompressBuffer;
02600     t += *t;
02601     mstop = t - ABS(t[-1]);
02602     m++;
02603     while ( *m != HAAKJE && m < mstop ) m += m[1];
02604     if ( m >= mstop ) { /* No deferred action! */
02605         numterms = 1;
02606         AR.DeferFlag = olddeferflag;
```

```
02607         AT.WorkPointer = oldwork;
02608         AR.GetOneFile = oldGetOneFile;
02609         return(numterms);
02610     }
02611     mstop = m + m[1];
02612     decr = WORDDIF(mstop,AR.CompressBuffer)-1;
02613
02614     m = AR.CompressBuffer;
02615     t = AR.CompressPointer;
02616     i = *m;
02617     NCOPY(t,m,i);
02618     AR.TePos = 0;
02619     AN.TeSuOut = 0;
02620 /*
02621         Status:
02622         First bracket content starts at mstop.
02623         Next term starts at startposition.
02624         Decompression information is in AR.CompressPointer.
02625         The outside of the bracket runs from AR.CompressBuffer+1 to mstop.
02626 */
02627     AR.CompressPointer = oldipointer;
02628     for(;;) {
02629         numterms++;
02630         retval = GetOneTerm(BHEAD AT.WorkPointer,AR.infile,&startposition,0);
02631         if ( retval >= 0 ) AR.CompressPointer = oldipointer;
02632         if ( retval <= 0 ) break;
02633         t = AR.CompressPointer;
02634         if ( *t < (1 + decr + ABS(*(t+*t-1))) ) break;
02635         t++;
02636         m = AR.CompressBuffer+1;
02637         while ( m < mstop ) {
02638             if ( *m != *t ) goto Thatsit;
02639             m++; t++;
02640         }
02641     }
02642 Thatsit:;
02643 /*
02644         Finished. Reposition the file, restore information and return.
02645 */
02646     AT.WorkPointer = oldwork;
02647     if ( AR.infile->handle >= 0 ) {
02648 /*
02649         SeekFile(AR.infile->handle,&oldposition,SEEK_SET);
02650 */
02651     }
02652     else {
02653         AR.infile->POfill = AR.infile->PObuffer + BASEPOSITION(oldposition);
02654     }
02655     AR.DeferFlag = olddeferflag;
02656     AR.GetOneFile = oldGetOneFile;
02657     return(numterms);
02658 }
02659
02660 /*
02661         #] CountTerms1 :
02662         #[ TermsInBracket :     LONG TermsInBracket(term,level)
02663
02664     The function TermsInBracket_()
02665     Syntax:
02666         TermsInBracket_() : The current bracket in a Keep Brackets
02667         TermsInBracket_(bracket) : This bracket in the current expression
02668         TermsInBracket_(expression,bracket) : This bracket in the given expression
02669     All other specifications don't have any effect.
02670 */
02671
02672 #define CURRENTBRACKET 1
02673 #define BRACKETCURRENTEXPR 2
02674 #define BRACKETOTHEREXPR 3
02675 #define NOBRACKETACTIVE 4
02676
02677 LONG TermsInBracket(PHEAD WORD *term, WORD level)
02678 {
02679     WORD *t, *tstop, *b, *tt, *n1, *n2;
02680     int type = 0, i, num;
02681     LONG numterms = 0;
02682     WORD *bracketbuffer = AT.WorkPointer;
02683     t = term; GETSTOP(t,tstop);
02684     t++; b = bracketbuffer;
02685     while ( t < tstop ) {
02686         if ( *t != TERMSINBRACKET ) { t += t[1]; continue; }
02687         if ( t[1] == FUNHEAD || (
02688              t[1] == FUNHEAD+2
02689             && t[FUNHEAD] == -SNUMBER
02690             && t[FUNHEAD+1] == 0
02691          ) ) {
02692             if ( AC.ComDefer == 0 ) {
02693                 type = NOBRACKETACTIVE;
```

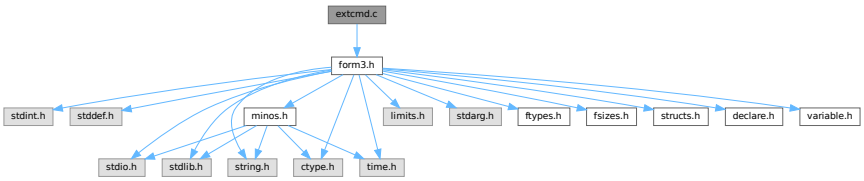
```
02694             }
02695             else {
02696                 type = CURRENTBRACKET;
02697             }
02698             *b = 0;
02699             break;
02700         }
02701         if ( t[FUNHEAD] == -EXPRESSION ) {
02702             if ( t[FUNHEAD+2] < 0 ) {
02703                 if ( ( t[FUNHEAD+2] <= -FUNCTION ) && ( t[1] == FUNHEAD+3 ) ) {
02704                     type = BRACKETOTHEREXPR;
02705                     *b++ = FUNHEAD+4; *b++ = -t[FUNHEAD+2]; *b++ = FUNHEAD;
02706                     for ( i = 2; i < FUNHEAD; i++ ) *b++ = 0;
02707                     *b++ = 1; *b++ = 1; *b++ = 3;
02708                     break;
02709                 }
02710                 else if ( ( t[FUNHEAD+2] > -FUNCTION ) && ( t[1] == FUNHEAD+4 ) ) {
02711                     type = BRACKETOTHEREXPR;
02712                     tt = t + FUNHEAD+2;
02713                     switch ( *tt ) {
02714                         case -SYMBOL:
02715                             *b++ = 8; *b++ = SYMBOL; *b++ = 4; *b++ = tt[1];
02716                             *b++ = 1; *b++ = 1; *b++ = 1; *b++ = 3;
02717                             break;
02718                         case -SNUMBER:
02719                             if ( tt[1] == 1 ) {
02720                                 *b++ = 4; *b++ = 1; *b++ = 1; *b++ = 3;
02721                             }
02722                             else goto IllBraReq;
02723                             break;
02724                         default:
02725                             goto IllBraReq;
02726                     }
02727                     break;
02728                 }
02729             }
02730             else if ( ( t[FUNHEAD+2] == (t[1]-FUNHEAD-2) ) &&
02731                       ( t[FUNHEAD+2+ARGHEAD] == (t[FUNHEAD+2]-ARGHEAD) ) ) {
02732                 type = BRACKETOTHEREXPR;
02733                 tt = t + FUNHEAD + ARGHEAD; num = *tt;
02734                 for ( i = 0; i < num; i++ ) *b++ = *tt++;
02735                 break;
02736             }
02737         }
02738         else {
02739             if ( t[FUNHEAD] < 0 ) {
02740                 if ( ( t[FUNHEAD] <= -FUNCTION ) && ( t[1] == FUNHEAD+1 ) ) {
02741                     type = BRACKETCURRENTEXPR;
02742                     *b++ = FUNHEAD+4; *b++ = -t[FUNHEAD+2]; *b++ = FUNHEAD;
02743                     for ( i = 2; i < FUNHEAD; i++ ) *b++ = 0;
02744                     *b++ = 1; *b++ = 1; *b++ = 3; *b = 0;
02745                     break;
02746                 }
02747                 else if ( ( t[FUNHEAD] > -FUNCTION ) && ( t[1] == FUNHEAD+2 ) ) {
02748                     type = BRACKETCURRENTEXPR;
02749                     tt = t + FUNHEAD+2;
02750                     switch ( *tt ) {
02751                         case -SYMBOL:
02752                             *b++ = 8; *b++ = SYMBOL; *b++ = 4; *b++ = tt[1];
02753                             *b++ = 1; *b++ = 1; *b++ = 1; *b++ = 3;
02754                             break;
02755                         case -SNUMBER:
02756                             if ( tt[1] == 1 ) {
02757                                 *b++ = 4; *b++ = 1; *b++ = 1; *b++ = 3;
02758                             }
02759                             else goto IllBraReq;
02760                             break;
02761                         default:
02762                             goto IllBraReq;
02763                     }
02764                     break;
02765                 }
02766             }
02767             else if ( ( t[FUNHEAD] == (t[1]-FUNHEAD) ) &&
02768                       ( t[FUNHEAD+ARGHEAD] == (t[FUNHEAD]-ARGHEAD) ) ) {
02769                 type = BRACKETCURRENTEXPR;
02770                 tt = t + FUNHEAD + ARGHEAD; num = *tt;
02771                 for ( i = 0; i < num; i++ ) *b++ = *tt++;
02772                 break;
02773             }
02774             else {
02775 IllBraReq:;
02776                 MLOCK(ErrorMessageLock);
02777                 MesPrint("Illegal bracket request in termsinbracket_ function.");
02778                 MUNLOCK(ErrorMessageLock);
02779                 Terminate(-1);
02780             }
```

```
02781         }
02782         t += t[1];
02783     }
02784     AT.WorkPointer = b;
02785     if ( AT.WorkPointer + *term +4 > AT.WorkTop ) {
02786         MLOCK(ErrorMessageLock);
02787         MesWork();
02788         MesPrint("Called from termsinbracket_ function.");
02789         MUNLOCK(ErrorMessageLock);
02790         return(-1);
02791     }
02792 /*
02793     We are now in the position to look for the bracket
02794 */
02795     switch ( type ) {
02796         case CURRENTBRACKET:
02797 /*
02798             The code here should be rather similar to when we pick up
02799             the contents of the bracket. In our case we only count the
02800             terms though.
02801 */
02802             numterms = CountTerms1(BHEAD0);
02803             break;
02804         case BRACKETCURRENTEXPR:
02805 /*
02806             Not implemented yet.
02807 */
02808             MLOCK(ErrorMessageLock);
02809             MesPrint("termsinbracket_ function currently only handles Keep Brackets.");
02810             MUNLOCK(ErrorMessageLock);
02811             return(-1);
02812         case BRACKETOTHEREXPR:
02813             MLOCK(ErrorMessageLock);
02814             MesPrint("termsinbracket_ function currently only handles Keep Brackets.");
02815             MUNLOCK(ErrorMessageLock);
02816             return(-1);
02817         case NOBRACKETACTIVE:
02818             numterms = 1;
02819             break;
02820     }
02821 /*
02822     Now we have the number in numterms. We replace the function by it.
02823 */
02824     n1 = term; n2 = AT.WorkPointer; tstop = n1 + *n1;
02825     while ( n1 < t ) *n2++ = *n1++;
02826     i = numterms » BITSINWORD;
02827     if ( i == 0 ) {
02828         *n2++ = LNUMBER; *n2++ = 4; *n2++ = 1; *n2++ = (WORD)(numterms & WORDMASK);
02829     }
02830     else {
02831         *n2++ = LNUMBER; *n2++ = 5; *n2++ = 2;
02832         *n2++ = (WORD)(numterms & WORDMASK); *n2++ = i;
02833     }
02834     n1 += n1[1];
02835     while ( n1 < tstop ) *n2++ = *n1++;
02836     AT.WorkPointer[0] = n2 - AT.WorkPointer;
02837     AT.WorkPointer = n2;
02838     if ( Generator(BHEAD n1,level) < 0 ) {
02839         AT.WorkPointer = bracketbuffer;
02840         MLOCK(ErrorMessageLock);
02841         MesPrint("Called from termsinbracket_ function.");
02842         MUNLOCK(ErrorMessageLock);
02843         return(-1);
02844     }
02845 /*
02846     Finished. Reset things and return.
02847 */
02848     AT.WorkPointer = bracketbuffer;
02849     return(numterms);
02850 }
02851 /*
02852         #] TermsInBracket :     LONG TermsInBracket(term,level)
02853     #] Expressions :
02854 */
```

4.33 extcmd.c File Reference

```
#include "form3.h"
```

Include dependency graph for extcmd.c:



Functions

- int openExternalChannel (UBYTE ∗cmd, int daemonize, UBYTE ∗shellname, UBYTE ∗stderrname)
- int initPresetExternalChannels (UBYTE ∗theline, int thetimeout)
- int closeExternalChannel (int n)
- int selectExternalChannel (int n)
- int getCurrentExternalChannel (void)
- void closeAllExternalChannels (void)

4.33.1 Detailed Description

The system that takes care of communication with external programs.

Definition in file extcmd.c.

4.33.2 Function Documentation

4.33.2.1 openExternalChannel()

```
int openExternalChannel (
            UBYTE * cmd,
            int daemonize,
            UBYTE * shellname,
            UBYTE * stderrname )
```

Definition at line 230 of file extcmd.c.

4.33.2.2 initPresetExternalChannels()

```
int initPresetExternalChannels (
            UBYTE * theline,
            int thetimeout )
```

Definition at line 232 of file extcmd.c.

4.33.2.3 closeExternalChannel()

```
int closeExternalChannel (
            int n )
```

Definition at line 233 of file extcmd.c.

4.33.2.4 selectExternalChannel()

```
int selectExternalChannel (
            int n )
```

Definition at line 234 of file extcmd.c.

4.33.2.5 getCurrentExternalChannel()

```
int getCurrentExternalChannel (
            void  )
```

Definition at line 235 of file extcmd.c.

4.33.2.6 closeAllExternalChannels()

```
void closeAllExternalChannels (
            void  )
```

Definition at line 236 of file extcmd.c.

4.34 extcmd.c

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
00026  *
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
00031 /*
00032     #[ Documentation :
```

```
00033
00034     This module is written by M.Tentyukov as a part of implementation of
00035     interaction between FORM and external processes, first release
00036     09.04.2004. A part of this code is copied from the DIANA project, which was
00037     written by M. Tentyukov and published under the GPL version 2 as
00038     published by the Free Software Foundation.
00039
00040     This file is completely re-written by M.Tentyukov in May 2006.
00041     Since the interface was changed, the public function were changed,
00042     also. A new public functions were added: initPresetExternalChannels()
00043     (see comments just before this function in the present file) and
00044     setKillModeForExternalChannel (a pointer, not a function).
00045
00046     If a macro WITHEXTERNALCHANNEL is not defined, all public functions
00047     are stubs returning failure.
00048
00049     The idea is to start an external command  swallowing
00050     its stdin and stdout. This can be done by means of the function
00051     int openExternalChannel(cmd,daemonize,shellname,stderrname), where
00052     cmd is a command to run,
00053     daemonize: if !=0 then start the command in the "daemon" mode,
00054     shellname: if !=NULL, execute the command in a subshell,
00055     stderrname: if != NULL, redirect stderr of the command to this file.
00056     The function returns some small positive integer number (the
00057     descriptor of a newly created external channel), or -1 on failure.
00058
00059     After the command is started, it becomes a _current_ opened external
00060     channel. The buffer can be sent to its stdin by a function
00061     int writeBufToExtChannel(buf, n)
00062     (here buf is a pointer to the buffer, n is the length in bytes; the
00063     function returns 0 in success, or -1 on failure),
00064     or one character can be read from its stdout by
00065     means of the function
00066     int getcFromExtChannel().
00067
00068     The latter returns the
00069     character casted to integer, or something <0. This can be -2 (if there
00070     is no current external channel) or EOF, if the external program closes
00071     its stdout, or if the external program outputs a string coinciding
00072     with a _terminator_.
00073
00074     By default, the terminator if an empty line. For the current external
00075     channel it can be set by means of the function
00076     int setTerminatorForExternalChannel(newterminator).
00077     The function returns 0 in success, or !0 if something is wrong (no
00078     current channel, too long terminator).
00079
00080     After getcFromExtChannel() returns EOF, the current channel becomes
00081     undefined. Any further attempts to read information by
00082     getcFromExtChannel() result in -2. To set (re-set) a current channel,
00083     the function
00084     int selectExternalChannel(n)
00085     can be used. This function accepts the valid external channel
00086     descriptor (returned by openExternalChannel) and returns the
00087     descriptor of a previous current channel (0, if there was no current
00088     channel, or -1, if the external channel descriptor is invalid).
00089     If n == 0, the function undefine the current external channel.
00090
00091     The function
00092     int closeExternalChannel(n)
00093     destroys the opened external channel with the descriptor n. It returns
00094     0 in success, or -1 on failure. If the corresponding external channel
00095     was the current one, the current channel becomes undefined. If n==0,
00096     the function closes the current external channel.
00097
00098     The function
00099     int getCurrentExternalChannel(void)
00100     returns the descriptor if the current external channel, or 0 , if
00101     there is no current external channel.
00102
00103     The function
00104     void closeAllExternalChannels(void)
00105
00106     destroys all opened external channels.
00107
00108     List of all public functions:
00109     int openExternalChannel(UBYTE *cmd,int daemonize,UBYTE *shellname, UBYTE * stderrname);
00110     int initPresetExternalChannels(UBYTE *theline, int thetimeout);
00111     int setTerminatorForExternalChannel(char *newterminator);
00112     int setKillModeForExternalChannel(int signum, int sentToWholeGroup);
00113     int closeExternalChannel(int n);
00114     int selectExternalChannel(int n);
00115     int writeBufToExtChannel(char *buf,int n);
00116     int getcFromExtChannel(void);
00117     int getCurrentExternalChannel(void);
00118     void closeAllExternalChannels(void);
00119
```

```
00120     ATTENTION!
00121
00122     Four of them:
00123     1 setTerminatorForExternalChannel
00124     2 setKillModeForExternalChannel
00125     3 writeBufToExtChannel
00126     4 getcFromExtChannel
00127
00128     are NOT functions, but variables (pointers) of a corrsponding type.
00129     They are initialised by proper values to avoid repeated error checking.
00130
00131     All public functions are independent of realization hidden in this module.
00132     All other functions may have a returned type/parameters type local w.r.t.
00133     this module; they are not declared outside of this file.
00134
00135     #] Documentation :
00136     #[ Selftest initializing:
00137 */
00138
00139 /*
00140 Uncomment to get a self-consistent program:
00141 #define SELFTEST 1
00142 */
00143
00144
00145 #ifdef SELFTEST
00146 #define WITHEXTERNALCHANNEL 1
00147 #ifdef _MSC_VER
00148 #define FORM_INLINE __inline
00149 #else
00150 #define FORM_INLINE inline
00151 #endif
00152
00153 /*
00154     From form3.h:
00155 */
00156 typedef unsigned char UBYTE;
00157
00158 /*The following variables should be defined in variable.h:*/
00159 extern int (*writeBufToExtChannel)(char *buffer, size_t n);
00160 extern int (*getcFromExtChannel)();
00161 extern int (*setTerminatorForExternalChannel)(char *buffer);
00162 extern int (*setKillModeForExternalChannel)(int signum, int sentToWholeGroup);
00163
00164 #else /*ifdef SELFTEST*/
00165 #include "form3.h"
00166 #endif /*ifdef SELFTEST ... else*/
00167 /*
00168 pid_t  getExternalChannelPid(void);
00169 */
00170 /*
00171     #] Selftest initializing:
00172     #[ Includes :
00173 */
00174 #ifdef WITHEXTERNALCHANNEL
00175 #include <stdio.h>
00176 #ifndef _MSC_VER
00177 #include <unistd.h>
00178 #endif
00179 #include <fcntl.h>
00180 #include <sys/types.h>
00181 #ifndef _MSC_VER
00182 #include <sys/time.h>
00183 #include <sys/wait.h>
00184 #endif
00185 #include <errno.h>
00186 #include <signal.h>
00187 #include <limits.h>
00188 /*
00189     #] Includes :
00190     #[ FailureFunctions:
00191 */
00192
00193 /*Non-initialized variant of public functions:*/
00194 int writeBufToExtChannelFailure(char *buf, size_t count)
00195 {
00196     DUMMYUSE(buf); DUMMYUSE(count);
00197     return(-1);
00198 }/*writeBufToExtChannelFailure*/
00199
00200 int setTerminatorForExternalChannelFailure(char *newTerminator)
00201 {
00202     DUMMYUSE(newTerminator);
00203     return(-1);
00204 }/*setTerminatorForExternalChannelFailure*/
00205
00206 int setKillModeForExternalChannelFailure(int signum, int sentToWholeGroup)
```

```
00207 {
00208     DUMMYUSE(signum); DUMMYUSE(sentToWholeGroup);
00209     return(-1);
00210 }/*setKillModeForExternalChannelFailure*/
00211
00212 int getcFromExtChannelFailure(void)
00213 {
00214     return(-2);
00215 }/*getcFromExtChannelFailure*/
00216
00217 int (*writeBufToExtChannel)(char *buffer, size_t n) = &writeBufToExtChannelFailure;
00218 int (*setTerminatorForExternalChannel)(char *buffer) =
00219     &setTerminatorForExternalChannelFailure;
00220 int (*setKillModeForExternalChannel)(int signum, int sentToWholeGroup) =
00221     &setKillModeForExternalChannelFailure;
00222 int (*getcFromExtChannel)(void) = &getcFromExtChannelFailure;
00223 #endif
00224 /*
00225     #] FailureFunctions:
00226     #[ Stubs :
00227 */
00228 #ifndef WITHEXTERNALCHANNEL
00229 /*Stubs for public functions:*/
00230 int openExternalChannel(UBYTE *cmd, int daemonize, UBYTE *shellname, UBYTE *stderrname)
00231 { DUMMYUSE(cmd); DUMMYUSE(daemonize); DUMMYUSE(shellname); DUMMYUSE(stderrname); return(-1); };
00232 int initPresetExternalChannels(UBYTE *theline, int thetimeout) { DUMMYUSE(theline);
      DUMMYUSE(thetimeout); return(-1); };
00233 int closeExternalChannel(int n) { DUMMYUSE(n); return(-1); };
00234 int selectExternalChannel(int n) { DUMMYUSE(n); return(-1); };
00235 int getCurrentExternalChannel(void) { return(0); };
00236 void closeAllExternalChannels(void) {};
00237 #else /*ifndef WITHEXTERNALCHANNEL*/
00238 /*
00239     #] Stubs :
00240     #[ Local types :
00241 */
00242 /*First argument for the function signal:*/
00243 #ifndef INTSIGHANDLER
00244 typedef void (*mysighandler_t)(int);
00245 #else
00246 /* Sometimes, this nonsense may occurs:*/
00247 /*typedef int (*mysighandler_t)(int);*/
00248 #endif
00249
00250 /*Input IO buffer size increment -- each time the buffer
00251 is expired it will be increased by this value (in bytes):*/
00252 #define DELTA_EXT_BUF 128
00253
00254 /*Re-allocatable array containing External Channel
00255 handlers increased each time by this value:*/
00256 #define DELTA_EXT_LIST 8
00257
00258 /*How many times I/O routines may attempt to continue their work
00259 in some failures:*/
00260 #define MAX_FAILS_IO 2
00261
00262 /*The external channel handler structure:*/
00263 typedef struct ExternalChannel {
00264     pid_t pid;       /*PID of the external process*/
00265     pid_t gpid;       /*process  group ID of the external process.
00266                         If <=0, not used, if >0, the kill signals
00267                         is sent to the whole group */
00268     FILE *frec;      /*stdout of the external process*/
00269     char *INbuf;     /*External channel buffer*/
00270     char *IBfill;    /*Position in INbuf from which the next input character will be read*/
00271     char *IBfull;    /*End of read INbuf*/
00272     char *IBstop;    /*end of allocated space for INbuf*/
00273     char *terminator;/* Terminator - when extern. program outputs ONLY this string,
00274                         it is assumed that the answer is ready, and getcFromExtChannel
00275                         returns EOF. Should not be longer then the minimal buffer!*/
00276     /*Info fields, not changable after creating a channel:*/
00277     char *cmd;       /*the command*/
00278     char *shellname;
00279     char *stderrname;/*filename to redirect stderr, or NULL*/
00280     int fsend;       /*stdin of the external process*/
00281     int killSignal; /*signal to kill*/
00282     int daemonize;/*0 --neither setsid nor daemonize, !=0 -- full daemonization*/
00283 } EXTHANDLE;
00284
00285
00286 static EXTHANDLE *externalChannelsList=0;
00287 /*Here integers are better than pointers: */
00288 static int externalChannelsListStop=0;
00289 static int externalChannelsListFill=0;
00290
00291 /*"current" external channel:*/
00292 static EXTHANDLE *externalChannelsListTop=0;
```

```
00293 /*
00294     #] Local types :
00295     #[ Selftest functions :
00296 */
00297 #ifdef SELFTEST
00298
00299 /*For malloc prototype:*/
00300 #include <stdlib.h>
00301
00302 /*StrLen, Malloc1, M_free and strDup1 are defined in tools.c -- here only emulation:*/
00303 int StrLen(char *pattern)
00304 {
00305 register  char *p=(char*)pattern;
00306      while(*p)p++;
00307      return((int) ((p-(char*)pattern)) );
00308 }/*StrLen*/
00309 void *Malloc1(int l, char *c)
00310 {
00311     return(malloc(l));
00312 }
00313 void M_free(void *p,char *c)
00314 {
00315     return(free(p));
00316 }
00317
00318 char *strDup1(UBYTE *instring, char *ifwrong)
00319 {
00320     UBYTE *s = instring, *to;
00321     while ( *s ) s++;
00322     to = s = (UBYTE *)Malloc1((s-instring)+1,ifwrong);
00323     while ( *instring ) *to++ = *instring++;
00324     *to = 0;
00325     return(s);
00326 }
00327
00328 /*PutPreVar from pre.c -- just ths stub:*/
00329 int PutPreVar(UBYTE *a,UBYTE *b,UBYTE *c,int i)
00330 {
00331     return(0);
00332 }
00333
00334 #endif
00335 /*
00336     #] Selftest functions :
00337     #[ Local functions :
00338 */
00339
00340 /*Initialize one cell of handler:*/
00341 static FORM_INLINE void extHandlerInit(EXTHANDLE *h)
00342 {
00343     h->pid=-1;
00344     h->gpid=-1;
00345     h->fsend=0;
00346     h->killSignal=SIGKILL;
00347     h->daemonize=1;
00348     h->frec=NULL;
00349     h->INbuf=h->IBfill=h->IBfull=h->IBstop=
00350     h->terminator=h->cmd=h->shellname=h->stderrname=NULL;
00351 }/*extHandlerInit*/
00352
00353 /* Copies each field of handler:*/
00354 static FORM_INLINE void extHandlerSwallowCopy(EXTHANDLE *to, EXTHANDLE *from)
00355 {
00356     to->pid=from->pid;
00357     to->gpid=from->gpid;
00358     to->fsend=from->fsend;
00359     to->killSignal=from->killSignal;
00360     to->daemonize=from->daemonize;
00361     to->frec=from->frec;
00362     to->INbuf=from->INbuf;
00363     to->IBfill=from->IBfill;
00364     to->IBfull=from->IBfull;
00365     to->IBstop=from->IBstop;
00366     to->terminator=from->terminator;
00367     to->cmd=from->cmd;
00368     to->shellname=from->shellname;
00369     to->stderrname=from->stderrname;
00370 }/*extHandlerSwallow*/
00371
00372 /*Allocates memory for fields of handler which have no fixed
00373  storage size and initializes some fields:*/
00374 static FORM_INLINE void
00375 extHandlerAlloc(EXTHANDLE *h, char *cmd, char *shellname, char *stderrname)
00376 {
00377     h->IBfill=h->IBfull=h->INbuf=
00378                 Malloc1(DELTA_EXT_BUF,"External channel buffer");
00379     h->IBstop=h->INbuf+DELTA_EXT_BUF;
```

```
00380     /*Initialize a terminator:*/
00381     *(h->terminator=Malloc1(DELTA_EXT_BUF,"External channel terminator"))='\n';
00382     (h->terminator)[1]='\0';/*By default the terminator is '\n'*/
00383     /*Deep copy all strings:*/
00384     if(cmd!=NULL)
00385         h->cmd=(char *)strDup1((UBYTE *)cmd,"External channel command");
00386     else/*cmd cannot be NULL! If this is NULL then force it to be something special*/
00387         h->cmd=(char *)strDup1((UBYTE *)"/","External channel command");
00388     if(shellname!=NULL)
00389         h->shellname=
00390               (char *)strDup1((UBYTE *)shellname,"External channel shell name");
00391     if(stderrname!=NULL)
00392         h->stderrname=
00393               (char *)strDup1((UBYTE *)stderrname,"External channel stderr name");
00394 }/*extHandlerAlloc*/
00395
00396 /*Disallocates dynamically allocated fields of a handler:*/
00397 static FORM_INLINE void extHandlerFree(EXTHANDLE *h)
00398 {
00399     if(h->stderrname) M_free(h->stderrname,"External channel stderr name");
00400     if(h->shellname) M_free(h->shellname,"External channel shell name");
00401     if(h->cmd) M_free(h->cmd,"External channel command");
00402     if(h->terminator)M_free(h->terminator,"External channel terminator");
00403     if(h->INbuf)M_free(h->INbuf,"External channel buffer");
00404     extHandlerInit(h);
00405 }/*extHandlerFree*/
00406 /* Closes all descriptors, kills the external process, frees all internal fields,
00407 BUT does NOT free the main container:*/
00408 static void destroyExternalChannel(EXTHANDLE *h)
00409 {
00410     /*Note, this function works in parallel mode correctly, see comments below.*/
00411
00412     /*Note, for slaves in a parallel mode h->pid == 0:*/
00413     if( (h->pid > 0) && (h->killSignal > 0) ){
00414         int chstatus;
00415         if( h->gpid > 0)
00416             chstatus=kill(-h->gpid,h->killSignal);
00417         else
00418             chstatus=kill(h->pid,h->killSignal);
00419         if(chstatus==0)
00420         /*If the process will not be killed by this signal, FORM hangs up here!:*/
00421             waitpid(h->pid, &chstatus, 0);
00422     }/*if( (h->pid > 0) && (h->killSignal > 0) )*/
00423
00424     /*Note, for slaves in a parallel mode h->frec == h->fsend == 0:*/
00425     if(h->frec) fclose(h->frec);
00426     if( h->fsend > 0) close(h->fsend);
00427
00428     extHandlerFree(h);
00429     /*Does not do "free(h)"!*/
00430 }/*destroyExternalChannel*/
00431
00432 /*Wrapper to the read() syscall, to handle possible interrupts by unblocked signals:*/
00433 static FORM_INLINE ssize_t read2b(int fd, char *buf, size_t count)
00434 {
00435 ssize_t res;
00436
00437     if( (res=read(fd,buf,count)) <1 )/*EOF or read is interrupted by a signal?:*/
00438          while( (errno == EINTR)&&(res <1) )
00439              /*The call was interrupted by  a  signal  before  any data was read, try again:*/
00440              res=read(fd,buf,count);
00441     return (res);
00442 }/*read2b*/
00443
00444 /*Wrapper to the write() syscall, to handle possible interrupts by unblocked signals:*/
00445 static FORM_INLINE ssize_t writeFromb(int fd, char *buf, size_t count)
00446 {
00447 ssize_t res;
00448     if( (res=write(fd,buf,count)) <1 )/*Is write interrupted by a signal?:*/
00449          while( (errno == EINTR)&&(res <1) )
00450              /*The call was interrupted by a signal before any data was written, try again:*/
00451              res=write(fd,buf,count);
00452     return (res);
00453 }/*writeFromb*/
00454
00455 /* Read one (binary) PID from the file descriptor fd:*/
00456 static FORM_INLINE pid_t readpid(int fd)
00457 {
00458 pid_t tmp;
00459     if(read2b(fd,(char*)&tmp,sizeof(pid_t))!=sizeof(pid_t))
00460       return (pid_t)-1;
00461     return tmp;
00462 }/*readpid*/
00463
00464 /* Writeone (binary) PID to the file descriptor fd:*/
00465 static FORM_INLINE pid_t writepid(int fd, pid_t thepid)
00466 {
```

```
00467     if(writeFromb(fd,(char*)&thepid,sizeof(pid_t))!=sizeof(pid_t))
00468       return (pid_t)-1;
00469     return (pid_t)0;
00470 }/*readpid*/
00471
00472 /*Wrtites exactly count bytes from the  buffer buf into the descriptor fd, independently on
00473   nonblocked signals and the MPU/buffer hits. Returns 0 or -1:
00474 */
00475 static FORM_INLINE int writexactly(int fd, char *buf, size_t count)
00476 {
00477 ssize_t i;
00478 int j=0,n=0;
00479
00480     for(;;){
00481         if(  (i=writeFromb(fd, buf+j, count-j)) < 0 ) return(-1);
00482         j+=i;
00483         if ( ((size_t)j) == count ) break;
00484         if(i==0)n++;
00485         else n=0;
00486         if(n>MAX_FAILS_IO)return (-1);
00487     }/*for(;;)*/
00488     return (0);
00489 }/*writexactly*/
00490
00491 /* Set the FD_CLOEXEC  flag of desc if value is nonzero,
00492  or clear the flag if value is 0.
00493  Return 0 on success, or -1 on error with errno  set. */
00494 static int set_cloexec_flag(int desc, int value)
00495 {
00496 int oldflags = fcntl (desc, F_GETFD, 0);
00497     /* If reading the flags failed, return error indication now.*/
00498     if (oldflags < 0)
00499         return (oldflags);
00500     /* Set just the flag we want to set. */
00501     if (value != 0)
00502         oldflags |= FD_CLOEXEC;
00503     else
00504       oldflags &= ~FD_CLOEXEC;
00505     /* Store modified flag word in the descriptor. */
00506     return (fcntl(desc, F_SETFD, oldflags));
00507 }/*set_cloexec_flag*/
00508
00509 /* Adds the integer fd to the array fifo of length top+1 so that
00510 the array is ascendantly ordered. It is supposed that all 0 -- top-1
00511 elements in the array are already ordered:*/
00512 static void pushDescriptor(int *fifo, int top, int fd)
00513 {
00514     if ( top == 0 ) {
00515         fifo[top] = fd;
00516     } else {
00517         int ins=top-1;
00518         if( fifo[ins]<=fd )
00519             fifo[top]=fd;
00520         else{
00521             /*Find the position:*/
00522             while( (ins>=0)&&(fifo[ins]>fd) )ins--;
00523             /*Move all elements starting from the position to the right:*/
00524             for(ins++;top>ins; top--)
00525                 fifo[top]=fifo[top-1];
00526             /*Put the element:*/
00527             fifo[ins]=fd;
00528         }
00529     }
00530 }/*pushDescriptor*/
00531
00532 /*Close all descriptors greater or equal than startFrom except those
00533   listed in the ascendantly ordered array usedFd of length top:*/
00534 static FORM_INLINE void closeAllDescriptors(int startFrom, int *usedFd, int top)
00535 {
00536 int n,maxfd;
00537     for(n=0;n<top; n++){
00538         maxfd=usedFd[n];
00539         for(;startFrom<maxfd;startFrom++)/*Close all less than maxfd*/
00540             close(startFrom);
00541         startFrom++;/*skip maxfd*/
00542     }/*for(;startFrom<maxfd;startFrom++)*/
00543     /*Close all the rest:*/
00544     maxfd=sysconf(_SC_OPEN_MAX);
00545     for(;startFrom<maxfd;startFrom++)
00546         close(startFrom);
00547 }/*closeAllDescriptors*/
00548
00549 typedef int L_APIPE[2];
00550 /*Closes both pipe descriptors if not -1:*/
00551 static void closepipe(L_APIPE *thepipe)
00552 {
00553   if( (*thepipe)[0] != -1) close ((*thepipe)[0]);
```

```
00554   if( (*thepipe)[1] != -1) close ((*thepipe)[1]);
00555 }/*closepipe*/
00556
00557 /*Parses the cmd line like "sh -c myprg", passes each option to the
00558   corresponding element of argv, ends agrv by NULL. Returns the
00559   number of stored argv elements, or -1 if fails:*/
00560 static FORM_INLINE int parseline(char **argv, char *cmd)
00561 {
00562 int n=0;
00563     while(*cmd != '\0'){
00564         for(; (*cmd <= ' ') && (*cmd != '\0') ;cmd++);
00565         if(*cmd != '\0'){
00566             argv[n]=cmd;
00567             while(*++cmd > ' ');
00568             if(*cmd != '\0')
00569                 *cmd++ = '\0';
00570             n++;
00571         }/*if(*cmd != '\0')*/
00572     }/*while(*cmd != '\0')*/
00573     argv[n]=NULL;
00574     if(n==0)return -1;
00575     return n;
00576 }/*parseline*/
00577
00578 /*Reads positive decimal number (not bigger than maxnum)
00579   from the string and returns it;
00580   the pointer *b is set to the next non-converted character:*/
00581 static LONG str2i(char *str, char **b, LONG maxnum)
00582 {
00583     LONG n=0;
00584     /*Eat trailing spaces:*/
00585     while(*str<=' ')if(*str++ == '\0')return(-1);
00586     (*b)=str;
00587     while (*str>='0'&&*str<='9')
00588         if( (n=10*n + *str++ - '0')>maxnum )
00589             return(-1);
00590     if((*b)==str)/*No single number!*/
00591         return(-1);
00592     (*b)=str;
00593     return(n);
00594 }
00595
00596 /*Converts long integer to a decimal representation.
00597   For portability reasons we cannot use LongCopy from tools.c
00598   since theoretically LONG may be smaller than pid_t:*/
00599 static char *l2s(LONG x, char *to)
00600 {
00601     char *s;
00602     int i = 0, j;
00603     s = to;
00604     do { *s++ = (x % 10)+'0'; i++; } while ( ( x /= 10 ) != 0 );
00605     *s-- = '\0';
00606     j = ( i - 1 ) » 1;
00607     while ( j >= 0 ) {
00608         i = to[j]; to[j] = s[-j]; s[-j] = (char)i; j--;
00609     }
00610     return(s+1);
00611 }
00612
00613 /*like strcat() but returns the pointer to the end of the
00614 resulting string:*/
00615 static  FORM_INLINE char *addStr(char *to, char *from)
00616 {
00617     while(  (*to++ = *from++)!='\0' );
00618     return(to-1);
00619 }/*addStr*/
00620
00621
00622 /*Try to write (atomically) short buffer (of length count) to fd.
00623   timeout is a timeout in millisecs. Returns number of written bytes or -1:*/
00624 static FORM_INLINE ssize_t writeSome(int fd, char *buf, size_t count, int timeout)
00625 {
00626     ssize_t res = 0;
00627     fd_set wfds;
00628     struct timeval tv;
00629     int nrep=5;/*five attempts it interrupted by a non-blocking signal*/
00630     int flags = fcntl(fd, F_GETFL,0);
00631
00632     /*Add O_NONBLOCK:*/
00633     fcntl(fd,F_SETFL, flags | O_NONBLOCK);
00634     /* important -- in order to avoid blocking of short receiver buffer*/
00635
00636     do{
00637         FD_ZERO(&wfds);
00638         FD_SET(fd, &wfds);
00639         /* Wait up to timeout. */
00640
```

```
00641         tv.tv_sec =timeout /1000;
00642         tv.tv_usec = (timeout % 1000)*1000;
00643         nrep--;
00644 
00645         switch(select(fd+1, NULL, &wfds, NULL, &tv)){
00646             case -1:
00647                 if((nrep == 0)||( errno != EINTR) ){
00648                     perror("select()");
00649                     res=-1;
00650                     nrep=0;
00651                 }/*else -- A non blocked signal was caught, just repeat*/
00652                 break;
00653             case 0:/*timeout*/
00654                 res=-1;
00655                 nrep=0;
00656                 break;
00657             default:
00658                 if( (res=write(fd,buf,count)) <0 )/*Signal?*/
00659                     while( (errno == EINTR)&&(res <0) )
00660                       res=write(fd,buf,count);
00661                 nrep=0;
00662         }/*switch*/
00663     }while(nrep);
00664 
00665     /*restore the flags:*/
00666     fcntl(fd,F_SETFL, flags);
00667     return (res);
00668 }/*writeSome*/
00669 
00670 /*Try to read short buffer (of length not more than count)
00671   from fd. timeout is a timeout in millisecs. Returns number
00672   of written bytes or -1: */
00673 static FORM_INLINE ssize_t readSome(int fd, char *buf, size_t count, int timeout)
00674 {
00675     ssize_t res = 0;
00676     fd_set rfds;
00677     struct timeval tv;
00678     int nrep=5;/*five attempts it interrupted by a non-blocking signal*/
00679 
00680     do{
00681         FD_ZERO(&rfds);
00682         FD_SET(fd, &rfds);
00683         /* Wait up to timeout. */
00684 
00685         tv.tv_sec = timeout/1000;
00686         tv.tv_usec = (timeout % 1000)*1000;
00687         nrep--;
00688 
00689         switch(select(fd+1, &rfds, NULL, NULL, &tv)){
00690             case -1:
00691                 if((nrep == 0)||( errno != EINTR) ){
00692                     perror("select()");
00693                     res=-1;
00694                     nrep=0;
00695                 }/*else -- A non blocked signal was caught, just repeat*/
00696                 break;
00697             case 0:/*timeout*/
00698                 res=-1;
00699                 nrep=0;
00700                 break;
00701             default:
00702                 if( (res=read(fd,buf,count)) <0 )/*Signal?*/
00703                     while( (errno == EINTR)&&(res <0) )
00704                       res=read(fd,buf,count);
00705                 nrep=0;
00706         }/*switch*/
00707     }while(nrep);
00708     return (res);
00709 }/*readSome*/
00710 
00711 /*
00712     #] Local functions :
00713     #[ Ok functions:
00714 */
00715 
00716 /*Copies (deep copy) newTerminator to thehandler->terminator. Returns 0 if
00717   newTerminator fits to the buffer, or !0 if it does not fit. ATT! In the
00718   latter case thehandler->terminator is NOT '\0' terminated! */
00719 int setTerminatorForExternalChannelOk(char *newTerminator)
00720 {
00721 int i=DELTA_EXT_BUF;
00722 /*
00723     No problems with externalChannelsListTop are
00724     possible since this function may be invoked only
00725     when the current channel is defined and externalChannelsListTop
00726   is set properly
00727 */
```

```
00728 char *t=externalChannelsListTop->terminator;
00729
00730     for(; i>1; i--)
00731         if( (*t++ = *newTerminator++)=='\0' )
00732             break;
00733     /*Add trailing '\n', if absent:*/
00734     if(  (i == DELTA_EXT_BUF)/*newTerminator == '\0'*/
00735         ||(*(t-2)!='\n')       ){
00736       *(t-1)='\n';*t='\0';
00737     }
00738     return(i==1);
00739 }/*setTerminatorForExternalChannelOk*/
00740
00741 /*Interface to change handler fields "killSignal" and "gpid"*/
00742 int setKillModeForExternalChannelOk(int signum, int sentToWholeGroup)
00743 {
00744     if(signum<0)
00745         return(-1);
00746     /*
00747         No problems with externalChannelsListTop are
00748         possible since this function may be invoked only
00749         when the current channel is defined and externalChannelsListTop
00750         is set properly
00751     */
00752     externalChannelsListTop->killSignal=signum;
00753     if(sentToWholeGroup){/*gpid must be >0*/
00754         if(externalChannelsListTop->gpid <= 0)
00755             externalChannelsListTop->gpid=-externalChannelsListTop->gpid;
00756     }else{/*gpid must be <=0*/
00757         if(externalChannelsListTop->gpid>0)
00758             externalChannelsListTop->gpid=-externalChannelsListTop->gpid;
00759     }
00760     return(0);
00761 }/*setKillModeForExternalChannelOk*/
00762
00763 /*
00764         #[ getcFromExtChannelOk
00765 */
00766 /*Returns one character from the external channel. It the input is expired,
00767 returns EOF. If the external process is finished completely, the function closes
00768 the channel (and returns EOF). If the external process was finished, the function
00769 returns EOF:*/
00770 int getcFromExtChannelOk(void)
00771 {
00772 mysighandler_t oldPIPE = 0;
00773 EXTHANDLE *h;
00774 int ret;
00775
00776     if (externalChannelsListTop->IBfill < externalChannelsListTop->IBfull)
00777         /*in buffer*/
00778         return( *(externalChannelsListTop->IBfill++) );
00779     /*else -- the buffer is empty*/
00780     ret=EOF;
00781     h= externalChannelsListTop;
00782 #ifdef WITHMPI
00783     if ( PF.me == MASTER ){
00784 #endif
00785     /* Temporary ignore this signal:*/
00786     /* if compiler fails here, try to change the definition of
00787         mysighandler_t on the beginning of this file
00788         (just define INTSIGHANDLER).*/
00789     oldPIPE=signal(SIGPIPE,SIG_IGN);
00790 #ifdef WITHMPI
00791     if(  fgets(h->INbuf,h->IBstop - h->INbuf, h->frec) == 0  )/*Fail! EOF?*/
00792         *(h->INbuf)='\0';/*Empty line may not appear here!*/
00793 #else
00794     if(  (fgets(h->INbuf,h->IBstop - h->INbuf, h->frec) == 0)/*Fail! EOF?*/
00795         ||( *(h->INbuf) == '\0')/*Empty line? This shouldn't be!*/
00796       ){
00797         closeExternalChannel(externalChannelsListTop-externalChannelsList+1);
00798         /*Note, this code is only for the sequential mode! */
00799         goto getcFromExtChannelReady;
00800         /*Here we assume that fgets is never interrupted by singals*/
00801     }/*if( fgets(h->INbuf,h->IBstop - h->INbuf, h->frec) == 0 )*/
00802 #endif
00803 #ifdef WITHMPI
00804     }/*if ( PF.me == MASTER */
00805
00806     /*Master broadcasts result to slaves, slaves read it from the master:*/
00807     if( PF_BroadcastString((UBYTE *)h->INbuf) ){/*Fail!*/
00808           MesPrint("Fail broadcasting external channel results");
00809           Terminate(-1);
00810     }/*if( PF_BroadcastString((UBYTE *)h->INbuf) )*/
00811
00812     if( *(h->INbuf) == '\0'){/*Empty line? This shouldn't be!*/
00813         closeExternalChannel(externalChannelsListTop-externalChannelsList+1);
00814         goto getcFromExtChannelReady;
```

```
00815     }/*if( *(h->INbuf) == '\0')*/
00816 #endif
00817
00818     {/*Block*/
00819         char *t=h->terminator;
00820         /*Move IBfull to the end of read line and compare the line with the terminator.
00821           Note, by construction the terminator fits to the first read line, see
00822           the function setTerminatorForExternalChannel.*/
00823         for(h->IBfull=h->INbuf; *(h->IBfull)!='\0'; (h->IBfull)++)
00824             if( *t== *(h->IBfull) )
00825                 t++;
00826             else
00827                 break;/*not a terminator*/
00828         /*Continue moving IBfullto the end of read line:*/
00829         while(*(h->IBfull)!='\0')(h->IBfull)++;
00830
00831         if( (t-h->terminator) == (h->IBfull-h->INbuf) ){
00832             /*Terminator!*/
00833             /*Reset the channel*/
00834             h->IBfull=h->IBfill=h->INbuf;
00835             externalChannelsListTop=0;/*Undefine the current channel*/
00836             writeBufToExtChannel=&writeBufToExtChannelFailure;
00837             getcFromExtChannel=&getcFromExtChannelFailure;
00838             setTerminatorForExternalChannel=&setTerminatorForExternalChannelFailure;
00839             setKillModeForExternalChannel=&setKillModeForExternalChannelFailure;
00840             goto getcFromExtChannelReady;
00841         }/*if(t == (h->IBfull-h->INbuf) )*/
00842     }/*Block*/
00843     /*Does the buffer have enough capacity?*/
00844     while( *(h->IBfull - 1) != '\n' ){/*Buffer is not enough!*/
00845         /*Extend the buffer:*/
00846         int l= (h->IBstop - h->INbuf)+DELTA_EXT_BUF;
00847         char *newbuf=Malloc1(l,"External channel buffer");
00848         /*We wouldn't like to use realloc.*/
00849         /*Copy the buffer:*/
00850         char *n=newbuf,*o=h->INbuf;
00851         while(  (*n++ = *o++)!='\0' );
00852         /*Att! The order of the following operators is important!:*/
00853         h->IBfull= newbuf+(h->IBfull-h->INbuf);
00854         M_free(h->INbuf,"External channel buffer");
00855         h->INbuf = newbuf;
00856         h->IBstop = h->INbuf+l;
00857 #ifdef WITHMPI
00858         if ( PF.me == MASTER ){
00859             (h->IBfull)[1]='\0';/*Will mark (h->IBfull)[1] as '!' for failure*/
00860             if(  fgets(h->IBfull,h->IBstop - h->IBfull, h->frec) == 0 ){
00861                 /*EOF! No trailing '\n'?*/
00862                 /*Mark:*/
00863                 (h->IBfull)[0]='\0';
00864                 (h->IBfull)[1]='!';
00865                 (h->IBfull)[2]='\0';
00866                  /*The string "\0!\0" is used as an image of NULL.*/
00867             }/*if(  fgets(h->IBfull,h->IBstop - h->IBfull, h->frec) == 0 )*/
00868         }/*if ( PF.me == MASTER )*/
00869
00870         /*Master broadcasts results to slaves, slaves read it from the master:*/
00871         if( PF_BroadcastString((UBYTE *)h->IBfull) ){/*Fail!*/
00872           MesPrint("Fail broadcasting external channel results");
00873           Terminate(-1);
00874         }/*if( PF_BroadcastString(h->IBfull) )*/
00875
00876         /*The string "\0!\0" is used as the image of NULL.*/
00877         if(
00878               ( (h->IBfull)[0]=='\0' )
00879             &&(  (h->IBfull)[1]=='!' )
00880             &&(  (h->IBfull)[2]=='\0' )
00881           )/*EOF! No trailing '\n'?*/
00882             break;
00883 #else
00884         if(  fgets(h->IBfull,h->IBstop - h->IBfull, h->frec) == 0 )
00885             /*EOF! No trailing '\n'?*/
00886             break;
00887 #endif
00888         while( *(h->IBfull)!='\0' )(h->IBfull)++;
00889     }/*while( *(h->IBfull - 1) != '\n' )*/
00890     /*In h->INbuf we have a fresh string.*/
00891     ret=*(h->IBfill=h->INbuf);
00892     h->IBfill++;/*Next time a new, isn't it?*/
00893
00894     getcFromExtChannelReady:
00895 #ifdef WITHMPI
00896     if ( PF.me == MASTER ){
00897 #endif
00898             signal(SIGPIPE,oldPIPE);
00899 #ifdef WITHMPI
00900     }/*if ( PF.me == MASTER )*/
00901 #endif
```

```
00902             return(ret);
00903 }/*getcFromExtChannelOk*/
00904 /*
00905         #] getcFromExtChannelOk
00906 */
00907 /*Writes exactly count bytes from the buffer buf to the external channel thehandler
00908   Returns 0 (on success) or -1:
00909 */
00910 int writeBufToExtChannelOk(char *buf, size_t count)
00911 {
00912
00913 int ret;
00914 mysighandler_t oldPIPE;
00915 #ifdef WITHMPI
00916     /*Only master communicates with the external program:*/
00917     if ( PF.me == MASTER ){
00918 #endif
00919         /* Temporary ignore this signal:*/
00920         /* if compiler fails here, try to change the definition of
00921             mysighandler_t on the beginning of this file
00922             (just define INTSIGHANDLER)*/
00923         oldPIPE=signal(SIGPIPE,SIG_IGN);
00924         ret=writexactly( externalChannelsListTop->fsend, buf, count);
00925         signal(SIGPIPE,oldPIPE);
00926 #ifdef WITHMPI
00927     }else{
00928         /*Do not wait the master status: this would be too slow!*/
00929         ret=0;
00930     }
00931 #endif
00932     return(ret);
00933 }/*writeBufToExtChannel*/
00934 /*
00935     #] Ok functions:
00936     #[ do_run_cmd :
00937 */
00938 /*The function returns PID of the started command*/
00939 static FORM_INLINE pid_t do_run_cmd(
00940     int *fdsend,
00941     int *fdreceive,
00942     int *gpid, /*returns group process ID*/
00943     int ttymode,
00944                     /*
00945                      &8 - daemonizeing
00946                      &16 - setsid()*/
00947     char *cmd,
00948     char *argv[],
00949     char *stderrname
00950     )
00951 {
00952 int fdin[2]={-1,-1}, fdout[2]={-1,-1}, fdsig[2]={-1,-1};
00953 /*initialised by -1 for possible rollback at failure, see closepipe() above*/
00954
00955 pid_t childpid,fatherchildpid = (pid_t)0;
00956 mysighandler_t oldPIPE=NULL;
00957
00958     if(
00959         (pipe(fdsig)!=0)/*This pipe will be used by a child to tell the father if fail.*/
00960         ||(pipe(fdin)!=0)
00961         ||(pipe(fdout)!=0)
00962     )goto fail_do_run_cmd;
00963
00964     if((childpid = fork()) == -1){
00965          perror("fork");
00966          goto fail_do_run_cmd;
00967     }/*if((childpid = fork()) == -1)*/
00968
00969     if(childpid == 0){/*Child.*/
00970         int fifo[3], top=0;
00971         /*
00972           To be thread safely we can't rely on ascendant order of opened
00973           file descriptors. So we put each of descriptor we have to
00974           preserve into the array fifo.
00975               Note, in _this_ process there are no any threads but descriptors
00976           were created in frame of the parent process which may have
00977           multiple threads.
00978         */
00979         /*Mark descriptors which will NOT be closed:*/
00980         pushDescriptor(fifo,top++,fdsig[1]);
00981         pushDescriptor(fifo,top++,fdin[0]);
00982         pushDescriptor(fifo,top++,fdout[1]);
00983         /*Close all except stdin, stdout, stderr and placed into fifo:*/
00984         closeAllDescriptors(3,fifo, top);
00985
00986         /*Now reopen stdin and stdout.*/
00987         /*thread-safety is not a problem here since there are no any threads up to now:*/
00988         if(
```

```
00989             (close(0) == -1 )||/* Use fdin as stdin :*/
00990             (dup(fdin[0]) == -1 )||
00991             (close(1)==-1)||/* Use fdout as stdout:*/
00992             (dup(fdout[1]) == -1 )
00993           )
00994         {/*Fail!*/
00995             /*Signal to parent:*/
00996             writepid(fdsig[1],(pid_t)-2);
00997             _exit(1);
00998         }
00999
01000         if(stderrname != NULL){
01001             if(
01002                 (close(2) != 0 )||
01003                 (open(stderrname,O_WRONLY)<0)
01004               )
01005             {/*Fail!*/
01006                 writepid(fdsig[1],(pid_t)-2);
01007                 _exit(1);
01008             }
01009         }/*if(stderrname != NULL)*/
01010
01011         if( ttymode & 16 )/* create a session and sets the process group ID */
01012             setsid();
01013
01014         /*   */
01015         if(set_cloexec_flag (fdsig[1], 1)!=0){/*Error?*/
01016             /*Signal to parent:*/
01017             writepid(fdsig[1],(pid_t)-2);
01018             _exit(1);
01019         }/*if(set_cloexec_flag (fdsig[1], 1)!=0)*/
01020
01021         if( ttymode & 8 ){/*Daemonize*/
01022             int fdsig2[2];/*To check exec() success*/
01023             if(
01024                 pipe(fdsig2)||
01025                 (set_cloexec_flag (fdsig2[1], 1)!=0)
01026               )
01027             {/*Error?*/
01028                 /*Signal to parent:*/
01029                 writepid(fdsig[1],(pid_t)-2);
01030                 _exit(1);
01031             }
01032             set_cloexec_flag (fdsig2[0], 1);
01033             switch(childpid=fork()){
01034                 case 0:/*grandchild*/
01035                   /*Execute external command:*/
01036                   execvp(cmd, argv);
01037                   /* Control can  reach this point only on error!*/
01038                   writepid(fdsig2[1],(pid_t)-2);
01039                   break;
01040                 case -1:
01041                   /* Control can  reach this point only on error!*/
01042                   /*Inform the father about the failure*/
01043                   writepid(fdsig[1],(pid_t)-2);
01044                   _exit(1);/*The child, just exit, not return*/
01045               default:/*Son of his father*/
01046                   close(fdsig2[1]);
01047                   /*Ignore SIGPIPE (up to the end of the process):*/
01048                   signal(SIGPIPE,SIG_IGN);
01049
01050                   /*Wait on read() while the grandchild close the pipe
01051                      (on success) or send -2 (if exec() fails).*/
01052                   /*There are two possibilities:
01053                       -1 -- this is ok, the pipe was closed on exec,
01054                       the program was successfully executed;
01055                       -2 -- something is wrong, exec failed since the
01056                       grandchild sends -2 after exec.
01057                   */
01058                   if( readpid(fdsig2[0]) != (pid_t)-1 )/*something is wrong*/
01059                       writepid(fdsig[1],(pid_t)-1);
01060                   else/*ok, send PID of the grandchild to the father:*/
01061                       writepid(fdsig[1],childpid);
01062                   /*Die and free the life space for the grandchild:*/
01063                   _exit(0);/*The child, just exit, not return*/
01064             }/*switch(childpid=fork())*/
01065         }else{/*if( ttymode & 8 )*/
01066           execvp(cmd, argv);
01067           /* Control can  reach this point only on error!*/
01068           writepid(fdsig[1],(pid_t)-2);
01069           _exit(2);/*The child, just exit, not return*/
01070         }/*if( ttymode & 8 )...else*/
01071     }else{/* The (grand)father*/
01072         close(fdsig[1]);
01073         /*To prevent closing fdsig in rollback:*/
01074         fdsig[1]=-1;
01075         close(fdin[0]);
```

```
01076         close(fdout[1]);
01077         *fdsend    = fdin[1];
01078         *fdreceive = fdout[0];
01079 
01080         /*Get the process group ID.*/
01081         /*Avoid to use getpgid() which is non-standard.*/
01082         if( ttymode & 16)/*setsid() was invoked, the child is a group leader:*/
01083             *gpid=childpid;
01084         else/*the child belongs to the same process group as the this process:*/
01085             *gpid=getpgrp();/*if compiler fails here, try getpgrp(0) instead!*/
01086         /*
01087           Rationale: getpgrp  conform  to POSIX.1 while 4.3BSD provides a
01088           getpgrp() function that returns the process group ID for a
01089           specified process.
01090         */
01091 
01092         /* Temporary ignore this signal:*/
01093         /* if compiler fails here, try to change the definition of
01094           mysighandler_t on the beginning of this file
01095           (just define INTSIGHANDLER)*/
01096         oldPIPE=signal(SIGPIPE,SIG_IGN);
01097 
01098         if( ttymode & 8 ){/*Daemonize*/
01099             /*Read the grandchild PID from the son.*/
01100             fatherchildpid=childpid;
01101             if(  (childpid=readpid(fdsig[0]))<0  ){
01102                 /*Daemonization process fails for some reasons!*/
01103                 childpid=fatherchildpid;/*for rollback*/
01104                 goto fail_do_run_cmd;
01105             }
01106         }else{
01107               /*fdsig[1] should be closed on exec and this read operation
01108                  must fail on success:*/
01109               if( readpid(fdsig[0])!= (pid_t)-1 )
01110                   goto fail_do_run_cmd;
01111         }/*if( ttymode & 8 ) ... else*/
01112     }/*if(childpid == 0)...else*/
01113 
01114     /*Here can be ONLY the father*/
01115     close(fdsig[0]);
01116     /*To prevent closing fdsig in rollback after goto fail_flnk_do_runcmd:*/
01117     fdsig[0]=-1;
01118 
01119     if( ttymode & 8 ){/*Daemonize*/
01120         int i;
01121         /*Wait while the father of a grandchild dies:*/
01122         waitpid(fatherchildpid,&i,0);
01123     }
01124 
01125     /*Restore the signal:*/
01126     signal(SIGPIPE,oldPIPE);
01127 
01128     return(childpid);
01129 
01130     fail_do_run_cmd:
01131         closepipe(&fdout);
01132         closepipe(&fdin);
01133         closepipe(&fdsig);
01134         return((pid_t)-1);
01135 }/*do_run_cmd*/
01136 /*
01137     #] do_run_cmd :
01138     #[ run_cmd :
01139 */
01140 /*Starts the command cmd (directly, if shellpath is NULL, or in a subshell),
01141   swallowing its stdin and stdout;
01142   stderr will be re-directed to stderrname (if !=NULL). Returns PID of
01143   the started process. Stdin will be available as fdsend, and stdout
01144   will be available as fdreceive:*/
01145 static FORM_INLINE pid_t run_cmd(char *cmd,
01146                                  int *fdsend,
01147                                  int *fdreceive,
01148                                  int *gpid,
01149                                  int daemonize,
01150                                  char *shellpath,
01151                                  char *stderrname )
01152 {
01153 char **argv;
01154 pid_t thepid;
01155 
01156     cmd=(char*)strDup1((UBYTE*)cmd, "run_cmd: cmd");/*detouch cmd*/
01157 
01158 
01159     /* Prepare arguments for execvp:*/
01160     if(shellpath != NULL){/*Run in a subshell.*/
01161         int nopt;
01162         /*Allocate space which is definitely enough:*/
```

```
01163         argv=Malloc1(StrLen((UBYTE*)shellpath)*sizeof(char*)+2,"run_cmd:argv");
01164         shellpath=(char*)strDup1((UBYTE*)shellpath, "run_cmd: shellpath");/*detouch shellpath*/
01165         /*Parse a shell (e.g., "/bin/sh -c"):*/
01166         nopt=parseline(argv, shellpath);
01167         /* and add the command as a shell argument:*/
01168         argv[nopt]=cmd;
01169         argv[nopt+1]=NULL;
01170     }else{/*Run the command directly:*/
01171         /*Allocate space which is definitely enough:*/
01172         argv=Malloc1(StrLen((UBYTE*)cmd)*sizeof(char*)+1,"run_cmd:argv");
01173         parseline(argv, cmd);
01174     }
01175
01176     thepid=do_run_cmd(
01177         fdsend,
01178         fdreceive,
01179         gpid,
01180         (daemonize)?(8|16):0,
01181         argv[0],
01182         argv,
01183         stderrname
01184     );
01185
01186     M_free(argv,"run_cmd:argv");
01187     if(shellpath)
01188         M_free(shellpath,"run_cmd:argv");
01189     M_free(cmd, "run_cmd: cmd");
01190
01191     return(thepid);
01192 }/*run_cmd*/
01193 /*
01194     #] run_cmd :
01195     #[ createExternalChannel :
01196 */
01197 /*The structure to pass parameters to createExternalChannel
01198   and openExternalChannel in case of preset channel (instead of
01199   shellname):*/
01200 typedef struct{
01201     int fdin;
01202     int fdout;
01203     pid_t theppid;
01204 }ECINFOSTRUCT;
01205
01206 /* Creates a new external channel starting the command cmd (if cmd !=NULL)
01207     or using information from (ECINFOSTRUCT *)shellname, if cmd ==NULL:*/
01208 static FORM_INLINE void *createExternalChannel(
01209                       EXTHANDLE *h,
01210                       char *cmd, /*Command to run or NULL*/
01211                       /*0 --neither setsid nor daemonize, !=0 -- full daemonization:*/
01212                       int daemonize,
01213                       char *shellname,/* The shell (like "/bin/sh -c") or NULL*/
01214                       char *stderrname/*filename to redirect stderr or NULL*/
01215                                                           )
01216 {
01217     int fdreceive=0;
01218     int gpid = 0;
01219     ECINFOSTRUCT *psetInfo;
01220 #ifdef WITHMPI
01221     char statusbuf[2]={'\0','\0'};/*'\0' if run_cmd retuns ok, '!' otherwise.*/
01222 #endif
01223     extHandlerInit(h);
01224
01225     h->pid=0;
01226
01227     if( cmd==NULL ){/*Instead of starting a new command, use preset channel:*/
01228         psetInfo=(ECINFOSTRUCT *)shellname;
01229         shellname=NULL;
01230         h->killSignal=0;
01231         h->daemonize=0;
01232     }
01233
01234     /*Create a channel:*/
01235 #ifdef WITHMPI
01236     if ( PF.me == MASTER ){
01237 #endif
01238         if(cmd!=NULL)
01239             h->pid=run_cmd (cmd, &(h->fsend),
01240                     &fdreceive,&gpid,daemonize,shellname,stderrname);
01241         else{
01242             gpid=-psetInfo->theppid;
01243             h->pid=psetInfo->theppid;
01244             h->fsend=psetInfo->fdout;
01245             fdreceive=psetInfo->fdin;
01246         }
01247 #ifdef WITHMPI
01248         if(h->pid<0)
01249             statusbuf[0]='!';/*Broadcast fail to slaves*/
```

```
01250     }
01251      /*else: Keep h->pid = 0 and h->fsend = 0 for slaves in parallel mode!*/
01252
01253     /*Master broadcasts status to slaves, slaves read it from the master:*/
01254     if( PF_BroadcastString((UBYTE *)statusbuf) ){/*Fail!*/
01255         h->pid=-1;
01256     }else if( statusbuf[0]=='!')/*Master fails*/
01257         h->pid=-1;
01258 #endif
01259
01260     if(h->pid<0)goto createExternalChannelFails;
01261 #ifdef WITHMPI
01262     if ( PF.me == MASTER ){
01263 #endif
01264         h->gpid=gpid;
01265     /*Open stdout of a newly created program as FILE* :*/
01266         if( (h->frec=fdopen(fdreceive,"r")) == 0 )goto createExternalChannelFails;
01267
01268 #ifdef WITHMPI
01269     }
01270 #endif
01271     /*Initialize buffers:*/
01272     extHandlerAlloc(h,cmd,shellname,stderrname);
01273     return(h);
01274     /*Something is wrong?*/
01275     createExternalChannelFails:
01276
01277         destroyExternalChannel(h);
01278         return(NULL);
01279 }/*createExternalChannel*/
01280
01281 /*
01282     #] createExternalChannel :
01283     #[ openExternalChannel :
01284 */
01285 int openExternalChannel(UBYTE *cmd, int daemonize, UBYTE *shellname, UBYTE *stderrname)
01286 {
01287 EXTHANDLE *h=externalChannelsListTop;
01288 int i=0;
01289
01290     for(externalChannelsListTop=0;i<externalChannelsListFill;i++)
01291         if(externalChannelsList[i].cmd==0){
01292             externalChannelsListTop=externalChannelsList+i;
01293             break;
01294         }/*if(externalChannelsList[i].cmd!=0)*/
01295
01296     if(externalChannelsListTop==0){
01297       /*No free cells, create the new one:*/
01298       if(externalChannelsListFill == externalChannelsListStop){
01299             /*The list is full, increase and reallocate it:*/
01300             EXTHANDLE *newbuf=Malloc1(
01301                                       (externalChannelsListStop+=DELTA_EXT_LIST)*sizeof(EXTHANDLE),
01302                                     "External channel list");
01303             for(i=0;i<externalChannelsListFill;i++)
01304                  extHandlerSwallowCopy(newbuf+i,externalChannelsList+i);
01305             if(externalChannelsList!=0)
01306                 M_free(externalChannelsList,"External channel list");
01307             for(;i<externalChannelsListStop;i++)
01308                 extHandlerInit(newbuf+i);
01309             externalChannelsList=newbuf;
01310       }/*if(externalChannelsListFill == externalChannelsListStop)*/
01311       externalChannelsListTop=externalChannelsList+externalChannelsListFill;
01312       externalChannelsListFill++;
01313     }/*if(externalChannelsListTop==0)*/
01314
01315     if(createExternalChannel(
01316                                     externalChannelsListTop,
01317                                     (char*) cmd,
01318                                     daemonize,
01319                                     (char*)shellname,
01320                                     (char*)stderrname            )!=NULL){
01321         writeBufToExtChannel=&writeBufToExtChannelOk;
01322         getcFromExtChannel=&getcFromExtChannelOk;
01323         setTerminatorForExternalChannel=&setTerminatorForExternalChannelOk;
01324         setKillModeForExternalChannel=&setKillModeForExternalChannelOk;
01325         return(externalChannelsListTop-externalChannelsList+1);
01326     }
01327     /*else - failure!*/
01328     externalChannelsListTop=h;
01329     return(-1);
01330 }/*openExternalChannel*/
01331 /*
01332     #] openExternalChannel :
01333     #[ initPresetExternalChannels :
01334 */
01335 /*Just simple wrapper to invoke  openExternalChannel()
01336     from initPresetExternalChannels():*/
```

```
01337 static FORM_INLINE int openPresetExternalChannel(int fdin, int fdout, pid_t theppid)
01338 {
01339 ECINFOSTRUCT inf;
01340     inf.fdin=fdin; inf.fdout=fdout;inf.theppid=theppid;
01341     return( openExternalChannel(NULL,0,(UBYTE *)&inf,NULL) );
01342 }/*openPresetExternalChannel*/
01343
01344 #define PIDTXTSIZE 23
01345 #define BOTHPIDSIZE 45
01346
01347 #ifndef LONG_MAX
01348 #define LONG_MAX 0x7FFFFFFFL
01349 #endif
01350
01351 /*There is a possibility to start FORM from another program providing
01352 one (or more) communication channels. These channels will be
01353 visible from a FORM program as ``pre-opened'' external channels existing
01354 after FORM starts.
01355 Before starting FORM, the parent application must create one or more
01356 pairs of pipes The read-only descriptor of the first pipe in the pair
01357 and the write-only descriptor of the second pipe must be passed to
01358 FORM as an argument of a command line option -pipe in ASCII decimal
01359 format. The argument of the option is a comma-separated list of pairs
01360 r#,w# where r# is a read-only descriptor and w# is a write-only
01361 descriptor. Alternatively, the environment variable FORM_PIPES can be
01362 used.
01363     The following function expects as the first argument
01364 this comma-separated list of the  descriptor pairs and tries to
01365 initialize each of channel during thetimeout milliseconds:*/
01366
01367 int initPresetExternalChannels(UBYTE *theline, int thetimeout)
01368 {
01369     int i, nchannels = 0;
01370     int pidln;           /*The length of FORM PID in pidtxt*/
01371     char pidtxt[PIDTXTSIZE],     /*64/Log_2[10] = 19.3, this is enough for any integer*/
01372          chdescriptor[PIDTXTSIZE],
01373          bothpidtxt[BOTHPIDSIZE],   /*"#,#\n\0"*/
01374          *c,*b = 0;
01375     int pip[2];
01376     pid_t ppid;
01377     if ( theline == NULL ) return(-1);
01378     /*Put into pidtxt PID\n:*/
01379     c = l2s((LONG)getpid(),pidtxt);
01380     *c++='\n';
01381     *c = '\0';
01382     pidln = c-pidtxt;
01383
01384     do {
01385         pip[0] = (int)str2i((char*)theline,&c,0xFFFF);
01386         if( ( pip[0] < 0 ) || ( *c != ',' ) ) goto presetFails;
01387
01388         theline = (UBYTE*)c + 1;
01389         pip[1] = (int)str2i((char*)theline,&c,0xFFFF);
01390         if ( (pip[1] < 0 ) || ( ( *c != ',' ) && ( *c != '\0' ) ) ) goto presetFails;
01391         theline = (UBYTE *)c + 1;
01392         /*Now we have two descriptors.
01393           According to the protocol, FORM must send to external channel
01394           it's PID with added '\n' and read back two comma-separated
01395           decimals with added '\n'. The first must be repeated FORM PID,
01396           the second must be the parent PID
01397         */
01398         if ( writeSome(pip[1],pidtxt,pidln,thetimeout) != pidln ) goto presetFails;
01399         i = readSome(pip[0],bothpidtxt,BOTHPIDSIZE,thetimeout);
01400         if( ( i < 4 )                /*at least 1,2\n*/
01401             || ( i == BOTHPIDSIZE )   /*No space for trailing '\0'*/
01402           ) goto presetFails;
01403         /*read the FORM PID:*/
01404         ppid = (pid_t)str2i(bothpidtxt,&b,getpid());
01405         if( ( *b != ',' ) || ( ppid != getpid() ) )goto presetFails;
01406         /*read the parent PID:*/
01407         /*The problem is that we do not know the the real type of pid_t.
01408           But long should be enough. On obsolete systems (when LONG_MAX
01409           is not defined) we assume pid_t is 32-bit integer.
01410           This can lead to problem with portability: */
01411         ppid = (pid_t)str2i(b+1,&b,LONG_MAX);
01412         if ( (*b != '\n') || (ppid<2) ) goto presetFails;
01413         i = openPresetExternalChannel(pip[0],pip[1],ppid);
01414         if ( i < 0 ) goto presetFails;
01415         nchannels++;
01416         /*Now use bothpidtxt as a buffer for preprovar, the space is enough:*/
01417         /*"PIPE#_" where # is ne order number of the channel:*/
01418         b = l2s(nchannels,addStr(bothpidtxt,"PIPE"));
01419         *b = '_';
01420         b[1] = '\0';
01421         *l2s(i,chdescriptor) = '\0';
01422         PutPreVar((UBYTE*)bothpidtxt,(UBYTE*)chdescriptor,0,0);
01423     } while ( *c != '\0' );
```

```
01424     /*Export proprovar "PIPES_":*/
01425     *l2s(nchannels,chdescriptor)='\0';
01426     PutPreVar((UBYTE*)"PIPES_",(UBYTE*)chdescriptor,0,0);
01427 
01428     /*success:*/
01429     return (nchannels);
01430 
01431 presetFails:
01432         /*Here we assume the descriptors the beginning of the list!*/
01433         for(i=0; i<nchannels; i++)
01434             destroyExternalChannel(externalChannelsList+i);
01435         return(-1);
01436 } /*initPresetExternalChannels*/
01437 /*
01438     #] initPresetExternalChannels :
01439     #[ selectExternalChannel :
01440 */
01441 /*
01442 Accepts the valid external channel descriptor (returned by
01443 openExternalChannel) and returns the descriptor of a previous current
01444 channel (0, if there was no current channel, or -1, if the external
01445 channel descriptor is invalid).  If n == 0, the function undefine the
01446 current external channel:
01447 */
01448 int selectExternalChannel(int n)
01449 {
01450 int ret=0;
01451     if(externalChannelsListTop!=0)
01452         ret=externalChannelsListTop-externalChannelsList+1;
01453 
01454     if(--n<0){
01455         if(n!=-1)
01456             return(-1);
01457         externalChannelsListTop=0;
01458         writeBufToExtChannel=&writeBufToExtChannelFailure;
01459         getcFromExtChannel=&getcFromExtChannelFailure;
01460         setTerminatorForExternalChannel=&setTerminatorForExternalChannelFailure;
01461         setKillModeForExternalChannel=&setKillModeForExternalChannelFailure;
01462         return(ret);
01463     }
01464     if(
01465         (n>=externalChannelsListFill)||
01466         (externalChannelsList[n].cmd==0)
01467     )
01468         return(-1);
01469 
01470     externalChannelsListTop=externalChannelsList+n;
01471     writeBufToExtChannel=&writeBufToExtChannelOk;
01472     getcFromExtChannel=&getcFromExtChannelOk;
01473     setTerminatorForExternalChannel=&setTerminatorForExternalChannelOk;
01474     setKillModeForExternalChannel=&setKillModeForExternalChannelOk;
01475     return(ret);
01476 }/*selectExternalChannel*/
01477 /*
01478     #] selectExternalChannel :
01479     #[ closeExternalChannel :
01480 */
01481 
01482 /*
01483 Destroys the opened external channel with the descriptor n. It returns
01484 0 in success, or -1 on failure. If the corresponding external channel
01485 was the current one, the current channel becomes undefined. If n==0,
01486 the function closes the current external channel.
01487 */
01488 int closeExternalChannel(int n)
01489 {
01490     if(n==0)
01491         n=externalChannelsListTop-externalChannelsList;
01492     else
01493         n--;/*Count from 0*/
01494 
01495     if(
01496         (n<0)||
01497         (n>=externalChannelsListFill)||
01498         (externalChannelsList[n].cmd==0)
01499     )/*No shuch a channel*/
01500         return(-1);
01501 
01502     destroyExternalChannel(externalChannelsList+n);
01503     /*If the current external channel was destroyed, undefine current channel:*/
01504     if(externalChannelsListTop==externalChannelsList+n){
01505         externalChannelsListTop=NULL;
01506         writeBufToExtChannel=&writeBufToExtChannelFailure;
01507         getcFromExtChannel=&getcFromExtChannelFailure;
01508         setTerminatorForExternalChannel=&setTerminatorForExternalChannelFailure;
01509         setKillModeForExternalChannel=&setKillModeForExternalChannelFailure;
01510     }/*if(externalChannelsListTop==externalChannelsList+n)*/
```

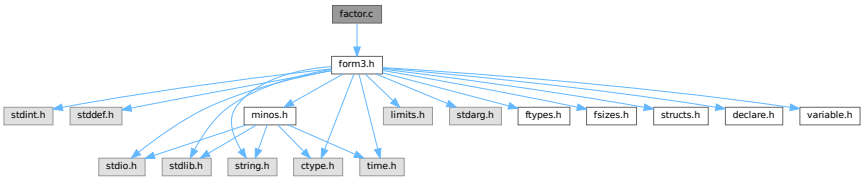
```
01511     return(0);
01512 }/*closeExternalChannel*/
01513 /*
01514     #] closeExternalChannel :
01515     #[ closeAllExternalChannels :
01516 */
01517 void closeAllExternalChannels(void)
01518 {
01519 int i;
01520     for(i=0; i<externalChannelsListFill; i++)
01521         destroyExternalChannel(externalChannelsList+i);
01522     externalChannelsListFill=externalChannelsListStop=0;
01523     externalChannelsListTop=NULL;
01524
01525     writeBufToExtChannel=&writeBufToExtChannelFailure;
01526     getcFromExtChannel=&getcFromExtChannelFailure;
01527     setTerminatorForExternalChannel=&setTerminatorForExternalChannelFailure;
01528     setKillModeForExternalChannel=&setKillModeForExternalChannelFailure;
01529
01530     if(externalChannelsList!=NULL){
01531         M_free(externalChannelsList,"External channel list");
01532         externalChannelsList=NULL;
01533     }
01534 }/*closeAllExternalChannels*/
01535 /*
01536     #] closeAllExternalChannels :
01537     #[ getExternalChannelPid :
01538 */
01539 #ifdef SELFTEST
01540 pid_t getExternalChannelPid(void)
01541 {
01542   if(externalChannelsListTop!=0)
01543         return(externalChannelsListTop ->pid);
01544   return(-1);
01545 }/*getExternalChannelPid*/
01546 #endif
01547 /*
01548     #] getExternalChannelPid :
01549     #[ getCurrentExternalChannel :
01550 */
01551
01552 int getCurrentExternalChannel(void)
01553 {
01554
01555     if ( externalChannelsListTop != 0 )
01556         return(externalChannelsListTop-externalChannelsList+1);
01557     return(0);
01558 }/*getCurrentExternalChannel*/
01559 /*
01560     #] getCurrentExternalChannel :
01561     #[ Selftest main :
01562 */
01563
01564 #ifdef SELFTEST
01565
01566 /*
01567     This is the example of how all these public functions may be used:
01568 */
01569
01570 char buf[1024];
01571 char buf2[1024];
01572
01573 void help(void)
01574 {
01575     printf("String started with a special letter is a command\n");
01576     printf("Known commands are:\n");
01577     printf("H or ? -- this help\n");
01578     printf("Nn<command> -- start a new command\n");
01579     printf("S<command> -- start a new command in a subshell,daemon,stderr>/dev/null\n");
01580     printf("C# -- destroy channel #\n");
01581     printf("R# -- set a new cahhel(number#) as a current one\n");
01582     printf("K#1 #2 --  set signal for kill and kill mode (0 or !=0)\n");
01583     printf("   ^d to quit\n");
01584 }/*help*/
01585
01586 int main (void)
01587 {
01588     int i, j, k,last;
01589     long long sum = 0;
01590
01591     /*openExternalChannel(UBYTE *cmd, int daemonize, UBYTE *shellname, UBYTE *stderrname)*/
01592
01593     help();
01594
01595     printf("Initial channel:%d\n",last=openExternalChannel((UBYTE*)"cat",0,NULL,NULL));
01596
01597     if( ( i = setTerminatorForExternalChannel("qu") ) != 0 ) return 1;
```

```
01598     printf("Terminator is 'qu'\n");
01599
01600     while ( fgets(buf, 1024, stdin) != NULL ) {
01601         if ( *buf == 'N' ) {
01602             printf("New channel:%d\n",j=openExternalChannel((UBYTE*)buf+1,0,NULL,NULL));
01603             continue;
01604         }
01605         else if ( *buf == 'C' ) {
01606             int n;
01607             sscanf(buf+1,"%d",&n);
01608             printf("Destroy last channel:%d\n",closeExternalChannel(n));
01609             continue;
01610         }
01611         else if ( *buf == 'R' ) {
01612             int n = 0;
01613             sscanf(buf+1,"%d",&n);
01614             printf("Reopen channel %d:%d\n",n,selectExternalChannel(n));
01615             continue;
01616         }else if( *buf == 'K' ) {
01617             int n=0,g = 0;
01618             sscanf(buf+1,"%d %d",&n,&g);
01619             printf("setKillMode %d\n",setKillModeForExternalChannel(n,g));
01620             continue;
01621         }else if( *buf == 'S' ) {
01622             printf("New channel with sh&d&stderr:%d\n",
01623                     j=openExternalChannel((UBYTE*)buf+1,1,(UBYTE*)"/bin/sh -c",(UBYTE*)"/dev/null"));
01624             continue;
01625         }
01626         else if( ( *buf == 'H' )|| ( *buf == '?' )  ){
01627             help();
01628             continue;
01629         }
01630
01631         writeBufToExtChannel(buf,k=StrLen(buf));
01632         sum += k;
01633         for ( j = 0; ( i = getcFromExtChannel() ) != '\n'; j++) {
01634             if ( i == EOF ) {
01635                 printf("EOF!\n");
01636                 break;
01637             }
01638             buf2[j] = i;
01639         }
01640         buf2[j] = '\0';
01641         printf("I got:'%s'; pid=%d\n",buf2,getExternalChannelPid());
01642     }
01643     printf("Total:%lld bytes\n",sum);
01644     closeAllExternalChannels();
01645     return 0;
01646 }
01647 #endif /*ifdef SELFTEST*/
01648 /*
01649     #] Selftest main :
01650 */
01651
01652 #endif /*ifndef WITHEXTERNALCHANNEL ... else*/
```

4.35 factor.c File Reference

`#include "form3.h"`

Include dependency graph for factor.c:



Functions

- int FactorIn (PHEAD WORD *term, WORD level)
- int FactorInExpr (PHEAD WORD *term, WORD level)

4.35.1 Detailed Description

The routines for finding (one term) factors in dollars and expressions.

Definition in file factor.c.

4.35.2 Function Documentation

4.35.2.1 FactorIn()

```
int FactorIn (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 46 of file factor.c.

4.35.2.2 FactorInExpr()

```
int FactorInExpr (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 423 of file factor.c.

4.36 factor.c

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
00026  *
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
00031 /*
00032     #[ Includes : factor.c
00033 */
00034
00035 #include "form3.h"
00036
00037 /*
00038     #] Includes :
00039     #[ FactorIn :
```

```
00040
00041     This routine tests for a factor in a dollar expression.
00042
00043     Note that unlike with regular active or hidden expressions we cannot
00044     add memory as easily as dollars are rather volatile.
00045 */
00046 int FactorIn(PHEAD WORD *term, WORD level)
00047 {
00048     GETBIDENTITY
00049     WORD *t, *tstop, *m, *mm, *oldwork, *mstop, *n1, *n2, *n3, *n4, *n1stop, *n2stop;
00050     WORD *r1, *r2, *r3, *r4, j, k, kGCD, kGCD2, kLCM, jGCD, kkLCM, jLCM, size;
00051     UWORD *GCDbuffer, *GCDbuffer2, *LCMbuffer, *LCMb, *LCMc;
00052     int fromwhere = 0, i;
00053     DOLLARS d;
00054     t = term; GETSTOP(t,tstop); t++;
00055     while ( ( t < tstop ) && ( *t != FACTORIN || ( ( *t == FACTORIN )
00056      && ( t[FUNHEAD] != -DOLLAREXPRESSION || t[1] != FUNHEAD+2 ) ) ) ) t += t[1];
00057     if ( t >= tstop ) {
00058 /* INTERNAL_ERROR_EXCL_START */
00059         MLOCK(ErrorMessageLock);
00060         MesPrint("!>Internal error. Could not find proper factorin_ function.");
00061         MUNLOCK(ErrorMessageLock);
00062         return(-1);
00063 /* INTERNAL_ERROR_EXCL_STOP */
00064     }
00065     oldwork = AT.WorkPointer;
00066     d = Dollars + t[FUNHEAD+1];
00067 #ifdef WITHPTHREADS
00068     {
00069         int nummodopt, dtype = -1;
00070         if ( AS.MultiThreaded ) {
00071             for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
00072                 if ( t[FUNHEAD+1] == ModOptdollars[nummodopt].number ) break;
00073             }
00074             if ( nummodopt < NumModOptdollars ) {
00075                 dtype = ModOptdollars[nummodopt].type;
00076                 if ( DollarLocalCopy(dtype) ) {
00077                     d = ModOptdollars[nummodopt].dstruct+AT.identity;
00078                 }
00079             }
00080         }
00081     }
00082 #endif
00083
00084     if ( d->type == DOLTERMS ) {
00085         fromwhere = 1;
00086     }
00087     else if ( ( d = DolToTerms(BHEAD t[FUNHEAD+1]) ) == 0 ) {
00088 /*
00089         The variable cannot convert to an expression
00090         We replace the function by 1.
00091 */
00092         m = oldwork; n1 = term;
00093         while ( n1 < t ) *m++ = *n1++;
00094         n1 = t + t[1]; tstop = term + *term;
00095         while ( n1 < tstop ) *m++ = *n1++;
00096         *oldwork = m - oldwork;
00097         AT.WorkPointer = m;
00098         if ( Generator(BHEAD oldwork,level) ) return(-1);
00099         AT.WorkPointer = oldwork;
00100         return(0);
00101     }
00102     if ( d->where[0] == 0 ) {
00103         if ( fromwhere == 0 ) {
00104             if ( d->factors ) M_free(d->factors,"Dollar factors");
00105             M_free(d,"Dollar in FactorIn_");
00106         }
00107         return(0);
00108     }
00109 /*
00110     Now we have an expression in d->where. Find the symbolic factor that
00111     divides the expression and the numerical factor that makes all
00112     coefficients integer.
00113
00114     For the symbolic factor we make a copy of the first term, and then
00115     go through all terms, scratching in the copy the objects that do not
00116     occur in the terms.
00117 */
00118     m = oldwork;
00119     mm = d->where;
00120     k = *mm - ABS((mm[*mm-1]));
00121     for ( j = 0; j < k; j++ ) *m++ = *mm++;
00122     mstop = m;
00123     *oldwork = k;
00124 /*
00125     The copy is in place. Now search through the terms. Start at the second term
00126 */
```

```
00127     mm = d->where + d->where[0];
00128     while ( *mm ) {
00129         m = oldwork+1;
00130         r2 = mm+*mm;
00131         r2 -= ABS(r2[-1]);
00132         r1 = mm+1;
00133         while ( m < mstop ) {
00134             while ( r1 < r2 ) {
00135                 if ( *r1 != *m ) {
00136                     r1 += r1[1]; continue;
00137                 }
00138 /*
00139                     Now the various cases
00140             #[ SYMBOL :
00141 */
00142                 if ( *m == SYMBOL ) {
00143                     n1 = m+2; n1stop = m+m[1];
00144                     n2stop = r1+r1[1];
00145                     while ( n1 < n1stop ) {
00146                         n2 = r1+2;
00147                         while ( n2 < n2stop ) {
00148                             if ( *n1 != *n2 ) { n2 += 2; continue; }
00149                             if ( n1[1] > 0 ) {
00150                                 if ( n2[1] < 0 ) { n2 += 2; continue; }
00151                                 if ( n2[1] < n1[1] ) n1[1] = n2[1];
00152                             }
00153                             else {
00154                                 if ( n2[1] > 0 ) { n2 += 2; continue; }
00155                                 if ( n2[1] > n1[1] ) n1[1] = n2[1];
00156                             }
00157                             break;
00158                         }
00159                         if ( n2 >= n2stop ) {   /* scratch symbol */
00160                             if ( m[1] == 4 ) goto scratch;
00161                             m[1] -= 2;
00162                             n3 = n1; n4 = n1+2;
00163                             while ( n4 < mstop ) *n3++ = *n4++;
00164                             *oldwork = n3 - oldwork;
00165                             mstop -= 2; n1stop -= 2;
00166                             continue;
00167                         }
00168                         n1 += 2;
00169                     }
00170                     break;
00171                 }
00172 /*
00173             #] SYMBOL :
00174             #[ DOTPRODUCT :
00175 */
00176                 else if ( *m == DOTPRODUCT ) {
00177                     n1 = m+2; n1stop = m+m[1];
00178                     n2stop = r1+r1[1];
00179                     while ( n1 < n1stop ) {
00180                         n2 = r1+2;
00181                         while ( n2 < n2stop ) {
00182                             if ( *n1 != *n2 || n1[1] != n2[1] ) { n2 += 3; continue; }
00183                             if ( n1[2] > 0 ) {
00184                                 if ( n2[2] < 0 ) { n2 += 3; continue; }
00185                                 if ( n2[2] < n1[2] ) n1[2] = n2[2];
00186                             }
00187                             else {
00188                                 if ( n2[2] > 0 ) { n2 += 3; continue; }
00189                                 if ( n2[2] > n1[2] ) n1[2] = n2[2];
00190                             }
00191                             break;
00192                         }
00193                         if ( n2 >= n2stop ) {   /* scratch symbol */
00194                             if ( m[1] == 5 ) goto scratch;
00195                             m[1] -= 3;
00196                             n3 = n1; n4 = n1+3;
00197                             while ( n4 < mstop ) *n3++ = *n4++;
00198                             *oldwork = n3 - oldwork;
00199                             mstop -= 3; n1stop -= 3;
00200                             continue;
00201                         }
00202                         n1 += 3;
00203                     }
00204                     break;
00205                 }
00206 /*
00207             #] DOTPRODUCT :
00208             #[ VECTOR :
00209 */
00210                 else if ( *m == VECTOR ) {
00211 /*
00212                     Here we have to be careful if there is more than
00213                     one of the same
```

```
00214 */
00215                     n1 = m+2; n1stop = m+m[1];
00216                     n2 = r1+2;n2stop = r1+r1[1];
00217                     while ( n1 < n1stop ) {
00218                         while ( n2 < n2stop ) {
00219                             if ( *n1 == *n2 && n1[1] == n2[1] ) {
00220                                 n2 += 2; goto nextn1;
00221                             }
00222                             n2 += 2;
00223                         }
00224                         if ( n2 >= n2stop ) {   /* scratch symbol */
00225                             if ( m[1] == 4 ) goto scratch;
00226                             m[1] -= 2;
00227                             n3 = n1; n4 = n1+2;
00228                             while ( n4 < mstop ) *n3++ = *n4++;
00229                             *oldwork = n3 - oldwork;
00230                             mstop -= 2; n1stop -= 2;
00231                             continue;
00232                         }
00233                         n2 = r1+2;
00234 nextn1:                 n1 += 2;
00235                     }
00236                     break;
00237                 }
00238 /*
00239             #] VECTOR :
00240             #[ REMAINDER :
00241 */
00242                 else {
00243 /*
00244                     Again: watch for multiple occurrences of the same object
00245 */
00246                     if ( m[1] != r1[1] ) { r1 += r1[1]; continue; }
00247                     for ( j = 2; j < m[1]; j++ ) {
00248                         if ( m[j] != r1[j] ) break;
00249                     }
00250                     if ( j < m[1] ) { r1 += r1[1]; continue; }
00251                     r1 += r1[1]; /* to restart at the next potential match */
00252                     goto nextm;  /* match */
00253                 }
00254 /*
00255             #] REMAINDER :
00256 */
00257             }
00258             if ( r1 >= r2 ) { /* no factor! */
00259 scratch:;
00260                 r3 = m + m[1]; r4 = m;
00261                 while ( r3 < mstop ) *r4++ = *r3++;
00262                 *oldwork = r4 - oldwork;
00263                 if ( *oldwork == 1 ) goto nofactor;
00264                 mstop = r4;
00265                 r1 = mm + 1;
00266                 continue;
00267             }
00268             r1 = mm + 1;
00269 nextm:      m += m[1];
00270         }
00271         mm = mm + *mm;
00272     }
00273
00274 nofactor:;
00275 /*
00276     For the coefficient we have to determine the LCM of the denominators
00277     and the GCD of the numerators.
00278 */
00279     GCDbuffer = NumberMalloc("FactorIn"); GCDbuffer2 = NumberMalloc("FactorIn");
00280     LCMbuffer = NumberMalloc("FactorIn"); LCMb = NumberMalloc("FactorIn"); LCMc =
      NumberMalloc("FactorIn");
00281     r1 = d->where;
00282 /*
00283     First take the first term to load up the LCM and the GCD
00284 */
00285     r2 = r1 + *r1;
00286     j = r2[-1];
00287     r3 = r2 - ABS(j);
00288     k = REDLENG(j);
00289     if ( k < 0 ) k = -k;
00290     while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00291     for ( kGCD = 0; kGCD < k; kGCD++ ) GCDbuffer[kGCD] = r3[kGCD];
00292     k = REDLENG(j);
00293     if ( k < 0 ) k = -k;
00294     r3 += k;
00295     while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00296     for ( kLCM = 0; kLCM < k; kLCM++ ) LCMbuffer[kLCM] = r3[kLCM];
00297     r1 = r2;
00298 /*
00299     Now go through the rest of the terms in this dollar buffer.
```

```
00300 */
00301     while ( *r1 ) {
00302         r2 = r1 + *r1;
00303         j = r2[-1];
00304         r3 = r2 - ABS(j);
00305         k = REDLENG(j);
00306         if ( k < 0 ) k = -k;
00307         while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00308         if ( ( ( GCDbuffer[0] == 1 ) && ( kGCD == 1 ) ) ) {
00309 /*
00310             GCD is already 1
00311 */
00312         }
00313         else if ( ( ( k != 1 ) || ( r3[0] != 1 ) ) ) {
00314             if ( GcdLong(BHEAD GCDbuffer,kGCD,(UWORD *)r3,k,GCDbuffer2,&kGCD2) ) {
00315                 goto onerror;
00316             }
00317             kGCD = kGCD2;
00318             for ( i = 0; i < kGCD; i++ ) GCDbuffer[i] = GCDbuffer2[i];
00319         }
00320         else {
00321             kGCD = 1; GCDbuffer[0] = 1;
00322         }
00323         k = REDLENG(j);
00324         if ( k < 0 ) k = -k;
00325         r3 += k;
00326         while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00327         if ( ( ( LCMbuffer[0] == 1 ) && ( kLCM == 1 ) ) ) {
00328             for ( kLCM = 0; kLCM < k; kLCM++ )
00329                 LCMbuffer[kLCM] = r3[kLCM];
00330         }
00331         else if ( ( ( k != 1 ) || ( r3[0] != 1 ) ) ) {
00332             if ( GcdLong(BHEAD LCMbuffer,kLCM,(UWORD *)r3,k,LCMb,&kkLCM) ) {
00333                 goto onerror;
00334             }
00335             DivLong((UWORD *)r3,k,LCMb,kkLCM,LCMb,&kkLCM,LCMc,&jLCM);
00336             MulLong(LCMbuffer,kLCM,LCMb,kkLCM,LCMc,&jLCM);
00337             for ( kLCM = 0; kLCM < jLCM; kLCM++ )
00338                 LCMbuffer[kLCM] = LCMc[kLCM];
00339         }
00340         else {} /* LCM doesn't change */
00341         r1 = r2;
00342     }
00343 /*
00344     Now put the factor together: GCD/LCM
00345 */
00346     r3 = (WORD *)(GCDbuffer);
00347     if ( kGCD == kLCM ) {
00348         for ( jGCD = 0; jGCD < kGCD; jGCD++ )
00349             r3[jGCD+kGCD] = LCMbuffer[jGCD];
00350         k = kGCD;
00351     }
00352     else if ( kGCD > kLCM ) {
00353         for ( jGCD = 0; jGCD < kLCM; jGCD++ )
00354             r3[jGCD+kGCD] = LCMbuffer[jGCD];
00355         for ( jGCD = kLCM; jGCD < kGCD; jGCD++ )
00356             r3[jGCD+kGCD] = 0;
00357         k = kGCD;
00358     }
00359     else {
00360         for ( jGCD = kGCD; jGCD < kLCM; jGCD++ )
00361             r3[jGCD] = 0;
00362         for ( jGCD = 0; jGCD < kLCM; jGCD++ )
00363             r3[jGCD+kLCM] = LCMbuffer[jGCD];
00364         k = kLCM;
00365     }
00366     j = 2*k+1;
00367     mm = m = oldwork + oldwork[0];
00368 /*
00369     Now compose the new term
00370 */
00371     n1 = term;
00372     while ( n1 < t ) *m++ = *n1++;
00373     n1 += n1[1];
00374     n2 = oldwork+1;
00375     while ( n2 < mm ) *m++ = *n2++;
00376     while ( n1 < tstop ) *m++ = *n1++;
00377 /*
00378     And the coefficient
00379 */
00380     size = term[*term-1];
00381     size = REDLENG(size);
00382     if ( MulRat(BHEAD (UWORD *)tstop,size,(UWORD *)r3,k,
00383                                 (UWORD *)m,&size) ) goto onerror;
00384     size = INCLENG(size);
00385     k = size < 0 ? -size: size;
00386     m[k-1] = size;
```

```
00387     m += k;
00388     *mm = (WORD)(m - mm);
00389     AT.WorkPointer = m;
00390     if ( Generator(BHEAD mm,level) ) goto onerror;
00391     AT.WorkPointer = oldwork;
00392     if ( fromwhere == 0 ) {
00393         if ( d->factors ) M_free(d->factors,"Dollar factors");
00394         M_free(d,"Dollar in FactorIn");
00395     }
00396     NumberFree(GCDbuffer,"FactorIn"); NumberFree(GCDbuffer2,"FactorIn");
00397     NumberFree(LCMbuffer,"FactorIn"); NumberFree(LCMb,"FactorIn"); NumberFree(LCMc,"FactorIn");
00398     return(0);
00399 onerror:
00400     AT.WorkPointer = oldwork;
00401     MLOCK(ErrorMessageLock);
00402     MesCall("FactorIn");
00403     MUNLOCK(ErrorMessageLock);
00404     NumberFree(GCDbuffer,"FactorIn"); NumberFree(GCDbuffer2,"FactorIn");
00405     NumberFree(LCMbuffer,"FactorIn"); NumberFree(LCMb,"FactorIn"); NumberFree(LCMc,"FactorIn");
00406     return(-1);
00407 }
00408
00409 /*
00410     #] FactorIn :
00411     #[ FactorInExpr :
00412
00413     This routine tests for a factor in an active or hidden expression.
00414
00415     The factor from the last call is stored in a cache.
00416     Main problem here is whether the cache is global or private to each thread.
00417     A global cache gives most likely the same traffic jam for the computation
00418     as a local cache. The local cache may stay valid longer.
00419     In the future we may make it such that threads can look at the cache
00420     of the others, or even whether a certain factor is under construction.
00421 */
00422
00423 int FactorInExpr(PHEAD WORD *term, WORD level)
00424 {
00425     GETBIDENTITY
00426     WORD *t, *tstop, *m, *oldwork, *mstop, *n1, *n2, *n3, *n4, *n1stop, *n2stop;
00427     WORD *r1, *r2, *r3, *r4, j, k, kGCD, kGCD2, kLCM, jGCD, kkLCM, jLCM, size, sign;
00428     WORD *newterm, expr = 0;
00429     WORD olddeferflag = AR.DeferFlag, oldgetfile = AR.GetFile, oldhold = AR.KeptInHold;
00430     WORD newgetfile, newhold;
00431     int i;
00432     EXPRESSIONS e;
00433     FILEHANDLE *file = 0;
00434     POSITION position, oldposition, startposition;
00435     WORD *oldcpointer = AR.CompressPointer;
00436     UWORD *GCDbuffer, *GCDbuffer2, *LCMbuffer, *LCMb, *LCMc;
00437     GCDbuffer = NumberMalloc("FactorInExpr"); GCDbuffer2 = NumberMalloc("FactorInExpr");
00438     LCMbuffer = NumberMalloc("FactorInExpr"); LCMb = NumberMalloc("FactorInExpr"); LCMc =
    NumberMalloc("FactorInExpr");
00439     t = term; GETSTOP(t,tstop); t++;
00440     while ( t < tstop ) {
00441         if ( *t == FACTORIN && t[1] == FUNHEAD+2 && t[FUNHEAD] == -EXPRESSION ) {
00442             expr = t[FUNHEAD+1];
00443             break;
00444         }
00445         t += t[1];
00446     }
00447     if ( t >= tstop ) {
00448 /* INTERNAL_ERROR_EXCL_START */
00449         MLOCK(ErrorMessageLock);
00450         MesPrint("!>Internal error. Could not find proper factorin_ function.");
00451         MUNLOCK(ErrorMessageLock);
00452         NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
00453         NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
    NumberFree(LCMc,"FactorInExpr");
00454         return(-1);
00455 /* INTERNAL_ERROR_EXCL_STOP */
00456     }
00457     oldwork = AT.WorkPointer;
00458     if ( AT.previousEfactor && ( expr == AT.previousEfactor[0] ) ) {
00459 /*
00460         We have a hit in the cache. Construct the new term.
00461         At the moment AT.previousEfactor[1] is reserved for future flags
00462 */
00463         goto PutTheFactor;
00464     }
00465 /*
00466     No hit. We have to do the work. We start with constructing the factor
00467     in the WorkSpace. Later we will move it to the cache.
00468     Finally we will jump to PutTheFactor.
00469 */
00470     e = Expressions + expr;
00471     switch ( e->status ) {
```

```
00472         case LOCALEXPRESSION:
00473         case SKIPLEXPRESSION:
00474         case DROPLEXPRESSION:
00475         case GLOBALEXPRESSION:
00476         case SKIPGEXPRESSION:
00477         case DROPGEXPRESSION:
00478 /*
00479             Expression is to be found in the input Scratch file.
00480             Set the file handle and the position.
00481             The rest is done by GetTerm.
00482 */
00483             newhold = 0;
00484             newgetfile = 1;
00485             file = AR.infile;
00486             break;
00487         case HIDDENLEXPRESSION:
00488         case HIDDENGEXPRESSION:
00489         case HIDELEXPRESSION:
00490         case HIDEGEXPRESSION:
00491         case DROPHLEXPRESSION:
00492         case DROPHGEXPRESSION:
00493         case UNHIDELEXPRESSION:
00494         case UNHIDEGEXPRESSION:
00495 /*
00496             Expression is to be found in the hide Scratch file.
00497             Set the file handle and the position.
00498             The rest is done by GetTerm.
00499 */
00500             newhold = 0;
00501             newgetfile = 2;
00502             file = AR.hidefile;
00503             break;
00504         case STOREDEXPRESSION:
00505 /*
00506             This is an 'illegal' case
00507 */
00508             MLOCK(ErrorMessageLock);
00509             MesPrint("Error: factorin_ cannot determine factors in stored expressions.");
00510             MUNLOCK(ErrorMessageLock);
00511             NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
00512             NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
      NumberFree(LCMc,"FactorInExpr");
00513             return(-1);
00514         case DROPPEDEXPRESSION:
00515 /*
00516             We replace the function by 1.
00517 */
00518             m = oldwork; n1 = term;
00519             while ( n1 < t ) *m++ = *n1++;
00520             n1 = t + t[1]; tstop = term + *term;
00521             while ( n1 < tstop ) *m++ = *n1++;
00522             *oldwork = m - oldwork;
00523             AT.WorkPointer = m;
00524             if ( Generator(BHEAD oldwork,level) ) {
00525                 NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
00526                 NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
      NumberFree(LCMc,"FactorInExpr");
00527                 return(-1);
00528             }
00529             AT.WorkPointer = oldwork;
00530             NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
00531             NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
      NumberFree(LCMc,"FactorInExpr");
00532             return(0);
00533         default:
00534             MLOCK(ErrorMessageLock);
00535             MesPrint("Error: Illegal expression in factorinexpr.");
00536             MUNLOCK(ErrorMessageLock);
00537             NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
00538             NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
      NumberFree(LCMc,"FactorInExpr");
00539             return(-1);
00540     }
00541 /*
00542     Before we start with the file we set the buffers for the coefficient
00543     For the coefficient we have to determine the LCM of the denominators
00544     and the GCD of the numerators.
00545 */
00546     position = AS.OldOnFile[expr];
00547     AR.DeferFlag = 0; AR.KeptInHold = newhold; AR.GetFile = newgetfile;
00548     SeekScratch(file,&oldposition);
00549     SetScratch(file,&position);
00550     if ( GetTerm(BHEAD oldwork) <= 0 ) {
00551 /* INTERNAL_ERROR_EXCL_START */
00552         MLOCK(ErrorMessageLock);
00553         MesPrint("!>(5) Expression %d has problems in scratchfile",expr);
00554         MUNLOCK(ErrorMessageLock);
```

```
00555         NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
00556         NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
    NumberFree(LCMc,"FactorInExpr");
00557         return(-1);
00558 /* INTERNAL_ERROR_EXCL_STOP */
00559     }
00560     SeekScratch(file,&startposition);
00561     SeekScratch(file,&position);
00562 /*
00563     Load the first term in the workspace
00564 */
00565     if ( GetTerm(BHEAD oldwork) == 0 ) {
00566         SetScratch(file,&oldposition); /* We still need this until Processor is clean */
00567         AR.DeferFlag = olddeferflag;
00568         oldwork[0] = 4; oldwork[1] = 1; oldwork[2] = 1; oldwork[3] = 3;
00569         goto Complete;
00570     }
00571     SeekScratch(file,&position);
00572     AR.DeferFlag = olddeferflag; AR.KeptInHold = oldhold; AR.GetFile = oldgetfile;
00573
00574     r2 = m = oldwork + *oldwork;
00575     j = m[-1];
00576     m -= ABS(j);
00577     *oldwork = (WORD)(m-oldwork);
00578     AT.WorkPointer = newterm = mstop = m;
00579 /*
00580     Now take the coefficient of the first term to load up the LCM and the GCD
00581 */
00582     r3 = m;
00583     k = REDLENG(j);
00584     if ( k < 0 ) { k = -k; sign = -1; }
00585     else { sign = 1; }
00586     while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00587     for ( kGCD = 0; kGCD < k; kGCD++ ) GCDbuffer[kGCD] = r3[kGCD];
00588     k = REDLENG(j);
00589     if ( k < 0 ) k = -k;
00590     r3 += k;
00591     while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00592     for ( kLCM = 0; kLCM < k; kLCM++ ) LCMbuffer[kLCM] = r3[kLCM];
00593 /*
00594     The copy and the coefficient are in place. Now search through the terms.
00595 */
00596     for (;;) {
00597         AR.DeferFlag = 0; AR.KeptInHold = newhold; AR.GetFile = newgetfile;
00598         SetScratch(file,&position);
00599         size = GetTerm(BHEAD newterm);
00600         SeekScratch(file,&position);
00601         AR.DeferFlag = olddeferflag; AR.KeptInHold = oldhold; AR.GetFile = oldgetfile;
00602         if ( size == 0 ) break;
00603         m = oldwork+1;
00604         r2 = newterm + *newterm;
00605         r2 -= ABS(r2[-1]);
00606         r1 = newterm+1;
00607         while ( m < mstop ) {
00608             while ( r1 < r2 ) {
00609                 if ( *r1 != *m ) {
00610                     r1 += r1[1]; continue;
00611                 }
00612 /*
00613                     Now the various cases
00614                 #[ SYMBOL :
00615 */
00616                 if ( *m == SYMBOL ) {
00617                     n1 = m+2; n1stop = m+m[1];
00618                     n2stop = r1+r1[1];
00619                     while ( n1 < n1stop ) {
00620                         n2 = r1+2;
00621                         while ( n2 < n2stop ) {
00622                             if ( *n1 != *n2 ) { n2 += 2; continue; }
00623                             if ( n1[1] > 0 ) {
00624                                 if ( n2[1] < 0 ) { n2 += 2; continue; }
00625                                 if ( n2[1] < n1[1] ) n1[1] = n2[1];
00626                             }
00627                             else {
00628                                 if ( n2[1] > 0 ) { n2 += 2; continue; }
00629                                 if ( n2[1] > n1[1] ) n1[1] = n2[1];
00630                             }
00631                             break;
00632                         }
00633                         if ( n2 >= n2stop ) {   /* scratch symbol */
00634                             if ( m[1] == 4 ) goto scratch;
00635                             m[1] -= 2;
00636                             n3 = n1; n4 = n1+2;
00637                             while ( n4 < mstop ) *n3++ = *n4++;
00638                             *oldwork = n3 - oldwork;
00639                             mstop -= 2; n1stop -= 2;
00640                             continue;
```

```
00641                         }
00642                         n1 += 2;
00643                     }
00644                     break;
00645                 }
00646 /*
00647             #] SYMBOL :
00648             #[ DOTPRODUCT :
00649 */
00650                 else if ( *m == DOTPRODUCT ) {
00651                     n1 = m+2; n1stop = m+m[1];
00652                     n2stop = r1+r1[1];
00653                     while ( n1 < n1stop ) {
00654                         n2 = r1+2;
00655                         while ( n2 < n2stop ) {
00656                             if ( *n1 != *n2 || n1[1] != n2[1] ) { n2 += 3; continue; }
00657                             if ( n1[2] > 0 ) {
00658                                 if ( n2[2] < 0 ) { n2 += 3; continue; }
00659                                 if ( n2[2] < n1[2] ) n1[2] = n2[2];
00660                             }
00661                             else {
00662                                 if ( n2[2] > 0 ) { n2 += 3; continue; }
00663                                 if ( n2[2] > n1[2] ) n1[2] = n2[2];
00664                             }
00665                             break;
00666                         }
00667                         if ( n2 >= n2stop ) {   /* scratch dotproduct */
00668                             if ( m[1] == 5 ) goto scratch;
00669                             m[1] -= 3;
00670                             n3 = n1; n4 = n1+3;
00671                             while ( n4 < mstop ) *n3++ = *n4++;
00672                             *oldwork = n3 - oldwork;
00673                             mstop -= 3; n1stop -= 3;
00674                             continue;
00675                         }
00676                         n1 += 3;
00677                     }
00678                     break;
00679                 }
00680 /*
00681             #] DOTPRODUCT :
00682             #[ VECTOR :
00683 */
00684                 else if ( *m == VECTOR ) {
00685 /*
00686                     Here we have to be careful if there is more than
00687                     one of the same
00688 */
00689                     n1 = m+2; n1stop = m+m[1];
00690                     n2 = r1+2;n2stop = r1+r1[1];
00691                     while ( n1 < n1stop ) {
00692                         while ( n2 < n2stop ) {
00693                             if ( *n1 == *n2 && n1[1] == n2[1] ) {
00694                                 n2 += 2; goto nextn1;
00695                             }
00696                             n2 += 2;
00697                         }
00698                         if ( n2 >= n2stop ) {   /* scratch vector */
00699                             if ( m[1] == 4 ) goto scratch;
00700                             m[1] -= 2;
00701                             n3 = n1; n4 = n1+2;
00702                             while ( n4 < mstop ) *n3++ = *n4++;
00703                             *oldwork = n3 - oldwork;
00704                             mstop -= 2; n1stop -= 2;
00705                             continue;
00706                         }
00707                         n2 = r1+2;
00708 nextn1:                 n1 += 2;
00709                     }
00710                     break;
00711                 }
00712 /*
00713             #] VECTOR :
00714             #[ REMAINDER :
00715 */
00716                 else {
00717 /*
00718                     Again: watch for multiple occurrences of the same object
00719 */
00720                     if ( m[1] != r1[1] ) { r1 += r1[1]; continue; }
00721                     for ( j = 2; j < m[1]; j++ ) {
00722                         if ( m[j] != r1[j] ) break;
00723                     }
00724                     if ( j < m[1] ) { r1 += r1[1]; continue; }
00725                     r1 += r1[1]; /* to restart at the next potential match */
00726                     goto nextm;  /* match */
00727                 }
```

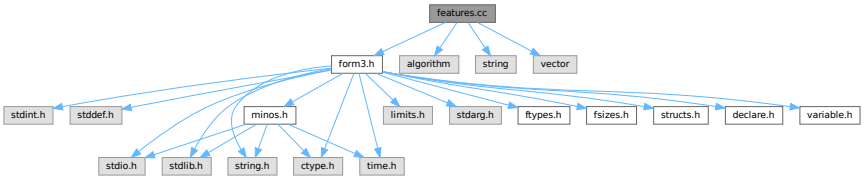
```
00728 /*
00729             #] REMAINDER :
00730 */
00731             }
00732             if ( r1 >= r2 ) { /* no factor! */
00733 scratch:;
00734                 r3 = m + m[1]; r4 = m;
00735                 while ( r3 < mstop ) *r4++ = *r3++;
00736                 *oldwork = r4 - oldwork;
00737                 if ( *oldwork == 1 ) goto nofactor;
00738                 mstop = r4;
00739                 r1 = newterm + 1;
00740                 continue;
00741             }
00742             r1 = newterm + 1;
00743 nextm:      m += m[1];
00744         }
00745 nofactor:;
00746 /*
00747         Now the coefficient
00748 */
00749         r2 = newterm + *newterm;
00750         j = r2[-1];
00751         r3 = r2 - ABS(j);
00752         k = REDLENG(j);
00753         if ( k < 0 ) k = -k;
00754         while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00755         if ( ( ( GCDbuffer[0] == 1 ) && ( kGCD == 1 ) ) ) {
00756 /*
00757             GCD is already 1
00758 */
00759         }
00760         else if ( ( ( k != 1 ) || ( r3[0] != 1 ) ) ) {
00761             if ( GcdLong(BHEAD GCDbuffer,kGCD,(UWORD *)r3,k,GCDbuffer2,&kGCD2) ) {
00762                 goto onerror;
00763             }
00764             kGCD = kGCD2;
00765             for ( i = 0; i < kGCD; i++ ) GCDbuffer[i] = GCDbuffer2[i];
00766         }
00767         else {
00768             kGCD = 1; GCDbuffer[0] = 1;
00769         }
00770         k = REDLENG(j);
00771         if ( k < 0 ) k = -k;
00772         r3 += k;
00773         while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00774         if ( ( ( LCMbuffer[0] == 1 ) && ( kLCM == 1 ) ) ) {
00775             for ( kLCM = 0; kLCM < k; kLCM++ )
00776                 LCMbuffer[kLCM] = r3[kLCM];
00777         }
00778         else if ( ( k != 1 ) || ( r3[0] != 1 ) ) {
00779             if ( GcdLong(BHEAD LCMbuffer,kLCM,(UWORD *)r3,k,LCMb,&kkLCM) ) {
00780                 goto onerror;
00781             }
00782             DivLong((UWORD *)r3,k,LCMb,kkLCM,LCMb,&kkLCM,LCMc,&jLCM);
00783             MulLong(LCMbuffer,kLCM,LCMb,kkLCM,LCMc,&jLCM);
00784             for ( kLCM = 0; kLCM < jLCM; kLCM++ )
00785                 LCMbuffer[kLCM] = LCMc[kLCM];
00786         }
00787         else {} /* LCM doesn't change */
00788     }
00789     SetScratch(file,&oldposition); /* Needed until Processor is thread safe */
00790     AR.DeferFlag = olddeferflag;
00791 /*
00792     Now put the term together in oldwork: GCD/LCM
00793     We have already the algebraic contents there.
00794 */
00795     r3 = (WORD *)(GCDbuffer);
00796     r4 = (WORD *)(LCMbuffer);
00797     r1 = oldwork + *oldwork;
00798     if ( kGCD == kLCM ) {
00799         for ( jGCD = 0; jGCD < kGCD; jGCD++ ) *r1++ = *r3++;
00800         for ( jGCD = 0; jGCD < kGCD; jGCD++ ) *r1++ = *r4++;
00801         k = 2*kGCD+1;
00802     }
00803     else if ( kGCD > kLCM ) {
00804         for ( jGCD = 0; jGCD < kGCD; jGCD++ ) *r1++ = *r3++;
00805         for ( jGCD = 0; jGCD < kLCM; jGCD++ ) *r1++ = *r4++;
00806         for ( jGCD = kLCM; jGCD < kGCD; jGCD++ ) *r1++ = 0;
00807         k = 2*kGCD+1;
00808     }
00809     else {
00810         for ( jGCD = 0; jGCD < kGCD; jGCD++ ) *r1++ = *r3++;
00811         for ( jGCD = kGCD; jGCD < kLCM; jGCD++ ) *r1++ = 0;
00812         for ( jGCD = 0; jGCD < kLCM; jGCD++ ) *r1++ = *r4++;
00813         k = 2*kLCM+1;
00814     }
```

```
00815     if ( sign < 0 ) *r1++ = -k;
00816     else *r1++ = k;
00817     *oldwork = (WORD)(r1-oldwork);
00818 /*
00819     Now put the new term in the cache
00820 */
00821 Complete:;
00822     if ( AT.previousEfactor ) M_free(AT.previousEfactor,"Efactor cache");
00823     AT.previousEfactor = (WORD *)Malloc1((*oldwork+2)*sizeof(WORD),"Efactor cache");
00824     AT.previousEfactor[0] = expr;
00825     r1 = oldwork; r2 = AT.previousEfactor + 2; k = *oldwork;
00826     NCOPY(r2,r1,k)
00827     AT.previousEfactor[1] = 0;
00828 /*
00829     Now we construct the new term in the workspace.
00830 */
00831 PutTheFactor:;
00832     if ( AT.WorkPointer + AT.previousEfactor[2] >= AT.WorkTop ) {
00833         MLOCK(ErrorMessageLock);
00834         MesWork();
00835         MesPrint("Called from factorin_");
00836         MUNLOCK(ErrorMessageLock);
00837         NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
00838         NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
      NumberFree(LCMc,"FactorInExpr");
00839         return(-1);
00840     }
00841     n1 = oldwork; n2 = term; while ( n2 < t ) *n1++ = *n2++;
00842     n2 = AT.previousEfactor+2; GETSTOP(n2,n2stop); n3 = n2 + *n2;
00843     n2++; while ( n2 < n2stop ) *n1++ = *n2++;
00844     n2 = t + t[1]; while ( n2 < tstop ) *n1++ = *n2++;
00845     size = term[*term-1];
00846     size = REDLENG(size);
00847     k = n3[-1]; k = REDLENG(k);
00848     if ( MulRat(BHEAD (UWORD *)tstop,size,(UWORD *)n2stop,k,
00849                               (UWORD *)n1,&size) ) goto onerror;
00850     size = INCLENG(size);
00851     k = size < 0 ? -size: size;
00852        n1 += k; n1[-1] = size;
00853     *oldwork = n1 - oldwork;
00854     AT.WorkPointer = n1;
00855     if ( Generator(BHEAD oldwork,level) ) {
00856         NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
00857         NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
      NumberFree(LCMc,"FactorInExpr");
00858         return(-1);
00859     }
00860     AT.WorkPointer = oldwork;
00861     AR.CompressPointer = oldcpointer;
00862     NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
00863     NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
      NumberFree(LCMc,"FactorInExpr");
00864     return(0);
00865 onerror:
00866     AT.WorkPointer = oldwork;
00867     AR.CompressPointer = oldcpointer;
00868     MLOCK(ErrorMessageLock);
00869     MesCall("FactorInExpr");
00870     MUNLOCK(ErrorMessageLock);
00871     NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
00872     NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
      NumberFree(LCMc,"FactorInExpr");
00873     return(-1);
00874 }
00875
00876 /*
00877     #] FactorInExpr :
00878 */
```

4.37 features.cc File Reference

```
#include "form3.h"
#include <algorithm>
#include <string>
#include <vector>
```

Include dependency graph for features.cc:



Functions

- void PrintFeatureList (void)

4.37.1 Detailed Description

Utility code for printing available features.

Definition in file features.cc.

4.37.2 Function Documentation

4.37.2.1 PrintFeatureList()

```
void PrintFeatureList (
            void  )
```

Prints the list of available features.

Definition at line 131 of file features.cc.

4.38 features.cc

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
```

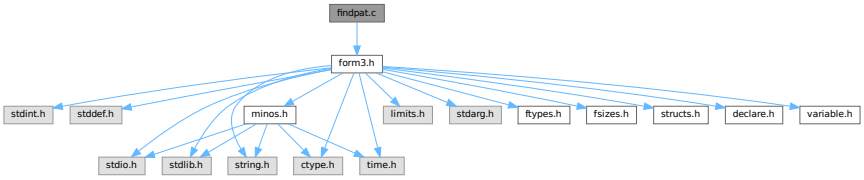
```
00026  *
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
00031 //  #[ Includes :
00032
00033 extern "C" {
00034 #include "form3.h"
00035 }
00036
00037 #include <algorithm>
00038 #include <string>
00039 #include <vector>
00040
00041 #ifdef WITHFLINT
00042 #include <flint/flint.h>
00043 #endif
00044
00045 #ifdef WITHGMP
00046 #include <gmp.h>
00047 #endif
00048
00049 #ifdef WITHMPFR
00050 #include <mpfr.h>
00051 #endif
00052
00053 #ifdef WITHZLIB
00054 #include <zlib.h>
00055 #endif
00056
00057 #ifdef WITHZSTD
00058 #include <zstd.h>
00059 #endif
00060
00061 //  #] Includes :
00062 //  #[ PrintFeatureList :
00063
00069 static void PrintMulticolumn(const std::vector<std::string>& lines)
00070 {
00071     const std::size_t n_lines = lines.size();
00072
00073     if ( n_lines == 0 ) return;
00074
00075     const std::size_t line_len = AC.LineLength > 0 ? AC.LineLength : 79;
00076     constexpr std::size_t max_cols = 6;
00077     constexpr std::size_t col_spacing = 2;
00078
00079     std::vector<std::size_t> col_lens(max_cols);
00080
00081     for ( std::size_t n_cols = max_cols; n_cols >= 2; n_cols-- ) {
00082         const std::size_t n_rows = (n_lines - 1) / n_cols + 1;
00083
00084         // Check whether `n_cols' columns fit within the line length.
00085         // Columns are filled top-to-bottom, then left-to-right.
00086
00087         std::fill(col_lens.begin(),col_lens.begin()+n_cols,0);
00088
00089         std::size_t total_len = 0;
00090         for ( std::size_t  j = 0; j < n_cols; j++ ) {
00091             for ( std::size_t i = 0; i < n_rows; i++ ) {
00092                 const std::size_t k = i + j * n_rows;
00093                 if ( k >= n_lines ) break;
00094                 col_lens[j] = std::max(col_lens[j],lines[k].size());
00095             }
00096             total_len += col_lens[j];
00097         }
00098         total_len += (n_cols - 1) * col_spacing;
00099
00100         if ( total_len > line_len ) continue;
00101
00102         // Output using `n_cols' columns.
00103
00104         std::string line;
00105         line.reserve(line_len);
00106         for ( std::size_t i = 0; i < n_rows; i++ ) {
00107             line.clear();
00108             for ( std::size_t j = 0; j < n_cols; j++ ) {
00109                 const std::size_t k = i + j * n_rows;
00110                 if ( k >= n_lines ) break;
00111                 line += lines[k];
00112                 if ( j < n_cols - 1 ) {
00113                     line += std::string(col_lens[j]-lines[k].size()+col_spacing,' ');
00114                 }
00115             }
00116             MesPrint("%s",line.c_str());
00117         }
```

```
00118         return;
00119     }
00120
00121     // Fallback to a single column.
00122
00123     for ( const auto& f : lines ) {
00124         MesPrint("%s",f.c_str());
00125     }
00126 }
00127
00131 void PrintFeatureList(void)
00132 {
00133     std::vector<std::string> feature_list = {
00134
00135 #ifdef ENABLE_BACKTRACE
00136         "+backtrace",
00137 #else
00138         "-backtrace",
00139 #endif
00140
00141 #ifdef DEBUGGING
00142         "+debugging",
00143 #else
00144         "-debugging",
00145 #endif
00146
00147 #ifdef WITHFLINT
00148         "+flint=" + std::string(flint_version),
00149 #else
00150         "-flint",
00151 #endif
00152
00153 #ifdef WITHFLOAT
00154         "+float",
00155 #else
00156         "-float",
00157 #endif
00158
00159 #ifdef WITHGMP
00160         "+gmp=" + std::string(gmp_version),
00161 #else
00162         "-gmp",
00163 #endif
00164
00165 #ifdef WITHMPFR
00166         "+mpfr=" + std::string(mpfr_get_version()),
00167 #else
00168         "-mpfr",
00169 #endif
00170
00171 #ifdef WITHMPI
00172         "+mpi",
00173 #else
00174         "-mpi",
00175 #endif
00176
00177 #ifdef UNIX
00178         "+posix",
00179 #else
00180         "-posix",
00181 #endif
00182
00183 #ifdef WITHPTHREADS
00184         "+pthreads",
00185 #else
00186         "-pthreads",
00187 #endif
00188
00189 #ifdef WINDOWS
00190         "+windows",
00191 #else
00192         "-windows",
00193 #endif
00194
00195 #ifdef WITHZLIB
00196         "+zlib=" + std::string(zlibVersion()),
00197 #else
00198         "-zlib",
00199 #endif
00200
00201 #ifdef WITHZSTD
00202         "+zstd=" + std::string(ZSTD_versionString()),
00203 #else
00204         "-zstd",
00205 #endif
00206
00207     };
```

```
00208
00209     PrintMulticolumn(feature_list);
00210 }
00211
00212 //  #] PrintFeatureList :
```

4.39 findpat.c File Reference

`#include "form3.h"`
Include dependency graph for findpat.c:



Functions

- int FindOnly (PHEAD WORD ∗term, WORD ∗pattern)
- int FindOnce (PHEAD WORD ∗term, WORD ∗pattern)
- WORD FindMulti (PHEAD WORD ∗term, WORD ∗pattern)
- int FindRest (PHEAD WORD ∗term, WORD ∗pattern)

4.39.1 Detailed Description

Pattern matching of symbols and dotproducts. There are various routines because of the options in the id-statements like once, only, multi and many. These are among the oldest routines in FORM and that can be noticed, because the interplay with the function matching is not complete. When we match functions and halfway we fail we can backtrack properly. With the symbols, the dotproducts and the vectors (in pattern.c) there is no proper backtracking. Hence the routines here need still quite some work or may even have to be rewritten.

Definition in file findpat.c.

4.39.2 Function Documentation

4.39.2.1 FindOnly()

```
int FindOnly (
            PHEAD WORD * term,
            WORD * pattern )
```

Definition at line 61 of file findpat.c.

4.39.2.2 FindOnce()

```
int FindOnce (
              PHEAD WORD * term,
              WORD * pattern )
```

Definition at line 419 of file findpat.c.

4.39.2.3 FindMulti()

```
WORD FindMulti (
              PHEAD WORD * term,
              WORD * pattern )
```

Definition at line 954 of file findpat.c.

4.39.2.4 FindRest()

```
int FindRest (
              PHEAD WORD * term,
              WORD * pattern )
```

Definition at line 1070 of file findpat.c.

4.40 findpat.c

Go to the documentation of this file.

```
00001
00013 /* #[ License : */
00014 /*
00015  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00016  *   When using this file you are requested to refer to the publication
00017  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00018  *   This is considered a matter of courtesy as the development was paid
00019  *   for by FOM the Dutch physics granting agency and we would like to
00020  *   be able to track its scientific use to convince FOM of its value
00021  *   for the community.
00022  *
00023  *   This file is part of FORM.
00024  *
00025  *   FORM is free software: you can redistribute it and/or modify it under the
00026  *   terms of the GNU General Public License as published by the Free Software
00027  *   Foundation, either version 3 of the License, or (at your option) any later
00028  *   version.
00029  *
00030  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00031  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00032  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00033  *   details.
00034  *
00035  *   You should have received a copy of the GNU General Public License along
00036  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00037  */
00038 /* #] License : */
00039 /*
00040     #[ Includes : findpat.c
00041 */
00042
00043 #include "form3.h"
00044
00045 /*
00046     #] Includes :
00047     #[ Patterns :
00048         #[ FindOnly :           WORD FindOnly(term,pattern)
```

```
00049
00050     The current version doesn't scan function arguments yet. 10-Apr-1988
00051
00052     This routine searches for an exact match. This means in particular:
00053     1:  x^# must match exactly.
00054     2:  x^n? must have a single value for n that cannot be adapted.
00055
00056     When setp != 0 it points to a collection of sets
00057     A match can occur only if no object will be left that belongs
00058     to any of these sets.
00059 */
00060
00061 int FindOnly(PHEAD WORD *term, WORD *pattern)
00062 {
00063     GETBIDENTITY
00064     WORD *t, *m;
00065     WORD *tstop, *mstop;
00066     WORD *xstop, *ystop, *setp = AN.ForFindOnly;
00067     WORD n, nt, *p, nq;
00068     WORD older[NORMSIZE], *q, newval1, newval2, newval3;
00069     AN.UsedOtherFind = 0;
00070     m = pattern;
00071     mstop = m + *m;
00072     m++;
00073     t = term;
00074     t += *term - 1;
00075     tstop = t - ABS(*t) + 1;
00076     t = term;
00077     t++;
00078     while ( t < tstop && *t > DOTPRODUCT ) t += t[1];
00079     while ( m < mstop && *m > DOTPRODUCT ) m += m[1];
00080     if ( m < mstop ) { do {
00081 /*
00082             #[ SYMBOLS :
00083 */
00084         if ( *m == SYMBOL ) {
00085             ystop = m + m[1];
00086             m += 2;
00087             n = 0;
00088             p = older;
00089             if ( t < tstop ) while ( *t != SYMBOL ) {
00090                 t += t[1];
00091                 if ( t >= tstop ) {
00092 OnlyZer1:
00093                     do {
00094                         if ( *m >= 2*MAXPOWER ) return(0);
00095                         if ( m[1] >= 2*MAXPOWER ) nt = m[1];
00096                         else if ( m[1] <= -2*MAXPOWER ) nt = -m[1];
00097                         else return(0);
00098                         nt -= 2*MAXPOWER;
00099                         if ( CheckWild(BHEAD nt,SYMTONUM,0,&newval3) ) return(0);
00100                         AddWild(BHEAD nt,SYMTONUM,0);
00101                         m += 2;
00102                     } while ( m < ystop );
00103                     goto EndLoop;
00104                 }
00105             }
00106             else goto OnlyZer1;
00107             xstop = t + t[1];
00108             t += 2;
00109             do {
00110                 if ( *m == *t && t < xstop ) {
00111                     if ( m[1] == t[1] ) { m += 2; t += 2; }
00112                     else if ( m[1] >= 2*MAXPOWER ) {
00113                         nt = t[1];
00114                         nq = m[1];
00115                         goto OnlyL2;
00116                     }
00117                     else if ( m[1] <= -2*MAXPOWER ) {
00118                         nt = -t[1];
00119                         nq = -m[1];
00120 OnlyL2:                 nq -= 2*MAXPOWER;
00121                         if ( CheckWild(BHEAD nq,SYMTONUM,nt,&newval3) ) return(0);
00122                         AddWild(BHEAD nq,SYMTONUM,nt);
00123                         m += 2;
00124                         t += 2;
00125                     }
00126                     else {
00127                         *p++ = *t++; *p++ = *t++; n += 2;
00128                     }
00129                 }
00130                 else if ( *m >= 2*MAXPOWER ) {
00131                     while ( t < xstop ) { *p++ = *t++; *p++ = *t++; n += 2; }
00132                     nq = n;
00133                     p = older;
00134                     while ( nq > 0 ) {
00135                         if ( !CheckWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,*p,&newval1) ) {
```

```
00136                               if ( m[1] == p[1] ) {
00137                                   AddWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,newval1);
00138                                   break;
00139                               }
00140                               else if ( m[1] >= 2*MAXPOWER && m[1] != *m ) {
00141                                   if ( !CheckWild(BHEAD m[1]-2*MAXPOWER,SYMTONUM,p[1],&newval3) ) {
00142                                       AddWild(BHEAD m[1]-2*MAXPOWER,SYMTONUM,p[1]);
00143                                       AddWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,newval1);
00144                                       break;
00145                                   }
00146                               }
00147                               else if ( m[1] <= -2*MAXPOWER && m[1] != -(*m) ) {
00148                                   if ( !CheckWild(BHEAD -m[1]-2*MAXPOWER,SYMTONUM,-p[1],&newval3) ) {
00149                                       AddWild(BHEAD -m[1]-2*MAXPOWER,SYMTONUM,-p[1]);
00150                                       AddWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,newval1);
00151                                       break;
00152                                   }
00153                               }
00154                           }
00155                           nq -= 2;
00156                           p += 2;
00157                       }
00158                       if ( nq <= 0 ) return(0);
00159                       nq -= 2;
00160                       n -= 2;
00161                       q = p + 2;
00162                       while ( --nq >= 0 ) *p++ = *q++;
00163                       m += 2;
00164                   }
00165                   else {
00166                       if ( t >= xstop || *m < *t ) {
00167                           if ( m[1] >= 2*MAXPOWER ) nt = m[1];
00168                           else if ( m[1] <= -2*MAXPOWER ) nt = -m[1];
00169                           else return(0);
00170                           nt -= 2*MAXPOWER;
00171                           if ( CheckWild(BHEAD nt,SYMTONUM,0,&newval3) ) return(0);
00172                           AddWild(BHEAD nt,SYMTONUM,0);
00173                           m += 2;
00174                       }
00175                       else {
00176                           *p++ = *t++; *p++ = *t++; n += 2;
00177                       }
00178                   }
00179               } while ( m < ystop );
00180               if ( setp ) {
00181                   while ( t < xstop ) { *p++ = *t++; *p++ = *t++; n+= 2; }
00182                   p = older;
00183                   while ( n > 0 ) {
00184                       nq = setp[1] - 2;
00185                       m = setp + 2;
00186                       while ( --nq >= 0 ) {
00187                           if ( Sets[*m].type != CSYMBOL ) { m++; continue; }
00188                           t = SetElements + Sets[*m].first;
00189                           tstop = SetElements + Sets[*m].last;
00190                           while ( t < tstop ) {
00191                               if ( *t++ == *p ) return(0);
00192                           }
00193                           m++;
00194                       }
00195                       n -= 2;
00196                       p += 2;
00197                   }
00198               }
00199               return(1);
00200           }
00201   /*
00202               #] SYMBOLS :
00203               #[ DOTPRODUCTS :
00204   */
00205           else if ( *m == DOTPRODUCT ) {
00206               ystop = m + m[1];
00207               m += 2;
00208               n = 0;
00209               p = older;
00210               if ( t < tstop ) {
00211                   if ( *t < DOTPRODUCT ) goto OnlyZer2;
00212                   while ( *t > DOTPRODUCT ) {
00213                       t += t[1];
00214                       if ( t >= tstop || *t < DOTPRODUCT ) {
00215   OnlyZer2:
00216                           do {
00217                               if ( *m >= (AM.OffsetVector+WILDOFFSET)
00218                               || m[1] >= (AM.OffsetVector+WILDOFFSET) ) return(0);
00219                               if ( m[2] >= 2*MAXPOWER ) nq = m[2];
00220                               else if ( m[2] <= -2*MAXPOWER ) nq = -m[2];
00221                               else return(0);
00222                               nq -= 2*MAXPOWER;
```

```
00223                         if ( CheckWild(BHEAD nq,SYMTONUM,0,&newval3) ) return(0);
00224                         AddWild(BHEAD nq,SYMTONUM,0);
00225                         m += 3;
00226                     } while ( m < ystop );
00227                     goto EndLoop;
00228                 }
00229                 }
00230             }
00231             else goto OnlyZer2;
00232             xstop = t + t[1];
00233             t += 2;
00234             do {
00235                 if ( *m == *t && m[1] == t[1] && t < xstop ) {
00236                     if ( t[2] != m[2] ) {
00237                         if ( m[2] >= 2*MAXPOWER ) {
00238                             nq = m[2];
00239                             nt = t[2];
00240                         }
00241                         else if ( m[2] <= -2*MAXPOWER ) {
00242                             nq = -m[2];
00243                             nt = -t[2];
00244                         }
00245                         else return(0);
00246                         nq -= 2*MAXPOWER;
00247                         if ( CheckWild(BHEAD nq,SYMTONUM,nt,&newval3) ) return(0);
00248                         AddWild(BHEAD nq,SYMTONUM,nt);
00249                     }
00250                     t += 3; m += 3;
00251                 }
00252                 else if ( *m >= (AM.OffsetVector+WILDOFFSET) ) {
00253                     while ( t < xstop ) {
00254                         *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
00255                     }
00256                     nq = n;
00257                     p = older;
00258                     while ( nq > 0 ) {
00259                         if ( *m == m[1] ) {
00260                             if ( *p != p[1] ) goto NextInDot;
00261                         }
00262                         if ( !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*p,&newval1) &&
00263                              !CheckWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,p[1],&newval2) ) {
00264                             if ( p[2] == m[2] ) {
00265 OnlyL9:                         AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval2);
00266                                 AddWild(BHEAD *m-WILDOFFSET,VECTOVEC,newval1);
00267                                 break;
00268                             }
00269                             if ( m[2] >= 2*MAXPOWER ) {
00270                                 if ( !CheckWild(BHEAD m[2]-2*MAXPOWER,SYMTONUM,p[2],&newval3) ) {
00271                                     AddWild(BHEAD m[2]-2*MAXPOWER,SYMTONUM,newval3);
00272                                     goto OnlyL9;
00273                                 }
00274                             }
00275                             else if ( m[2] <= -2*MAXPOWER ) {
00276                                 if ( !CheckWild(BHEAD -m[2]-2*MAXPOWER,SYMTONUM,-p[2],&newval3) ) {
00277                                     AddWild(BHEAD -m[2]-2*MAXPOWER,SYMTONUM,-p[2]);
00278                                     goto OnlyL9;
00279                                 }
00280                             }
00281                         }
00282                         if ( !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,p[1],&newval1) &&
00283                              !CheckWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,*p,&newval2) ) {
00284                             if ( p[2] == m[2] ) {
00285 OnlyL10:                        AddWild(BHEAD *m-WILDOFFSET,VECTOVEC,newval1);
00286                                 AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval2);
00287                                 break;
00288                             }
00289                             if ( m[2] >= 2*MAXPOWER ) {
00290                                 if ( !CheckWild(BHEAD m[2]-2*MAXPOWER,SYMTONUM,p[2],&newval3) ) {
00291                                     AddWild(BHEAD m[2]-2*MAXPOWER,SYMTONUM,p[2]);
00292                                     goto OnlyL10;
00293                                 }
00294                             }
00295                             else if ( m[2] <= -2*MAXPOWER ) {
00296                                 if ( !CheckWild(BHEAD -m[2]-2*MAXPOWER,SYMTONUM,-p[2],&newval3) ) {
00297                                     AddWild(BHEAD -m[2]-2*MAXPOWER,SYMTONUM,-p[2]);
00298                                     goto OnlyL10;
00299                                 }
00300                             }
00301                         }
00302 NextInDot:
00303                         p += 3; nq -= 3;
00304                     }
00305                     if ( nq <= 0 ) return(0);
00306                     q = p+3;
00307                     nq -= 3;
00308                     n -= 3;
00309                     while ( --nq >= 0 ) *p++ = *q++;
```

```
00310                   m += 3;
00311               }
00312               else if ( m[1] >= (AM.OffsetVector+WILDOFFSET) ) {
00313                   while ( *m >= *t && t < xstop ) {
00314                       *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
00315                   }
00316                   nq = n;
00317                   p = older;
00318                   while ( nq > 0 ) {
00319                       if ( *m == *p && !CheckWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,p[1],&newval1) ) {
00320                           if ( p[2] == m[2] ) {
00321                               AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
00322                               break;
00323                           }
00324                           else if ( m[2] >= 2*MAXPOWER ) {
00325                               if ( !CheckWild(BHEAD m[2]-2*MAXPOWER,SYMTONUM,p[2],&newval3) ) {
00326                                   AddWild(BHEAD m[2]-2*MAXPOWER,SYMTONUM,p[2]);
00327                                   AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
00328                                   break;
00329                               }
00330                           }
00331                           else if ( m[2] <= -2*MAXPOWER ) {
00332                               if ( !CheckWild(BHEAD -m[2]-2*MAXPOWER,SYMTONUM,-p[2],&newval3) ) {
00333                                   AddWild(BHEAD -m[2]-2*MAXPOWER,SYMTONUM,-p[2]);
00334                                   AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
00335                                   break;
00336                               }
00337                           }
00338                       }
00339                       if ( *m == p[1] && !CheckWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,*p,&newval1) ) {
00340                           if ( p[2] == m[2] ) {
00341                               AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
00342                               break;
00343                           }
00344                           if ( m[2] >= 2*MAXPOWER ) {
00345                               if ( !CheckWild(BHEAD m[2]-2*MAXPOWER,SYMTONUM,p[2],&newval3) ) {
00346                                   AddWild(BHEAD m[2]-2*MAXPOWER,SYMTONUM,p[2]);
00347                                   AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
00348                                   break;
00349                               }
00350                           }
00351                           else if ( m[2] <= -2*MAXPOWER ) {
00352                               if ( !CheckWild(BHEAD -m[2]-2*MAXPOWER,SYMTONUM,-p[2],&newval3) ) {
00353                                   AddWild(BHEAD -m[2]-2*MAXPOWER,SYMTONUM,-p[2]);
00354                                   AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
00355                                   break;
00356                               }
00357                           }
00358                       }
00359                       p += 3; nq -= 3;
00360                   }
00361                   if ( nq <= 0 ) return(0);
00362                   q = p+3;
00363                   nq -= 3;
00364                   n -= 3;
00365                   while ( --nq >= 0 ) *p++ = *q++;
00366                   m += 3;
00367               }
00368               else {
00369                   if ( t >= xstop || *m < *t || ( *m == *t && m[1] < t[1] ) ) {
00370                       if ( m[2] > 2*MAXPOWER ) nt = m[2];
00371                       else if ( m[2] <= -2*MAXPOWER ) nt = -m[2];
00372                       else return(0);
00373                       nt -= 2*MAXPOWER;
00374                       if ( CheckWild(BHEAD nt,SYMTONUM,0,&newval3) ) return(0);
00375                       AddWild(BHEAD nt,SYMTONUM,0);
00376                       m += 3;
00377                   }
00378                   else {
00379                       *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
00380                   }
00381               }
00382           } while ( m < ystop );
00383           t = xstop;
00384       }
00385 /*
00386           #] DOTPRODUCTS :
00387 */
00388       else {
00389           MLOCK(ErrorMessageLock);
00390           MesPrint("Error in pattern(1)");
00391           MUNLOCK(ErrorMessageLock);
00392           Terminate(-1);
00393       }
00394 EndLoop:;
00395     } while ( m < mstop ); }
00396     if ( setp ) {
```

```
00397 /*
00398         while ( t < tstop && *t > SYMBOL ) t += t[1];
00399         if ( t < tstop && setp[1] > 2 ) return(0);
00400 */
00401                         /* There were nonempty sets */
00402                         /* Empty sets are rejected by the compiler */
00403     }
00404     return(1);
00405 }
00406
00407 /*
00408         #] FindOnly :
00409         #[ FindOnce :           WORD FindOnce(term,pattern)
00410
00411     Searches for a single match in term. The difference with multi
00412     lies mainly in the fact that here functions may occur.
00413     The functions have not been implemented yet. (10-Apr-1988)
00414     Wildcard powers are adjustable. The value closer to zero is taken.
00415     Positive and negative gives (o surprise) zero.
00416
00417 */
00418
00419 int FindOnce(PHEAD WORD *term, WORD *pattern)
00420 {
00421     GETBIDENTITY
00422     WORD *t, *m;
00423     WORD *tstop, *mstop;
00424     WORD *xstop, *ystop;
00425     WORD n, nt, *p, nq, mt, ch;
00426     WORD older[2*NORMSIZE], *q, newval1, newval2, newval3;
00427     AN.UsedOtherFind = 0;
00428     m = pattern;
00429     mstop = m + *m;
00430     m++;
00431     t = term;
00432     t += *term - 1;
00433     tstop = t - ABS(*t) + 1;
00434     t = term;
00435     t++;
00436     while ( t < tstop && *t > DOTPRODUCT ) t += t[1];
00437     while ( m < mstop && *m > DOTPRODUCT ) m += m[1];
00438     if ( m < mstop ) { do {
00439 /*
00440             #[ SYMBOLS :
00441 */
00442         if ( *m == SYMBOL ) {
00443             ystop = m + m[1];
00444             m += 2;
00445             n = 0;
00446             p = older;
00447             if ( t < tstop ) while ( *t != SYMBOL ) {
00448                 t += t[1];
00449                 if ( t >= tstop ) {
00450 TryZero:
00451                     do {
00452                         if ( *m >= 2*MAXPOWER ) return(0);
00453                         if ( m[1] >= 2*MAXPOWER ) nt = m[1];
00454                         else if ( m[1] <= -2*MAXPOWER ) nt = -m[1];
00455                         else return(0);
00456                         nt -= 2*MAXPOWER;
00457                         if ( ( ch = CheckWild(BHEAD nt,SYMTONUM,0,&newval3) ) != 0 ) {
00458                             if ( ch > 1 ) return(0);
00459                             if ( AN.oldtype != SYMTONUM ) return(0);
00460                             if ( *AN.MaskPointer == 2 ) return(0);
00461                         }
00462                         AddWild(BHEAD nt,SYMTONUM,0);
00463                         m += 2;
00464                     } while ( m < ystop );
00465                     goto EndLoop;
00466                 }
00467             }
00468             else goto TryZero;
00469             xstop = t + t[1];
00470             t += 2;
00471             do {
00472                 if ( *m == *t && t < xstop ) {
00473                     nt = t[1];
00474                     mt = m[1];
00475                     if ( ( mt > 0 && mt <= nt ) ||
00476                          ( mt < 0 && mt >= nt ) ) { m += 2; t += 2; }
00477                     else if ( mt >= 2*MAXPOWER ) goto OnceL2;
00478                     else if ( mt <= -2*MAXPOWER ) {
00479                         nt = -nt;
00480                         mt = -mt;
00481 OnceL2:                 mt -= 2*MAXPOWER;
00482                         if ( ( ch = CheckWild(BHEAD mt,SYMTONUM,nt,&newval3) ) != 0 ) {
00483                             if ( ch > 1 ) return(0);
```

```
00484                               if ( AN.oldtype != SYMTONUM ) return(0);
00485                               if ( AN.oldvalue <= 0 ) {
00486                                   if ( nt < AN.oldvalue ) nt = AN.oldvalue;
00487                                   else {
00488                                       if ( *AN.MaskPointer == 2 ) return(0);
00489                                       if ( nt > 0 ) nt = 0;
00490                                   }
00491                               }
00492                               if ( AN.oldvalue >= 0 ) {
00493                                   if ( nt > AN.oldvalue ) nt = AN.oldvalue;
00494                                   else {
00495                                       if ( *AN.MaskPointer == 2 ) return(0);
00496                                       if ( nt < 0 ) nt = 0;
00497                                   }
00498                               }
00499                           }
00500                           AddWild(BHEAD mt,SYMTONUM,nt);
00501                           m += 2;
00502                           t += 2;
00503                       }
00504                       else {
00505                           *p++ = *t++; *p++ = *t++; n += 2;
00506                       }
00507                   }
00508                   else if ( *m >= 2*MAXPOWER ) {
00509                       while ( t < xstop ) { *p++ = *t++; *p++ = *t++; n += 2; }
00510                       nq = n;
00511                       p = older;
00512                       while ( nq > 0 ) {
00513                           nt = p[1];
00514                           if ( !CheckWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,*p,&newval1) ) {
00515                               mt = m[1];
00516                               if ( ( mt > 0 && mt <= nt ) ||
00517                                    ( mt < 0 && mt >= nt ) ) {
00518                                   AddWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,newval1);
00519                                   break;
00520                               }
00521                               else if ( mt >= 2*MAXPOWER && mt != *m ) {
00522 OnceL4a:                           mt -= 2*MAXPOWER;
00523                                   if ( ( ch = CheckWild(BHEAD mt,SYMTONUM,nt,&newval3) ) != 0 ) {
00524                                       if ( ch > 1 ) return(0);
00525                                       if ( AN.oldtype == SYMTONUM ) {
00526                                           if ( AN.oldvalue >= 0 ) {
00527                                               if ( nt > AN.oldvalue ) nt = AN.oldvalue;
00528                                               else {
00529                                                   if ( *AN.MaskPointer == 2 ) return(0);
00530                                                   if ( nt < 0 ) nt = 0;
00531                                               }
00532                                           }
00533                                           else {
00534                                               if ( nt < AN.oldvalue ) nt = AN.oldvalue;
00535                                               else {
00536                                                   if ( *AN.MaskPointer == 2 ) return(0);
00537                                                   if ( nt > 0 ) nt = 0;
00538                                               }
00539                                           }
00540                                           AddWild(BHEAD mt,SYMTONUM,nt);
00541                                           AddWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,newval1);
00542                                           break;
00543                                       }
00544                                   }
00545                                   else {
00546                                       AddWild(BHEAD mt,SYMTONUM,nt);
00547                                       AddWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,newval1);
00548                                       break;
00549                                   }
00550                               }
00551                               else if ( mt <= -2*MAXPOWER && mt != -(*m) ) {
00552                                   nt = -nt;
00553                                   mt = -mt;
00554                                   goto OnceL4a;
00555                               }
00556                           }
00557                           nq -= 2;
00558                           p += 2;
00559                       }
00560                       if ( nq <= 0 ) return(0);
00561                       nq -= 2;
00562                       n -= 2;
00563                       q = p + 2;
00564                       while ( --nq >= 0 ) *p++ = *q++;
00565                       m += 2;
00566                   }
00567                   else {
00568                       if ( t >= xstop || *m < *t ) {
00569                           if ( m[1] >= 2*MAXPOWER ) nt = m[1];
00570                           else if ( m[1] <= -2*MAXPOWER ) nt = -m[1];
```

```
00571                           else return(0);
00572                           nt -= 2*MAXPOWER;
00573                           if ( ( ch = CheckWild(BHEAD nt,SYMTONUM,0,&newval3) ) != 0 ) {
00574                               if ( ch > 1 ) return(0);
00575                               if ( AN.oldtype != SYMTONUM ) return(0);
00576                               if ( *AN.MaskPointer == 2 ) return(0);
00577                           }
00578                           AddWild(BHEAD nt,SYMTONUM,0);
00579                           m += 2;
00580                       }
00581                       else {
00582                           *p++ = *t++; *p++ = *t++; n += 2;
00583                       }
00584                   }
00585               } while ( m < ystop );
00586           }
00587 /*
00588               #] SYMBOLS :
00589               #[ DOTPRODUCTS :
00590 */
00591           else if ( *m == DOTPRODUCT ) {
00592               ystop = m + m[1];
00593               m += 2;
00594               n = 0;
00595               p = older;
00596               if ( t < tstop ) {
00597                   if ( *t < DOTPRODUCT ) goto OnceOp;
00598                   while ( *t > DOTPRODUCT ) {
00599                   t += t[1];
00600                   if ( t >= tstop || *t < DOTPRODUCT ) {
00601 OnceOp:
00602                       do {
00603                           if ( *m >= (AM.OffsetVector+WILDOFFSET)
00604                           || m[1] >= (AM.OffsetVector+WILDOFFSET) ) return(0);
00605                           if ( m[2] >= 2*MAXPOWER ) {
00606                               nq = m[2] - 2*MAXPOWER;
00607                           }
00608                           else if ( m[2] <= -2*MAXPOWER ) {
00609                               nq = -m[2] - 2*MAXPOWER;
00610                           }
00611                           else return(0);
00612                           if ( CheckWild(BHEAD nq,SYMTONUM,(WORD)0,&newval3) ) {
00613                               if ( AN.oldtype != SYMTONUM ) return(0);
00614                               if ( *AN.MaskPointer == 2 ) return(0);
00615                           }
00616                           AddWild(BHEAD nq,SYMTONUM,(WORD)0);
00617                           m += 3;
00618                       } while ( m < ystop );
00619                       goto EndLoop;
00620                   }
00621                   }
00622               }
00623               else goto OnceOp;
00624               xstop = t + t[1];
00625               t += 2;
00626               do {
00627                   if ( *m == *t && m[1] == t[1] && t < xstop ) {
00628                       nt = t[2];
00629                       mt = m[2];
00630 /*
00631                       if ( ( nt > 0 && nt < mt ) ||
00632                            ( nt < 0 && nt > mt ) ) {
00633                           if ( mt <= -2*MAXPOWER ) {
00634                               mt = -mt;
00635                               nt = -nt;
00636                           }
00637                           else if ( mt < 2*MAXPOWER ) return(0);
00638                           mt -= 2*MAXPOWER;
00639                           if ( CheckWild(BHEAD mt,SYMTONUM,nt,&newval3) ) {
00640                               if ( AN.oldtype != SYMTONUM ) return(0);
00641                               if ( AN.oldvalue <= 0 ) {
00642                                   if ( nt < AN.oldvalue ) nt = AN.oldvalue;
00643                                   else {
00644                                       if ( *AN.MaskPointer == 2 ) return(0);
00645                                       if ( nt > 0 ) nt = 0;
00646                                   }
00647                               }
00648                               if ( AN.oldvalue >= 0 ) {
00649                                   if ( nt > AN.oldvalue ) nt = AN.oldvalue;
00650                                   else {
00651                                       if ( *AN.MaskPointer == 2 ) return(0);
00652                                       if ( nt < 0 ) nt = 0;
00653                                   }
00654                               }
00655                           }
00656                           AddWild(BHEAD mt,SYMTONUM,nt);
00657                           m += 3; t += 3;
```

```
00658                     }
00659                     else if ( ( nt > 0 && nt >= mt && mt > -2*MAXPOWER )
00660                     || ( nt < 0 && nt <= mt && mt < 2*MAXPOWER ) ) {
00661                         m += 3; t += 3;
00662                     }
00663 */
00664                     if ( ( mt > 0 && mt <= nt ) ||
00665                          ( mt < 0 && mt >= nt ) ) { m += 3; t += 3; }
00666                     else if ( mt >= 2*MAXPOWER ) goto OnceL7;
00667                     else if ( mt <= -2*MAXPOWER ) {
00668                         nt = -nt;
00669                         mt = -mt;
00670 OnceL7:                 mt -= 2*MAXPOWER;
00671                         if ( CheckWild(BHEAD mt,SYMTONUM,nt,&newval3) ) {
00672                             if ( AN.oldtype != SYMTONUM ) return(0);
00673                             if ( AN.oldvalue <= 0 ) {
00674                                 if ( nt < AN.oldvalue ) nt = AN.oldvalue;
00675                                 else {
00676                                     if ( *AN.MaskPointer == 2 ) return(0);
00677                                     if ( nt > 0 ) nt = 0;
00678                                 }
00679                             }
00680                             if ( AN.oldvalue >= 0 ) {
00681                                 if ( nt > AN.oldvalue ) nt = AN.oldvalue;
00682                                 else {
00683                                     if ( *AN.MaskPointer == 2 ) return(0);
00684                                     if ( nt < 0 ) nt = 0;
00685                                 }
00686                             }
00687                         }
00688                         AddWild(BHEAD mt,SYMTONUM,nt);
00689                         m += 3;
00690                         t += 3;
00691                     }
00692                     else {
00693                         *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
00694                     }
00695                 }
00696                 else if ( *m >= (AM.OffsetVector+WILDOFFSET) ) {
00697                     while ( t < xstop ) {
00698                         *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
00699                     }
00700                     nq = n;
00701                     p = older;
00702                     while ( nq > 0 ) {
00703                         if ( *m == m[1] ) {
00704                             if ( *p != p[1] ) goto NextInDot;
00705                         }
00706                         if ( !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*p,&newval1) &&
00707                              !CheckWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,p[1],&newval2) ) {
00708                             nt = p[2];
00709                             mt = m[2];
00710                             if ( ( mt > 0 && nt >= mt ) ||
00711                                  ( mt < 0 && nt <= mt ) ) {
00712 OnceL9:                         AddWild(BHEAD *m-WILDOFFSET,VECTOVEC,newval1);
00713                                 AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval2);
00714                                 break;
00715                             }
00716                             if ( mt >= 2*MAXPOWER ) {
00717 OnceL9a:                        mt -= 2*MAXPOWER;
00718                                 if ( CheckWild(BHEAD mt,SYMTONUM,nt,&newval3) ) {
00719                                     if ( AN.oldtype == SYMTONUM ) {
00720                                         if ( AN.oldvalue >= 0 ) {
00721                                             if ( nt > AN.oldvalue ) nt = AN.oldvalue;
00722                                             else {
00723                                                 if ( *AN.MaskPointer == 2 ) return(0);
00724                                                 if ( nt < 0 ) nt = 0;
00725                                             }
00726                                         }
00727                                         else {
00728                                             if ( nt < AN.oldvalue ) nt = AN.oldvalue;
00729                                             else {
00730                                                 if ( *AN.MaskPointer == 2 ) return(0);
00731                                                 if ( nt > 0 ) nt = 0;
00732                                             }
00733                                         }
00734                                         AddWild(BHEAD mt,SYMTONUM,nt);
00735                                         goto OnceL9;
00736                                     }
00737                                 }
00738                                 else {
00739                                     AddWild(BHEAD mt,SYMTONUM,nt);
00740                                     goto OnceL9;
00741                                 }
00742                             }
00743                             else if ( mt <= -2*MAXPOWER ) {
00744                                 mt = -mt;
```

```
00745                                   nt = -nt;
00746                                   goto OnceL9a;
00747                               }
00748                           }
00749                           if ( !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,p[1],&newval1) &&
00750                                !CheckWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,*p,&newval2) ) {
00751                               nt = p[2];
00752                               mt = m[2];
00753                               if ( ( mt > 0 && nt >= mt ) ||
00754                                    ( mt < 0 && nt <= mt ) ) {
00755 OnceL10:                         AddWild(BHEAD *m-WILDOFFSET,VECTOVEC,newval1);
00756                                   AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval2);
00757                                   break;
00758                               }
00759                               if ( mt >= 2*MAXPOWER ) {
00760 OnceL10a:                         mt -= 2*MAXPOWER;
00761                                   if ( CheckWild(BHEAD mt,SYMTONUM,nt,&newval3) ) {
00762                                       if ( AN.oldtype == SYMTONUM ) {
00763                                           if ( AN.oldvalue >= 0 ) {
00764                                               if ( nt > AN.oldvalue ) nt = AN.oldvalue;
00765                                               else {
00766                                                   if ( *AN.MaskPointer == 2 ) return(0);
00767                                                   if ( nt < 0 ) nt = 0;
00768                                               }
00769                                           }
00770                                           else {
00771                                               if ( nt < AN.oldvalue ) nt = AN.oldvalue;
00772                                               else {
00773                                                   if ( *AN.MaskPointer == 2 ) return(0);
00774                                                   if ( nt > 0 ) nt = 0;
00775                                               }
00776                                           }
00777                                           AddWild(BHEAD mt,SYMTONUM,nt);
00778                                           goto OnceL10;
00779                                       }
00780                                   }
00781                                   else {
00782                                       AddWild(BHEAD mt,SYMTONUM,nt);
00783                                       goto OnceL10;
00784                                   }
00785                               }
00786                               else if ( mt <= -2*MAXPOWER ) {
00787                                   mt = -mt;
00788                                   nt = -nt;
00789                                   goto OnceL10a;
00790                               }
00791                           }
00792 NextInDot:
00793                       p += 3; nq -= 3;
00794                   }
00795                   if ( nq <= 0 ) return(0);
00796                   else {
00797                       q = p+3;
00798                       nq -= 3;
00799                       n -= 3;
00800                       while ( --nq >= 0 ) *p++ = *q++;
00801                   }
00802                   m += 3;
00803               }
00804               else if ( m[1] >= (AM.OffsetVector+WILDOFFSET) ) {
00805                   while ( *m >= *t && t < xstop ) {
00806                       *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
00807                   }
00808                   nq = n;
00809                   p = older;
00810                   while ( nq > 0 ) {
00811                       if ( *m == *p && !CheckWild(BHEAD m[1]-WILDOFFSET,
00812                       VECTOVEC,p[1],&newval1) ) {
00813                           nt = p[2];
00814                           mt = m[2];
00815                           if ( ( mt > 0 && nt >= mt ) ||
00816                                ( mt < 0 && nt <= mt ) ) {
00817                               AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
00818                               break;
00819                           }
00820                           else if ( mt >= 2*MAXPOWER ) {
00821 OnceL7a:                       mt -= 2*MAXPOWER;
00822                               if ( CheckWild(BHEAD mt,SYMTONUM,nt,&newval3) ) {
00823                                   if ( AN.oldtype == SYMTONUM ) {
00824                                       if ( AN.oldvalue >= 0 ) {
00825                                           if ( nt > AN.oldvalue ) nt = AN.oldvalue;
00826                                           else {
00827                                               if ( *AN.MaskPointer == 2 ) return(0);
00828                                               if ( nt < 0 ) nt = 0;
00829                                           }
00830                                       }
00831                                       else {
```

```
00832                                           if ( nt < AN.oldvalue ) nt = AN.oldvalue;
00833                                           else {
00834                                               if ( *AN.MaskPointer == 2 ) return(0);
00835                                               if ( nt > 0 ) nt = 0;
00836                                           }
00837                                       }
00838                                       AddWild(BHEAD mt,SYMTONUM,nt);
00839                                       AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
00840                                       break;
00841                                   }
00842                               }
00843                               else {
00844                                   AddWild(BHEAD mt,SYMTONUM,nt);
00845                                   AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
00846                                   break;
00847                               }
00848                           }
00849                           else if ( mt <= -2*MAXPOWER ) {
00850                               mt = -mt;
00851                               nt = -nt;
00852                               goto OnceL7a;
00853                           }
00854                       }
00855                       if ( *m == p[1] && !CheckWild(BHEAD m[1]-WILDOFFSET,
00856                       VECTOVEC,*p,&newval1) ) {
00857                           nt = p[2];
00858                           mt = m[2];
00859                           if ( ( mt > 0 && nt >= mt ) ||
00860                                ( mt < 0 && nt <= mt ) ) {
00861                               AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
00862                               break;
00863                           }
00864                           if ( mt >= 2*MAXPOWER ) {
00865 OnceL8a:                       mt -= 2*MAXPOWER;
00866                               if ( CheckWild(BHEAD mt,SYMTONUM,nt,&newval3) ) {
00867                                   if ( AN.oldtype == SYMTONUM ) {
00868                                       if ( AN.oldvalue >= 0 ) {
00869                                           if ( nt > AN.oldvalue ) nt = AN.oldvalue;
00870                                           else {
00871                                               if ( *AN.MaskPointer == 2 ) return(0);
00872                                               if ( nt < 0 ) nt = 0;
00873                                           }
00874                                       }
00875                                       else {
00876                                           if ( nt < AN.oldvalue ) nt = AN.oldvalue;
00877                                           else {
00878                                               if ( *AN.MaskPointer == 2 ) return(0);
00879                                               if ( nt > 0 ) nt = 0;
00880                                           }
00881                                       }
00882                                       AddWild(BHEAD mt,SYMTONUM,nt);
00883                                       AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
00884                                       break;
00885                                   }
00886                               }
00887                               else {
00888                                   AddWild(BHEAD mt,SYMTONUM,nt);
00889                                   AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
00890                                   break;
00891                               }
00892                           }
00893                           else if ( mt < -2*MAXPOWER ) {
00894                               mt = -mt;
00895                               nt = -nt;
00896                               goto OnceL8a;
00897                           }
00898                       }
00899                       p += 3; nq -= 3;
00900                   }
00901                   if ( nq <= 0 ) return(0);
00902                   q = p+3;
00903                   nq -= 3;
00904                   n -= 3;
00905                   while ( --nq >= 0 ) *p++ = *q++;
00906                   m += 3;
00907               }
00908               else {
00909                   if ( t >= xstop || *m < *t || ( *m == *t && m[1] < t[1] ) ) {
00910                       if ( m[2] >= 2*MAXPOWER ) nt = m[2];
00911                       else if ( m[2] <= -2*MAXPOWER ) nt = -m[2];
00912                       else return(0);
00913                       nt -= 2*MAXPOWER;
00914                       if ( CheckWild(BHEAD nt,SYMTONUM,0,&newval3) ) {
00915                           if ( AN.oldtype != SYMTONUM ) return(0);
00916                           if ( *AN.MaskPointer == 2 ) return(0);
00917                       }
00918                       AddWild(BHEAD nt,SYMTONUM,0);
```

```
00919                       m += 3;
00920                   }
00921                   else {
00922                       *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
00923                   }
00924               }
00925           } while ( m < ystop );
00926           t = xstop;
00927       }
00928 /*
00929           #] DOTPRODUCTS :
00930 */
00931       else {
00932           MLOCK(ErrorMessageLock);
00933           MesPrint("Error in pattern(2)");
00934           MUNLOCK(ErrorMessageLock);
00935           Terminate(-1);
00936       }
00937 EndLoop:;
00938   } while ( m < mstop ); }
00939   else {
00940       return(-1);
00941   }
00942   return(1);
00943 }
00944
00945 /*
00946       #] FindOnce :
00947       #[ FindMulti :          WORD FindMulti(term,pattern)
00948
00949   Note that multi cannot deal with wildcards. Those patterns revert
00950   to many which gives subsequent calls to once.
00951
00952 */
00953
00954 WORD FindMulti(PHEAD WORD *term, WORD *pattern)
00955 {
00956   GETBIDENTITY
00957   WORD *t, *m, *p;
00958   WORD *tstop, *mstop;
00959   WORD *xstop, *ystop;
00960   WORD mt, power, n, nq;
00961   WORD older[2*NORMSIZE], *q, newval1;
00962   AN.UsedOtherFind = 0;
00963   m = pattern;
00964   mstop = m + *m;
00965   m++;
00966   t = term;
00967   t += *term - 1;
00968   tstop = t - ABS(*t) + 1;
00969   t = term;
00970   t++;
00971   while ( t < tstop && *t > DOTPRODUCT ) t += t[1];
00972   while ( m < mstop && *m > DOTPRODUCT ) m += m[1];
00973   power = -1;             /* No power yet */
00974   if ( m < mstop ) { do {
00975 /*
00976           #[ SYMBOLS :
00977 */
00978       if ( *m == SYMBOL ) {
00979           ystop = m + m[1];
00980           m += 2;
00981           if ( t >= tstop ) return(0);
00982           while ( *t != SYMBOL ) { t += t[1]; if ( t >= tstop ) return(0); }
00983           xstop = t + t[1];
00984           t += 2;
00985           p = older;
00986           n = 0;
00987           do {
00988               if ( *m >= 2*MAXPOWER ) {
00989                   while ( t < xstop ) { *p++ = *t++; *p++ = *t++; n += 2; }
00990                   nq = n;
00991                   p = older;
00992                   while ( nq > 0 ) {
00993                       if ( !CheckWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,*p,&newval1) ) {
00994                           mt = p[1]/m[1];
00995                           if ( mt > 0 ) {
00996                               if ( power < 0 || mt < power ) power = mt;
00997                               AddWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,newval1);
00998                               break;
00999                           }
01000                       }
01001                       nq -= 2;
01002                       p += 2;
01003                   }
01004                   if ( nq <= 0 ) return(0);
01005                   nq -= 2;
```

```
01006                       n -= 2;
01007                       q = p + 2;
01008                       while ( --nq >= 0 ) *p++ = *q++;
01009                       m += 2;
01010                   }
01011                   else if ( t >= xstop ) return(0);
01012                   else if ( *m == *t ) {
01013                       if ( ( mt = t[1]/m[1] ) <= 0 ) return(0);
01014                       if ( power < 0 || mt < power ) power = mt;
01015                       m += 2;
01016                       t += 2;
01017                   }
01018                   else if ( *m < *t ) return(0);
01019                   else { *p++ = *t++; *p++ = *t++; n += 2; }
01020               } while ( m < ystop );
01021           }
01022 /*
01023               #] SYMBOLS :
01024               #[ DOTPRODUCTS :
01025 */
01026           else if ( *m == DOTPRODUCT ) {
01027               ystop = m + m[1];
01028               m += 2;
01029               if ( t >= tstop ) return(0);
01030               while ( *t != DOTPRODUCT ) { t += t[1]; if ( t >= tstop ) return(0); }
01031               xstop = t + t[1];
01032               t += 2;
01033               do {
01034                   if ( t >= xstop ) return(0);
01035                   if ( *t == *m ) {
01036                       if ( t[1] == m[1] ) {
01037                           if ( ( mt = t[2]/m[2] ) <= 0 ) return(0);
01038                           if ( power < 0 || mt < power ) power = mt;
01039                           m += 3;
01040                       }
01041                       else if ( t[1] > m[1] ) return(0);
01042                   }
01043                   else if ( *t > *m ) return(0);
01044                   t += 3;
01045               } while ( m < ystop );
01046               t = xstop;
01047           }
01048 /*
01049               #] DOTPRODUCTS :
01050 */
01051           else {
01052               MLOCK(ErrorMessageLock);
01053               MesPrint("Error in pattern(3)");
01054               MUNLOCK(ErrorMessageLock);
01055               Terminate(-1);
01056           }
01057       } while ( m < mstop ); }
01058       if ( power < 0 ) power = 0;
01059       return(power);
01060 }
01061
01062 /*
01063           #] FindMulti :
01064           #[ FindRest :           WORD FindRest(term,pattern)
01065
01066       This routine scans for anything but dotproducts and symbols.
01067
01068 */
01069
01070 int FindRest(PHEAD WORD *term, WORD *pattern)
01071 {
01072       GETBIDENTITY
01073       WORD *t, *m, *tt, wild, regular;
01074       WORD *tstop, *mstop;
01075       WORD *xstop, *ystop;
01076       WORD n, *p, nq;
01077       WORD older[NORMSIZE], *q, newval1, newval2;
01078       int i, ntwa;
01079       AN.UsedOtherFind = 0;
01080       AN.findTerm = term; AN.findPattern = pattern;
01081       m = AN.WildValue;
01082       i = (m[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
01083       ntwa = 0;
01084       while ( i > 0 ) {
01085           if ( m[0] == ARGTOARG ) ntwa++;
01086           m += m[1];
01087           i--;
01088       }
01089       t = term;
01090       t += *term - 1;
01091       tstop = t - ABS(*t) + 1;
01092       t = term;
```

```
01093     t++; p = t;
01094     while ( t < tstop && *t > DOTPRODUCT ) t += t[1];
01095     tstop = t;
01096     t = p;
01097     m = pattern;
01098     mstop = m + *m;
01099     m++;
01100     p = m;
01101     while ( m < mstop && *m > DOTPRODUCT ) m += m[1];
01102     mstop = m;
01103     m = p;
01104     if ( m < mstop ) {
01105     do {
01106 /*
01107             #[ FUNCTIONS :
01108 */
01109         if ( *m >= FUNCTION ) {
01110             if ( *mstop > 5 && !MatchIsPossible(pattern,term) ) return(0);
01111             ystop = m;
01112             n = 0;
01113             do {
01114                 m += m[1]; n++;
01115             } while ( m < mstop && *m >= FUNCTION );
01116             AT.WorkPointer += n;
01117             while ( t < tstop && *t == SUBEXPRESSION ) t += t[1];
01118             tt = xstop = t;
01119             nq = 0;
01120             while ( t < tstop && ( *t >= FUNCTION || *t == SUBEXPRESSION ) ) {
01121                 if ( *t != SUBEXPRESSION ) {
01122                     nq++;
01123                     if ( functions[*t-FUNCTION].commute ) tt = t + t[1];
01124                 }
01125                 t += t[1];
01126             }
01127             if ( nq < n ) return(0);
01128             AN.terstart = term;
01129             AN.terstop = t;
01130             AN.terfirstcomm = tt;
01131             AN.patstop = m;
01132             AN.NumTotWildArgs = ntwa;
01133             if ( !ScanFunctions(BHEAD ystop,xstop,0) ) return(0);
01134         }
01135 /*
01136             #] FUNCTIONS :
01137             #[ VECTORS :
01138 */
01139         else if ( *m == VECTOR ) {
01140             while ( t < tstop && *t != VECTOR ) t += t[1];
01141             if ( t >= tstop ) return(0);
01142             xstop = t + t[1];
01143             ystop = m + m[1];
01144             t += 2;
01145             m += 2;
01146             n = 0;
01147             p = older;
01148             do {
01149                 if ( *m == *t && m[1] == t[1] && t < xstop ) {
01150                     m += 2; t += 2;
01151                 }
01152                 else if ( *m >= (AM.OffsetVector+WILDOFFSET) ) {
01153                     if ( t < xstop ) {
01154                         p = older + n;
01155                         do { *p++ = *t++; n++; } while ( t < xstop );
01156                     }
01157                     p = older;
01158                     nq = n;
01159                     if ( ( m[1] < (AM.OffsetIndex+WILDOFFSET) )
01160                     || ( m[1] >= (AM.OffsetIndex+2*WILDOFFSET) ) ) {
01161                         while ( nq > 0 ) {
01162                             if ( m[1] == p[1] ) {
01163                                 if ( !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*p,&newval1) ) {
01164 RestL11:                            AddWild(BHEAD *m-WILDOFFSET,VECTOVEC,newval1);
01165                                     break;
01166                                 }
01167                             }
01168                             p += 2;
01169                             nq -= 2;
01170                         }
01171                     }
01172                     else {      /* Double wildcard */
01173                         while ( nq > 0 ) {
01174                             if ( !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*p,&newval1) &&
01175                                  !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,p[1],&newval2) ) {
01176                                 AddWild(BHEAD m[1]-WILDOFFSET,INDTOIND,newval2);
01177                                 goto RestL11;
01178                             }
01179                             p += 2;
```

```
01180                           nq -= 2;
01181                       }
01182                   }
01183                   if ( nq > 0 ) {
01184                       nq -= 2; q = p + 2; n -= 2;
01185                       while ( --nq >= 0 ) *p++ = *q++;
01186                   }
01187                   else return(0);
01188                   m += 2;
01189               }
01190               else if ( ( *m <= *t )
01191               && ( m[1] >= (AM.OffsetIndex + WILDOFFSET) )
01192               && ( m[1] <  (AM.OffsetIndex + 2*WILDOFFSET) ) ) {
01193                   if ( *m == *t && t < xstop ) {
01194                       p = older;
01195                       p += n;
01196                       *p++ = *t++;
01197                       *p++ = *t++;
01198                       n += 2;
01199                   }
01200                   p = older;
01201                   nq = n;
01202                   while ( nq > 0 ) {
01203                       if ( *m == *p ) {
01204                           if ( !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,p[1],&newval1) ) {
01205                               AddWild(BHEAD m[1]-WILDOFFSET,INDTOIND,newval1);
01206                               break;
01207                           }
01208                       }
01209                       p += 2;
01210                       nq -= 2;
01211                   }
01212                   if ( nq > 0 ) {
01213                       nq -= 2; q = p + 2; n -= 2;
01214                       while ( --nq >= 0 ) *p++ = *q++;
01215                   }
01216                   else return(0);
01217                   m += 2;
01218               }
01219               else {
01220                   if ( t >= xstop ) return(0);
01221                   *p++ = *t++; *p++ = *t++; n += 2;
01222               }
01223           } while ( m < ystop );
01224       }
01225 /*
01226           #] VECTORS :
01227           #[ INDICES :
01228 */
01229       else if ( *m == INDEX ) {
01230 /*
01231           This needs only to say that there is a match, after matching
01232           a 'wildcard'. This has to be prepared in TestMatch. The C->rhs
01233           should provide the replacement inside the prototype!
01234           Next question: id,p=q/2+r/2
01235 */
01236           while ( *t != INDEX ) { t += t[1]; if ( t >= tstop ) return(0); }
01237           xstop = t + t[1];
01238           ystop = m + m[1];
01239           t += 2;
01240           m += 2;
01241           n = 0;
01242           p = older;
01243           do {
01244               if ( *m == *t && t < xstop && m < ystop ) {
01245                   t++; m++;
01246               }
01247               else if ( ( *m >= (AM.OffsetIndex+WILDOFFSET) )
01248               && ( *m < (AM.OffsetIndex+2*WILDOFFSET) ) ) {
01249                   while ( t < xstop ) {
01250                       *p++ = *t++; n++;
01251                   }
01252                   if ( !n ) return(0);
01253                   nq = n;
01254                   q = older;
01255                   do {
01256                       if ( !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,*q,&newval1) ) {
01257                           AddWild(BHEAD *m-WILDOFFSET,INDTOIND,newval1);
01258                           break;
01259                       }
01260                       q++;
01261                       nq--;
01262                   } while ( nq > 0 );
01263                   if ( nq <= 0 ) return (0);
01264                   n--;
01265                   nq--;
01266                   p = q + 1;
```

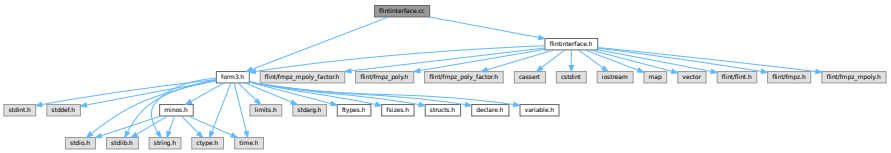
```
01267                     while ( nq > 0 ) { *q++ = *p++; nq--; }
01268                     p--;
01269                     m++;
01270                 }
01271                 else if ( ( *m >= (AM.OffsetVector+WILDOFFSET) )
01272                 && ( *m < (AM.OffsetVector+2*WILDOFFSET) ) ) {
01273                     while ( t < xstop ) {
01274                         *p++ = *t++; n++;
01275                     }
01276                     if ( !n ) return(0);
01277                     nq = n;
01278                     q = older;
01279                     do {
01280                         if ( !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*q,&newval1) ) {
01281                             AddWild(BHEAD *m-WILDOFFSET,VECTOVEC,newval1);
01282                             break;
01283                         }
01284                         q++;
01285                         nq--;
01286                     } while ( nq > 0 );
01287                     if ( nq <= 0 ) return (0);
01288                     n--;
01289                     nq--;
01290                     p = q + 1;
01291                     while ( nq > 0 ) { *q++ = *p++; nq--; }
01292                     p--;
01293                     m++;
01294                 }
01295                 else {
01296                     if ( t >= xstop ) return(0);
01297                     *p++ = *t++; n++;
01298                 }
01299             } while ( m < ystop );
01300 
01301 /*
01302             return(0);
01303 */
01304         }
01305 /*
01306             #] INDICES :
01307             #[ DELTAS :
01308 */
01309         else if ( *m == DELTA ) {
01310             while ( *t != DELTA ) { t += t[1]; if ( t >= tstop ) return(0); }
01311             xstop = t + t[1];
01312             ystop = m + m[1];
01313             t += 2;
01314             m += 2;
01315             n = 0;
01316             p = older;
01317             do {
01318                 if ( *t == *m && t[1] == m[1] && t < xstop ) {
01319                     m += 2;
01320                     t += 2;
01321                 }
01322                 else if ( ( *m >= (AM.OffsetIndex+WILDOFFSET) )
01323                 && ( *m < (AM.OffsetIndex+2*WILDOFFSET) )
01324                 && ( m[1] >= (AM.OffsetIndex+WILDOFFSET) )
01325                 && ( m[1] < (AM.OffsetIndex+2*WILDOFFSET) ) ) { /* Two dummies */
01326                     while ( t < xstop ) {
01327                         *p++ = *t++; *p++ = *t++; n += 2;
01328                     }
01329                     if ( !n ) return(0);
01330                     nq = n;
01331                     q = older;
01332                     do {
01333                         if ( !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,*q,&newval1) &&
01334                              !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,q[1],&newval2) ) {
01335                             AddWild(BHEAD *m-WILDOFFSET,INDTOIND,newval1);
01336                             AddWild(BHEAD m[1]-WILDOFFSET,INDTOIND,newval2);
01337                             break;
01338                         }
01339                         if ( !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,q[1],&newval1) &&
01340                              !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,*q,&newval2) ) {
01341                             AddWild(BHEAD *m-WILDOFFSET,INDTOIND,newval1);
01342                             AddWild(BHEAD m[1]-WILDOFFSET,INDTOIND,newval2);
01343                             break;
01344                         }
01345                         q += 2;
01346                         nq -= 2;
01347                     } while ( nq > 0 );
01348                     if ( nq <= 0 ) return(0);
01349                     n -= 2;
01350                     nq -= 2;
01351                     p = q + 2;
01352                     while ( nq > 0 ) { *q++ = *p++; nq--; }
01353                     p -= 2;
```

```
01354                     m += 2;
01355                 }
01356                 else if ( ( m[1] >= (AM.OffsetIndex+WILDOFFSET) )
01357                 && ( m[1] < (AM.OffsetIndex+2*WILDOFFSET) ) ) {
01358                     wild = m[1]; regular = *m;
01359 OneWild:
01360                     while ( ( regular == *t || regular == t[1] ) && t < xstop ) {
01361                         *p++ = *t++; *p++ = *t++; n += 2;
01362                     }
01363                     if ( !n ) return(0);
01364                     nq = n;
01365                     q = older;
01366                     do {
01367                         if ( regular == *q && !CheckWild(BHEAD wild-WILDOFFSET,INDTOIND,q[1],&newval1)
     ) {
01368                             AddWild(BHEAD wild-WILDOFFSET,INDTOIND,newval1);
01369                             break;
01370                         }
01371                         if ( regular == q[1] && !CheckWild(BHEAD wild-WILDOFFSET,INDTOIND,*q,&newval1)
     ) {
01372                             AddWild(BHEAD wild-WILDOFFSET,INDTOIND,newval1);
01373                             break;
01374                         }
01375                         q += 2;
01376                         nq -= 2;
01377                     } while ( nq > 0 );
01378                     if ( nq <= 0 ) return(0);
01379                     n -= 2;
01380                     nq -= 2;
01381                     p = q + 2;
01382                     while ( nq > 0 ) { *q++ = *p++; nq--; }
01383                     p -= 2;
01384                     m += 2;
01385                 }
01386                 else if ( ( *m >= (AM.OffsetIndex+WILDOFFSET) )
01387                 && ( *m < (AM.OffsetIndex+2*WILDOFFSET) ) ) {
01388                     wild = *m; regular = m[1];
01389                     goto OneWild;
01390                 }
01391                 else {
01392                     if ( t >= tstop || *m < *t || ( *m == *t && m[1] < t[1] ) )
01393                         return(0);
01394                     *p++ = *t++; *p++ = *t++; n += 2;
01395                 }
01396             } while ( m < ystop );
01397         }
01398 /*
01399             #] DELTAS :
01400 */
01401         else {
01402 /* INTERNAL_ERROR_EXCL_START */
01403             MLOCK(ErrorMessageLock);
01404             MesPrint("!>Pattern not yet implemented");
01405             MUNLOCK(ErrorMessageLock);
01406             Terminate(-1);
01407 /* INTERNAL_ERROR_EXCL_STOP */
01408         }
01409     } while ( m < mstop );
01410     return(1);
01411     }
01412     else return(-1);
01413 }
01414
01415 /*
01416         #] FindRest :
01417     #] Patterns :
01418 */
```

4.41 flintinterface.cc File Reference

```
#include "form3.h"
#include "flintinterface.h"
```

Include dependency graph for flintinterface.cc:



Macros

- #define UNIVARIATE_DENSITY_THR 0.02f
- #define IFW(x) { if (write) {x;} }

4.41.1 Detailed Description

Contains functions which interface with FLINT functions to perform arithmetic with uni- and multi-variate polynomials and perform factorization.

Definition in file flintinterface.cc.

4.41.2 Macro Definition Documentation

4.41.2.1 UNIVARIATE_DENSITY_THR

```
#define UNIVARIATE_DENSITY_THR 0.02f
```

Definition at line 71 of file flintinterface.cc.

4.41.2.2 IFW

```
#define IFW(
              x ) { if ( write ) {x;} }
```

Definition at line 1856 of file flintinterface.cc.

4.42 flintinterface.cc

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032
00033 #ifdef HAVE_CONFIG_H
00034 #ifndef CONFIG_H_INCLUDED
00035 #define CONFIG_H_INCLUDED
00036 #include <config.h>
00037 #endif
00038 #endif
00039
00040 extern "C" {
00041 #include "form3.h"
00042 }
00043
00044 #include "flintinterface.h"
00045
00046 /*
00047     #[ Types
00048  * Use fixed-size integer types (int32_t, uint32_t, int64_t, uint64_t) except when we refer
00049  * specifically to FORM term data, when we use WORD. This code has only been tested on 64bit
00050  * systems where WORDs are int32_t, and some functions (fmpz_get_form, fmpz_set_form) require this
00051  * to be the case. Enforce this:
00052  */
00053 static_assert(sizeof(WORD) == sizeof(int32_t), "flint interface expects 32bit WORD");
00054 static_assert(sizeof(UWORD) == sizeof(uint32_t), "flint interface expects 32bit WORD");
00055 static_assert(BITSINWORD == 32, "flint interface expects 32bit WORD");
00056 static_assert(sizeof(ulong) == sizeof(uint64_t), "flint interface expects ulong is uint64_t");
00057 static_assert(sizeof(slong) == sizeof(int64_t), "flint interface expects slong is int64_t");
00058
00059 /*
00060  * Flint functions take arguments or return values which may be "slong" or "ulong" in its
00061  * documentation, and these are int64_t and uint64_t respectively.
00062  *
00063     #] Types
00064 */
00065
00066 /*
00067  * FLINT's univariate poly has a dense representation. For sufficiently sparse polynomials it is
00068  * faster to use mpoly instead, which is sparse. For a density <= this threshold we switch, where
00069  * the density defined as is "number of terms" / "maximum degree".
00070  */
00071 #define UNIVARIATE_DENSITY_THR 0.02f
00072
00073 /*
00074     #[ flint::cleanup :
00075 */
00076 void flint::cleanup(void) {
00077     flint_cleanup();
00078 }
00079 /*
00080     #] flint::cleanup :
00081     #[ flint::cleanup_master :
00082 */
00083 void flint::cleanup_master(void) {
00084     flint_cleanup_master();
00085 }
00086 /*
```

```
00087     #] flint::cleanup_master :
00088
00089     #[ flint::divmod_mpoly :
00090 */
00091 WORD* flint::divmod_mpoly(PHEAD const WORD *a, const WORD *b, const bool return_rem,
00092     const WORD must_fit_term, const var_map_t &var_map, const bool sort_vars) {
00093
00094     flint::mpoly_ctx ctx(var_map.size());
00095     flint::mpoly pa(ctx.d), pb(ctx.d), denpa(ctx.d), denpb(ctx.d);
00096
00097     flint::from_argument_mpoly(pa.d, denpa.d, a, false, var_map, ctx.d);
00098     flint::from_argument_mpoly(pb.d, denpb.d, b, false, var_map, ctx.d);
00099
00100     // The input won't have any symbols with negative powers, but there may be rational
00101     // coefficients. Verify this:
00102     if ( fmpz_mpoly_is_fmpz(denpa.d, ctx.d) != 1 ) {
00103         // INTERNAL_ERROR_EXCL_START
00104         MLOCK(ErrorMessageLock);
00105         MesPrint("!>flint::divmod_mpoly: error: denpa is non-constant");
00106         MUNLOCK(ErrorMessageLock);
00107         Terminate(-1);
00108         // INTERNAL_ERROR_EXCL_STOP
00109     }
00110     if ( fmpz_mpoly_is_fmpz(denpb.d, ctx.d) != 1 ) {
00111         // INTERNAL_ERROR_EXCL_START
00112         MLOCK(ErrorMessageLock);
00113         MesPrint("!>flint::divmod_mpoly: error: denpb is non-constant");
00114         MUNLOCK(ErrorMessageLock);
00115         Terminate(-1);
00116         // INTERNAL_ERROR_EXCL_STOP
00117     }
00118
00119
00120     flint::fmpz scale;
00121     flint::mpoly div(ctx.d), rem(ctx.d);
00122
00123     fmpz_mpoly_quasidivrem(scale.d, div.d, rem.d, pa.d, pb.d, ctx.d);
00124
00125     // The quotient must be multiplied by the denominator of the divisor
00126     fmpz_mpoly_mul(div.d, div.d, denpb.d, ctx.d);
00127
00128     // The overall denominator of both div and rem is given by scale*denpa. This we will pass to
00129     // to_argument_mpoly's "denscale" argument which performs the division in the result. We have
00130     // already checked denpa is just an fmpz.
00131     fmpz_mul(scale.d, scale.d, fmpz_mpoly_term_coeff_ref(denpa.d, 0, ctx.d));
00132
00133
00134     // Determine the size of the output by passing write = false.
00135     const bool with_arghead = false;
00136     bool write = false;
00137     const uint64_t prev_size = 0;
00138     const uint64_t out_size = return_rem ?
00139         (uint64_t)flint::to_argument_mpoly(BHEAD NULL, with_arghead, must_fit_term, write, prev_size,
00140             rem.d, var_map, ctx.d, sort_vars, scale.d)
00141         :
00142         (uint64_t)flint::to_argument_mpoly(BHEAD NULL, with_arghead, must_fit_term, write, prev_size,
00143             div.d, var_map, ctx.d, sort_vars, scale.d)
00144         ;
00145     WORD* res = (WORD *)Malloc1(sizeof(WORD)*out_size, "flint::divrem_mpoly");
00146
00147
00148     // Write out the result
00149     write = true;
00150     if ( return_rem ) {
00151         flint::to_argument_mpoly(BHEAD res, with_arghead, must_fit_term, write, prev_size,
00152             rem.d, var_map, ctx.d, sort_vars, scale.d);
00153     }
00154     else {
00155         flint::to_argument_mpoly(BHEAD res, with_arghead, must_fit_term, write, prev_size,
00156             div.d, var_map, ctx.d, sort_vars, scale.d);
00157     }
00158
00159     return res;
00160 }
00161 /*
00162     #] flint::divmod_mpoly :
00163     #[ flint::divmod_poly :
00164 */
00165 WORD* flint::divmod_poly(PHEAD const WORD *a, const WORD *b, const bool return_rem,
00166     const WORD must_fit_term, const var_map_t &var_map) {
00167
00168     flint::poly pa, pb, denpa, denpb;
00169
00170     flint::from_argument_poly(pa.d, denpa.d, a, false);
00171     flint::from_argument_poly(pb.d, denpb.d, b, false);
00172
00173     // The input won't have any symbols with negative powers, but there may be rational
```

```
00174     // coefficients. Verify this:
00175     if ( fmpz_poly_length(denpa.d) != 1 ) {
00176         // INTERNAL_ERROR_EXCL_START
00177         MLOCK(ErrorMessageLock);
00178         MesPrint("!>flint::divmod_poly: error: denpa is non-constant");
00179         MUNLOCK(ErrorMessageLock);
00180         Terminate(-1);
00181         // INTERNAL_ERROR_EXCL_STOP
00182     }
00183     if ( fmpz_poly_length(denpb.d) != 1 ) {
00184         // INTERNAL_ERROR_EXCL_START
00185         MLOCK(ErrorMessageLock);
00186         MesPrint("!>flint::divmod_poly: error: denpb is non-constant");
00187         MUNLOCK(ErrorMessageLock);
00188         Terminate(-1);
00189         // INTERNAL_ERROR_EXCL_STOP
00190     }
00191
00192     flint::fmpz scale;
00193     uint64_t scale_power = 0;
00194     flint::poly div, rem;
00195
00196     // Get the leading coefficient of pb. fmpz_poly_pseudo_divrem returns only the scaling power.
00197     fmpz_poly_get_coeff_fmpz(scale.d, pb.d, fmpz_poly_degree(pb.d));
00198
00199     fmpz_poly_pseudo_divrem(div.d, rem.d, (ulong*)&scale_power, pa.d, pb.d);
00200
00201     // The quotient must be multiplied by the denominator of the divisor
00202     fmpz_poly_mul(div.d, div.d, denpb.d);
00203
00204     // The overall denominator of both div and rem is given by scale^scale_power*denpa. This we will
00205     // pass to to_argument_poly's "denscale" argument which performs the division in the result. We
00206     // have already checked denpa is just an fmpz.
00207     fmpz_pow_ui(scale.d, scale.d, scale_power);
00208     fmpz_mul(scale.d, scale.d, fmpz_poly_get_coeff_ptr(denpa.d, 0));
00209
00210
00211     // Determine the size of the output by passing write = false.
00212     const bool with_arghead = false;
00213     bool write = false;
00214     const uint64_t prev_size = 0;
00215     const uint64_t out_size = return_rem ?
00216         (uint64_t)flint::to_argument_poly(BHEAD NULL, with_arghead, must_fit_term, write, prev_size,
00217             rem.d, var_map, scale.d)
00218         :
00219         (uint64_t)flint::to_argument_poly(BHEAD NULL, with_arghead, must_fit_term, write, prev_size,
00220             div.d, var_map, scale.d)
00221         ;
00222     WORD* res = (WORD *)Malloc1(sizeof(WORD)*out_size, "flint::divrem_poly");
00223
00224
00225     // Write out the result
00226     write = true;
00227     if ( return_rem ) {
00228         flint::to_argument_poly(BHEAD res, with_arghead, must_fit_term, write, prev_size,
00229             rem.d, var_map, scale.d);
00230     }
00231     else {
00232         flint::to_argument_poly(BHEAD res, with_arghead, must_fit_term, write, prev_size,
00233             div.d, var_map, scale.d);
00234     }
00235
00236     return res;
00237 }
00238 /*
00239     #] flint::divmod_poly :
00240
00241     #[ flint::factorize_mpoly :
00242 */
00243 WORD* flint::factorize_mpoly(PHEAD const WORD *argin, WORD *argout, const bool with_arghead,
00244     const bool is_fun_arg, const var_map_t &var_map, const bool sort_vars) {
00245
00246     flint::mpoly_ctx ctx(var_map.size());
00247     flint::mpoly arg(ctx.d), den(ctx.d), base(ctx.d);
00248
00249     flint::from_argument_mpoly(arg.d, den.d, argin, with_arghead, var_map, ctx.d);
00250     // The denominator must be 1:
00251     if ( fmpz_mpoly_is_one(den.d, ctx.d) != 1 ) {
00252         // INTERNAL_ERROR_EXCL_START
00253         MLOCK(ErrorMessageLock);
00254         MesPrint("!>flint::factorize_mpoly error: den != 1");
00255         MUNLOCK(ErrorMessageLock);
00256         Terminate(-1);
00257         // INTERNAL_ERROR_EXCL_STOP
00258     }
00259
00260
```

```
00261     // Now we can factor the mpoly:
00262     flint::mpoly_factor arg_fac(ctx.d);
00263     fmpz_mpoly_factor(arg_fac.d, arg.d, ctx.d);
00264     const int64_t num_factors = fmpz_mpoly_factor_length(arg_fac.d, ctx.d);
00265
00266     flint::fmpz overall_constant;
00267     fmpz_mpoly_factor_get_constant_fmpz(overall_constant.d, arg_fac.d, ctx.d);
00268     // FORM should always have taken the overall constant out in the content. Thus this overall
00269     // constant factor should be +- 1 here. Verify this:
00270     if ( ! ( fmpz_equal_si(overall_constant.d, 1) || fmpz_equal_si(overall_constant.d, -1) ) ) {
00271         // INTERNAL_ERROR_EXCL_START
00272         MLOCK(ErrorMessageLock);
00273         MesPrint("!>flint::factorize_mpoly error: overall constant factor != +-1");
00274         MUNLOCK(ErrorMessageLock);
00275         Terminate(-1);
00276         // INTERNAL_ERROR_EXCL_STOP
00277     }
00278
00279     // Construct the output. If argout is not NULL, we write the result there.
00280     // Otherwise, allocate memory.
00281     // The output is zero-terminated list of factors. If with_arghead, each has an arghead which
00282     // contains its size. Otherwise, the factors are zero separated.
00283
00284     // We only need to determine the size of the output if we are allocating memory, but we need to
00285     // loop through the factors to fix their signs anyway. Do both together in one loop:
00286
00287     // Initially 1, for the final trailing 0.
00288     uint64_t output_size = 1;
00289
00290     // For finding the highest symbol, in FORM's lexicographic ordering
00291     vector<uint32_t> var_map_inv;
00292     var_map_inv.resize(var_map.size());
00293     for ( auto x : var_map ) {
00294         var_map_inv[x.second] = x.first;
00295     }
00296
00297     // Store whether we should flip the factor sign in the ouput:
00298     vector<int32_t> base_sign(num_factors, 0);
00299
00300     for ( int64_t i = 0; i < num_factors; i++ ) {
00301         fmpz_mpoly_factor_get_base(base.d, arg_fac.d, i, ctx.d);
00302         const int64_t exponent = fmpz_mpoly_factor_get_exp_si(arg_fac.d, i, ctx.d);
00303
00304         // poly_factorize makes sure the highest power of the "highest symbol" (in FORM's
00305         // lexicographic ordering) has a positive coefficient. Check this, update overall_constant
00306         // of the factorization if necessary.
00307         // Store the sign per factor, so that we can flip the signs in the output without re-checking
00308         // the individual terms again.
00309         uint32_t max_var = 0; // FORM symbols start at 20, 0 is a good initial value.
00310         int32_t max_pow = -1;
00311         vector<int64_t> base_term_exponents(var_map.size(), 0);
00312
00313         for ( int64_t j = 0; j < fmpz_mpoly_length(base.d, ctx.d); j++ ) {
00314             fmpz_mpoly_get_term_exp_si((slong*)base_term_exponents.data(), base.d, (slong)j, ctx.d);
00315
00316             for ( size_t k = 0; k < var_map.size(); k++ ) {
00317                 if ( base_term_exponents[k] > 0 && ( var_map_inv[k] > max_var ||
00318                     ( var_map_inv[k] == max_var && base_term_exponents[k] > max_pow ) ) ) {
00319
00320                     max_var = var_map_inv[k];
00321                     max_pow = base_term_exponents[k];
00322                     base_sign[i] = fmpz_sgn(fmpz_mpoly_term_coeff_ref(base.d, j, ctx.d));
00323                 }
00324             }
00325         }
00326         // If this base's sign will be flipped an odd number of times, there is a contribution to
00327         // the overall sign of the whole factorization:
00328         if ( ( base_sign[i] == -1 ) && ( exponent % 2 == 1 ) ) {
00329             fmpz_neg(overall_constant.d, overall_constant.d);
00330         }
00331
00332         // Now determine the output size of the factor, if we are allocating the memory
00333         if ( argout == NULL ) {
00334             const bool write = false;
00335             for ( int64_t j = 0; j < exponent; j++ ) {
00336                 output_size += (uint64_t)flint::to_argument_mpoly(BHEAD NULL, with_arghead,
00337                     is_fun_arg, write, 0, base.d, var_map, ctx.d, sort_vars);
00338             }
00339         }
00340     }
00341     if ( fmpz_sgn(overall_constant.d) == -1 && argout == NULL ) {
00342         // Add space for a fast-notation number or a normal-notation number and zero separator
00343         output_size += with_arghead ? 2 : 4+1;
00344     }
00345
00346     // Now make the allocation if necessary:
00347     if ( argout == NULL ) {
```

```
00348         argout = (WORD*)Malloc1(sizeof(WORD)*output_size, "flint::factorize_mpoly");
00349     }
00350
00351
00352     // And now comes the actual output:
00353     WORD* old_argout = argout;
00354
00355     // If the overall sign is negative, first write a full-notation -1. It will be absorbed into the
00356     // overall factor in the content by the caller.
00357     if ( fmpz_sgn(overall_constant.d) == -1 ) {
00358         if ( with_arghead ) {
00359             // poly writes in fast notation in this case. Fast notation is expected by the caller, to
00360             // properly merge it with the overall factor of the content.
00361             *argout++ = -SNUMBER;
00362             *argout++ = -1;
00363         }
00364         else {
00365             *argout++ = 4; // term size
00366             *argout++ = 1; // numerator
00367             *argout++ = 1; // denominator
00368             *argout++ = -3; // coeff size, negative number
00369             *argout++ = 0; // factor separator
00370         }
00371     }
00372
00373     for ( int64_t i = 0; i < num_factors; i++ ) {
00374         fmpz_mpoly_factor_get_base(base.d, arg_fac.d, i, ctx.d);
00375         const int64_t exponent = fmpz_mpoly_factor_get_exp_si(arg_fac.d, i, ctx.d);
00376
00377         if ( base_sign[i] == -1 ) {
00378             fmpz_mpoly_neg(base.d, base.d, ctx.d);
00379         }
00380
00381         const bool write = true;
00382         for ( int64_t j = 0; j < exponent; j++ ) {
00383             argout += flint::to_argument_mpoly(BHEAD argout, with_arghead, is_fun_arg, write,
00384                 argout-old_argout, base.d, var_map, ctx.d, sort_vars);
00385         }
00386     }
00387     // Final trailing zero to denote the end of the factors.
00388     *argout++ = 0;
00389
00390     return old_argout;
00391 }
00392 /*
00393     #] flint::factorize_mpoly :
00394     #[ flint::factorize_poly :
00395 */
00396 WORD* flint::factorize_poly(PHEAD const WORD *argin, WORD *argout, const bool with_arghead,
00397     const bool is_fun_arg, const var_map_t &var_map) {
00398
00399     flint::poly arg, den;
00400
00401     flint::from_argument_poly(arg.d, den.d, argin, with_arghead);
00402     // The denominator must be 1:
00403     if ( fmpz_poly_is_one(den.d) != 1 ) {
00404         // INTERNAL_ERROR_EXCL_START
00405         MLOCK(ErrorMessageLock);
00406         MesPrint("!>flint::factorize_poly error: den != 1");
00407         MUNLOCK(ErrorMessageLock);
00408         Terminate(-1);
00409         // INTERNAL_ERROR_EXCL_STOP
00410     }
00411
00412
00413     // Now we can factor the poly:
00414     flint::poly_factor arg_fac;
00415     fmpz_poly_factor(arg_fac.d, arg.d);
00416     // fmpz_poly_factor_t lacks some convenience functions which fmpz_mpoly_factor_t has.
00417     // I have worked out how to get the factors by looking at how fmpz_poly_factor_print works.
00418     const long num_factors = (arg_fac.d)->num;
00419
00420
00421     // Construct the output. If argout is not NULL, we write the result there.
00422     // Otherwise, allocate memory.
00423     // The output is zero-terminated list of factors. If with_arghead, each has an arghead which
00424     // contains its size. Otherwise, the factors are zero separated.
00425     if ( argout == NULL ) {
00426         // First we need to determine the size of the output. This is the same procedure as the
00427         // loop below, but we don't write anything in to_argument_poly (write arg: false).
00428         // Initially 1, for the final trailing 0.
00429         uint64_t output_size = 1;
00430
00431         // If the overall constant is not 1, make an fmpz_poly out of it and include it. Typically
00432         // this does not happen: FORM takes out the overall factor. But we do sometimes have this
00433         // case, for eg awkward interactions with "On highfirst;" and non-dirty arguments.
00434         if ( ! fmpz_is_one(&(arg_fac.d)->c) ) {
```

```
00435             flint::poly overall_factor;
00436             fmpz_poly_set_fmpz(overall_factor.d, &(arg_fac.d)->c);
00437             const bool write = false;
00438             output_size += (uint64_t)flint::to_argument_poly(BHEAD NULL, with_arghead, is_fun_arg,
00439                 write, 0, overall_factor.d, var_map);
00440         }
00441 
00442         for ( long i = 0; i < num_factors; i++ ) {
00443             fmpz_poly_struct* base = (arg_fac.d)->p + i;
00444 
00445             const long exponent = (arg_fac.d)->exp[i];
00446 
00447             const bool write = false;
00448             for ( long j = 0; j < exponent; j++ ) {
00449                 output_size += (uint64_t)flint::to_argument_poly(BHEAD NULL, with_arghead,
00450                     is_fun_arg, write, 0, base, var_map);
00451             }
00452         }
00453 
00454         argout = (WORD*)Malloc1(sizeof(WORD)*output_size, "flint::factorize_poly");
00455     }
00456 
00457     WORD* old_argout = argout;
00458 
00459     // If the overall constant is not 1, make an fmpz_poly out of it and include it. Typically
00460     // this does not happen: FORM takes out the overall factor. But we do sometimes have this
00461     // case, for eg awkward interactions with "On highfirst;" and non-dirty arguments.
00462     if ( ! fmpz_is_one(&(arg_fac.d)->c) ) {
00463         flint::poly overall_factor;
00464         fmpz_poly_set_fmpz(overall_factor.d, &(arg_fac.d)->c);
00465         const bool write = true;
00466         argout += (uint64_t)flint::to_argument_poly(BHEAD argout, with_arghead, is_fun_arg, write,
00467             argout-old_argout, overall_factor.d, var_map);
00468     }
00469 
00470     for ( long i = 0; i < num_factors; i++ ) {
00471         fmpz_poly_struct* base = (arg_fac.d)->p + i;
00472 
00473         const long exponent = (arg_fac.d)->exp[i];
00474 
00475         const bool write = true;
00476         for ( long j = 0; j < exponent; j++ ) {
00477             argout += flint::to_argument_poly(BHEAD argout, with_arghead, is_fun_arg, write,
00478                 argout-old_argout, base, var_map);
00479         }
00480     }
00481     *argout = 0;
00482 
00483     return old_argout;
00484 }
00485 /*
00486     #] flint::factorize_poly :
00487 
00488     #[ flint::form_sort :
00489 */
00490 // Sort terms using form's sorting routines. Uses a custom (faster) compare routine, since here
00491 // only symbols can appear.
00492 // This is a modified poly_sort from polywrap.cc.
00493 void flint::form_sort(PHEAD WORD *terms) {
00494 
00495     if ( terms[0] < 0 ) {
00496         // Fast notation, there is nothing to do
00497         return;
00498     }
00499 
00500     const WORD oldsorttype = AR.SortType;
00501     AR.SortType = SORTHIGHFIRST;
00502 
00503     const WORD in_size = terms[0] - ARGHEAD;
00504     WORD out_size;
00505 
00506     if ( NewSort(BHEAD0) ) {
00507         Terminate(-1);
00508     }
00509     AR.CompareRoutine = (COMPAREDUMMY)(&CompareSymbols);
00510 
00511     // Make sure the symbols are in the right order within the terms
00512     for ( WORD i = ARGHEAD; i < terms[0]; i += terms[i] ) {
00513         if ( SymbolNormalize(terms+i) < 0 || StoreTerm(BHEAD terms+i) ) {
00514             AR.SortType = oldsorttype;
00515             AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
00516             LowerSortLevel();
00517             Terminate(-1);
00518         }
00519     }
00520 
00521     if ( ( out_size = EndSort(BHEAD terms+ARGHEAD, 1) ) < 0 ) {
```

```
00522         AR.SortType = oldsorttype;
00523         AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
00524         Terminate(-1);
00525     }
00526 
00527     // Check the final size
00528     if ( in_size != out_size  ) {
00529         // INTERNAL_ERROR_EXCL_START
00530         MLOCK(ErrorMessageLock);
00531         MesPrint("!>flint::form_sort: error: unexpected sorted arg length change %d->%d", in_size,
00532             out_size);
00533         MUNLOCK(ErrorMessageLock);
00534         Terminate(-1);
00535         // INTERNAL_ERROR_EXCL_STOP
00536     }
00537 
00538     AR.SortType = oldsorttype;
00539     AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
00540     terms[1] = 0; // set dirty flag to zero
00541 }
00542 /*
00543     #] flint::form_sort :
00544 
00545     #[ flint::from_argument_mpoly :
00546 */
00547 // Convert a FORM argument (or 0-terminated list of terms: with_arghead == false) to a
00548 // (multi-variate) fmpz_mpoly_t poly. The "denominator" is return in denpoly, and contains the
00549 // overall negative-power numeric and symbolic factor.
00550 uint64_t flint::from_argument_mpoly(fmpz_mpoly_t poly, fmpz_mpoly_t denpoly, const WORD *args,
00551     const bool with_arghead, const var_map_t &var_map, const fmpz_mpoly_ctx_t ctx) {
00552 
00553     // Some callers re-use their poly, denpoly to avoid calling init/clear unnecessarily.
00554     // Make sure they are 0 to begin.
00555     fmpz_mpoly_set_si(poly, 0, ctx);
00556     fmpz_mpoly_set_si(denpoly, 0, ctx);
00557 
00558     // First check for "fast notation" arguments:
00559     if ( *args == -SNUMBER ) {
00560         fmpz_mpoly_set_si(poly, *(args+1), ctx);
00561         fmpz_mpoly_set_si(denpoly, 1, ctx);
00562         return 2;
00563     }
00564 
00565     if ( *args == -SYMBOL ) {
00566         // A "fast notation" SYMBOL has a power and coefficient of 1:
00567         vector<uint64_t> exponents(var_map.size(), 0);
00568         exponents[var_map.at(*(args+1))] = 1;
00569 
00570         fmpz_mpoly_set_coeff_ui_ui(poly, (ulong)1, (ulong*)exponents.data(), ctx);
00571         fmpz_mpoly_set_ui(denpoly, (ulong)1, ctx);
00572         return 2;
00573     }
00574 
00575 
00576     // Now we can iterate through the terms of the argument. If we have
00577     // an ARGHEAD, we already know where to terminate. Otherwise we'll have
00578     // to loop until the terminating 0.
00579     const WORD* arg_stop = with_arghead ? args+args[0] : (WORD*)UINT64_MAX;
00580     uint64_t arg_size = 0;
00581     if ( with_arghead ) {
00582         arg_size = args[0];
00583         args += ARGHEAD;
00584     }
00585 
00586 
00587     // Search for numerical or symbol denominators to create "denpoly".
00588     flint::fmpz den_coeff, tmp;
00589     fmpz_set_si(den_coeff.d, 1);
00590     vector<uint64_t> exponents(var_map.size(), 0);
00591     vector<uint64_t> neg_exponents(var_map.size(), 0);
00592 
00593     for ( const WORD* term = args; term < arg_stop; term += term[0] ) {
00594         const WORD* term_stop = term+term[0];
00595         const WORD  coeff_size = (term_stop)[-1];
00596         const WORD* symbol_stop = term_stop - ABS(coeff_size);
00597         const WORD* t = term;
00598 
00599         t++;
00600         if ( t == symbol_stop ) {
00601             // Just a number, no symbols
00602         }
00603         else {
00604             t++; // this entry is SYMBOL
00605             t++; // this entry just has the size of the symbol array, but we can use symbol_stop
00606 
00607             for ( const WORD* s = t; s < symbol_stop; s += 2 ) {
00608                 if ( *(s+1) < 0 ) {
```

```
00609                     neg_exponents[var_map.at(*s)] =
00610                         MaX(neg_exponents[var_map.at(*s)], (uint64_t)(-(*(s+1))) );
00611                 }
00612             }
00613         }
00614
00615         // Now check for a denominator in the coefficient:
00616         if ( *(symbol_stop+ABS(coeff_size/2)) != 1 ) {
00617             flint::fmpz_set_form(tmp.d, (UWORD*)(symbol_stop+ABS(coeff_size/2)), ABS(coeff_size/2));
00618             // Record the LCM of the coefficient denominators:
00619             fmpz_lcm(den_coeff.d, den_coeff.d, tmp.d);
00620         }
00621
00622         if ( !with_arghead && *term_stop == 0 ) {
00623             // + 1 for the terminating 0
00624             arg_size = term_stop - args + 1;
00625             break;
00626         }
00627
00628     }
00629     // Assemble denpoly.
00630     fmpz_mpoly_set_coeff_fmpz_ui(denpoly, den_coeff.d, (ulong*)neg_exponents.data(), ctx);
00631
00632
00633     // For the term coefficients
00634     flint::fmpz coeff;
00635
00636     for ( const WORD* term = args; term < arg_stop; term += term[0] ) {
00637         const WORD* term_stop = term+term[0];
00638         const WORD  coeff_size = (term_stop)[-1];
00639         const WORD* symbol_stop = term_stop - ABS(coeff_size);
00640         const WORD* t = term;
00641
00642         fill(exponents.begin(), exponents.end(), 0);
00643
00644         t++; // skip over the total size entry
00645         if ( t == symbol_stop ) {
00646             // Just a number, no symbols
00647         }
00648         else {
00649             t++; // this entry is SYMBOL
00650             t++; // this entry just has the size of the symbol array, but we can use symbol_stop
00651             for ( const WORD* s = t; s < symbol_stop; s += 2 ) {
00652                 exponents[var_map.at(*s)] = *(s+1);
00653             }
00654         }
00655         // Now read the coefficient
00656         flint::fmpz_set_form(coeff.d, (UWORD*)symbol_stop, coeff_size/2);
00657
00658         // Multiply by denominator LCM
00659         fmpz_mul(coeff.d, coeff.d, den_coeff.d);
00660
00661         // Shift by neg_exponents
00662         for ( size_t i = 0; i < var_map.size(); i++ ) {
00663             exponents[i] += neg_exponents[i];
00664         }
00665
00666         // Read the denominator if there is one, and divide it out of the coefficient
00667         if ( *(symbol_stop+ABS(coeff_size/2)) != 1 ) {
00668             flint::fmpz_set_form(tmp.d, (UWORD*)(symbol_stop+ABS(coeff_size/2)), ABS(coeff_size/2));
00669             // By construction, this is an exact division
00670             fmpz_divexact(coeff.d, coeff.d, tmp.d);
00671         }
00672
00673         // Push the term to the mpoly, remember to sort when finished! This is much faster than using
00674         // fmpz_mpoly_set_coeff_fmpz_ui when the terms arrive in the "wrong order".
00675         fmpz_mpoly_push_term_fmpz_ui(poly, coeff.d, (ulong*)exponents.data(), ctx);
00676
00677         if ( !with_arghead && *term_stop == 0 ) {
00678             break;
00679         }
00680
00681     }
00682     // And now sort the mpoly if necessary:
00683     if ( ! fmpz_mpoly_is_canonical(poly, ctx) ) {
00684         fmpz_mpoly_sort_terms(poly, ctx);
00685     }
00686
00687     return arg_size;
00688 }
00689 /*
00690     #] flint::from_argument_mpoly :
00691     #[ flint::from_argument_poly :
00692 */
00693 // Convert a FORM argument (or 0-terminated list of terms: with_arghead == false) to a
00694 // (uni-variate) fmpz_poly_t poly. The "denominator" is return in denpoly, and contains the
00695 // overall negative-power numeric and symbolic factor.
```

```
00696 uint64_t flint::from_argument_poly(fmpz_poly_t poly, fmpz_poly_t denpoly, const WORD *args,
00697     const bool with_arghead) {
00698
00699     // Some callers re-use their poly, denpoly to avoid calling init/clear unnecessarily.
00700     // Make sure they are 0 to begin.
00701     fmpz_poly_set_si(poly, 0);
00702     fmpz_poly_set_si(denpoly, 0);
00703
00704     // First check for "fast notation" arguments:
00705     if ( *args == -SNUMBER ) {
00706         fmpz_poly_set_si(poly, *(args+1));
00707         fmpz_poly_set_si(denpoly, 1);
00708         return 2;
00709     }
00710
00711     if ( *args == -SYMBOL ) {
00712         // A "fast notation" SYMBOL has a power and coefficient of 1:
00713         fmpz_poly_set_coeff_si(poly, 1, 1);
00714         fmpz_poly_set_si(denpoly, 1);
00715         return 2;
00716     }
00717
00718     // Now we can iterate through the terms of the argument. If we have
00719     // an ARGHEAD, we already know where to terminate. Otherwise we'll have
00720     // to loop until the terminating 0.
00721     const WORD* arg_stop = with_arghead ? args+args[0] : (WORD*)UINT64_MAX;
00722     uint64_t arg_size = 0;
00723     if ( with_arghead ) {
00724         arg_size = args[0];
00725         args += ARGHEAD;
00726     }
00727
00728     // Search for numerical or symbol denominators to create "denpoly".
00729     flint::fmpz den_coeff, tmp;
00730     fmpz_set_si(den_coeff.d, 1);
00731     uint64_t neg_exponent = 0;
00732
00733     for ( const WORD* term = args; term < arg_stop; term += term[0] ) {
00734
00735         const WORD* term_stop = term+term[0];
00736         const WORD  coeff_size = (term_stop)[-1];
00737         const WORD* symbol_stop = term_stop - ABS(coeff_size);
00738         const WORD* t = term;
00739
00740         t++; // skip over the total size entry
00741         if ( t == symbol_stop ) {
00742             // Just a number, no symbols
00743         }
00744         else {
00745             t++; // this entry is SYMBOL
00746             t++; // this entry is the size of the symbol array
00747             t++; // this is the first (and only) symbol code
00748             if ( *t < 0 ) {
00749                 neg_exponent = MaX(neg_exponent, (uint64_t)(-(*t)) );
00750             }
00751         }
00752
00753         // Now check for a denominator in the coefficient:
00754         if ( *(symbol_stop+ABS(coeff_size/2)) != 1 ) {
00755             flint::fmpz_set_form(tmp.d, (UWORD*)(symbol_stop+ABS(coeff_size/2)), ABS(coeff_size/2));
00756             // Record the LCM of the coefficient denominators:
00757             fmpz_lcm(den_coeff.d, den_coeff.d, tmp.d);
00758         }
00759
00760         if ( *term_stop == 0 ) {
00761             // + 1 for the terminating 0
00762             arg_size = term_stop - args + 1;
00763             break;
00764         }
00765     }
00766     // Assemble denpoly.
00767     fmpz_poly_set_coeff_fmpz(denpoly, neg_exponent, den_coeff.d);
00768
00769
00770     // For the term coefficients
00771     flint::fmpz coeff;
00772
00773     for ( const WORD* term = args; term < arg_stop; term += term[0] ) {
00774
00775         const WORD* term_stop = term+term[0];
00776         const WORD  coeff_size = (term_stop)[-1];
00777         const WORD* symbol_stop = term_stop - ABS(coeff_size);
00778         const WORD* t = term;
00779
00780         uint64_t exponent = 0;
00781
00782         t++; // skip over the total size entry
```

```
00783         if ( t == symbol_stop ) {
00784             // Just a number, no symbols
00785         }
00786         else {
00787             t++; // this entry is SYMBOL
00788             t++; // this entry is the size of the symbol array
00789             t++; // this is the first (and only) symbol code
00790             exponent = *t++;
00791         }
00792
00793         // Now read the coefficient
00794         flint::fmpz_set_form(coeff.d, (UWORD*)symbol_stop, coeff_size/2);
00795
00796         // Multiply by denominator LCM
00797         fmpz_mul(coeff.d, coeff.d, den_coeff.d);
00798
00799         // Shift by neg_exponent
00800         exponent += neg_exponent;
00801
00802         // Read the denominator if there is one, and divide it out of the coefficient
00803         if ( *(symbol_stop+ABS(coeff_size/2)) != 1 ) {
00804             flint::fmpz_set_form(tmp.d, (UWORD*)(symbol_stop+ABS(coeff_size/2)), ABS(coeff_size/2));
00805             // By construction, this is an exact division
00806             fmpz_divexact(coeff.d, coeff.d, tmp.d);
00807         }
00808
00809         // Add the term to the poly
00810         fmpz_poly_set_coeff_fmpz(poly, exponent, coeff.d);
00811
00812         if ( *term_stop == 0 ) {
00813             break;
00814         }
00815     }
00816
00817     return arg_size;
00818 }
00819 /*
00820     #] flint::from_argument_poly :
00821
00822     #[ flint::fmpz_get_form :
00823 */
00824 // Write FORM's long integer representation of an fmpz at a, and put the number of WORDs at na.
00825 // na carries the sign of the integer.
00826 WORD flint::fmpz_get_form(fmpz_t z, WORD *a) {
00827
00828     WORD na = 0;
00829     const int32_t sgn = fmpz_sgn(z);
00830     if ( sgn == -1 ) {
00831         fmpz_neg(z, z);
00832     }
00833     const int64_t nlimbs = fmpz_size(z);
00834
00835     // This works but is UB?
00836     //fmpz_get_ui_array(reinterpret_cast<uint64_t*>(a), nlimbs, z);
00837
00838     // Use fixed-size functions to get limb data where possible. These probably cover most real
00839     // cases.
00840     if ( nlimbs == 1 ) {
00841         const uint64_t limb = fmpz_get_ui(z);
00842         a[0] = (WORD)(limb & 0xFFFFFFFF);
00843         na++;
00844         a[1] = (WORD)(limb » BITSINWORD);
00845         if ( a[1] != 0 ) {
00846             na++;
00847         }
00848     }
00849     else if ( nlimbs == 2 ) {
00850         uint64_t limb_hi = 0, limb_lo = 0;
00851         fmpz_get_uiui((ulong*)&limb_hi, (ulong*)&limb_lo, z);
00852         a[0] = (WORD)(limb_lo & 0xFFFFFFFF);
00853             na++;
00854         a[1] = (WORD)(limb_lo » BITSINWORD);
00855             na++;
00856         a[2] = (WORD)(limb_hi & 0xFFFFFFFF);
00857             na++;
00858         a[3] = (WORD)(limb_hi » BITSINWORD);
00859         if ( a[3] != 0 ) {
00860             na++;
00861         }
00862     }
00863     else {
00864         vector<uint64_t> limb_data(nlimbs, 0);
00865         fmpz_get_ui_array((ulong*)limb_data.data(), (slong)nlimbs, z);
00866         for ( long i = 0; i < nlimbs; i++ ) {
00867             a[2*i] = (WORD)(limb_data[i] & 0xFFFFFFFF);
00868             na++;
00869             a[2*i+1] = (WORD)(limb_data[i] » BITSINWORD);
```

```
00870             if ( a[2*i+1] != 0 || i < (nlimbs-1) ) {
00871                 // The final limb might fit in a single 32bit WORD. Only
00872                 // increment na if the final WORD is non zero.
00873                 na++;
00874             }
00875         }
00876     }
00877
00878     // And now put the sign in the number of limbs
00879     if ( sgn == -1 ) {
00880         na = -na;
00881     }
00882
00883     return na;
00884 }
00885 /*
00886     #] flint::fmpz_get_form :
00887     #[ flint::fmpz_set_form :
00888 */
00889 // Create an fmpz directly from FORM's long integer representation. fmpz uses 64bit unsigned limbs,
00890 // but FORM uses 32bit UWORDs on 64bit architectures so we can't use fmpz_set_ui_array directly.
00891 void flint::fmpz_set_form(fmpz_t z, UWORD *a, WORD na) {
00892
00893     if ( na == 0 ) {
00894         fmpz_zero(z);
00895         return;
00896     }
00897
00898     // Negative na represenents a negative number
00899     int32_t sgn = 1;
00900     if ( na < 0 ) {
00901         sgn = -1;
00902         na = -na;
00903     }
00904
00905     // Remove padding. FORM stores numerators and denominators with equal numbers of limbs but we
00906     // don't need to do this within the fmpz. It is not necessary to do this really, the fmpz
00907     // doesn't add zero limbs unnecessarily, but we might be able to avoid creating the limb_data
00908     // array below.
00909     while ( a[na-1] == 0 ) {
00910         na--;
00911     }
00912
00913     // If the number fits in fixed-size fmpz_set functions, we don't need to use additional memory
00914     // to convert to uint64_t. These probably cover most real cases.
00915     if ( na == 1 ) {
00916         fmpz_set_ui(z, (uint64_t)a[0]);
00917     }
00918     else if ( na == 2 ) {
00919         fmpz_set_ui(z, (((uint64_t)a[1])«BITSINWORD) + (uint64_t)a[0]);
00920     }
00921     else if ( na == 3 ) {
00922         fmpz_set_uiui(z, (uint64_t)a[2], (((uint64_t)a[1])«BITSINWORD) + (uint64_t)a[0]);
00923     }
00924     else if ( na == 4 ) {
00925         fmpz_set_uiui(z, (((uint64_t)a[3])«BITSINWORD) + (uint64_t)a[2],
00926             (((uint64_t)a[1])«BITSINWORD) + (uint64_t)a[0]);
00927     }
00928     else {
00929         const int32_t nlimbs = (na+1)/2;
00930         vector<uint64_t> limb_data(nlimbs, 0);
00931         for ( int32_t i = 0; i < nlimbs; i++ ) {
00932             if ( 2*i+1 <= na-1 ) {
00933                 limb_data[i] = (uint64_t)a[2*i] + (((uint64_t)a[2*i+1])«BITSINWORD);
00934             }
00935             else {
00936                 limb_data[i] = (uint64_t)a[2*i];
00937             }
00938         }
00939         fmpz_set_ui_array(z, (ulong*)limb_data.data(), nlimbs);
00940     }
00941
00942     // Finally set the sign.
00943     if ( sgn == -1 ) {
00944         fmpz_neg(z, z);
00945     }
00946
00947     return;
00948 }
00949 /*
00950     #] flint::fmpz_set_form :
00951
00952     #[ flint::gcd_mpoly :
00953 */
00954 // Return a pointer to a buffer containing the GCD of the 0-terminated term lists at a and b.
00955 // If must_fit_term, this should be a TermMalloc buffer. Otherwise Malloc1 the buffer.
00956 // For multi-variate cases.
```

```
00957 WORD* flint::gcd_mpoly(PHEAD const WORD *a, const WORD *b, const WORD must_fit_term,
00958     const var_map_t &var_map, const bool sort_vars) {
00959
00960     flint::mpoly_ctx ctx(var_map.size());
00961     flint::mpoly pa(ctx.d), pb(ctx.d), denpa(ctx.d), denpb(ctx.d), gcd(ctx.d);
00962
00963     flint::from_argument_mpoly(pa.d, denpa.d, a, false, var_map, ctx.d);
00964     flint::from_argument_mpoly(pb.d, denpb.d, b, false, var_map, ctx.d);
00965
00966     // denpa, denpb must be 1:
00967     if ( fmpz_mpoly_is_one(denpa.d, ctx.d) != 1 ) {
00968         // INTERNAL_ERROR_EXCL_START
00969         MLOCK(ErrorMessageLock);
00970         MesPrint("!>flint::gcd_mpoly: error: denpa != 1");
00971         MUNLOCK(ErrorMessageLock);
00972         Terminate(-1);
00973         // INTERNAL_ERROR_EXCL_STOP
00974     }
00975     if ( fmpz_mpoly_is_one(denpb.d, ctx.d) != 1 ) {
00976         // INTERNAL_ERROR_EXCL_START
00977         MLOCK(ErrorMessageLock);
00978         MesPrint("!>flint::gcd_mpoly: error: denpb != 1");
00979         MUNLOCK(ErrorMessageLock);
00980         Terminate(-1);
00981         // INTERNAL_ERROR_EXCL_STOP
00982     }
00983
00984     // poly returns pa if pa == pb, regardless of the lcoeff sign
00985     if ( fmpz_mpoly_equal(pa.d, pb.d, ctx.d) ) {
00986         fmpz_mpoly_set(gcd.d, pa.d, ctx.d);
00987     }
00988     else {
00989         // We need some gymnastics to have the same sign conventions as the poly class. It takes the
00990         // integer or univar content out of a,b, with the convention that the content sign matches
00991         // the lcoeff sign. Since FORM has already taken out the content, we are left with +-1. In
00992         // Flint, the content always has a positive sign so here we should find +1. Check this:
00993         flint::mpoly tmp(ctx.d);
00994         fmpz_mpoly_term_content(tmp.d, pa.d, ctx.d);
00995         if ( fmpz_mpoly_is_one(tmp.d, ctx.d) != 1 ) {
00996             // INTERNAL_ERROR_EXCL_START
00997             MLOCK(ErrorMessageLock);
00998             MesPrint("!>flint::gcd_mpoly: error: content of 1st arg != 1");
00999             MUNLOCK(ErrorMessageLock);
01000             Terminate(-1);
01001             // INTERNAL_ERROR_EXCL_STOP
01002         }
01003         fmpz_mpoly_term_content(tmp.d, pb.d, ctx.d);
01004         if ( fmpz_mpoly_is_one(tmp.d, ctx.d) != 1 ) {
01005             // INTERNAL_ERROR_EXCL_START
01006             MLOCK(ErrorMessageLock);
01007             MesPrint("!>flint::gcd_mpoly: error: content of 2nd arg != 1");
01008             MUNLOCK(ErrorMessageLock);
01009             Terminate(-1);
01010             // INTERNAL_ERROR_EXCL_STOP
01011         }
01012
01013         // The poly class now divides the content out of a,b so that they have a positive lcoeff.
01014         // Then it multiplies the final gcd (which is given a positive lcoeff also) by
01015         // gcd(cont a, cont b). There it has gcd(1,1) = gcd(-1,1) = gcd(1,-1) = 1, and
01016         // gcd(-1,-1) = -1 (because of the pa==pb early return). So: if both input polys have a
01017         // negative lcoeff, we will flip the sign in the final result.
01018         bool flip_sign = 0;
01019         if ( ( fmpz_sgn(fmpz_mpoly_term_coeff_ref(pa.d, 0, ctx.d)) == -1 ) &&
01020             ( fmpz_sgn(fmpz_mpoly_term_coeff_ref(pb.d, 0, ctx.d)) == -1 ) ) {
01021             flip_sign = 1;
01022         }
01023
01024         fmpz_mpoly_gcd(gcd.d, pa.d, pb.d, ctx.d);
01025         if ( flip_sign ) {
01026             fmpz_mpoly_neg(gcd.d, gcd.d, ctx.d);
01027         }
01028     }
01029
01030     // This is freed by the caller
01031     WORD *res;
01032     if ( must_fit_term ) {
01033         res = TermMalloc("flint::gcd_mpoly");
01034     }
01035     else {
01036         // Determine the size of the GCD by passing write = false.
01037         const bool with_arghead = false;
01038         const bool write = false;
01039         const uint64_t prev_size = 0;
01040         const uint64_t gcd_size = (uint64_t)flint::to_argument_mpoly(BHEAD NULL,
01041             with_arghead, must_fit_term, write, prev_size, gcd.d, var_map, ctx.d, sort_vars);
01042
01043         res = (WORD *)Malloc1(sizeof(WORD)*gcd_size, "flint::gcd_mpoly");
```

```
01044     }
01045
01046     const bool with_arghead = false;
01047     const bool write = true;
01048     const uint64_t prev_size = 0;
01049     flint::to_argument_mpoly(BHEAD res, with_arghead, must_fit_term, write, prev_size, gcd.d,
01050         var_map, ctx.d, sort_vars);
01051
01052     return res;
01053 }
01054 /*
01055     #] flint::gcd_mpoly :
01056     #[ flint::gcd_poly :
01057 */
01058 // Return a pointer to a buffer containing the GCD of the 0-terminated term lists at a and b.
01059 // If must_fit_term, this should be a TermMalloc buffer. Otherwise Malloc1 the buffer.
01060 // For uni-variate cases.
01061 WORD* flint::gcd_poly(PHEAD const WORD *a, const WORD *b, const WORD must_fit_term,
01062     const var_map_t &var_map) {
01063
01064     flint::poly pa, pb, denpa, denpb, gcd;
01065
01066     flint::from_argument_poly(pa.d, denpa.d, a, false);
01067     flint::from_argument_poly(pb.d, denpb.d, b, false);
01068
01069     // denpa, denpb must be 1:
01070     if ( fmpz_poly_is_one(denpa.d) != 1 ) {
01071         // INTERNAL_ERROR_EXCL_START
01072         MLOCK(ErrorMessageLock);
01073         MesPrint("!>flint::gcd_poly: error: denpa != 1");
01074         MUNLOCK(ErrorMessageLock);
01075         Terminate(-1);
01076         // INTERNAL_ERROR_EXCL_STOP
01077     }
01078     if ( fmpz_poly_is_one(denpb.d) != 1 ) {
01079         // INTERNAL_ERROR_EXCL_START
01080         MLOCK(ErrorMessageLock);
01081         MesPrint("!>flint::gcd_poly: error: denpb != 1");
01082         MUNLOCK(ErrorMessageLock);
01083         Terminate(-1);
01084         // INTERNAL_ERROR_EXCL_STOP
01085     }
01086
01087     // poly returns pa if pa == pb, regardless of the lcoeff sign
01088     if ( fmpz_poly_equal(pa.d, pb.d) ) {
01089         fmpz_poly_set(gcd.d, pa.d);
01090     }
01091     else {
01092         // Here, we don't have to make any sign flips like the mpoly case, because poly's
01093         // integer_gcd(1,1) = integer_gcd(-1,1) = integer_gcd(1,-1) = integer_gcd(-1,-1) = +1.
01094         // Still, verify that the content is 1:
01095         flint::fmpz tmp;
01096         fmpz_poly_content(tmp.d, pa.d);
01097         if ( fmpz_is_one(tmp.d) != 1 ) {
01098             // INTERNAL_ERROR_EXCL_START
01099             MLOCK(ErrorMessageLock);
01100             MesPrint("!>flint::gcd_poly: error: content of 1st arg != 1");
01101             MUNLOCK(ErrorMessageLock);
01102             Terminate(-1);
01103             // INTERNAL_ERROR_EXCL_STOP
01104         }
01105         fmpz_poly_content(tmp.d, pb.d);
01106         if ( fmpz_is_one(tmp.d) != 1 ) {
01107             // INTERNAL_ERROR_EXCL_START
01108             MLOCK(ErrorMessageLock);
01109             MesPrint("!>flint::gcd_poly: error: content of 2nd arg != 1");
01110             MUNLOCK(ErrorMessageLock);
01111             Terminate(-1);
01112             // INTERNAL_ERROR_EXCL_STOP
01113         }
01114
01115         fmpz_poly_gcd(gcd.d, pa.d, pb.d);
01116     }
01117
01118     // This is freed by the caller
01119     WORD *res;
01120     if ( must_fit_term ) {
01121         res = TermMalloc("flint::gcd_poly");
01122     }
01123     else {
01124         // Determine the size of the GCD by passing write = false.
01125         const bool with_arghead = false;
01126         const bool write = false;
01127         const uint64_t prev_size = 0;
01128         const uint64_t gcd_size = (uint64_t)flint::to_argument_poly(BHEAD NULL,
01129             with_arghead, must_fit_term, write, prev_size, gcd.d, var_map);
01130
```

```
01131         res = (WORD *)Malloc1(sizeof(WORD)*gcd_size, "flint::gcd_poly");
01132     }
01133
01134     const bool with_arghead = false;
01135     const uint64_t prev_size = 0;
01136     const bool write = true;
01137     flint::to_argument_poly(BHEAD res, with_arghead, must_fit_term, write, prev_size, gcd.d,
01138         var_map);
01139
01140     return res;
01141 }
01142 /*
01143     #] flint::gcd_poly :
01144
01145     #[ flint::get_variables :
01146 */
01147 // Get the list of symbols which appear in the vector of expressions. These are polyratfun
01148 // numerators, denominators or expressions from calls to gcd_ etc. Return this list as a map
01149 // between indices and symbol codes.
01150 // TODO FACTORSYMBOL last?
01151 flint::var_map_t flint::get_variables(const vector <WORD *> &es, const bool with_arghead,
01152     const bool sort_vars) {
01153
01154     int32_t num_vars = 0;
01155     // We count the total number of terms to determine "density".
01156     uint32_t num_terms = 0;
01157     // To be used if we sort by highest degree, as the poly code does.
01158     vector<int32_t> degrees;
01159     var_map_t var_map;
01160
01161     // extract all variables
01162     for ( size_t ei = 0; ei < es.size(); ei++ ) {
01163         WORD *e = es[ei];
01164
01165         // fast notation
01166         if ( *e == -SNUMBER ) {
01167             num_terms++;
01168         }
01169         else if ( *e == -SYMBOL ) {
01170             num_terms++;
01171             if ( !var_map.count(e[1]) ) {
01172                 var_map[e[1]] = num_vars++;
01173                 degrees.push_back(1);
01174             }
01175         }
01176         // JD: Here we need to check for non-symbol/number terms in fast notation.
01177         else if ( *e < 0 ) {
01178             MLOCK(ErrorMessageLock);
01179             MesPrint("ERROR: polynomials and polyratfuns must contain symbols only");
01180             MUNLOCK(ErrorMessageLock);
01181             Terminate(1);
01182         }
01183         else {
01184             for ( WORD i = with_arghead ? ARGHEAD:0; with_arghead ? i < e[0]:e[i] != 0; i += e[i] ) {
01185                 num_terms++;
01186                 const WORD coeff_size = e[i+e[i]-1];
01187                 const WORD symbols_size = e[i] - ABS(coeff_size);
01188                 if ( e[i+1] != SYMBOL && 1 < symbols_size ) {
01189                     MLOCK(ErrorMessageLock);
01190                     MesPrint("ERROR: polynomials and polyratfuns must contain symbols only");
01191                     MUNLOCK(ErrorMessageLock);
01192                     Terminate(1);
01193                 }
01194
01195                 for ( WORD j = i+3; j < i+symbols_size; j += 2 ) {
01196                     const WORD symbol = e[j];
01197                     const WORD degree = e[j+1];
01198                     auto it = var_map.find(symbol);
01199                     if ( it == var_map.end() ) {
01200                         var_map[symbol] = num_vars++;
01201                         degrees.push_back(degree);
01202                     }
01203                     else {
01204                         degrees[it->second] = MaX(degrees[it->second], degree);
01205                     }
01206                 }
01207             }
01208         }
01209     }
01210
01211     if ( sort_vars ) {
01212         // bubble sort variables in decreasing order of degree
01213         // (this seems better for factorization)
01214         for ( size_t i = 0; i < var_map.size(); i++ ) {
01215             for ( size_t j = 0; j+1 < var_map.size(); j++ ) {
01216                 if ( degrees[j] < degrees[j+1] ) {
01217                     swap(degrees[j], degrees[j+1]);
```

```
01218
01219                     // Find the map keys associated with the values we want to swap
01220                     uint32_t j0 = 0;
01221                     uint32_t j1 = 0;
01222                     for ( auto x: var_map ) {
01223                         if ( x.second == j ) {
01224                             j0 = x.first;
01225                         }
01226                         else if ( x.second == j+1 ) {
01227                             j1 = x.first;
01228                         }
01229                     }
01230                     swap(var_map.at(j0), var_map.at(j1));
01231                 }
01232             }
01233         }
01234     }
01235     // Otherwise, sort lexicographically in FORM's ordering
01236     else {
01237         for ( size_t i = 0; i < var_map.size(); i++ ) {
01238             for ( size_t j = 0; j+1 < var_map.size(); j++ ) {
01239                 uint32_t j0 = 0;
01240                 uint32_t j1 = 0;
01241                 for ( auto x: var_map ) {
01242                     if ( x.second == j ) {
01243                         j0 = x.first;
01244                     }
01245                     else if ( x.second == j+1 ) {
01246                         j1 = x.first;
01247                     }
01248                 }
01249                 if ( j0 > j1 ) {
01250                     swap(var_map.at(j0), var_map.at(j1));
01251                 }
01252             }
01253         }
01254     }
01255
01256     if ( var_map.size() == 1 ) {
01257         // In the univariate case, if the polynomials are sufficiently sparse force the use of the
01258         // multivariate routines, which use a sparse representation, by adding a dummy map entry.
01259         if ( (float)num_terms <= UNIVARIATE_DENSITY_THR * (float)degrees[0] ) {
01260             // -1 will never be a symbol code. Built-in symbols from 0 to 19, and 20 is the first
01261             // user symbol.
01262             var_map[-1] = num_vars;
01263         }
01264     }
01265
01266     return var_map;
01267 }
01268 /*
01269     #] flint::get_variables :
01270
01271     #[ flint::inverse_poly :
01272 */
01273 WORD* flint::inverse_poly(PHEAD const WORD *a, const WORD *b, const var_map_t &var_map) {
01274
01275     flint::poly pa, pb, denpa, denpb;
01276
01277     flint::from_argument_poly(pa.d, denpa.d, a, false);
01278     flint::from_argument_poly(pb.d, denpb.d, b, false);
01279
01280     // fmpz_poly_xgcd is undefined if the content of pa and pb are not 1. Take the content out.
01281     // fmpz_poly_content gives the non-negative content and fmpz_poly_primitive normalizes to a
01282     // non-negative lcoeff, so we need to add the sign to the content if the polys have a negative
01283     // lcoeff. We don't need to keep the content of pb, it is a numerical multiple of the modulus.
01284     flint::fmpz content_a, resultant;
01285     flint::poly inverse, tmp;
01286     fmpz_poly_content(content_a.d, pa.d);
01287     if ( fmpz_sgn(fmpz_poly_lead(pa.d)) == -1 ) {
01288         fmpz_neg(content_a.d, content_a.d);
01289     }
01290     fmpz_poly_primitive_part(pa.d, pa.d);
01291     fmpz_poly_primitive_part(pb.d, pb.d);
01292
01293     // Special cases:
01294     // Possibly strange that we give 1 for inverse_(x1,1) but here we take MMA's convention.
01295     if ( fmpz_poly_is_one(pa.d) && fmpz_poly_is_one(pb.d) ) {
01296         fmpz_poly_one(inverse.d);
01297         fmpz_one(resultant.d);
01298     }
01299     else if ( fmpz_poly_is_one(pb.d) ) {
01300         fmpz_poly_zero(inverse.d);
01301         fmpz_one(resultant.d);
01302     }
01303     else {
01304         // Now use the extended Euclidean algorithm to find inverse, resultant, tmp of the Bezout
```

```
01305         // identity: inverse*pa + tmp*pb = resultant. Then inverse/resultant is the multiplicative
01306         // inverse of pa mod pb. We'll divide by resultant in the denscale argument of
    to_argument_poly.
01307         fmpz_poly_xgcd(resultant.d, inverse.d, tmp.d, pa.d, pb.d);
01308 
01309         // If the resultant is zero, the inverse does not exist:
01310         if ( fmpz_is_zero(resultant.d) ) {
01311             MLOCK(ErrorMessageLock);
01312             MesPrint("flint::inverse_poly error: inverse does not exist");
01313             MUNLOCK(ErrorMessageLock);
01314             Terminate(-1);
01315         }
01316     }
01317 
01318     // Multiply inverse by denpa. denpb is a numerical multiple of the modulus, so doesn't matter.
01319     // We also need to divide by content_a, which we do in the denscale argument of to_argument_poly
01320     // by multiplying resultant by content_a here.
01321     fmpz_poly_mul(inverse.d, inverse.d, denpa.d);
01322     fmpz_mul(resultant.d, resultant.d, content_a.d);
01323 
01324 
01325     WORD* res;
01326     // First determine the result size, and malloc. The result should have no arghead. Here we use
01327     // the "scale" argument of to_argument_poly since resultant might not be 1.
01328     const bool with_arghead = false;
01329     bool write = false;
01330     const bool must_fit_term = false;
01331     const uint64_t prev_size = 0;
01332     const uint64_t res_size = (uint64_t)flint::to_argument_poly(BHEAD NULL,
01333         with_arghead, must_fit_term, write, prev_size, inverse.d, var_map, resultant.d);
01334     res = (WORD*)Malloc1(sizeof(WORD)*res_size, "flint::inverse_poly");
01335 
01336     write = true;
01337     flint::to_argument_poly(BHEAD res, with_arghead, must_fit_term, write, prev_size, inverse.d,
01338         var_map, resultant.d);
01339 
01340     return res;
01341 }
01342 /*
01343     #] flint::inverse_poly :
01344 
01345     #[ flint::mul_mpoly :
01346 */
01347 // Return a pointer to a buffer containing the product of the 0-terminated term lists at a and b.
01348 // For multi-variate cases.
01349 WORD* flint::mul_mpoly(PHEAD const WORD *a, const WORD *b, const var_map_t &var_map,
01350     const bool sort_vars) {
01351 
01352     flint::mpoly_ctx ctx(var_map.size());
01353     flint::mpoly pa(ctx.d), pb(ctx.d), denpa(ctx.d), denpb(ctx.d);
01354 
01355     flint::from_argument_mpoly(pa.d, denpa.d, a, false, var_map, ctx.d);
01356     flint::from_argument_mpoly(pb.d, denpb.d, b, false, var_map, ctx.d);
01357 
01358     // denpa, denpb must be integers. Negative symbol powers have been converted to extra symbols.
01359     if ( fmpz_mpoly_is_fmpz(denpa.d, ctx.d) != 1 ) {
01360         // INTERNAL_ERROR_EXCL_START
01361         MLOCK(ErrorMessageLock);
01362         MesPrint("!>flint::mul_mpoly: error: denpa is non-constant");
01363         MUNLOCK(ErrorMessageLock);
01364         Terminate(-1);
01365         // INTERNAL_ERROR_EXCL_STOP
01366     }
01367     if ( fmpz_mpoly_is_fmpz(denpb.d, ctx.d) != 1 ) {
01368         // INTERNAL_ERROR_EXCL_START
01369         MLOCK(ErrorMessageLock);
01370         MesPrint("!>flint::mul_mpoly: error: denpb is non-constant");
01371         MUNLOCK(ErrorMessageLock);
01372         Terminate(-1);
01373         // INTERNAL_ERROR_EXCL_STOP
01374     }
01375 
01376     // Multiply numerators, store result in pa
01377     fmpz_mpoly_mul(pa.d, pa.d, pb.d, ctx.d);
01378     // Multiply denominators, store result in denpa, and convert to an fmpz:
01379     fmpz_mpoly_mul(denpa.d, denpa.d, denpb.d, ctx.d);
01380     flint::fmpz den;
01381     fmpz_mpoly_get_fmpz(den.d, denpa.d, ctx.d);
01382 
01383     WORD* res;
01384     // First determine the result size, and malloc. The result should have no arghead. Here we use
01385     // the "scale" argument of to_argument_mpoly since den might not be 1.
01386     const bool with_arghead = false;
01387     bool write = false;
01388     const bool must_fit_term = false;
01389     const uint64_t prev_size = 0;
01390     const uint64_t mul_size = (uint64_t)flint::to_argument_mpoly(BHEAD NULL,
```

```
01391         with_arghead, must_fit_term, write, prev_size, pa.d, var_map, ctx.d, sort_vars, den.d);
01392     res = (WORD*)Malloc1(sizeof(WORD)*mul_size, "flint::mul_mpoly");
01393 
01394     write = true;
01395     flint::to_argument_mpoly(BHEAD res, with_arghead, must_fit_term, write, prev_size, pa.d,
01396         var_map, ctx.d, sort_vars, den.d);
01397 
01398     return res;
01399 }
01400 /*
01401     #] flint::mul_mpoly :
01402     #[ flint::mul_poly :
01403 */
01404 // Return a pointer to a buffer containing the product of the 0-terminated term lists at a and b.
01405 // For uni-variate cases.
01406 WORD* flint::mul_poly(PHEAD const WORD *a, const WORD *b, const var_map_t &var_map) {
01407 
01408     flint::poly pa, pb, denpa, denpb;
01409 
01410     flint::from_argument_poly(pa.d, denpa.d, a, false);
01411     flint::from_argument_poly(pb.d, denpb.d, b, false);
01412 
01413     // denpa, denpb must be integers. Negative symbol powers have been converted to extra symbols.
01414     if ( fmpz_poly_degree(denpa.d) != 0 ) {
01415         // INTERNAL_ERROR_EXCL_START
01416         MLOCK(ErrorMessageLock);
01417         MesPrint("!>flint::mul_poly: error: denpa is non-constant");
01418         MUNLOCK(ErrorMessageLock);
01419         Terminate(-1);
01420         // INTERNAL_ERROR_EXCL_STOP
01421     }
01422     if ( fmpz_poly_degree(denpb.d) != 0 ) {
01423         // INTERNAL_ERROR_EXCL_START
01424         MLOCK(ErrorMessageLock);
01425         MesPrint("!>flint::mul_poly: error: denpb is non-constant");
01426         MUNLOCK(ErrorMessageLock);
01427         Terminate(-1);
01428         // INTERNAL_ERROR_EXCL_STOP
01429     }
01430 
01431     // Multiply numerators, store result in pa
01432     fmpz_poly_mul(pa.d, pa.d, pb.d);
01433     // Multiply denominators, store result in denpa, and convert to an fmpz:
01434     fmpz_poly_mul(denpa.d, denpa.d, denpb.d);
01435     flint::fmpz den;
01436     fmpz_poly_get_coeff_fmpz(den.d, denpa.d, 0);
01437 
01438     WORD* res;
01439     // First determine the result size, and malloc. The result should have no arghead. Here we use
01440     // the "scale" argument of to_argument_poly since den might not be 1.
01441     const bool with_arghead = false;
01442     bool write = false;
01443     const bool must_fit_term = false;
01444     const uint64_t prev_size = 0;
01445     const uint64_t mul_size = (uint64_t)flint::to_argument_poly(BHEAD NULL,
01446         with_arghead, must_fit_term, write, prev_size, pa.d, var_map, den.d);
01447     res = (WORD*)Malloc1(sizeof(WORD)*mul_size, "flint::mul_poly");
01448 
01449     write = true;
01450     flint::to_argument_poly(BHEAD res, with_arghead, must_fit_term, write, prev_size, pa.d,
01451         var_map, den.d);
01452 
01453     return res;
01454 }
01455 /*
01456     #] flint::mul_poly :
01457 
01458     #[ flint::ratfun_add_mpoly :
01459 */
01460 // Add the multi-variate FORM rational polynomials at t1 and t2. The result is written at out.
01461 void flint::ratfun_add_mpoly(PHEAD const WORD *t1, const WORD *t2, WORD *out,
01462     const var_map_t &var_map, const bool sort_vars) {
01463 
01464     flint::mpoly_ctx ctx(var_map.size());
01465     flint::mpoly gcd(ctx.d), num1(ctx.d), den1(ctx.d), num2(ctx.d), den2(ctx.d);
01466 
01467     flint::ratfun_read_mpoly(t1, num1.d, den1.d, var_map, ctx.d);
01468     flint::ratfun_read_mpoly(t2, num2.d, den2.d, var_map, ctx.d);
01469 
01470     if ( fmpz_mpoly_cmp(den1.d, den2.d, ctx.d) != 0 ) {
01471         fmpz_mpoly_gcd_cofactors(gcd.d, den1.d, den2.d, den1.d, den2.d, ctx.d);
01472 
01473         fmpz_mpoly_mul(num1.d, num1.d, den2.d, ctx.d);
01474         fmpz_mpoly_mul(num2.d, num2.d, den1.d, ctx.d);
01475 
01476         fmpz_mpoly_add(num1.d, num1.d, num2.d, ctx.d);
01477         fmpz_mpoly_mul(den1.d, den1.d, den2.d, ctx.d);
```

```
01478         fmpz_mpoly_mul(den1.d, den1.d, gcd.d,  ctx.d);
01479     }
01480     else {
01481         fmpz_mpoly_add(num1.d, num1.d, num2.d, ctx.d);
01482     }
01483 
01484     // Finally divide out any common factors between the resulting num1, den1:
01485     fmpz_mpoly_gcd_cofactors(gcd.d, num1.d, den1.d, num1.d, den1.d, ctx.d);
01486 
01487     flint::util::fix_sign_fmpz_mpoly_ratfun(num1.d, den1.d, ctx.d);
01488 
01489     // Result in FORM notation:
01490     *out++ = AR.PolyFun;
01491     WORD* args_size = out++;
01492     WORD* args_flag = out++;
01493     *args_flag = 0; // clean prf
01494     FILLFUN3(out); // Remainder of funhead, if it is larger than 3
01495 
01496     const bool with_arghead = true;
01497     const bool must_fit_term = true;
01498     const bool write = true;
01499     // prev_size + 4, to account for final term size and coeff of "1/1"
01500     out += flint::to_argument_mpoly(BHEAD out, with_arghead, must_fit_term, write, out-args_size+4,
01501         num1.d, var_map, ctx.d, sort_vars);
01502     out += flint::to_argument_mpoly(BHEAD out, with_arghead, must_fit_term, write, out-args_size+4,
01503         den1.d, var_map, ctx.d, sort_vars);
01504 
01505     *args_size = out - args_size + 1; // The +1 is to include the function ID
01506     AT.WorkPointer = out;
01507 }
01508 /*
01509     #] flint::ratfun_add_mpoly :
01510     #[ flint::ratfun_add_poly :
01511 */
01512 // Add the uni-variate FORM rational polynomials at t1 and t2. The result is written at out.
01513 void flint::ratfun_add_poly(PHEAD const WORD *t1, const WORD *t2, WORD *out,
01514     const var_map_t &var_map) {
01515 
01516     flint::poly gcd, num1, den1, num2, den2;
01517 
01518     flint::ratfun_read_poly(t1, num1.d, den1.d);
01519     flint::ratfun_read_poly(t2, num2.d, den2.d);
01520 
01521     if ( fmpz_poly_equal(den1.d, den2.d) == 0 ) {
01522         flint::util::simplify_fmpz_poly(den1.d, den2.d, gcd.d);
01523 
01524         fmpz_poly_mul(num1.d, num1.d, den2.d);
01525         fmpz_poly_mul(num2.d, num2.d, den1.d);
01526 
01527         fmpz_poly_add(num1.d, num1.d, num2.d);
01528         fmpz_poly_mul(den1.d, den1.d, den2.d);
01529         fmpz_poly_mul(den1.d, den1.d, gcd.d);
01530     }
01531     else {
01532         fmpz_poly_add(num1.d, num1.d, num2.d);
01533     }
01534 
01535     // Finally divide out any common factors between the resulting num1, den1:
01536     flint::util::simplify_fmpz_poly(num1.d, den1.d, gcd.d);
01537 
01538     flint::util::fix_sign_fmpz_poly_ratfun(num1.d, den1.d);
01539 
01540     // Result in FORM notation:
01541     *out++ = AR.PolyFun;
01542     WORD* args_size = out++;
01543     WORD* args_flag = out++;
01544     *args_flag = 0; // clean prf
01545     FILLFUN3(out); // Remainder of funhead, if it is larger than 3
01546 
01547     const bool with_arghead = true;
01548     const bool must_fit_term = true;
01549     const bool write = true;
01550     // prev_size + 4, to account for final term size and coeff of "1/1"
01551     out += flint::to_argument_poly(BHEAD out, with_arghead, must_fit_term, write, out-args_size+4,
01552         num1.d, var_map);
01553     out += flint::to_argument_poly(BHEAD out, with_arghead, must_fit_term, write, out-args_size+4,
01554         den1.d, var_map);
01555 
01556     *args_size = out - args_size + 1; // The +1 is to include the function ID
01557     AT.WorkPointer = out;
01558 }
01559 /*
01560     #] flint::ratfun_add_poly :
01561 
01562     #[ flint::ratfun_normalize_mpoly :
01563 */
01564 // Multiply and simplify occurrences of the multi-variate FORM rational polynomials found in term.
```

```
01565 // The final term is written in place, with the rational polynomial at the end.
01566 void flint::ratfun_normalize_mpoly(PHEAD WORD *term, const var_map_t &var_map,
01567     const bool sort_vars) {
01568
01569     // The length of the coefficient
01570     const WORD ncoeff = (term + *term)[-1];
01571     // The end of the term data, before the coefficient:
01572     const WORD *tstop = term + *term - ABS(ncoeff);
01573
01574     flint::mpoly_ctx ctx(var_map.size());
01575     flint::mpoly num1(ctx.d), den1(ctx.d), num2(ctx.d), den2(ctx.d), gcd(ctx.d);
01576
01577     // Start with "trivial" polynomials, and multiply in the num and den of the prf which appear.
01578     flint::fmpz tmpNum, tmpDen;
01579     flint::fmpz_set_form(tmpNum.d, (UWORD*)tstop, ncoeff/2);
01580     flint::fmpz_set_form(tmpDen.d, (UWORD*)tstop+ABS(ncoeff/2), ABS(ncoeff/2));
01581     fmpz_mpoly_set_fmpz(num1.d, tmpNum.d, ctx.d);
01582     fmpz_mpoly_set_fmpz(den1.d, tmpDen.d, ctx.d);
01583
01584     // Loop over the occurrences of PolyFun in the term, and multiply in to num1, den1.
01585     // s tracks where we are writing the non-PolyFun term data. The final PolyFun will
01586     // go at the end.
01587     WORD* term_size = term;
01588     WORD* s = term + 1;
01589     for ( WORD *t = term + 1; t < tstop; ) {
01590         if ( *t == AR.PolyFun ) {
01591             flint::ratfun_read_mpoly(t, num2.d, den2.d, var_map, ctx.d);
01592
01593             // get gcd of num1,den2 and num2,den1 and then assemble
01594             fmpz_mpoly_gcd_cofactors(gcd.d, num1.d, den2.d, num1.d, den2.d, ctx.d);
01595             fmpz_mpoly_gcd_cofactors(gcd.d, num2.d, den1.d, num2.d, den1.d, ctx.d);
01596
01597             fmpz_mpoly_mul(num1.d, num1.d, num2.d, ctx.d);
01598             fmpz_mpoly_mul(den1.d, den1.d, den2.d, ctx.d);
01599
01600             t += t[1];
01601         }
01602
01603         else {
01604             // Not a PolyFun, just copy or skip over
01605             WORD i = t[1];
01606             if ( s != t ) { NCOPY(s,t,i); }
01607             else { t += i; s += i; }
01608         }
01609     }
01610
01611     flint::util::fix_sign_fmpz_mpoly_ratfun(num1.d, den1.d, ctx.d);
01612
01613     // Result in FORM notation:
01614     WORD* out = s;
01615     *out++ = AR.PolyFun;
01616     WORD* args_size = out++;
01617     WORD* args_flag = out++;
01618     *args_flag &= ~MUSTCLEANPRF;
01619
01620     const bool with_arghead = true;
01621     const bool must_fit_term = true;
01622     const bool write = true;
01623     out += flint::to_argument_mpoly(BHEAD out, with_arghead, must_fit_term, write, out-term_size,
01624         num1.d, var_map, ctx.d, sort_vars);
01625     out += flint::to_argument_mpoly(BHEAD out, with_arghead, must_fit_term, write, out-term_size,
01626         den1.d, var_map, ctx.d, sort_vars);
01627
01628     *args_size = out - args_size + 1; // The +1 is to include the function ID
01629
01630     // +3 for the coefficient of 1/1, which is added after the check
01631     if ( sizeof(WORD)*(out-term_size+3) > (size_t)AM.MaxTer ) {
01632         MLOCK(ErrorMessageLock);
01633         MesPrint("flint::ratfun_normalize: output exceeds MaxTermSize");
01634         MUNLOCK(ErrorMessageLock);
01635         Terminate(-1);
01636     }
01637
01638     *out++ = 1;
01639     *out++ = 1;
01640     *out++ = 3; // the term's coefficient is now 1/1
01641
01642     *term_size = out - term_size;
01643 }
01644 /*
01645     #] flint::ratfun_normalize_mpoly :
01646     #[ flint::ratfun_normalize_poly :
01647 */
01648 // Multiply and simplify occurrences of the uni-variate FORM rational polynomials found in term.
01649 // The final term is written in place, with the rational polynomial at the end.
01650 void flint::ratfun_normalize_poly(PHEAD WORD *term, const var_map_t &var_map) {
01651
```

```
01652     // The length of the coefficient
01653     const WORD ncoeff = (term + *term)[-1];
01654     // The end of the term data, before the coefficient:
01655     const WORD *tstop = term + *term - ABS(ncoeff);
01656 
01657     flint::poly num1, den1, num2, den2, gcd;
01658 
01659     // Start with "trivial" polynomials, and multiply in the num and den of the prf which appear.
01660     flint::fmpz tmpNum, tmpDen;
01661     flint::fmpz_set_form(tmpNum.d, (UWORD*)tstop, ncoeff/2);
01662     flint::fmpz_set_form(tmpDen.d, (UWORD*)tstop+ABS(ncoeff/2), ABS(ncoeff/2));
01663     fmpz_poly_set_fmpz(num1.d, tmpNum.d);
01664     fmpz_poly_set_fmpz(den1.d, tmpDen.d);
01665 
01666     // Loop over the occurrences of PolyFun in the term, and multiply in to num1, den1.
01667     // s tracks where we are writing the non-PolyFun term data. The final PolyFun will
01668     // go at the end.
01669     WORD* term_size = term;
01670     WORD* s = term + 1;
01671     for ( WORD *t = term + 1; t < tstop; ) {
01672         if ( *t == AR.PolyFun ) {
01673             flint::ratfun_read_poly(t, num2.d, den2.d);
01674 
01675             // get gcd of num1,den2 and num2,den1 and then assemble
01676             flint::util::simplify_fmpz_poly(num1.d, den2.d, gcd.d);
01677             flint::util::simplify_fmpz_poly(num2.d, den1.d, gcd.d);
01678 
01679             fmpz_poly_mul(num1.d, num1.d, num2.d);
01680             fmpz_poly_mul(den1.d, den1.d, den2.d);
01681 
01682             t += t[1];
01683         }
01684 
01685         else {
01686             // Not a PolyFun, just copy or skip over
01687             WORD i = t[1];
01688             if ( s != t ) { NCOPY(s,t,i); }
01689             else { t += i; s += i; }
01690         }
01691     }
01692 
01693     flint::util::fix_sign_fmpz_poly_ratfun(num1.d, den1.d);
01694 
01695     // Result in FORM notation:
01696     WORD* out = s;
01697     *out++ = AR.PolyFun;
01698     WORD* args_size = out++;
01699     WORD* args_flag = out++;
01700     *args_flag &= ~MUSTCLEANPRF;
01701 
01702     const bool with_arghead = true;
01703     const bool must_fit_term = true;
01704     const bool write = true;
01705     out += flint::to_argument_poly(BHEAD out, with_arghead, must_fit_term, write, out-term_size,
01706         num1.d, var_map);
01707     out += flint::to_argument_poly(BHEAD out, with_arghead, must_fit_term, write, out-term_size,
01708         den1.d, var_map);
01709 
01710     *args_size = out - args_size + 1; // The +1 is to include the function ID
01711 
01712     // +3 for the coefficient of 1/1, which is added after the check
01713     if ( sizeof(WORD)*(out-term_size+3) > (size_t)AM.MaxTer ) {
01714         MLOCK(ErrorMessageLock);
01715         MesPrint("flint::ratfun_normalize: output exceeds MaxTermSize");
01716         MUNLOCK(ErrorMessageLock);
01717         Terminate(-1);
01718     }
01719 
01720     *out++ = 1;
01721     *out++ = 1;
01722     *out++ = 3; // the term's coefficient is now 1/1
01723 
01724     *term_size = out - term_size;
01725 }
01726 /*
01727     #] flint::ratfun_normalize_poly :
01728 
01729     #[ flint::ratfun_read_mpoly :
01730 */
01731 // Read the multi-variate FORM rational polynomial at a and create fmpz_mpoly_t numerator and
01732 // denominator.
01733 void flint::ratfun_read_mpoly(const WORD *a, fmpz_mpoly_t num, fmpz_mpoly_t den,
01734     const var_map_t &var_map, fmpz_mpoly_ctx_t ctx) {
01735 
01736     // The end of the arguments:
01737     const WORD* arg_stop = a+a[1];
01738 
```

```
01739     const bool must_normalize = (a[2] & MUSTCLEANPRF) != 0;
01740
01741     a += FUNHEAD;
01742     if ( a >= arg_stop ) {
01743         MLOCK(ErrorMessageLock);
01744         MesPrint("ERROR: PolyRatFun cannot have zero arguments");
01745         MUNLOCK(ErrorMessageLock);
01746         Terminate(-1);
01747     }
01748
01749     // Polys to collect the "den of the num" and "den of the den".
01750     // Input can arrive like this when enabling the PolyRatFun or moving things into it.
01751     flint::mpoly den_num(ctx), den_den(ctx);
01752
01753     // Read the numerator
01754     flint::from_argument_mpoly(num, den_num.d, a, true, var_map, ctx);
01755     NEXTARG(a);
01756
01757     if ( a < arg_stop ) {
01758         // Read the denominator
01759         flint::from_argument_mpoly(den, den_den.d, a, true, var_map, ctx);
01760         NEXTARG(a);
01761     }
01762     else {
01763         // The denominator is 1
01764 /* INTERNAL_ERROR_EXCL_START */
01765         MLOCK(ErrorMessageLock);
01766         MesPrint("!>implement this");
01767         MUNLOCK(ErrorMessageLock);
01768         Terminate(-1);
01769 /* INTERNAL_ERROR_EXCL_STOP */
01770     }
01771     if ( a < arg_stop ) {
01772         MLOCK(ErrorMessageLock);
01773         MesPrint("ERROR: PolyRatFun cannot have more than two arguments");
01774         MUNLOCK(ErrorMessageLock);
01775         Terminate(-1);
01776     }
01777
01778     // Multiply the num by den_den and den by den_num:
01779     fmpz_mpoly_mul(num, num, den_den.d, ctx);
01780     fmpz_mpoly_mul(den, den, den_num.d, ctx);
01781
01782     if ( must_normalize ) {
01783         flint::mpoly gcd(ctx);
01784         fmpz_mpoly_gcd_cofactors(gcd.d, num, den, num, den, ctx);
01785     }
01786 }
01787 /*
01788     #] flint::ratfun_read_mpoly :
01789     #[ flint::ratfun_read_poly :
01790 */
01791 // Read the uni-variate FORM rational polynomial at a and create fmpz_mpoly_t numerator and
01792 // denominator.
01793 void flint::ratfun_read_poly(const WORD *a, fmpz_poly_t num, fmpz_poly_t den) {
01794
01795     // The end of the arguments:
01796     const WORD* arg_stop = a+a[1];
01797
01798     const bool must_normalize = (a[2] & MUSTCLEANPRF) != 0;
01799
01800     a += FUNHEAD;
01801     if ( a >= arg_stop ) {
01802         MLOCK(ErrorMessageLock);
01803         MesPrint("ERROR: PolyRatFun cannot have zero arguments");
01804         MUNLOCK(ErrorMessageLock);
01805         Terminate(-1);
01806     }
01807
01808     // Polys to collect the "den of the num" and "den of the den".
01809     // Input can arrive like this when enabling the PolyRatFun or moving things into it.
01810     flint::poly den_num, den_den;
01811
01812     // Read the numerator
01813     flint::from_argument_poly(num, den_num.d, a, true);
01814     NEXTARG(a);
01815
01816     if ( a < arg_stop ) {
01817         // Read the denominator
01818         flint::from_argument_poly(den, den_den.d, a, true);
01819         NEXTARG(a);
01820     }
01821     else {
01822         // The denominator is 1
01823 /* INTERNAL_ERROR_EXCL_START */
01824         MLOCK(ErrorMessageLock);
01825         MesPrint("!>implement this");
```

```
01826         MUNLOCK(ErrorMessageLock);
01827         Terminate(-1);
01828 /* INTERNAL_ERROR_EXCL_STOP */
01829     }
01830     if ( a < arg_stop ) {
01831         MLOCK(ErrorMessageLock);
01832         MesPrint("ERROR: PolyRatFun cannot have more than two arguments");
01833         MUNLOCK(ErrorMessageLock);
01834         Terminate(-1);
01835     }
01836 
01837     // Multiply the num by den_den and den by den_num:
01838     fmpz_poly_mul(num, num, den_den.d);
01839     fmpz_poly_mul(den, den, den_num.d);
01840 
01841     if ( must_normalize ) {
01842         flint::poly gcd;
01843         flint::util::simplify_fmpz_poly(num, den, gcd.d);
01844     }
01845 }
01846 /*
01847     #] flint::ratfun_read_poly :
01848     #[ flint::to_argument_mpoly :
01849 */
01850 // Convert a fmpz_mpoly_t to a FORM argument (or 0-terminated list of terms: with_arghead==false).
01851 // If the caller is building an output term, prev_size contains the size of the term so far, to
01852 // check that the output fits in AM.MaxTer if must_fit_term.
01853 // All coefficients will be divided by denscale (which might just be 1).
01854 // If write is false, we never write to out but only track the total would-be size. This lets this
01855 // function be repurposed as a "size of FORM notation" function without duplicating the code.
01856 #define IFW(x) { if ( write ) {x;} }
01857 uint64_t flint::to_argument_mpoly(PHEAD WORD *out, const bool with_arghead,
01858     const bool must_fit_term, const bool write, const uint64_t prev_size, const fmpz_mpoly_t poly,
01859     const var_map_t &var_map, const fmpz_mpoly_ctx_t ctx, const bool sort_vars,
01860     const fmpz_t denscale) {
01861 
01862     // out is modified later, keep the pointer at entry
01863     const WORD* out_entry = out;
01864 
01865     // Track the total size written. We could do this with pointer differences, but if
01866     // write == false we don't write to or move out to be able to find the size that way.
01867     uint64_t ws = 0;
01868 
01869     // Check there is at least space for ARGHEAD WORDs (the arghead or fast-notation number/symbol)
01870     if ( write && must_fit_term && (sizeof(WORD)*(prev_size + ARGHEAD) > (size_t)AM.MaxTer) ) {
01871         MLOCK(ErrorMessageLock);
01872         MesPrint("flint::to_argument_mpoly: output exceeds MaxTermSize");
01873         MUNLOCK(ErrorMessageLock);
01874         Terminate(-1);
01875     }
01876 
01877     // Create the inverse of var_map, so we don't have to search it for each symbol written
01878     vector<uint32_t> var_map_inv;
01879     var_map_inv.resize(var_map.size());
01880     for ( auto x : var_map ) {
01881         var_map_inv[x.second] = x.first;
01882     }
01883 
01884     vector<int64_t> exponents(var_map.size());
01885     const int64_t n_terms = fmpz_mpoly_length(poly, ctx);
01886 
01887     if ( n_terms == 0 ) {
01888         if ( with_arghead ) {
01889             IFW(*out++ = -SNUMBER); ws++;
01890             IFW(*out++ = 0); ws++;
01891             return ws;
01892         }
01893         else {
01894             IFW(*out++ = 0); ws++;
01895             return ws;
01896         }
01897     }
01898 
01899     // For dividing out denscale
01900     flint::fmpz coeff, den, gcd;
01901 
01902     // The mpoly might be constant or a single symbol with coeff 1. Use fast notation if possible.
01903     if ( with_arghead && n_terms == 1 ) {
01904 
01905         if ( fmpz_mpoly_is_fmpz(poly, ctx) ) {
01906             // The mpoly is constant. Use fast notation if the number is an integer and small enough:
01907 
01908             fmpz_mpoly_get_term_coeff_fmpz(coeff.d, poly, 0, ctx);
01909             fmpz_set(den.d, denscale);
01910             flint::util::simplify_fmpz(coeff.d, den.d, gcd.d);
01911 
01912             if ( fmpz_is_one(den.d) && fmpz_fits_si(coeff.d) ) {
```

```
01913                 const int64_t fast_coeff = fmpz_get_si(coeff.d);
01914                 // While ">=", could work here, FORM does not use fast notation for INT_MIN
01915                 if ( fast_coeff > INT32_MIN && fast_coeff <= INT32_MAX ) {
01916                     IFW(*out++ = -SNUMBER); ws++;
01917                     IFW(*out++ = (WORD)fast_coeff); ws++;
01918                     return ws;
01919                 }
01920             }
01921         }
01922 
01923         else {
01924             fmpz_mpoly_get_term_coeff_fmpz(coeff.d, poly, 0, ctx);
01925             fmpz_set(den.d, denscale);
01926             flint::util::simplify_fmpz(coeff.d, den.d, gcd.d);
01927 
01928             if ( fmpz_is_one(coeff.d) && fmpz_is_one(den.d) ) {
01929                 // The coefficient is one. Now check the symbol powers:
01930 
01931                 fmpz_mpoly_get_term_exp_si((slong*)exponents.data(), poly, 0, ctx);
01932                 int64_t use_fast = 0;
01933                 uint32_t fast_symbol = 0;
01934 
01935                 for ( size_t i = 0; i < var_map.size(); i++ ) {
01936                     if ( exponents[i] == 1 ) fast_symbol = var_map_inv[i];
01937                     use_fast += exponents[i];
01938                 }
01939 
01940                 // use_fast has collected the total degree. If it is 1, then fast_symbol holds the
      code
01941                 if ( use_fast == 1 ) {
01942                     IFW(*out++ = -SYMBOL); ws++;
01943                     IFW(*out++ = fast_symbol); ws++;
01944                     return ws;
01945                 }
01946             }
01947         }
01948     }
01949 
01950 
01951     WORD *tmp_coeff = (WORD *)NumberMalloc("flint::to_argument_mpoly");
01952     WORD *tmp_den = (WORD *)NumberMalloc("flint::to_argument_mpoly");
01953 
01954     WORD* arg_size = 0;
01955     WORD* arg_flag = 0;
01956     if ( with_arghead ) {
01957         IFW(arg_size = out++); ws++; // total arg size
01958         IFW(arg_flag = out++); ws++;
01959         IFW(*arg_flag = 0); // clean argument
01960     }
01961 
01962     for ( int64_t i = 0; i < n_terms; i++ ) {
01963 
01964         fmpz_mpoly_get_term_exp_si((slong*)exponents.data(), poly, i, ctx);
01965 
01966         fmpz_mpoly_get_term_coeff_fmpz(coeff.d, poly, i, ctx);
01967         fmpz_set(den.d, denscale);
01968         flint::util::simplify_fmpz(coeff.d, den.d, gcd.d);
01969 
01970         uint32_t num_symbols = 0;
01971         for ( size_t j = 0; j < var_map.size(); j++ ) {
01972             if ( exponents[j] != 0 ) { num_symbols += 1; }
01973         }
01974 
01975         // Convert the coefficient, write in temporary space
01976         const WORD num_size = flint::fmpz_get_form(coeff.d, tmp_coeff);
01977         const WORD den_size = flint::fmpz_get_form(den.d, tmp_den);
01978         const WORD coeff_size = [num_size, den_size] () -> WORD {
01979             WORD size = ABS(num_size) > ABS(den_size) ? ABS(num_size) : ABS(den_size);
01980             return size * SGN(num_size) * SGN(den_size);
01981         }();
01982 
01983         // Now we have the number of symbols and the coeff size, we can determine the output size.
01984         // Check it fits if necessary: term size, num,den of "coeff_size", +1 for total coeff size
01985         uint64_t current_size = prev_size + ws + 1 + 2*ABS(coeff_size) + 1;
01986         if ( num_symbols ) {
01987             // symbols header, code,power of each symbol:
01988             current_size += 2 + 2*num_symbols;
01989         }
01990         if ( write && must_fit_term && (sizeof(WORD)*current_size > (size_t)AM.MaxTer) ) {
01991             MLOCK(ErrorMessageLock);
01992             MesPrint("flint::to_argument_mpoly: output exceeds MaxTermSize");
01993             MUNLOCK(ErrorMessageLock);
01994             Terminate(-1);
01995         }
01996 
01997         WORD* term_size = 0;
01998         IFW(term_size = out++); ws++;
```

```
01999         if ( num_symbols ) {
02000             IFW(*out++ = SYMBOL); ws++;
02001             WORD* symbol_size = 0;
02002             IFW(symbol_size = out++); ws++;
02003             IFW(*symbol_size = 2);
02004 
02005             for ( size_t j = 0; j < var_map.size(); j++ ) {
02006                 if ( exponents[j] != 0 ) {
02007                     IFW(*out++ = var_map_inv[j]); ws++;
02008                     IFW(*out++ = exponents[j]); ws++;
02009                     IFW(*symbol_size += 2);
02010                 }
02011             }
02012         }
02013 
02014         // Copy numerator
02015         for ( WORD j = 0; j < ABS(num_size); j++ ) {
02016             IFW(*out++ = tmp_coeff[j]); ws++;
02017         }
02018         for ( WORD j = ABS(num_size); j < ABS(coeff_size); j++ ) {
02019             IFW(*out++ = 0); ws++;
02020         }
02021         // Copy denominator
02022         for ( WORD j = 0; j < ABS(den_size); j++ ) {
02023             IFW(*out++ = tmp_den[j]); ws++;
02024         }
02025         for ( WORD j = ABS(den_size); j < ABS(coeff_size); j++ ) {
02026             IFW(*out++ = 0); ws++;
02027         }
02028 
02029         IFW(*out = 2*ABS(coeff_size) + 1); // the size of the coefficient
02030         IFW(if ( coeff_size < 0 ) { *out = -(*out); });
02031         IFW(out++); ws++;
02032 
02033         IFW(*term_size = out - term_size);
02034     }
02035 
02036     if ( with_arghead ) {
02037         IFW(*arg_size = out - arg_size);
02038         if ( write && sort_vars ) {
02039             // Sort into form highfirst ordering, if we have potentially re-ordered the variables by
02040             // degree, in flint::get_variables. Otherwise, we can rely on FLINT's sorting with
02041             // ORD_LEX and the variables ordered in FORM's lexicographic order.
02042             flint::form_sort(BHEAD (WORD*)(out_entry));
02043         }
02044     }
02045     else {
02046         // with no arghead, we write a terminating zero
02047         IFW(*out++ = 0); ws++;
02048     }
02049 
02050     NumberFree(tmp_coeff, "flint::to_argument_mpoly");
02051     NumberFree(tmp_den, "flint::to_argument_mpoly");
02052 
02053     return ws;
02054 }
02055 
02056 // If no denscale argument is supplied, just set it to 1 and call the usual function
02057 uint64_t flint::to_argument_mpoly(PHEAD WORD *out, const bool with_arghead,
02058     const bool must_fit_term, const bool write, const uint64_t prev_size, const fmpz_mpoly_t poly,
02059     const var_map_t &var_map, const fmpz_mpoly_ctx_t ctx, const bool sort_vars) {
02060 
02061     flint::fmpz tmp;
02062     fmpz_set_ui(tmp.d, 1);
02063 
02064     uint64_t ret = flint::to_argument_mpoly(BHEAD out, with_arghead, must_fit_term, write,
02065         prev_size, poly, var_map, ctx, sort_vars, tmp.d);
02066 
02067     return ret;
02068 }
02069 /*
02070     #] flint::to_argument_mpoly :
02071     #[ flint::to_argument_poly :
02072 */
02073 // Convert a fmpz_poly_t to a FORM argument (or 0-terminated list of terms: with_arghead==false).
02074 // If the caller is building an output term, prev_size contains the size of the term so far, to
02075 // check that the output fits in AM.MaxTer if must_fit_term.
02076 // All coefficients will be divided by denscale (which might just be 1).
02077 // If write is false, we never write to out but only track the total would-be size. This lets this
02078 // function be repurposed as a "size of FORM notation" function without duplicating the code.
02079 uint64_t flint::to_argument_poly(PHEAD WORD *out, const bool with_arghead,
02080     const bool must_fit_term, const bool write, const uint64_t prev_size, const fmpz_poly_t poly,
02081     const var_map_t &var_map, const fmpz_t denscale) {
02082 
02083     // Track the total size written. We could do this with pointer differences, but if
02084     // write == false we don't write to or move out to be able to find the size that way.
02085     uint64_t ws = 0;
```

```
02086
02087     // Check there is at least space for ARGHEAD WORDs (the arghead or fast-notation number/symbol)
02088     if ( write && must_fit_term && (sizeof(WORD)*(prev_size + ARGHEAD) > (size_t)AM.MaxTer) ) {
02089         MLOCK(ErrorMessageLock);
02090         MesPrint("flint::to_argument_poly: output exceeds MaxTermSize");
02091         MUNLOCK(ErrorMessageLock);
02092         Terminate(-1);
02093     }
02094
02095     // Create the inverse of var_map, so we don't have to search it for each symbol written
02096     vector<uint32_t> var_map_inv;
02097     var_map_inv.resize(var_map.size());
02098     for ( auto x : var_map ) {
02099         var_map_inv[x.second] = x.first;
02100     }
02101
02102     const int64_t n_terms = fmpz_poly_length(poly);
02103
02104     // The poly is zero
02105     if ( n_terms == 0 ) {
02106         if ( with_arghead ) {
02107             IFW(*out++ = -SNUMBER); ws++;
02108             IFW(*out++ = 0); ws++;
02109             return ws;
02110         }
02111         else {
02112             IFW(*out++ = 0); ws++;
02113             return ws;
02114         }
02115     }
02116
02117     // For dividing out denscale
02118     flint::fmpz coeff, den, gcd;
02119
02120     // The poly is constant, use fast notation if the coefficient is integer and small enough
02121     if ( with_arghead && n_terms == 1 ) {
02122
02123         fmpz_poly_get_coeff_fmpz(coeff.d, poly, 0);
02124         fmpz_set(den.d, denscale);
02125         flint::util::simplify_fmpz(coeff.d, den.d, gcd.d);
02126
02127         if ( fmpz_is_one(den.d) && fmpz_fits_si(coeff.d) ) {
02128             const long fast_coeff = fmpz_get_si(coeff.d);
02129             // While ">=", could work here, FORM does not use fast notation for INT_MIN
02130             if ( fast_coeff > INT_MIN && fast_coeff <= INT_MAX ) {
02131                 IFW(*out++ = -SNUMBER); ws++;
02132                 IFW(*out++ = (WORD)fast_coeff); ws++;
02133                 return ws;
02134             }
02135         }
02136     }
02137
02138     // The poly might be a single symbol with coeff 1, use fast notation if so.
02139     if ( with_arghead && n_terms == 2 ) {
02140         if ( fmpz_is_zero(fmpz_poly_get_coeff_ptr(poly, 0)) ) {
02141             // The constant term is zero
02142
02143             fmpz_poly_get_coeff_fmpz(coeff.d, poly, 1);
02144             fmpz_set(den.d, denscale);
02145             flint::util::simplify_fmpz(coeff.d, den.d, gcd.d);
02146
02147             if ( fmpz_is_one(coeff.d) && fmpz_is_one(den.d) ) {
02148                 // Single symbol with coeff 1. Use fast notation:
02149                 IFW(*out++ = -SYMBOL); ws++;
02150                 IFW(*out++ = var_map_inv[0]); ws++;
02151                 return ws;
02152             }
02153         }
02154     }
02155
02156     WORD *tmp_coeff = (WORD *)NumberMalloc("flint::to_argument_poly");
02157     WORD *tmp_den = (WORD *)NumberMalloc("flint::to_argument_mpoly");
02158
02159     WORD* arg_size = 0;
02160     WORD* arg_flag = 0;
02161     if ( with_arghead ) {
02162         IFW(arg_size = out++); ws++; // total arg size
02163         IFW(arg_flag = out++); ws++;
02164         IFW(*arg_flag = 0); // clean argument
02165     }
02166
02167     // In reverse, since we want a "highfirst" output
02168     for ( int64_t i = n_terms-1; i >= 0; i-- ) {
02169
02170         // fmpz_poly is dense, there might be many zero coefficients:
02171         if ( !fmpz_is_zero(fmpz_poly_get_coeff_ptr(poly, i)) ) {
02172
```

```
02173             fmpz_poly_get_coeff_fmpz(coeff.d, poly, i);
02174             fmpz_set(den.d, denscale);
02175             flint::util::simplify_fmpz(coeff.d, den.d, gcd.d);
02176 
02177             // Convert the coefficient, write in temporary space
02178             const WORD num_size = flint::fmpz_get_form(coeff.d, tmp_coeff);
02179             const WORD den_size = flint::fmpz_get_form(den.d, tmp_den);
02180             const WORD coeff_size = [num_size, den_size] () -> WORD {
02181                 WORD size = ABS(num_size) > ABS(den_size) ? ABS(num_size) : ABS(den_size);
02182                 return size * SGN(num_size) * SGN(den_size);
02183             }();
02184 
02185             // Now we have the coeff size, we can determine the output size
02186             // Check it fits if necessary: symbol code,power, num,den of "coeff_size",
02187             // +1 for total coeff size
02188             uint64_t current_size = prev_size + ws + 1 + 2*ABS(coeff_size) + 1;
02189             if ( i > 0 ) {
02190                 // and also symbols header, code,power of the symbol
02191                 current_size += 4;
02192             }
02193             if ( write && must_fit_term && (sizeof(WORD)*current_size > (size_t)AM.MaxTer) ) {
02194                 MLOCK(ErrorMessageLock);
02195                 MesPrint("flint::to_argument_poly: output exceeds MaxTermSize");
02196                 MUNLOCK(ErrorMessageLock);
02197                 Terminate(-1);
02198             }
02199 
02200             WORD* term_size = 0;
02201             IFW(term_size = out++); ws++;
02202 
02203             if ( i > 0 ) {
02204                 IFW(*out++ = SYMBOL); ws++;
02205                 IFW(*out++ = 4); ws++; // The symbol array size, it is univariate
02206                 IFW(*out++ = var_map_inv[0]); ws++;
02207                 IFW(*out++ = i); ws++;
02208             }
02209 
02210             // Copy numerator
02211             for ( WORD j = 0; j < ABS(num_size); j++ ) {
02212                 IFW(*out++ = tmp_coeff[j]); ws++;
02213             }
02214             for ( WORD j = ABS(num_size); j < ABS(coeff_size); j++ ) {
02215                 IFW(*out++ = 0); ws++;
02216             }
02217             // Copy denominator
02218             for ( WORD j = 0; j < ABS(den_size); j++ ) {
02219                 IFW(*out++ = tmp_den[j]); ws++;
02220             }
02221             for ( WORD j = ABS(den_size); j < ABS(coeff_size); j++ ) {
02222                 IFW(*out++ = 0); ws++;
02223             }
02224 
02225             IFW(*out = 2*ABS(coeff_size) + 1); // the size of the coefficient
02226             IFW(if ( coeff_size < 0 ) { *out = -(*out); });
02227             IFW(out++); ws++;
02228 
02229             IFW(*term_size = out - term_size);
02230         }
02231 
02232     }
02233 
02234     if ( with_arghead ) {
02235         IFW(*arg_size = out - arg_size);
02236     }
02237     else {
02238         // with no arghead, we write a terminating zero
02239         IFW(*out++ = 0); ws++;
02240     }
02241 
02242     NumberFree(tmp_coeff, "flint::to_argument_poly");
02243     NumberFree(tmp_den, "flint::to_argument_poly");
02244 
02245     return ws;
02246 }
02247 
02248 // If no denscale argument is supplied, just set it to 1 and call the usual function
02249 uint64_t flint::to_argument_poly(PHEAD WORD *out, const bool with_arghead,
02250     const bool must_fit_term, const bool write, const uint64_t prev_size, const fmpz_poly_t poly,
02251     const var_map_t &var_map) {
02252 
02253     flint::fmpz tmp;
02254     fmpz_set_ui(tmp.d, 1);
02255 
02256     uint64_t ret = flint::to_argument_poly(BHEAD out, with_arghead, must_fit_term, write, prev_size,
02257         poly, var_map, tmp.d);
02258 
02259     return ret;
```

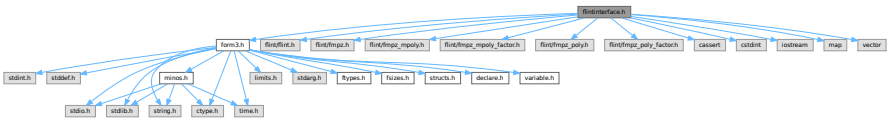
```
02260 }
02261 /*
02262     #] flint::to_argument_poly :
02263 */
02264
02265 // Utility functions
02266 /*
02267     #[ flint::util::simplify_fmpz :
02268 */
02269 // Divide the GCD out of num and den
02270 inline void flint::util::simplify_fmpz(fmpz_t num, fmpz_t den, fmpz_t gcd) {
02271     fmpz_gcd(gcd, num, den);
02272     if ( !fmpz_is_one(gcd) ) {
02273         fmpz_divexact(num, num, gcd);
02274         fmpz_divexact(den, den, gcd);
02275     }
02276 }
02277 /*
02278     #] flint::util::simplify_fmpz :
02279     #[ flint::util::simplify_fmpz_poly :
02280 */
02281 // Divide the GCD out of num and den
02282 inline void flint::util::simplify_fmpz_poly(fmpz_poly_t num, fmpz_poly_t den, fmpz_poly_t gcd) {
02283     fmpz_poly_gcd(gcd, num, den);
02284     if ( !fmpz_poly_is_one(gcd) ) {
02285 #if __FLINT_RELEASE >= 30100
02286         // This should be faster than fmpz_poly_div, see https://github.com/flintlib/flint/pull/1766
02287         fmpz_poly_divexact(num, num, gcd);
02288         fmpz_poly_divexact(den, den, gcd);
02289 #else
02290         fmpz_poly_div(num, num, gcd);
02291         fmpz_poly_div(den, den, gcd);
02292 #endif
02293     }
02294 }
02295 /*
02296     #] flint::util::simplify_fmpz_poly :
02297     #[ flint::util::fix_sign_fmpz_mpoly_ratfun :
02298 */
02299 inline void flint::util::fix_sign_fmpz_mpoly_ratfun(fmpz_mpoly_t num, fmpz_mpoly_t den,
02300     const fmpz_mpoly_ctx_t ctx) {
02301     // Fix sign to align with poly: the leading denominator term should have a positive coeff
02302     if ( fmpz_sgn(fmpz_mpoly_term_coeff_ref(den, 0, ctx)) == -1 ) {
02303         fmpz_mpoly_neg(num, num, ctx);
02304         fmpz_mpoly_neg(den, den, ctx);
02305     }
02306 }
02307 /*
02308     #] flint::util::fix_sign_fmpz_mpoly_ratfun :
02309     #[ flint::util::fix_sign_fmpz_poly_ratfun :
02310 */
02311 inline void flint::util::fix_sign_fmpz_poly_ratfun(fmpz_poly_t num, fmpz_poly_t den) {
02312     // Fix sign to align with poly: the leading denominator term should have a positive coeff
02313     if ( fmpz_sgn(fmpz_poly_get_coeff_ptr(den, fmpz_poly_degree(den))) == -1 ) {
02314         fmpz_poly_neg(num, num);
02315         fmpz_poly_neg(den, den);
02316     }
02317 }
02318 /*
02319     #] flint::util::fix_sign_fmpz_poly_ratfun :
02320 */
```

4.43 flintinterface.h File Reference

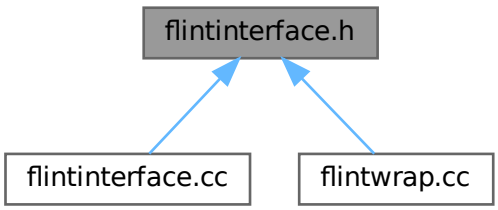
```
#include "form3.h"
#include <flint/flint.h>
#include <flint/fmpz.h>
#include <flint/fmpz_mpoly.h>
#include <flint/fmpz_mpoly_factor.h>
#include <flint/fmpz_poly.h>
#include <flint/fmpz_poly_factor.h>
#include <cassert>
#include <cstdint>
#include <iostream>
#include <map>
```

```
#include <vector>
```

Include dependency graph for flintinterface.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class flint::fmpz
- class flint::poly
- class flint::poly_factor
- class flint::mpoly
- class flint::mpoly_factor
- class flint::mpoly_ctx

Typedefs

- typedef std::map< uint32_t, uint32_t > flint::var_map_t

Functions

- void flint::cleanup (void)
- void flint::cleanup_master (void)
- WORD * flint::divmod_mpoly (PHEAD const WORD *, const WORD *, const bool, const WORD, const var↩ _map_t &, const bool)
- WORD * flint::divmod_poly (PHEAD const WORD *, const WORD *, const bool, const WORD, const var_↩ map_t &)
- WORD * flint::factorize_mpoly (PHEAD const WORD *, WORD *, const bool, const bool, const var_map_t &, const bool)
- WORD * flint::factorize_poly (PHEAD const WORD *, WORD *, const bool, const bool, const var_map_t &)
- void flint::form_sort (PHEAD WORD *)

- uint64_t flint::from_argument_mpoly (fmpz_mpoly_t, fmpz_mpoly_t, const WORD *, const bool, const var_map_t &, const fmpz_mpoly_ctx_t)
- uint64_t flint::from_argument_poly (fmpz_poly_t, fmpz_poly_t, const WORD *, const bool)
- WORD flint::fmpz_get_form (fmpz_t, WORD *)
- void flint::fmpz_set_form (fmpz_t, UWORD *, WORD)
- WORD * flint::gcd_mpoly (PHEAD const WORD *, const WORD *, const WORD, const var_map_t &, const bool)
- WORD * flint::gcd_poly (PHEAD const WORD *, const WORD *, const WORD, const var_map_t &)
- var_map_t flint::get_variables (const vector< WORD * > &, const bool, const bool)
- WORD * flint::inverse_poly (PHEAD const WORD *, const WORD *, const var_map_t &)
- WORD * flint::mul_mpoly (PHEAD const WORD *, const WORD *, const var_map_t &, const bool)
- WORD * flint::mul_poly (PHEAD const WORD *, const WORD *, const var_map_t &)
- void flint::ratfun_add_mpoly (PHEAD const WORD *, const WORD *, WORD *, const var_map_t &, const bool)
- void flint::ratfun_add_poly (PHEAD const WORD *, const WORD *, WORD *, const var_map_t &)
- void flint::ratfun_normalize_mpoly (PHEAD WORD *, const var_map_t &, const bool)
- void flint::ratfun_normalize_poly (PHEAD WORD *, const var_map_t &)
- void flint::ratfun_read_mpoly (const WORD *, fmpz_mpoly_t, fmpz_mpoly_t, const var_map_t &, fmpz_mpoly_ctx_t)
- void flint::ratfun_read_poly (const WORD *, fmpz_poly_t, fmpz_poly_t)
- uint64_t flint::to_argument_mpoly (PHEAD WORD *, const bool, const bool, const bool, const uint64_t, const fmpz_mpoly_t, const var_map_t &, const fmpz_mpoly_ctx_t, const bool)
- uint64_t flint::to_argument_mpoly (PHEAD WORD *, const bool, const bool, const bool, const uint64_t, const fmpz_mpoly_t, const var_map_t &, const fmpz_mpoly_ctx_t, const bool, const fmpz_t)
- uint64_t flint::to_argument_poly (PHEAD WORD *, const bool, const bool, const bool, const uint64_t, const fmpz_poly_t, const var_map_t &)
- uint64_t flint::to_argument_poly (PHEAD WORD *, const bool, const bool, const bool, const uint64_t, const fmpz_poly_t, const var_map_t &, const fmpz_t)
- void flint::util::simplify_fmpz (fmpz_t, fmpz_t, fmpz_t)
- void flint::util::simplify_fmpz_poly (fmpz_poly_t, fmpz_poly_t, fmpz_poly_t)
- void flint::util::fix_sign_fmpz_mpoly_ratfun (fmpz_mpoly_t, fmpz_mpoly_t, const fmpz_mpoly_ctx_t)
- void flint::util::fix_sign_fmpz_poly_ratfun (fmpz_poly_t, fmpz_poly_t)

4.43.1 Detailed Description

Prototypes for functions in flintinterface.cc

Definition in file flintinterface.h.

4.43.2 Typedef Documentation

4.43.2.1 var_map_t

```
typedef std::map<uint32_t,uint32_t> flint::var_map_t
```

Definition at line 77 of file flintinterface.h.

4.43.3 Function Documentation

4.43.3.1 cleanup()

```
void flint::cleanup (
            void  )
```

Definition at line 76 of file flintinterface.cc.

4.43.3.2 cleanup_master()

```
void flint::cleanup_master (
            void  )
```

Definition at line 83 of file flintinterface.cc.

4.43.3.3 divmod_mpoly()

```
WORD * flint::divmod_mpoly (
            PHEAD const WORD * a,
            const WORD * b,
            const bool return_rem,
            const WORD must_fit_term,
            const var_map_t & var_map,
            const bool sort_vars )
```

Definition at line 91 of file flintinterface.cc.

4.43.3.4 divmod_poly()

```
WORD * flint::divmod_poly (
            PHEAD const WORD * a,
            const WORD * b,
            const bool return_rem,
            const WORD must_fit_term,
            const var_map_t & var_map )
```

Definition at line 165 of file flintinterface.cc.

4.43.3.5 factorize_mpoly()

```
WORD * flint::factorize_mpoly (
            PHEAD const WORD * argin,
            WORD * argout,
            const bool with_arghead,
            const bool is_fun_arg,
            const var_map_t & var_map,
            const bool sort_vars )
```

Definition at line 243 of file flintinterface.cc.

4.43.3.6 factorize_poly()

```
WORD * flint::factorize_poly (
            PHEAD const WORD * argin,
            WORD * argout,
            const bool with_arghead,
            const bool is_fun_arg,
            const var_map_t & var_map )
```

Definition at line 396 of file flintinterface.cc.

4.43.3.7 form_sort()

```
void flint::form_sort (
            PHEAD WORD * terms )
```

Definition at line 493 of file flintinterface.cc.

4.43.3.8 from_argument_mpoly()

```
uint64_t flint::from_argument_mpoly (
            fmpz_mpoly_t poly,
            fmpz_mpoly_t denpoly,
            const WORD * args,
            const bool with_arghead,
            const var_map_t & var_map,
            const fmpz_mpoly_ctx_t ctx )
```

Definition at line 550 of file flintinterface.cc.

4.43.3.9 from_argument_poly()

```
uint64_t flint::from_argument_poly (
            fmpz_poly_t poly,
            fmpz_poly_t denpoly,
            const WORD * args,
            const bool with_arghead )
```

Definition at line 696 of file flintinterface.cc.

4.43.3.10 fmpz_get_form()

```
WORD flint::fmpz_get_form (
            fmpz_t z,
            WORD * a )
```

Definition at line 826 of file flintinterface.cc.

4.43.3.11 fmpz_set_form()

```
void flint::fmpz_set_form (
            fmpz_t z,
            UWORD * a,
            WORD na )
```

Definition at line 891 of file flintinterface.cc.

4.43.3.12 gcd_mpoly()

```
WORD * flint::gcd_mpoly (
            PHEAD const WORD * a,
            const WORD * b,
            const WORD must_fit_term,
            const var_map_t & var_map,
            const bool sort_vars )
```

Definition at line 957 of file flintinterface.cc.

4.43.3.13 gcd_poly()

```
WORD * flint::gcd_poly (
            PHEAD const WORD * a,
            const WORD * b,
            const WORD must_fit_term,
            const var_map_t & var_map )
```

Definition at line 1061 of file flintinterface.cc.

4.43.3.14 get_variables()

```
flint::var_map_t flint::get_variables (
            const vector< WORD * > & es,
            const bool with_arghead,
            const bool sort_vars )
```

Definition at line 1151 of file flintinterface.cc.

4.43.3.15 inverse_poly()

```
WORD * flint::inverse_poly (
            PHEAD const WORD * a,
            const WORD * b,
            const var_map_t & var_map )
```

Definition at line 1273 of file flintinterface.cc.

4.43.3.16 mul_mpoly()

```
WORD * flint::mul_mpoly (
            PHEAD const WORD * a,
            const WORD * b,
            const var_map_t & var_map,
            const bool sort_vars )
```

Definition at line 1349 of file flintinterface.cc.

4.43.3.17 mul_poly()

```
WORD * flint::mul_poly (
            PHEAD const WORD * a,
            const WORD * b,
            const var_map_t & var_map )
```

Definition at line 1406 of file flintinterface.cc.

4.43.3.18 ratfun_add_mpoly()

```
void flint::ratfun_add_mpoly (
            PHEAD const WORD * t1,
            const WORD * t2,
            WORD * out,
            const var_map_t & var_map,
            const bool sort_vars )
```

Definition at line 1461 of file flintinterface.cc.

4.43.3.19 ratfun_add_poly()

```
void flint::ratfun_add_poly (
            PHEAD const WORD * t1,
            const WORD * t2,
            WORD * out,
            const var_map_t & var_map )
```

Definition at line 1513 of file flintinterface.cc.

4.43.3.20 ratfun_normalize_mpoly()

```
void flint::ratfun_normalize_mpoly (
            PHEAD WORD * term,
            const var_map_t & var_map,
            const bool sort_vars )
```

Definition at line 1566 of file flintinterface.cc.

4.43.3.21 ratfun_normalize_poly()

```
void flint::ratfun_normalize_poly (
            PHEAD WORD * term,
            const var_map_t & var_map )
```

Definition at line 1650 of file flintinterface.cc.

4.43.3.22 ratfun_read_mpoly()

```
void flint::ratfun_read_mpoly (
            const WORD * a,
            fmpz_mpoly_t num,
            fmpz_mpoly_t den,
            const var_map_t & var_map,
            fmpz_mpoly_ctx_t ctx )
```

Definition at line 1733 of file flintinterface.cc.

4.43.3.23 ratfun_read_poly()

```
void flint::ratfun_read_poly (
            const WORD * a,
            fmpz_poly_t num,
            fmpz_poly_t den )
```

Definition at line 1793 of file flintinterface.cc.

4.43.3.24 to_argument_mpoly() [1/2]

```
uint64_t flint::to_argument_mpoly (
            PHEAD WORD * out,
            const bool with_arghead,
            const bool must_fit_term,
            const bool write,
            const uint64_t prev_size,
            const fmpz_mpoly_t poly,
            const var_map_t & var_map,
            const fmpz_mpoly_ctx_t ctx,
            const bool sort_vars )
```

Definition at line 2057 of file flintinterface.cc.

4.43.3.25 to_argument_mpoly() [2/2]

```
uint64_t flint::to_argument_mpoly (
            PHEAD WORD * out,
            const bool with_arghead,
            const bool must_fit_term,
            const bool write,
            const uint64_t prev_size,
            const fmpz_mpoly_t poly,
            const var_map_t & var_map,
            const fmpz_mpoly_ctx_t ctx,
            const bool sort_vars,
            const fmpz_t denscale )
```

Definition at line 1857 of file flintinterface.cc.

4.43.3.26 to_argument_poly() [1/2]

```
uint64_t flint::to_argument_poly (
            PHEAD WORD * out,
            const bool with_arghead,
            const bool must_fit_term,
            const bool write,
            const uint64_t prev_size,
            const fmpz_poly_t poly,
            const var_map_t & var_map )
```

Definition at line 2249 of file flintinterface.cc.

4.43.3.27 to_argument_poly() [2/2]

```
uint64_t flint::to_argument_poly (
            PHEAD WORD * out,
            const bool with_arghead,
            const bool must_fit_term,
            const bool write,
            const uint64_t prev_size,
            const fmpz_poly_t poly,
            const var_map_t & var_map,
            const fmpz_t denscale )
```

Definition at line 2079 of file flintinterface.cc.

4.43.3.28 simplify_fmpz()

```
void flint::util::simplify_fmpz (
            fmpz_t num,
            fmpz_t den,
            fmpz_t gcd )  [inline]
```

Definition at line 2270 of file flintinterface.cc.

4.43.3.29 simplify_fmpz_poly()

```
void flint::util::simplify_fmpz_poly (
            fmpz_poly_t num,
            fmpz_poly_t den,
            fmpz_poly_t gcd )  [inline]
```

Definition at line 2282 of file flintinterface.cc.

4.43.3.30 fix_sign_fmpz_mpoly_ratfun()

```
void flint::util::fix_sign_fmpz_mpoly_ratfun (
            fmpz_mpoly_t num,
            fmpz_mpoly_t den,
            const fmpz_mpoly_ctx_t ctx )  [inline]
```

Definition at line 2299 of file flintinterface.cc.

4.43.3.31 fix_sign_fmpz_poly_ratfun()

```
void flint::util::fix_sign_fmpz_poly_ratfun (
             fmpz_poly_t num,
             fmpz_poly_t den )  [inline]
```

Definition at line 2311 of file flintinterface.cc.

4.44 flintinterface.h

Go to the documentation of this file.

```
00001 #pragma once
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032
00033 extern "C" {
00034 #include "form3.h"
00035 }
00036
00037 #if defined(WINDOWS)
00038 // flint.h defines WORD(xx), which conflicts with the one defined in form3.h.
00039 #undef WORD
00040 #endif
00041
00042 #include <flint/flint.h>
00043 #if __FLINT_RELEASE >= 30000
00044 #include <flint/gr.h>
00045 #endif
00046 #include <flint/fmpz.h>
00047 #include <flint/fmpz_mpoly.h>
00048 #include <flint/fmpz_mpoly_factor.h>
00049 #include <flint/fmpz_poly.h>
00050 #include <flint/fmpz_poly_factor.h>
00051
00052 #if defined(WINDOWS)
00053 // Redefine WORD here to match form3.h.
00054 #undef WORD
00055 #define WORD FORM_WORD
00056 #endif
00057
00058 #include <cassert>
00059 #include <cstdint>
00060 #include <iostream>
00061 #include <map>
00062 #include <vector>
00063
00064
00065 // The bits of std that are needed:
00066 using std::cout;
00067 using std::endl;
00068 using std::fill;
00069 using std::map;
00070 using std::swap;
```

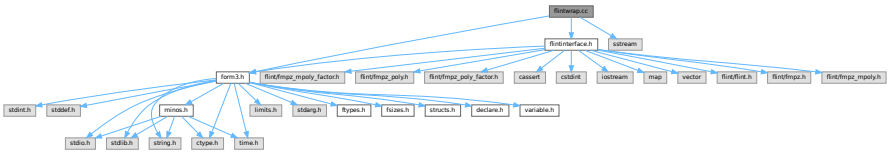
```
00071 using std::string;
00072 using std::vector;
00073
00074
00075 namespace flint {
00076
00077     typedef std::map<uint32_t,uint32_t> var_map_t;
00078
00079     // Small wrappers around the flint structs to enable RAII init and clear. "d" represents the
00080     // data, and this member will be passed to flint functions.
00081     class fmpz {
00082         public:
00083             fmpz_t d;
00084             fmpz() { fmpz_init(d); }
00085             ~fmpz() { fmpz_clear(d); }
00086             void print(const string& text) { cout << text; fmpz_print(d); cout << endl; }
00087     };
00088     class poly {
00089         public:
00090             fmpz_poly_t d;
00091             poly() { fmpz_poly_init(d); }
00092             ~poly() { fmpz_poly_clear(d); }
00093             void print(const string& text) {
00094                 cout << text; fmpz_poly_print_pretty(d, "x"); cout << endl;
00095             }
00096     };
00097     class poly_factor {
00098         public:
00099             fmpz_poly_factor_t d;
00100             poly_factor() { fmpz_poly_factor_init(d); }
00101             ~poly_factor() { fmpz_poly_factor_clear(d); }
00102     };
00103     class mpoly {
00104         private:
00105             fmpz_mpoly_ctx_struct *ctx; // We need to keep a copy of the context pointer for clearing.
00106         public:
00107             fmpz_mpoly_t d;
00108             explicit mpoly(fmpz_mpoly_ctx_struct *ctx_in) : ctx(ctx_in) { fmpz_mpoly_init(d, ctx); }
00109             ~mpoly() { fmpz_mpoly_clear(d, ctx); }
00110             void print(const string& text) {
00111                 cout << text;
00112                 fmpz_mpoly_print_pretty(d, 0, ctx);
00113                 cout << endl;
00114             }
00115     };
00116     class mpoly_factor {
00117         private:
00118             fmpz_mpoly_ctx_struct *ctx; // We need to keep a copy of the context pointer for clearing.
00119         public:
00120             fmpz_mpoly_factor_t d;
00121             explicit mpoly_factor(fmpz_mpoly_ctx_struct *ctx_in) : ctx(ctx_in) {
00122                 fmpz_mpoly_factor_init(d, ctx);
00123             }
00124             ~mpoly_factor() { fmpz_mpoly_factor_clear(d, ctx); }
00125     };
00126     class mpoly_ctx {
00127         public:
00128             fmpz_mpoly_ctx_t d;
00129             explicit mpoly_ctx(int64_t nvars) { fmpz_mpoly_ctx_init(d, nvars, ORD_LEX); }
00130             ~mpoly_ctx() { fmpz_mpoly_ctx_clear(d); }
00131     };
00132
00133     void cleanup(void);
00134     void cleanup_master(void);
00135
00136     WORD* divmod_mpoly(PHEAD const WORD *, const WORD *, const bool, const WORD, const var_map_t &,
00137         const bool);
00138     WORD* divmod_poly(PHEAD const WORD *, const WORD *, const bool, const WORD, const var_map_t &);
00139
00140     WORD* factorize_mpoly(PHEAD const WORD *, WORD *, const bool, const bool, const var_map_t &,
00141         const bool);
00142     WORD* factorize_poly(PHEAD const WORD *, WORD *, const bool, const bool, const var_map_t &);
00143
00144     void form_sort(PHEAD WORD *);
00145
00146     uint64_t from_argument_mpoly(fmpz_mpoly_t, fmpz_mpoly_t, const WORD *, const bool,
00147         const var_map_t &, const fmpz_mpoly_ctx_t);
00148     uint64_t from_argument_poly(fmpz_poly_t, fmpz_poly_t, const WORD *, const bool);
00149
00150     WORD fmpz_get_form(fmpz_t, WORD *);
00151     void fmpz_set_form(fmpz_t, UWORD *, WORD);
00152
00153     WORD* gcd_mpoly(PHEAD const WORD *, const WORD *, const WORD, const var_map_t &, const bool);
00154     WORD* gcd_poly(PHEAD const WORD *, const WORD *, const WORD, const var_map_t &);
00155
00156     var_map_t get_variables(const vector <WORD *> &, const bool, const bool);
00157
```

```
00158     WORD* inverse_poly(PHEAD const WORD *, const WORD *, const var_map_t &);
00159
00160     WORD* mul_mpoly(PHEAD const WORD *, const WORD *, const var_map_t &, const bool);
00161     WORD* mul_poly(PHEAD const WORD *, const WORD *, const var_map_t &);
00162
00163     void ratfun_add_mpoly(PHEAD const WORD *, const WORD *, WORD *, const var_map_t &, const bool);
00164     void ratfun_add_poly(PHEAD const WORD *, const WORD *, WORD *, const var_map_t &);
00165
00166     void ratfun_normalize_mpoly(PHEAD WORD *, const var_map_t &, const bool);
00167     void ratfun_normalize_poly(PHEAD WORD *, const var_map_t &);
00168
00169     void ratfun_read_mpoly(const WORD *, fmpz_mpoly_t, fmpz_mpoly_t, const var_map_t &,
00170         fmpz_mpoly_ctx_t);
00171     void ratfun_read_poly(const WORD *, fmpz_poly_t, fmpz_poly_t);
00172
00173     uint64_t to_argument_mpoly(PHEAD WORD *, const bool, const bool, const bool, const uint64_t,
00174         const fmpz_mpoly_t, const var_map_t &, const fmpz_mpoly_ctx_t, const bool);
00175     uint64_t to_argument_mpoly(PHEAD WORD *, const bool, const bool, const bool, const uint64_t,
00176         const fmpz_mpoly_t, const var_map_t &, const fmpz_mpoly_ctx_t, const bool, const fmpz_t);
00177     uint64_t to_argument_poly(PHEAD WORD *, const bool, const bool, const bool, const uint64_t,
00178         const fmpz_poly_t, const var_map_t &);
00179     uint64_t to_argument_poly(PHEAD WORD *, const bool, const bool, const bool, const uint64_t,
00180         const fmpz_poly_t, const var_map_t &, const fmpz_t);
00181
00182
00183     namespace util {
00184
00185         void simplify_fmpz(fmpz_t, fmpz_t, fmpz_t);
00186         void simplify_fmpz_poly(fmpz_poly_t, fmpz_poly_t, fmpz_poly_t);
00187
00188         void fix_sign_fmpz_mpoly_ratfun(fmpz_mpoly_t, fmpz_mpoly_t, const fmpz_mpoly_ctx_t);
00189         void fix_sign_fmpz_poly_ratfun(fmpz_poly_t, fmpz_poly_t);
00190
00191     }
00192
00193 }
```

4.45 flintwrap.cc File Reference

```
#include "form3.h"
#include <sstream>
#include "flintinterface.h"
```

Include dependency graph for flintwrap.cc:



Functions

- void flint_final_cleanup_thread (void)
- void flint_final_cleanup_master (void)
- WORD * flint_div (PHEAD WORD *a, WORD *b, const WORD must_fit_term)
- int flint_factorize_argument (PHEAD WORD *argin, WORD *argout)
- WORD * flint_factorize_dollar (PHEAD WORD *argin)
- WORD * flint_gcd (PHEAD WORD *a, WORD *b, const WORD must_fit_term)
- WORD * flint_inverse (PHEAD WORD *a, WORD *b)
- WORD * flint_mul (PHEAD WORD *a, WORD *b)
- WORD * flint_ratfun_add (PHEAD WORD *t1, WORD *t2)
- int flint_ratfun_normalize (PHEAD WORD *term)
- WORD * flint_rem (PHEAD WORD *a, WORD *b, const WORD must_fit_term)
- void flint_check_version (void)

4.45.1 Detailed Description

Contains functions to call FLINT interface from the rest of FORM.

Definition in file flintwrap.cc.

4.45.2 Function Documentation

4.45.2.1 flint_final_cleanup_thread()

```
void flint_final_cleanup_thread (
            void  )
```

Definition at line 43 of file flintwrap.cc.

4.45.2.2 flint_final_cleanup_master()

```
void flint_final_cleanup_master (
            void  )
```

Definition at line 50 of file flintwrap.cc.

4.45.2.3 flint_div()

```
WORD * flint_div (
            PHEAD WORD * a,
            WORD * b,
            const WORD must_fit_term )
```

Definition at line 57 of file flintwrap.cc.

4.45.2.4 flint_factorize_argument()

```
int flint_factorize_argument (
            PHEAD WORD * argin,
            WORD * argout )
```

Definition at line 79 of file flintwrap.cc.

4.45.2.5 flint_factorize_dollar()

```
WORD * flint_factorize_dollar (
            PHEAD WORD * argin )
```

Definition at line 100 of file flintwrap.cc.

4.45.2.6 flint_gcd()

```
WORD * flint_gcd (
            PHEAD WORD * a,
            WORD * b,
            const WORD must_fit_term )
```

Definition at line 119 of file flintwrap.cc.

4.45.2.7 flint_inverse()

```
WORD * flint_inverse (
            PHEAD WORD * a,
            WORD * b )
```

Definition at line 140 of file flintwrap.cc.

4.45.2.8 flint_mul()

```
WORD * flint_mul (
            PHEAD WORD * a,
            WORD * b )
```

Definition at line 161 of file flintwrap.cc.

4.45.2.9 flint_ratfun_add()

```
WORD * flint_ratfun_add (
            PHEAD WORD * t1,
            WORD * t2 )
```

Definition at line 182 of file flintwrap.cc.

4.45.2.10 flint_ratfun_normalize()

```
int flint_ratfun_normalize (
            PHEAD WORD * term )
```

Definition at line 223 of file flintwrap.cc.

4.45.2.11 flint_rem()

```
WORD * flint_rem (
            PHEAD WORD * a,
            WORD * b,
            const WORD must_fit_term )
```

Definition at line 296 of file flintwrap.cc.

4.45.2.12 flint_check_version()

```
void flint_check_version (
             void  )
```

Checks the FLINT library version at runtime.

This function should be called at startup. The program will terminate if a known buggy version of FLINT is detected.

Definition at line 325 of file flintwrap.cc.

4.46 flintwrap.cc

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
00026  *
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
00031
00032 extern "C" {
00033 #include "form3.h"
00034 }
00035
00036 #include <sstream>
00037 #include "flintinterface.h"
00038
00039
00040 /*
00041     #[ flint_final_cleanup_thread :
00042 */
00043 void flint_final_cleanup_thread(void) {
00044     flint::cleanup();
00045 }
00046 /*
00047     #] flint_final_cleanup_thread :
00048     #[ flint_final_cleanup_master :
00049 */
00050 void flint_final_cleanup_master(void) {
00051     flint::cleanup_master();
00052 }
00053 /*
00054     #] flint_final_cleanup_master :
00055     #[ flint_div :
00056 */
00057 WORD* flint_div(PHEAD WORD *a, WORD *b, const WORD must_fit_term) {
00058     // Extract expressions
00059     vector<WORD *> e;
00060     e.reserve(2);
00061     e.push_back(a);
00062     e.push_back(b);
00063     const bool with_arghead = false;
00064     const bool sort_vars = false;
00065     const flint::var_map_t var_map = flint::get_variables(e, with_arghead, sort_vars);
```

```
00066
00067     const bool return_rem = false;
00068     if ( var_map.size() > 1 ) {
00069         return flint::divmod_mpoly(BHEAD a, b, return_rem, must_fit_term, var_map, sort_vars);
00070     }
00071     else {
00072         return flint::divmod_poly(BHEAD a, b, return_rem, must_fit_term, var_map);
00073     }
00074 }
00075 /*
00076     #] flint_div :
00077     #[ flint_factorize_argument :
00078 */
00079 int flint_factorize_argument(PHEAD WORD *argin, WORD *argout) {
00080
00081     const bool with_arghead = true;
00082     const bool sort_vars = true;
00083     const flint::var_map_t var_map = flint::get_variables(vector<WORD*>(1,argin), with_arghead,
00084         sort_vars);
00085
00086     const bool is_fun_arg = true;
00087     if ( var_map.size() > 1 ) {
00088         flint::factorize_mpoly(BHEAD argin, argout, with_arghead, is_fun_arg, var_map, sort_vars);
00089     }
00090     else {
00091         flint::factorize_poly(BHEAD argin, argout, with_arghead, is_fun_arg, var_map);
00092     }
00093
00094     return 0;
00095 }
00096 /*
00097     #] flint_factorize_argument :
00098     #[ flint_factorize_dollar :
00099 */
00100 WORD* flint_factorize_dollar(PHEAD WORD *argin) {
00101
00102     const bool with_arghead = false;
00103     const bool sort_vars = true;
00104     const flint::var_map_t var_map = flint::get_variables(vector<WORD*>(1,argin), with_arghead,
00105         sort_vars);
00106
00107     const bool is_fun_arg = false;
00108     if ( var_map.size() > 1 ) {
00109         return flint::factorize_mpoly(BHEAD argin, NULL, with_arghead, is_fun_arg, var_map,
      sort_vars);
00110     }
00111     else {
00112         return flint::factorize_poly(BHEAD argin, NULL, with_arghead, is_fun_arg, var_map);
00113     }
00114 }
00115 /*
00116     #] flint_factorize_dollar :
00117     #[ flint_gcd :
00118 */
00119 WORD* flint_gcd(PHEAD WORD *a, WORD *b, const WORD must_fit_term) {
00120     // Extract expressions
00121     vector<WORD *> e;
00122     e.reserve(2);
00123     e.push_back(a);
00124     e.push_back(b);
00125     const bool with_arghead = false;
00126     const bool sort_vars = true;
00127     const flint::var_map_t var_map = flint::get_variables(e, with_arghead, sort_vars);
00128
00129     if ( var_map.size() > 1 ) {
00130         return flint::gcd_mpoly(BHEAD a, b, must_fit_term, var_map, sort_vars);
00131     }
00132     else {
00133         return flint::gcd_poly(BHEAD a, b, must_fit_term, var_map);
00134     }
00135 }
00136 /*
00137     #] flint_gcd :
00138     #[ flint_inverse :
00139 */
00140 WORD* flint_inverse(PHEAD WORD *a, WORD *b) {
00141     // Extract expressions
00142     vector<WORD *> e;
00143     e.reserve(2);
00144     e.push_back(a);
00145     e.push_back(b);
00146     const flint::var_map_t var_map = flint::get_variables(e, false, false);
00147
00148     if ( var_map.size() > 1 ) {
00149         MLOCK(ErrorMessageLock);
00150         MesPrint("flint_inverse: error: only univariate polynomials are supported.");
00151         MUNLOCK(ErrorMessageLock);
```

```
00152         Terminate(-1);
00153     }
00154 
00155     return flint::inverse_poly(BHEAD a, b, var_map);
00156 }
00157 /*
00158     #] flint_inverse :
00159     #[ flint_mul :
00160 */
00161 WORD* flint_mul(PHEAD WORD *a, WORD *b) {
00162     // Extract expressions
00163     vector<WORD *> e;
00164     e.reserve(2);
00165     e.push_back(a);
00166     e.push_back(b);
00167     const bool with_arghead = false;
00168     const bool sort_vars = false;
00169     const flint::var_map_t var_map = flint::get_variables(e, with_arghead, sort_vars);
00170 
00171     if ( var_map.size() > 1 ) {
00172         return flint::mul_mpoly(BHEAD a, b, var_map, sort_vars);
00173     }
00174     else {
00175         return flint::mul_poly(BHEAD a, b, var_map);
00176     }
00177 }
00178 /*
00179     #] flint_mul :
00180     #[ flint_ratfun_add :
00181 */
00182 WORD* flint_ratfun_add(PHEAD WORD *t1, WORD *t2) {
00183 
00184     if ( AR.PolyFunExp == 1 ) {
00185 /* INTERNAL_ERROR_EXCL_START */
00186         MLOCK(ErrorMessageLock);
00187         MesPrint("!>flint_ratfun_add: PolyFunExp unimplemented.");
00188         MUNLOCK(ErrorMessageLock);
00189         Terminate(-1);
00190 /* INTERNAL_ERROR_EXCL_STOP */
00191     }
00192 
00193     WORD *oldworkpointer = AT.WorkPointer;
00194 
00195     // Extract expressions: the num and den of both prf
00196     vector<WORD *> e;
00197     e.reserve(4);
00198     for (WORD *t=t1+FUNHEAD; t<t1+t1[1];) {
00199         e.push_back(t);
00200         NEXTARG(t);
00201     }
00202     for (WORD *t=t2+FUNHEAD; t<t2+t2[1];) {
00203         e.push_back(t);
00204         NEXTARG(t);
00205     }
00206     const bool with_arghead = true;
00207     const bool sort_vars = AC.OldPRFSignFlag ? true : false;
00208     const flint::var_map_t var_map = flint::get_variables(e, with_arghead, sort_vars);
00209 
00210     if ( var_map.size() > 1 ) {
00211         flint::ratfun_add_mpoly(BHEAD t1, t2, oldworkpointer, var_map, sort_vars);
00212     }
00213     else {
00214         flint::ratfun_add_poly(BHEAD t1, t2, oldworkpointer, var_map);
00215     }
00216 
00217     return oldworkpointer;
00218 }
00219 /*
00220     #] flint_ratfun_add :
00221     #[ flint_ratfun_normalize :
00222 */
00223 int flint_ratfun_normalize(PHEAD WORD *term) {
00224 
00225     // The length of the coefficient
00226     const WORD ncoeff = (term + *term)[-1];
00227     // The end of the term data, before the coefficient:
00228     const WORD *tstop = term + *term - ABS(ncoeff);
00229 
00230     // Search the term for multiple PolyFun or one dirty one.
00231     unsigned num_polyratfun = 0;
00232     for (WORD *t = term+1; t < tstop; t += t[1]) {
00233         if (*t == AR.PolyFun) {
00234             // Found one!
00235             num_polyratfun++;
00236             if ((t[2] & MUSTCLEANPRF) != 0) {
00237                 // Dirty, increment again to force normalisation for single PolyFun
00238                 num_polyratfun++;
```

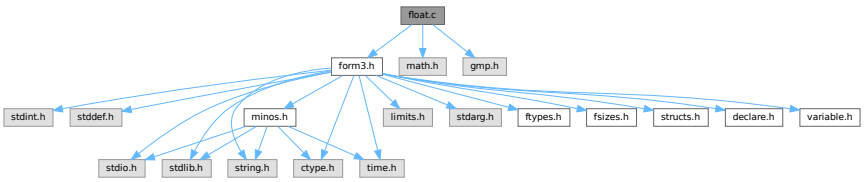
```
00239             }
00240             if (num_polyratfun > 1) {
00241                 // We're not counting all occurrences, just determinining if there
00242                 // is something to be done.
00243                 break;
00244             }
00245         }
00246     }
00247     if (num_polyratfun <= 1) {
00248         // There is nothing to do, return early
00249         return 0;
00250     }
00251
00252     // When there are polyratfun with only one argument: rename them temporarily to TMPPOLYFUN.
00253     for (WORD *t = term+1; t < tstop; t += t[1]) {
00254         if (*t == AR.PolyFun && (t[1] == FUNHEAD+t[FUNHEAD] || t[1] == FUNHEAD+2 ) ) {
00255             *t = TMPPOLYFUN;
00256         }
00257     }
00258
00259
00260     // Extract all variables in the polyfuns
00261     // Collect pointers to each relevant argument
00262     vector<WORD *> e;
00263     e.reserve(4); // There could be more than 4 args in principle, but 4 is probably common.
00264     for (WORD *t=term+1; t<tstop; t+=t[1]) {
00265         if (*t == AR.PolyFun) {
00266             for (WORD *t2 = t+FUNHEAD; t2<t+t[1];) {
00267                 e.push_back(t2);
00268                 NEXTARG(t2);
00269             }
00270         }
00271     }
00272     const bool with_arghead = true;
00273     const bool sort_vars = AC.OldPRFSignFlag ? true : false;
00274     const flint::var_map_t var_map = flint::get_variables(e, with_arghead, sort_vars);
00275
00276     if ( var_map.size() > 1 ) {
00277         flint::ratfun_normalize_mpoly(BHEAD term, var_map, sort_vars);
00278     }
00279     else {
00280         flint::ratfun_normalize_poly(BHEAD term, var_map);
00281     }
00282
00283
00284     // Undo renaming of single-argument PolyFun
00285     const WORD *new_tstop = term + *term - ABS((term + *term)[-1]);
00286     for (WORD *t=term+1; t<new_tstop; t+=t[1]) {
00287         if (*t == TMPPOLYFUN ) *t = AR.PolyFun;
00288     }
00289
00290     return 0;
00291 }
00292 /*
00293     #] flint_ratfun_normalize :
00294     #[ flint_rem :
00295 */
00296 WORD* flint_rem(PHEAD WORD *a, WORD *b, const WORD must_fit_term) {
00297     // Extract expressions
00298     vector<WORD *> e;
00299     e.reserve(2);
00300     e.push_back(a);
00301     e.push_back(b);
00302     const bool with_arghead = false;
00303     const bool sort_vars = false;
00304     const flint::var_map_t var_map = flint::get_variables(e, with_arghead, sort_vars);
00305
00306     const bool return_rem = true;
00307     if ( var_map.size() > 1 ) {
00308         return flint::divmod_mpoly(BHEAD a, b, return_rem, must_fit_term, var_map, sort_vars);
00309     }
00310     else {
00311         return flint::divmod_poly(BHEAD a, b, return_rem, must_fit_term, var_map);
00312     }
00313 }
00314 /*
00315     #] flint_rem :
00316     #[ flint_check_version :
00317 */
00318
00325 void flint_check_version(void) {
00326     bool ok = true;
00327     std::stringstream ss(flint_version);
00328     int major, minor, patch;
00329     char dot1, dot2;
00330     if ( ss » major » dot1 » minor » dot2 » patch ) {
00331         if ( dot1 != '.' || dot2 != '.' || major < 0 || minor < 0 || patch < 0 ) {
```

```
00332             ok = false;
00333         }
00334         else if ( major * 10000 + minor * 100 + patch < 30200 ) {
00335             // flint < 3.2.0: https://github.com/form-dev/form/issues/679
00336             ok = false;
00337         }
00338     }
00339     else {
00340         ok = false;
00341     }
00342     if ( !ok ) {
00343         MesPrint("Bad FLINT version detected at runtime: %s",flint_version);
00344         Terminate(-2);
00345     }
00346 }
00347
00348 /*
00349     #] flint_check_version :
00350 */
```

4.47 float.c File Reference

```
#include "form3.h"
#include <math.h>
#include <gmp.h>
```

Include dependency graph for float.c:



Macros

- #define WITHCUTOFF
- #define GMPSPREAD (GMP_LIMB_BITS/BITSINWORD)

Functions

- void Form_mpf_init (mpf_t t)
- void Form_mpf_clear (mpf_t t)
- void Form_mpf_set_prec_raw (mpf_t t, ULONG newprec)
- void FormtoZ (mpz_t z, UWORD *a, WORD na)
- void ZtoForm (UWORD *a, WORD *na, mpz_t z)
- long FloatToInteger (UWORD *out, mpf_t floatin, long *bitsused)
- void IntegerToFloat (mpf_t result, UWORD *formlong, int longsize)
- int FloatToRat (UWORD *ratout, WORD *nratout, mpf_t floatin)
- int AddFloats (PHEAD WORD *fun3, WORD *fun1, WORD *fun2)
- int MulFloats (PHEAD WORD *fun3, WORD *fun1, WORD *fun2)
- int DivFloats (PHEAD WORD *fun3, WORD *fun1, WORD *fun2)
- int AddRatToFloat (PHEAD WORD *outfun, WORD *infun, UWORD *formrat, WORD nrat)
- int MulRatToFloat (PHEAD WORD *outfun, WORD *infun, UWORD *formrat, WORD nrat)
- void SimpleDelta (mpf_t sum, int m)

- void SimpleDeltaC (mpf_t sum, int m)
- void SingleTable (mpf_t *tabl, int N, int m, int pow)
- void DoubleTable (mpf_t *tabout, mpf_t *tabin, int N, int m, int pow)
- void EndTable (mpf_t sum, mpf_t *tabin, int N, int m, int pow)
- void deltaMZV (mpf_t, WORD *, int)
- void deltaEuler (mpf_t, WORD *, int)
- void deltaEulerC (mpf_t, WORD *, int)
- void CalculateMZVhalf (mpf_t, WORD *, int)
- void CalculateMZV (mpf_t, WORD *, int)
- void CalculateEuler (mpf_t, WORD *, int)
- int ExpandMZV (WORD *term, WORD level)
- int ExpandEuler (WORD *term, WORD level)
- int PackFloat (WORD *, mpf_t)
- int UnpackFloat (mpf_t, WORD *)
- void RatToFloat (mpf_t result, UWORD *formrat, int ratsize)
- void Form_mpf_empty (mpf_t t)
- int TestFloat (WORD *fun)
- WORD FloatFunToRat (PHEAD UWORD *ratout, WORD *in)
- void ZeroTable (mpf_t *tab, int N)
- SBYTE * ReadFloat (SBYTE *s)
- UBYTE * CheckFloat (UBYTE *ss, int *spec)
- int SetFloatPrecision (WORD prec)
- int PrintFloat (WORD *fun, int numdigits)
- void SetupMZVTables (void)
- void SetupMPFTables (void)
- void ClearMZVTables (void)
- int CoToFloat (UBYTE *s)
- int CoToRat (UBYTE *s)
- int CoStrictRounding (UBYTE *s)
- int CoChop (UBYTE *s)
- int ToFloat (PHEAD WORD *term, WORD level)
- int ToRat (PHEAD WORD *term, WORD level)
- int StrictRounding (PHEAD WORD *term, WORD level, WORD prec, WORD base)
- int Chop (PHEAD WORD *term, WORD level)
- int AddWithFloat (PHEAD WORD **ps1, WORD **ps2)
- int MergeWithFloat (PHEAD WORD **interm1, WORD **interm2)
- int EvaluateEuler (PHEAD WORD *term, WORD level, WORD par)
- int CoEvaluate (UBYTE *s)

4.47.1 Detailed Description

This file contains numerical routines that deal with floating point numbers. We use the GMP for arbitrary precision floating point arithmetic. The functions of the type sin, cos, ln, sqrt etc are handled by the mpfr library. The reason that for MZV's and the general notation we use the GMP (mpf_) is because the contents of the structs have been frozen and can be used for storage in a Form function float_. With mpfr_ this is not safely possible. All mpfr_ related code is in the file evaluate.c.

Definition in file float.c.

4.47.2 Macro Definition Documentation

4.47.2.1 WITHCUTOFF

```
#define WITHCUTOFF
```

Definition at line 45 of file float.c.

4.47.2.2 GMPSPREAD

```
#define GMPSPREAD (GMP_LIMB_BITS/BITSINWORD)
```

Definition at line 47 of file float.c.

4.47.3 Function Documentation

4.47.3.1 Form_mpf_init()

```
void Form_mpf_init (
            mpf_t t )
```

Definition at line 177 of file float.c.

4.47.3.2 Form_mpf_clear()

```
void Form_mpf_clear (
            mpf_t t )
```

Definition at line 201 of file float.c.

4.47.3.3 Form_mpf_set_prec_raw()

```
void Form_mpf_set_prec_raw (
            mpf_t t,
            ULONG newprec )
```

Definition at line 227 of file float.c.

4.47.3.4 FormtoZ()

```
void FormtoZ (
            mpz_t z,
            UWORD * a,
            WORD na )
```

Definition at line 515 of file float.c.

4.47.3.5 ZtoForm()

```
void ZtoForm (
            UWORD * a,
            WORD * na,
            mpz_t z )
```

Definition at line 559 of file float.c.

4.47.3.6 FloatToInteger()

```
long FloatToInteger (
            UWORD * out,
            mpf_t floatin,
            long * bitsused )
```

Definition at line 585 of file float.c.

4.47.3.7 IntegerToFloat()

```
void IntegerToFloat (
            mpf_t result,
            UWORD * formlong,
            int longsize )
```

Definition at line 608 of file float.c.

4.47.3.8 FloatToRat()

```
int FloatToRat (
            UWORD * ratout,
            WORD * nratout,
            mpf_t floatin )
```

Definition at line 672 of file float.c.

4.47.3.9 AddFloats()

```
int AddFloats (
            PHEAD WORD * fun3,
            WORD * fun1,
            WORD * fun2 )
```

Definition at line 995 of file float.c.

4.47.3.10 MulFloats()

```
int MulFloats (
            PHEAD WORD * fun3,
            WORD * fun1,
            WORD * fun2 )
```

Definition at line 1011 of file float.c.

4.47.3.11 DivFloats()

```
int DivFloats (
              PHEAD WORD * fun3,
              WORD * fun1,
              WORD * fun2 )
```

Definition at line 1027 of file float.c.

4.47.3.12 AddRatToFloat()

```
int AddRatToFloat (
              PHEAD WORD * outfun,
              WORD * infun,
              UWORD * formrat,
              WORD nrat )
```

Definition at line 1045 of file float.c.

4.47.3.13 MulRatToFloat()

```
int MulRatToFloat (
              PHEAD WORD * outfun,
              WORD * infun,
              UWORD * formrat,
              WORD nrat )
```

Definition at line 1064 of file float.c.

4.47.3.14 SimpleDelta()

```
void SimpleDelta (
              mpf_t sum,
              int m )
```

Definition at line 1885 of file float.c.

4.47.3.15 SimpleDeltaC()

```
void SimpleDeltaC (
              mpf_t sum,
              int m )
```

Definition at line 1944 of file float.c.

4.47.3.16 SingleTable()

```
void SingleTable (
            mpf_t * tabl,
            int N,
            int m,
            int pow )
```

Definition at line 2003 of file float.c.

4.47.3.17 DoubleTable()

```
void DoubleTable (
            mpf_t * tabout,
            mpf_t * tabin,
            int N,
            int m,
            int pow )
```

Definition at line 2059 of file float.c.

4.47.3.18 EndTable()

```
void EndTable (
            mpf_t sum,
            mpf_t * tabin,
            int N,
            int m,
            int pow )
```

Definition at line 2114 of file float.c.

4.47.3.19 deltaMZV()

```
void deltaMZV (
            mpf_t result,
            WORD * indexes,
            int depth )
```

Definition at line 2163 of file float.c.

4.47.3.20 deltaEuler()

```
void deltaEuler (
            mpf_t result,
            WORD * indexes,
            int depth )
```

Definition at line 2230 of file float.c.

4.47.3.21 deltaEulerC()

```
void deltaEulerC (
            mpf_t result,
            WORD * indexes,
            int depth )
```

Definition at line 2286 of file float.c.

4.47.3.22 CalculateMZVhalf()

```
void CalculateMZVhalf (
            mpf_t result,
            WORD * indexes,
            int depth )
```

Definition at line 2362 of file float.c.

4.47.3.23 CalculateMZV()

```
void CalculateMZV (
            mpf_t result,
            WORD * indexes,
            int depth )
```

Definition at line 2387 of file float.c.

4.47.3.24 CalculateEuler()

```
void CalculateEuler (
            mpf_t result,
            WORD * Zindexes,
            int depth )
```

Definition at line 2458 of file float.c.

4.47.3.25 ExpandMZV()

```
int ExpandMZV (
            WORD * term,
            WORD level )
```

Definition at line 2540 of file float.c.

4.47.3.26 ExpandEuler()

```
int ExpandEuler (
            WORD * term,
            WORD level )
```

Definition at line 2552 of file float.c.

4.47.3.27 PackFloat()

```
int PackFloat (
            WORD * fun,
            mpf_t infloat )
```

Definition at line 271 of file float.c.

4.47.3.28 UnpackFloat()

```
int UnpackFloat (
            mpf_t outfloat,
            WORD * fun )
```

Definition at line 366 of file float.c.

4.47.3.29 RatToFloat()

```
void RatToFloat (
            mpf_t result,
            UWORD * formrat,
            int ratsize )
```

Definition at line 624 of file float.c.

4.47.3.30 Form_mpf_empty()

```
void Form_mpf_empty (
            mpf_t t )
```

Definition at line 214 of file float.c.

4.47.3.31 TestFloat()

```
int TestFloat (
            WORD * fun )
```

Definition at line 456 of file float.c.

4.47.3.32 FloatFunToRat()

```
WORD FloatFunToRat (
            PHEAD UWORD * ratout,
            WORD * in )
```

Definition at line 646 of file float.c.

4.47.3.33 ZeroTable()

```
void ZeroTable (
            mpf_t * tab,
            int N )
```

Definition at line 813 of file float.c.

4.47.3.34 ReadFloat()

```
SBYTE * ReadFloat (
            SBYTE * s )
```

Definition at line 831 of file float.c.

4.47.3.35 CheckFloat()

```
UBYTE * CheckFloat (
            UBYTE * ss,
            int * spec )
```

Definition at line 857 of file float.c.

4.47.3.36 SetFloatPrecision()

```
int SetFloatPrecision (
            WORD prec )
```

Definition at line 910 of file float.c.

4.47.3.37 PrintFloat()

```
int PrintFloat (
            WORD * fun,
            int numdigits )
```

Definition at line 939 of file float.c.

4.47.3.38 SetupMZVTables()

```
void SetupMZVTables (
            void  )
```

Definition at line 1081 of file float.c.

4.47.3.39 SetupMPFTables()

```
void SetupMPFTables (
            void  )
```

Definition at line 1145 of file float.c.

4.47.3.40 ClearMZVTables()

```
void ClearMZVTables (
            void  )
```

Definition at line 1179 of file float.c.

4.47.3.41 CoToFloat()

```
int CoToFloat (
            UBYTE * s )
```

Definition at line 1256 of file float.c.

4.47.3.42 CoToRat()

```
int CoToRat (
            UBYTE * s )
```

Definition at line 1278 of file float.c.

4.47.3.43 CoStrictRounding()

```
int CoStrictRounding (
            UBYTE * s )
```

Definition at line 1305 of file float.c.

4.47.3.44 CoChop()

```
int CoChop (
            UBYTE * s )
```

Definition at line 1352 of file float.c.

4.47.3.45 ToFloat()

```
int ToFloat (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 1448 of file float.c.

4.47.3.46 ToRat()

```
int ToRat (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 1480 of file float.c.

4.47.3.47 StrictRounding()

```
int StrictRounding (
            PHEAD WORD * term,
            WORD level,
            WORD prec,
            WORD base )
```

Definition at line 1520 of file float.c.

4.47.3.48 Chop()

```
int Chop (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 1582 of file float.c.

4.47.3.49 AddWithFloat()

```
int AddWithFloat (
            PHEAD WORD ** ps1,
            WORD ** ps2 )
```

Definition at line 1651 of file float.c.

4.47.3.50 MergeWithFloat()

```
int MergeWithFloat (
            PHEAD WORD ** interm1,
            WORD ** interm2 )
```

Definition at line 1772 of file float.c.

4.47.3.51 EvaluateEuler()

```
int EvaluateEuler (
            PHEAD WORD * term,
            WORD level,
            WORD par )
```

Definition at line 2580 of file float.c.

4.47.3.52 CoEvaluate()

```
int CoEvaluate (
            UBYTE * s )
```

Definition at line 2700 of file float.c.

4.48 float.c

Go to the documentation of this file.

```
00001
00011 /* #[ License : */
00012 /*
00013  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00014  *   When using this file you are requested to refer to the publication
00015  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00016  *   This is considered a matter of courtesy as the development was paid
00017  *   for by FOM the Dutch physics granting agency and we would like to
00018  *   be able to track its scientific use to convince FOM of its value
00019  *   for the community.
00020  *
00021  *   This file is part of FORM.
00022  *
00023  *   FORM is free software: you can redistribute it and/or modify it under the
00024  *   terms of the GNU General Public License as published by the Free Software
00025  *   Foundation, either version 3 of the License, or (at your option) any later
00026  *   version.
00027  *
00028  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00029  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00030  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00031  *   details.
00032  *
00033  *   You should have received a copy of the GNU General Public License along
00034  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00035  */
00036 /* #] License : */
00037 /*
00038     #[ Includes : float.c
00039 */
00040
00041 #include "form3.h"
00042 #include <math.h>
00043 #include <gmp.h>
00044
00045 #define WITHCUTOFF
00046
00047 #define GMPSPREAD (GMP_LIMB_BITS/BITSINWORD)
00048
00049
00050 void Form_mpf_init(mpf_t t);
00051 void Form_mpf_clear(mpf_t t);
00052 void Form_mpf_set_prec_raw(mpf_t t,ULONG newprec);
00053 void FormtoZ(mpz_t z,UWORD *a,WORD na);
00054 void ZtoForm(UWORD *a,WORD *na,mpz_t z);
00055 long FloatToInteger(UWORD *out, mpf_t floatin, long *bitsused);
00056 void IntegerToFloat(mpf_t result, UWORD *formlong, int longsize);
00057 int FloatToRat(UWORD *ratout, WORD *nratout, mpf_t floatin);
00058 int AddFloats(PHEAD WORD *fun3, WORD *fun1, WORD *fun2);
00059 int MulFloats(PHEAD WORD *fun3, WORD *fun1, WORD *fun2);
00060 int DivFloats(PHEAD WORD *fun3, WORD *fun1, WORD *fun2);
00061 int AddRatToFloat(PHEAD WORD *outfun, WORD *infun, UWORD *formrat, WORD nrat);
00062 int MulRatToFloat(PHEAD WORD *outfun, WORD *infun, UWORD *formrat, WORD nrat);
00063 void SimpleDelta(mpf_t sum, int m);
00064 void SimpleDeltaC(mpf_t sum, int m);
00065 void SingleTable(mpf_t *tabl, int N, int m, int pow);
00066 void DoubleTable(mpf_t *tabout, mpf_t *tabin, int N, int m, int pow);
00067 void EndTable(mpf_t sum, mpf_t *tabin, int N, int m, int pow);
00068 void deltaMZV(mpf_t, WORD *, int);
00069 void deltaEuler(mpf_t, WORD *, int);
00070 void deltaEulerC(mpf_t, WORD *, int);
00071 void CalculateMZVhalf(mpf_t, WORD *, int);
00072 void CalculateMZV(mpf_t, WORD *, int);
00073 void CalculateEuler(mpf_t, WORD *, int);
00074 int ExpandMZV(WORD *term, WORD level);
00075 int ExpandEuler(WORD *term, WORD level);
00076 int PackFloat(WORD *,mpf_t);
00077 int UnpackFloat(mpf_t, WORD *);
```

```
00078 void RatToFloat(mpf_t result, UWORD *formrat, int ratsize);
00079
00080 /*
00081     #] Includes :
00082     #[ Some GMP float info :
00083
00084     From gmp.h:
00085
00086     typedef struct
00087     {
00088       int _mp_prec;     Max precision, in number of `mp_limb_t's.
00089                         Set by mpf_init and modified by
00090                         mpf_set_prec.  The area pointed to by the
00091                         _mp_d field contains `prec' + 1 limbs.
00092       int _mp_size;     abs(_mp_size) is the number of limbs the
00093                         last field points to.  If _mp_size is
00094                         negative this is a negative number.
00095       mp_exp_t _mp_exp; Exponent, in the base of `mp_limb_t'.
00096       mp_limb_t *_mp_d; Pointer to the limbs.
00097     } __mpf_struct;
00098
00099     typedef __mpf_struct mpf_t[1];
00100
00101     Currently:
00102         sizeof(int) = 4
00103         sizeof(mpf_t) = 24
00104         sizeof(mp_limb_t) = 8,
00105         sizeof(mp_exp_t) = 8,
00106         sizeof a pointer is 8.
00107     If any of this changes in the future we would have to change the
00108     PackFloat and UnpackFloat routines.
00109
00110     Our float_ function is packed with the 4 arguments:
00111         -SNUMBER _mp_prec
00112         -SNUMBER _mp_size
00113         exponent which can be -SNUMBER or a regular term
00114         the limbs as n/1 in regular term format, or just an -SNUMBER.
00115     During normalization the sign is taken away from the second argument
00116     and put as the sign of the complete term. This is easier for the
00117     WriteInnerTerm routine in sch.c
00118
00119     Wildcarding of functions excludes matches with float_
00120     Float_ always ends up inside the bracket, just like poly(rat)fun.
00121
00122     From mpfr.h
00123
00124     The formats here are different. This has, among others, to do with
00125     the rounding. The result is that we need different aux variables
00126     when working with mpfr and we need a different conversion routine
00127     to float. It also means that we need to treat the mzv_ etc functions
00128     completely separately. For now we try to develop that in the file
00129     Evaluate.c. At a later stage we may merge the two files.
00130     Originally we had the sqrt in float.c but it seems better to put
00131     it in Evaluate.c
00132
00133     #] Some GMP float info :
00134     #[ Low Level :
00135
00136         In the low level routines we interact directly with the content
00137         of the GMP structs. This can be done safely, because their
00138         layout is documented. We pay particular attention to the init
00139         and clear routines, because they involve malloc/free calls.
00140
00141         #[ Explanations :
00142
00143     We need a number of facilities inside Form to deal with floating point
00144     numbers, taking into account that they are only meant for sporadic use.
00145     1: We need a special function float_ who's arguments contain a limb
00146        representation of the mpf_t of the gmp.
00147     2: Some functions may return a floating point number. This is then in
00148        the form of an occurrence of the function float_.
00149     3: We need a print routine to print this float_.
00150     4: We need routines for conversions:
00151         a: (long) int to float.
00152         b: rat to float.
00153         c: float to rat (if possible)
00154         d: float to integer (with rounding toward zero).
00155     5: We need calculational operations for add, sub, mul, div, exp.
00156     6: We need service routines to pack and unpack the function float_.
00157     7: The coefficient should be pulled inside float_.
00158     8: Float_ cannot occur inside PolyRatFun.
00159     9: We need to be able to treat float_ as the coefficient during sorting.
00160     10: For now, we need the functions mzvf_ and eulerf_ for the evaluation
00161         of mzv's and euler sums.
00162     11: We need a setting for the default precision.
00163     12: We need a special flag for the dirty field of arguments to avoid
00164         the normalization of the argument with the limbs.
```

```
00165         Alternatively, the float_ should be not a regular function, but
00166         get a status < FUNCTION. Just like SNUMBER, LNUMBER etc.
00167
00168     Note that we can keep this thread-safe. The only routine that is not
00169     thread-safe is the print routine, but that is in accordance with all
00170     other print routines in Form. When called from MesPrint it needs to
00171     be protected by a lock, as is done standard inside Form.
00172
00173         #] Explanations :
00174         #[ Form_mpf_init :
00175 */
00176 /* UNFINISHED_FEATURE_EXCL_START */
00177 void Form_mpf_init(mpf_t t)
00178 {
00179     mp_limb_t *d;
00180     int i, prec;
00181     if ( t->_mp_d ) { M_free(t->_mp_d,"Form_mpf_init"); t->_mp_d = 0; }
00182     prec = (AC.DefaultPrecision + 8*sizeof(mp_limb_t)-1)/(8*sizeof(mp_limb_t)) + 1;
00183     t->_mp_prec = prec;
00184     t->_mp_size = 0;
00185     t->_mp_exp = 0;
00186     d = (mp_limb_t *)Malloc1(prec*sizeof(mp_limb_t),"Form_mpf_init");
00187     if ( d == 0 ) {
00188         MesPrint("Fatal error in Malloc1 call in Form_mpf_init. Trying to allocate %ld bytes.",
00189             prec*sizeof(mp_limb_t));
00190         Terminate(-1);
00191     }
00192     t->_mp_d = d;
00193     for ( i = 0; i < prec; i++ ) d[i] = 0;
00194 }
00195 /* UNFINISHED_FEATURE_EXCL_STOP */
00196 /*
00197         #] Form_mpf_init :
00198         #[ Form_mpf_clear :
00199 */
00200 /* UNFINISHED_FEATURE_EXCL_START */
00201 void Form_mpf_clear(mpf_t t)
00202 {
00203     if ( t->_mp_d ) { M_free(t->_mp_d,"Form_mpf_init"); t->_mp_d = 0; }
00204     t->_mp_prec = 0;
00205     t->_mp_size = 0;
00206     t->_mp_exp = 0;
00207 }
00208 /* UNFINISHED_FEATURE_EXCL_STOP */
00209 /*
00210         #] Form_mpf_clear :
00211         #[ Form_mpf_empty :
00212 */
00213
00214 void Form_mpf_empty(mpf_t t)
00215 {
00216     int i;
00217     for ( i = 0; i < t->_mp_prec; i++ ) t->_mp_d[i] = 0;
00218     t->_mp_size = 0;
00219     t->_mp_exp = 0;
00220 }
00221
00222 /*
00223         #] Form_mpf_empty :
00224         #[ Form_mpf_set_prec_raw :
00225 */
00226 /* UNFINISHED_FEATURE_EXCL_START */
00227 void Form_mpf_set_prec_raw(mpf_t t,ULONG newprec)
00228 {
00229     ULONG newpr = (newprec + 8*sizeof(mp_limb_t)-1)/(8*sizeof(mp_limb_t)) + 1;
00230     if ( newpr <= (ULONG)(t->_mp_prec) ) {
00231         int i, used = ABS(t->_mp_size) ;
00232         if ( newpr > (ULONG)used ) {
00233             for ( i = used; i < (int)newpr; i++ ) t->_mp_d[i] = 0;
00234         }
00235         if ( t->_mp_size < 0 ) newpr = -newpr;
00236         t->_mp_size = newpr;
00237     }
00238     else {
00239 /*
00240         We can choose here between "forget about it" or making a new malloc.
00241         If the program is well designed, we should never get here though.
00242         For now we choose the "forget it" option.
00243 */
00244         MesPrint("Trying too big a precision %ld in Form_mpf_set_prec_raw.",newprec);
00245         MesPrint("Old maximal precision is %ld.",
00246             (size_t)(t->_mp_prec-1)*sizeof(mp_limb_t)*8);
00247         Terminate(-1);
00248     }
00249 }
00250 /* UNFINISHED_FEATURE_EXCL_STOP */
00251 /*
```

```
00252         #] Form_mpf_set_prec_raw :
00253         #[ PackFloat :
00254
00255     Puts an object of type mpf_t inside a function float_
00256     float_(prec, nlimbs, exp, limbs);
00257     Complication: the limbs and the exponent are long's. That means
00258     two times the size of a Form (U)WORD.
00259     To save space we could split the limbs in two because Form stores
00260     coefficients as rationals with equal space for numerator and denominator.
00261     Hence for each limb we could put the most significant UWORD in the
00262     numerator and the least significant limb in the denominator.
00263     This gives a problem with the compilation and subsequent potential
00264     normalization of the 'fraction'. Hence for now we take the wasteful
00265     approach. At a later stage we can try to veto normalization of FLOATFUN.
00266     Caution: gmp/fmp is little endian and Form is big endian.
00267
00268     Zero is represented by zero limbs (and zero exponent).
00269 */
00270
00271 int PackFloat(WORD *fun,mpf_t infloat)
00272 {
00273     WORD *t, nlimbs, num, nnum;
00274     mp_limb_t *d = infloat->_mp_d; /* Pointer to the limbs.  */
00275     int i;
00276     long e = infloat->_mp_exp;
00277     t = fun;
00278     *t++ = FLOATFUN;
00279     t++;
00280     FILLFUN(t);
00281     *t++ = -SNUMBER;
00282     *t++ = infloat->_mp_prec;
00283     *t++ = -SNUMBER;
00284     *t++ = infloat->_mp_size;
00285 /*
00286     Now the exponent which is a signed long
00287 */
00288     if ( e < 0 ) {
00289         e = -e;
00290         if ( ( e » (BITSINWORD-1) ) == 0 ) {
00291             *t++ = -SNUMBER; *t++ = -e;
00292         }
00293         else {
00294             *t++ = ARGHEAD+6; *t++ = 0; FILLARG(t);
00295             *t++ = 6;
00296             *t++ = (UWORD)e;
00297             *t++ = (UWORD)(e » BITSINWORD);
00298             *t++ = 1; *t++ = 0; *t++ = -5;
00299         }
00300     }
00301     else {
00302         if ( ( e » (BITSINWORD-1) ) == 0 ) {
00303             *t++ = -SNUMBER; *t++ = e;
00304         }
00305         else {
00306             *t++ = ARGHEAD+6; *t++ = 0; FILLARG(t);
00307             *t++ = 6;
00308             *t++ = (UWORD)e;
00309             *t++ = (UWORD)(e » BITSINWORD);
00310             *t++ = 1; *t++ = 0; *t++ = 5;
00311         }
00312     }
00313 /*
00314     and now the limbs. Note that the number of active limbs could be less
00315     than prec+1 in which case we copy the smaller number.
00316 */
00317     nlimbs = infloat->_mp_size < 0 ? -infloat->_mp_size: infloat->_mp_size;
00318     if ( nlimbs == 0 ) {
00319         *t++ = -SNUMBER;
00320         *t++ = 0;
00321     }
00322     else if ( nlimbs == 1 && (ULONG)(*d) < ((ULONG)1)«(BITSINWORD-1) ) {
00323         *t++ = -SNUMBER;
00324         *t++ = (UWORD)(*d);
00325     }
00326     else {
00327       if ( d[nlimbs-1] < ((ULONG)1)«BITSINWORD ) {
00328         num = 2*nlimbs-1; /* number of UWORDS needed */
00329       }
00330       else {
00331         num = 2*nlimbs;   /* number of UWORDS needed */
00332       }
00333       nnum = num;
00334       *t++ = 2*num+2+ARGHEAD;
00335       *t++ = 0;
00336       FILLARG(t);
00337       *t++ = 2*num+2;
00338       while ( num > 1 ) {
```

```
00339         *t++ = (UWORD)(*d); *t++ = (UWORD)(((ULONG)(*d))»BITSINWORD); d++;
00340         num -= 2;
00341       }
00342       if ( num == 1 ) { *t++ = (UWORD)(*d); }
00343       *t++ = 1;
00344       for ( i = 1; i < nnum; i++ ) *t++ = 0;
00345       *t++ = 2*nnum+1; /* the sign is hidden in infloat->_mp_size */
00346     }
00347     fun[1] = t-fun;
00348     return(fun[1]);
00349 }
00350
00351 /*
00352         #] PackFloat :
00353         #[ UnpackFloat :
00354
00355     Takes the arguments of a function float_ and puts it inside an mpf_t.
00356     For notation see commentary for PutInFloat.
00357
00358     We should make a new allocation for the limbs if the precision exceeds
00359     the present precision of outfloat, or when outfloat has not been
00360     initialized yet.
00361
00362     The return value is -1 if the contents of the FLOATFUN cannot be converted.
00363     Otherwise the return value is the number of limbs.
00364 */
00365
00366 int UnpackFloat(mpf_t outfloat,WORD *fun)
00367 {
00368     UWORD *t;
00369     WORD *f;
00370     int num, i;
00371     mp_limb_t *d;
00372 /*
00373     Very first step: check whether we can use float_:
00374 */
00375     GETIDENTITY
00376     if ( AT.aux_ == 0 ) {
00377         MLOCK(ErrorMessageLock);
00378         MesPrint("Illegal attempt at using a float_ function without proper startup.");
00379         MesPrint("Please use %#StartFloat <options> first.");
00380         MUNLOCK(ErrorMessageLock);
00381         Terminate(-1);
00382     }
00383 /*
00384     Check first the number and types of the arguments
00385     There should be 4. -SNUMBER,-SNUMBER,-SNUMBER or a long number.
00386     + the limbs, either -SNUMBER or Long number in the form of a Form rational.
00387 */
00388     if ( TestFloat(fun) == 0 ) goto Incorrect;
00389     f = fun + FUNHEAD + 2;
00390     if ( ABS(f[1]) > f[-1]+1 ) goto Incorrect;
00391     if ( f[1] > outfloat->_mp_prec+1
00392       || outfloat->_mp_d == 0 ) {
00393         /*
00394             We need to make a new allocation.
00395             Unfortunately we cannot use Malloc1 because that is not
00396             recognised by gmp and hence we cannot clear with mpf_clear.
00397             Maybe we can do something about it by making our own
00398             mpf_init and mpf_clear?
00399         */
00400         if ( outfloat->_mp_d != 0 ) free(outfloat->_mp_d);
00401         outfloat->_mp_d = (mp_limb_t *)malloc((f[1]+1)*sizeof(mp_limb_t));
00402     }
00403     num = f[1];
00404     outfloat->_mp_size = num;
00405     if ( num < 0 ) { num = -num; }
00406     f += 2;
00407     if ( *f == -SNUMBER ) {
00408         outfloat->_mp_exp = (mp_exp_t)(f[1]);
00409         f += 2;
00410     }
00411     else {
00412         f += ARGHEAD+6;
00413         if ( f[-1] == -5 ) {
00414             outfloat->_mp_exp =
00415                 -(mp_exp_t)((((ULONG)(f[-4]))«BITSINWORD)+(UWORD)f[-5]);
00416         }
00417         else if ( f[-1] == 5 ) {
00418             outfloat->_mp_exp =
00419                  (mp_exp_t)((((ULONG)(f[-4]))«BITSINWORD)+(UWORD)f[-5]);
00420         }
00421     }
00422 /*
00423     And now the limbs if needed
00424 */
00425     d = outfloat->_mp_d;
```

```
00426     if ( outfloat->_mp_size == 0 ) {
00427         for ( i = 0; i < outfloat->_mp_prec; i++ ) *d++ = 0;
00428         return(0);
00429     }
00430     else if ( *f == -SNUMBER ) {
00431         *d++ = (mp_limb_t)f[1];
00432         for ( i = 0; i < outfloat->_mp_prec; i++ ) *d++ = 0;
00433         return(0);
00434     }
00435     num = (*f-ARGHEAD-2)/2; /* 2*number of limbs in the argument */
00436     t = (UWORD *)(f+ARGHEAD+1);
00437     while ( num > 1 ) {
00438         *d++ = (mp_limb_t)((((ULONG)(t[1]))«BITSINWORD)+t[0]);
00439         t += 2; num -= 2;
00440     }
00441     if ( num == 1 ) *d++ = (mp_limb_t)((UWORD)(*t));
00442     for ( i = d-outfloat->_mp_d; i <= outfloat->_mp_prec; i++ ) *d++ = 0;
00443     return(0);
00444 Incorrect:
00445     return(-1);
00446 }
00447
00448 /*
00449         #] UnpackFloat :
00450         #[ TestFloat :
00451
00452         Tells whether the function (with its arguments) is a legal float_.
00453         We assume, as in the whole float library that one limb is 2 WORDs.
00454 */
00455
00456 int TestFloat(WORD *fun)
00457 {
00458     WORD *fstop, *f, num, nnum, i;
00459     fstop = fun+fun[1];
00460     f = fun + FUNHEAD;
00461     num = 0;
00462 /*
00463     Count arguments
00464 */
00465     while ( f < fstop ) { num++; NEXTARG(f); }
00466     if ( num != 4 ) return(0);
00467     f = fun + FUNHEAD;
00468     if ( *f != -SNUMBER ) return(0);
00469     if ( f[1] < 0 ) return(0);
00470     f += 2;
00471     if ( *f != -SNUMBER ) return(0);
00472     num = ABS(f[1]); /* number of limbs */
00473     f += 2;
00474     if ( *f == -SNUMBER ) { f += 2; }
00475     else {
00476         if ( *f != ARGHEAD+6 ) return(0);
00477         if ( ABS(f[ARGHEAD+5]) != 5 ) return(0);
00478         if ( f[ARGHEAD+3] != 1 ) return(0);
00479         if ( f[ARGHEAD+4] != 0 ) return(0);
00480         f += *f;
00481     }
00482     if ( num == 0 ) return(1);
00483     if ( *f == -SNUMBER ) { if ( num != 1 ) return(0); }
00484     else {
00485         nnum = (ABS(f[*f-1])-1)/2;
00486         if ( ( *f != 4*num+2+ARGHEAD ) && ( *f != 4*num+ARGHEAD ) ) return(0);
00487         if ( ( nnum != 2*num ) && ( nnum != 2*num-1 ) ) return(0);
00488         f += ARGHEAD+nnum+1;
00489         if ( f[0] != 1 ) return(0);
00490         for ( i = 1; i < nnum; i++ ) {
00491             if ( f[i] != 0 ) return(0);
00492         }
00493     }
00494     return(1);
00495 }
00496
00497 /*
00498         #] TestFloat :
00499         #[ FormtoZ :
00500
00501         Converts a Form long integer to a GMP long integer.
00502
00503         typedef struct
00504         {
00505           int _mp_alloc;   Number of *limbs* allocated and pointed
00506                            to by the _mp_d field.
00507           int _mp_size;    abs(_mp_size) is the number of limbs the
00508                            last field points to.  If _mp_size is
00509                            negative this is a negative number.
00510           mp_limb_t *_mp_d;Pointer to the limbs.
00511         } __mpz_struct;
00512         typedef __mpz_struct mpz_t[1];
```

```
00513 */
00514
00515 void FormtoZ(mpz_t z,UWORD *a,WORD na)
00516 {
00517     int nlimbs, sgn = 1;
00518     mp_limb_t *d;
00519     UWORD *b = a;
00520     WORD nb = na;
00521
00522     if ( nb == 0 ) {
00523         z->_mp_size = 0;
00524         z->_mp_d[0] = 0;
00525         return;
00526     }
00527     if ( nb < 0 ) { sgn = -1; nb = -nb; }
00528     nlimbs = (nb+1)/2;
00529     if ( mpz_cmp_si(z,0L) <= 1 ) {
00530         mpz_set_ui(z,10000000);
00531     }
00532     if ( z->_mp_alloc <= nlimbs ) {
00533         mpz_t zz;
00534         mpz_init(zz);
00535         while ( z->_mp_alloc <= nlimbs ) {
00536             mpz_mul(zz,z,z);
00537             mpz_set(z,zz);
00538         }
00539         mpz_clear(zz);
00540     }
00541     z->_mp_size = sgn*nlimbs;
00542     d = z->_mp_d;
00543     while ( nb > 1 ) {
00544         *d++ = (((mp_limb_t)(b[1]))«BITSINWORD)+(mp_limb_t)(b[0]);
00545         b += 2; nb -= 2;
00546     }
00547     if ( nb == 1 ) { *d = (mp_limb_t)(*b); }
00548 }
00549
00550 /*
00551         #] FormtoZ :
00552         #[ ZtoForm :
00553
00554         Converts a GMP long integer to a Form long integer.
00555         Mainly an exercise of going from little endian ULONGs.
00556         to big endian UWORDs
00557 */
00558
00559 void ZtoForm(UWORD *a,WORD *na,mpz_t z)
00560 {
00561     mp_limb_t *d = z->_mp_d;
00562     int nlimbs = ABS(z->_mp_size), i;
00563     UWORD j;
00564     if ( nlimbs == 0 ) { *na = 0; return; }
00565     *na = 0;
00566     for ( i = 0; i < nlimbs-1; i++ ) {
00567         *a++ = (UWORD)(*d);
00568         *a++ = (UWORD)((*d++)»BITSINWORD);
00569         *na += 2;
00570     }
00571     *na += 1;
00572     *a++ = (UWORD)(*d);
00573     j = (UWORD)((*d)»BITSINWORD);
00574     if ( j != 0 ) { *a = j; *na += 1; }
00575     if ( z->_mp_size < 0 ) *na = -*na;
00576 }
00577
00578 /*
00579         #] ZtoForm :
00580         #[ FloatToInteger :
00581
00582         Converts a GMP float to a long Form integer.
00583 */
00584
00585 long FloatToInteger(UWORD *out, mpf_t floatin, long *bitsused)
00586 {
00587     mpz_t z;
00588     WORD nout, nx;
00589     UWORD x;
00590     mpz_init(z);
00591     mpz_set_f(z,floatin);
00592     ZtoForm(out,&nout,z);
00593     mpz_clear(z);
00594     x = out[nout-1]; nx = 0;
00595     while ( x ) { nx++; x »= 1; }
00596     *bitsused = (nout-1)*BITSINWORD + nx;
00597     return(nout);
00598 }
00599
```

```
00600 /*
00601         #] FloatToInteger :
00602         #[ IntegerToFloat :
00603
00604         Converts a Form long integer to a gmp float of default size.
00605         We assume that sizeof(unsigned long int) = 2*sizeof(UWORD).
00606 */
00607
00608 void IntegerToFloat(mpf_t result, UWORD *formlong, int longsize)
00609 {
00610     mpz_t z;
00611     mpz_init(z);
00612     FormtoZ(z,formlong,longsize);
00613     mpf_set_z(result,z);
00614     mpz_clear(z);
00615 }
00616
00617 /*
00618         #] IntegerToFloat :
00619         #[ RatToFloat :
00620
00621         Converts a Form rational to a gmp float of default size.
00622 */
00623
00624 void RatToFloat(mpf_t result, UWORD *formrat, int ratsize)
00625 {
00626     GETIDENTITY
00627     int nnum, nden;
00628     UWORD *num, *den;
00629     int sgn = 0;
00630     if ( ratsize < 0 ) { ratsize = -ratsize; sgn = 1; }
00631     nnum = nden = (ratsize-1)/2;
00632     num = formrat; den = formrat+nnum;
00633     while ( num[nnum-1] == 0 ) { nnum--; }
00634     while ( den[nden-1] == 0 ) { nden--; }
00635     IntegerToFloat(aux2,num,nnum);
00636     IntegerToFloat(aux3,den,nden);
00637     mpf_div(result,aux2,aux3);
00638     if ( sgn > 0 ) mpf_neg(result,result);
00639 }
00640
00641 /*
00642         #] RatToFloat :
00643         #[ FloatFunToRat :
00644 */
00645 /* UNFINISHED_FEATURE_EXCL_START */
00646 WORD FloatFunToRat(PHEAD UWORD *ratout,WORD *in)
00647 {
00648     WORD oldin = in[0], nratout;
00649     in[0] = FLOATFUN;
00650     UnpackFloat(aux4,in);
00651     FloatToRat(ratout,&nratout,aux4);
00652     in[0] = oldin;
00653     return(nratout);
00654 }
00655 /* UNFINISHED_FEATURE_EXCL_STOP */
00656 /*
00657         #] FloatFunToRat :
00658         #[ FloatToRat :
00659
00660     Converts a gmp float to a Form rational.
00661     The algorithm we use is 'repeated fractions' of the style
00662         n0+1/(n1+1/(n2+1/(n3+.....)))
00663     The moment we get an n that is very large we should have the proper fraction
00664     Main problem: what if the sequence keeps going?
00665                   (there are always rounding errors)
00666     We need a cutoff with an error condition.
00667     Basically the product n0*n1*n2*... is an underlimit on the number of
00668     digits in the answer. We can put a limit on this.
00669     A return value of zero means that everything went fine.
00670 */
00671
00672 int FloatToRat(UWORD *ratout, WORD *nratout, mpf_t floatin)
00673 {
00674     GETIDENTITY
00675     WORD *out = AT.WorkPointer;
00676     WORD s; /* the sign. */
00677     UWORD *a, *b, *c, *d, *mc;
00678     WORD na, nb, nc, nd, i;
00679     int nout;
00680     LONG oldpWorkPointer = AT.pWorkPointer;
00681     long bitsused = 0, totalbitsused = 0, totalbits = AC.DefaultPrecision-AC.MaxWeight-1;
00682     int retval = 0, startnul;
00683     WantAddPointers(AC.DefaultPrecision);  /* Horrible overkill */
00684     AT.pWorkSpace[AT.pWorkPointer++] = out;
00685     a = NumberMalloc("FloatToRat");
00686     b = NumberMalloc("FloatToRat");
```

```
00687
00688     s = mpf_sgn(floatin);
00689     if ( s < 0 ) mpf_neg(floatin,floatin);
00690
00691     Form_mpf_empty(aux1);
00692     Form_mpf_empty(aux2);
00693     Form_mpf_empty(aux3);
00694
00695     mpf_trunc(aux1,floatin);      /* This should now be an integer */
00696     mpf_sub(aux2,floatin,aux1);   /* and this >= 0 */
00697     if ( mpf_sgn(aux1) == 0 ) { *out++ = 0; startnul = 1; }
00698     else {
00699         nout = FloatToInteger((UWORD *)out,aux1,&totalbitsused);
00700         out += nout;
00701         startnul = 0;
00702     }
00703     AT.pWorkSpace[AT.pWorkPointer++] = out;
00704     if ( mpf_sgn(aux2) ) {
00705       for(;;) {
00706         mpf_ui_div(aux3,1,aux2);
00707         mpf_trunc(aux1,aux3);
00708         mpf_sub(aux2,aux3,aux1);
00709         if ( mpf_sgn(aux1) == 0 ) { *out++ = 0; }
00710         else {
00711             nout = FloatToInteger((UWORD *)out,aux1,&bitsused);
00712             out += nout;
00713         }
00714         if ( bitsused > (totalbits-totalbitsused)/2 ) { break; }
00715         if ( mpf_sgn(aux2) == 0 ) {
00716             /*if ( startnul == 1 )*/ AT.pWorkSpace[AT.pWorkPointer++] = out;
00717             break;
00718         }
00719         totalbitsused += bitsused;
00720         AT.pWorkSpace[AT.pWorkPointer++] = out;
00721       }
00722       /*
00723         For |floatin| « 1, we have startnul = 1 and hit the precision guard
00724         already on the first continued-fraction step. The resulting float is
00725         therefore close to the rational 0/1 and we can immediately return.
00726        */
00727       if ( startnul == 1 && AT.pWorkPointer == oldpWorkPointer + 2 ) {
00728         *ratout++ = 0; *ratout++ = 1; *ratout = *nratout = 3;
00729         goto ret;
00730       }
00731 /*
00732       At this point we have the function with the repeated fraction.
00733       Now we should work out the fraction. Form code would be:
00734         repeat id   dum_(?a,x1?,x2?) = dum_(?a,x1+1/x2);
00735         id  dum_(x?) = x;
00736       We have to work backwards. This is why we made a list of pointers
00737       in AT.pWorkSpace
00738
00739       Now we need the long integers a and b.
00740       Note that by construction there should never be a GCD!
00741 */
00742       out = (WORD *)(AT.pWorkSpace[--AT.pWorkPointer]);
00743       na = 1; a[0] = 1;
00744       c = (UWORD *)(AT.pWorkSpace[--AT.pWorkPointer]);
00745       nc = nb = ((UWORD *)out)-c;
00746       for ( i = 0; i < nb; i++ ) b[i] = c[i];
00747       mc = c = NumberMalloc("FloatToRat");
00748       while ( AT.pWorkPointer > oldpWorkPointer ) {
00749         d = (UWORD *)(AT.pWorkSpace[--AT.pWorkPointer]);
00750         nd = (UWORD *)(AT.pWorkSpace[AT.pWorkPointer+1])-d; /* This is the x1 in the formula */
00751         if ( nd == 1 && *d == 0 ) break;
00752         c = a; a = b; b = c; nc = na; na = nb; nb = nc; /* 1/x2 */
00753         MulLong(d,nd,a,na,mc,&nc);
00754         AddLong(mc,nc,b,nb,b,&nb);
00755       }
00756       NumberFree(mc,"FloatToRat");
00757       if ( startnul == 0 ) {
00758         c = a; a = b; b = c; nc = na; na = nb; nb = nc;
00759       }
00760     }
00761     else {
00762         c = (UWORD *)(AT.pWorkSpace[oldpWorkPointer]);
00763         na = (UWORD *)out - c;
00764         for ( i = 0; i < na; i++ ) a[i] = c[i];
00765         b[0] = 1; nb = 1;
00766     }
00767 /*
00768     Now we have the fraction in a/b. Create the rational and add the sign.
00769 */
00770     d = ratout;
00771     if ( na == nb ) {
00772         *nratout = (2*na+1)*s;
00773         for ( i = 0; i < na; i++ ) *d++ = a[i];
```

```
00774         for ( i = 0; i < nb; i++ ) *d++ = b[i];
00775     }
00776     else if ( na < nb ) {
00777         *nratout = (2*nb+1)*s;
00778         for ( i = 0; i < na; i++ ) *d++ = a[i];
00779         for ( ; i < nb; i++ ) *d++ = 0;
00780         for ( i = 0; i < nb; i++ ) *d++ = b[i];
00781     }
00782     else {
00783         *nratout = (2*na+1)*s;
00784         for ( i = 0; i < na; i++ ) *d++ = a[i];
00785         for ( i = 0; i < nb; i++ ) *d++ = b[i];
00786         for ( ; i < na; i++ ) *d++ = 0;
00787     }
00788     *d = (UWORD)(*nratout);
00789     NumberFree(b,"FloatToRat");
00790     NumberFree(a,"FloatToRat");
00791     AT.pWorkPointer = oldpWorkPointer;
00792 /*
00793     Just for check we go back to float
00794 */
00795     if ( s < 0 ) mpf_neg(floatin,floatin);
00796 /*
00797     {
00798         WORD n = *nratout;
00799         RatToFloat(aux1,ratout,n);
00800         mpf_sub(aux2,floatin,aux1);
00801         gmp_printf("Diff is %.*Fe\n",40,aux2);
00802     }
00803 */
00804 ret:
00805     return(retval);
00806 }
00807
00808 /*
00809         #] FloatToRat :
00810         #[ ZeroTable :
00811 */
00812
00813 void ZeroTable(mpf_t *tab, int N)
00814 {
00815     int i, j;
00816     for ( i = 0; i < N; i++ ) {
00817         for ( j = 0; j < tab[i]->_mp_prec; j++ ) tab[i]->_mp_d[j] = 0;
00818     }
00819 }
00820
00821 /*
00822         #] ZeroTable :
00823         #[ ReadFloat :
00824
00825     Used by the compiler. It reads a floating point number and
00826     outputs it at AT.WorkPointer as if it were a float_ function
00827     with its arguments.
00828     The call enters with s[-1] == TFLOAT.
00829 */
00830
00831 SBYTE *ReadFloat(SBYTE *s)
00832 {
00833     GETIDENTITY
00834     SBYTE *ss, c;
00835     ss = s;
00836     while ( *ss > 0 ) ss++;
00837     c = *ss; *ss = 0;
00838     gmp_sscanf((char *)s,"%Ff",aux1);
00839     if ( AT.WorkPointer > AT.WorkTop ) {
00840         MLOCK(ErrorMessageLock);
00841         MesWork();
00842         MUNLOCK(ErrorMessageLock);
00843         Terminate(-1);
00844     }
00845     PackFloat(AT.WorkPointer,aux1);
00846     *ss = c;
00847     return(ss);
00848 }
00849
00850 /*
00851         #] ReadFloat :
00852         #[ CheckFloat :
00853
00854     For the tokenizer. Tests whether the input string can be a float.
00855 */
00856
00857 UBYTE *CheckFloat(UBYTE *ss, int *spec)
00858 {
00859     GETIDENTITY
00860     UBYTE *s = ss;
```

```
00861     int gotdot = 0;
00862     /* A single dot is not a valid float */
00863     if ( *s == '.' && FG.cTable[s[-1]] != 1 && FG.cTable[s[1]] != 1 ) return(ss);
00864     while ( FG.cTable[s[-1]] == 1 ) s--;
00865     /* This cannot be a valid float */
00866     if ( *s != '.' && FG.cTable[*s] != 1 ) return(ss);
00867     *spec = 0;
00868     while ( FG.cTable[*s] == 1 ) s++;
00869     if ( *s == '.' ) {
00870         gotdot = 1;
00871         s++;
00872         while ( FG.cTable[*s] == 1 ) s++;
00873     }
00874 /*
00875     Now we have had the mantissa part, which may be zero.
00876     Check for an exponent.
00877 */
00878     if ( *s == 'e' || *s == 'E' ) {
00879         s++;
00880         if ( *s == '-' || *s == '+' ) s++;
00881         if ( FG.cTable[*s] == 1 ) {
00882             s++;
00883             while ( FG.cTable[*s] == 1 ) s++;
00884         }
00885         else { return(ss); }
00886     }
00887     else if ( gotdot == 0 ) return(ss); /* No radix point and no exponent */
00888     if ( AT.aux_ == 0 ) { /* no floating point system */
00889         *spec = -1;
00890         return(s);
00891     }
00892     return(s);
00893 }
00894
00895 /*
00896         #] CheckFloat :
00897     #] Low Level :
00898     #[ Float Routines :
00899         #[ SetFloatPrecision :
00900
00901         Sets the default precision (in bits) of the floats and allocate
00902         buffer space for an output string.
00903         The buffer is used by PrintFloat (decimal output) and Strictrounding
00904         (binary or decimal output), so it must accommodate the larger space
00905         requirement:
00906         exponent: up to 12 chars.
00907         mantissa: prec + a little bit extra
00908 */
00909
00910 int SetFloatPrecision(WORD prec)
00911 {
00912     if ( prec <= 0 ) {
00913         MesPrint("&Illegal value for number of bits for floating point constants: %d",prec);
00914         return(-1);
00915     }
00916     else {
00917         AC.DefaultPrecision = prec;
00918         if ( AO.floatspace != 0 ) M_free(AO.floatspace,"floatspace");
00919         AO.floatsize = (prec+20)*sizeof(char);
00920         AO.floatspace = (UBYTE *)Malloc1(AO.floatsize,"floatspace");
00921         mpf_set_default_prec(prec);
00922         return(0);
00923     }
00924 }
00925
00926 /*
00927         #] SetFloatPrecision :
00928         #[ PrintFloat :
00929
00930     Print the float_ function with its arguments as a floating point
00931     number with numdigits digits.
00932     If however we use rawfloat as a format option it prints it as a
00933     regular Form function and it should never come here because the
00934     print routines in sch.c should intercept it.
00935     The buffer AO.floatspace is allocated when the precision of the
00936     floats is set in the routine SetFloatPrecision.
00937 */
00938
00939 int PrintFloat(WORD *fun,int numdigits)
00940 {
00941     GETIDENTITY
00942     UBYTE *s1, *s2;
00943     int n = 0;
00944     int prec = (AC.DefaultPrecision-AC.MaxWeight-1)*log10(2.0);
00945     if ( numdigits > prec || numdigits == 0 ) {
00946         numdigits = prec;
00947     }
```

```
00948 /*
00949     GMP's gmp_snprintf always prints a non-zero number before the decimal point, so we
00950     ask for one digit less.
00951 */
00952     if ( UnpackFloat(aux4,fun) == 0 )
00953         n = gmp_snprintf((char *)(AO.floatspace),AO.floatsize,"%.*Fe",numdigits-1,aux4);
00954     if ( n > 0 ) {
00955         int n1, n2 = n;
00956         s1 = AO.floatspace+n;
00957         while ( s1 > AO.floatspace && s1[-1] != 'e'
00958          && s1[-1] != 'E' && s1[-1] != '.' ) { s1--; n2--; }
00959         if ( s1 > AO.floatspace && s1[-1] != '.' ) {
00960             s1--; n2--;
00961             s2 = s1; n1 = n2;
00962             while ( s1[-1] == '0' ) { s1--; n1--; }
00963             if ( s1[-1] == '.' ) { s1++; n1++; }
00964             n -= (n2-n1);
00965             while ( n1 < n ) { *s1++ = *s2++; n1++; }
00966             *s1 = 0;
00967         }
00968         if ( AC.OutputMode == FORTRANMODE ) {
00969             s1 = AO.floatspace+n;
00970             while ( s1 > AO.floatspace && *s1 != 'e' && *s1 != 'E' ) {
00971                 s1--;
00972             }
00973             if ( ( AO.DoubleFlag & 2 ) == 2 ) { *s1 = 'Q'; } /* Quadruple precision fortran */
00974             else if ( ( AO.DoubleFlag & 1 ) == 1 ) { *s1 = 'D'; } /* Double precision fortran */
00975             else { *s1 = 'E'; } /* Single precision fortran */
00976         }
00977         if ( AC.OutputMode == MATHEMATICAMODE ) {
00978             s1 = AO.floatspace+n;
00979             s2 = s1+2; *s2-- = 0;
00980             while ( s1 > AO.floatspace && *s1 != 'e' && *s1 != 'E' ) {
00981                 *s2-- = *s1--;
00982             }
00983             *s2-- = '^'; *s2 = '*'; /* Replace 'e' by '^*' */
00984             n++;
00985         }
00986     }
00987     return(n);
00988 }
00989
00990 /*
00991         #] PrintFloat :
00992         #[ AddFloats :
00993 */
00994 /* UNFINISHED_FEATURE_EXCL_START */
00995 int AddFloats(PHEAD WORD *fun3, WORD *fun1, WORD *fun2)
00996 {
00997     int retval = 0;
00998     if ( UnpackFloat(aux1,fun1) == 0 && UnpackFloat(aux2,fun2) == 0 ) {
00999         mpf_add(aux1,aux1,aux2);
01000         PackFloat(fun3,aux1);
01001     }
01002     else { retval = -1; }
01003     return(retval);
01004 }
01005 /* UNFINISHED_FEATURE_EXCL_STOP */
01006 /*
01007         #] AddFloats :
01008         #[ MulFloats :
01009 */
01010 /* UNFINISHED_FEATURE_EXCL_START */
01011 int MulFloats(PHEAD WORD *fun3, WORD *fun1, WORD *fun2)
01012 {
01013     int retval = 0;
01014     if ( UnpackFloat(aux1,fun1) == 0 && UnpackFloat(aux2,fun2) == 0 ) {
01015         mpf_mul(aux1,aux1,aux2);
01016         PackFloat(fun3,aux1);
01017     }
01018     else { retval = -1; }
01019     return(retval);
01020 }
01021 /* UNFINISHED_FEATURE_EXCL_STOP */
01022 /*
01023         #] MulFloats :
01024         #[ DivFloats :
01025 */
01026 /* UNFINISHED_FEATURE_EXCL_START */
01027 int DivFloats(PHEAD WORD *fun3, WORD *fun1, WORD *fun2)
01028 {
01029     int retval = 0;
01030     if ( UnpackFloat(aux1,fun1) == 0 && UnpackFloat(aux2,fun2) == 0 ) {
01031         mpf_div(aux1,aux1,aux2);
01032         PackFloat(fun3,aux1);
01033     }
01034     else { retval = -1; }
```

```
01035     return(retval);
01036 }
01037 /* UNFINISHED_FEATURE_EXCL_STOP */
01038 /*
01039         #] DivFloats :
01040         #[ AddRatToFloat :
01041
01042         Note: this can be optimized, because often the rat is rather simple.
01043 */
01044 /* UNFINISHED_FEATURE_EXCL_START */
01045 int AddRatToFloat(PHEAD WORD *outfun, WORD *infun, UWORD *formrat, WORD nrat)
01046 {
01047     int retval = 0;
01048     if ( UnpackFloat(aux2,infun) == 0 ) {
01049         RatToFloat(aux1,formrat,nrat);
01050         mpf_add(aux2,aux2,aux1);
01051         PackFloat(outfun,aux2);
01052     }
01053     else retval = -1;
01054     return(retval);
01055 }
01056 /* UNFINISHED_FEATURE_EXCL_STOP */
01057 /*
01058         #] AddRatToFloat :
01059         #[ MulRatToFloat :
01060
01061         Note: this can be optimized, because often the rat is rather simple.
01062 */
01063 /* UNFINISHED_FEATURE_EXCL_START */
01064 int MulRatToFloat(PHEAD WORD *outfun, WORD *infun, UWORD *formrat, WORD nrat)
01065 {
01066     int retval = 0;
01067     if ( UnpackFloat(aux2,infun) == 0 ) {
01068         RatToFloat(aux1,formrat,nrat);
01069         mpf_mul(aux2,aux2,aux1);
01070         PackFloat(outfun,aux2);
01071     }
01072     else retval = -1;
01073     return(retval);
01074 }
01075 /* UNFINISHED_FEATURE_EXCL_STOP */
01076 /*
01077         #] MulRatToFloat :
01078         #[ SetupMZVTables :
01079 */
01080
01081 void SetupMZVTables(void)
01082 {
01083 /*
01084     Sets up a table of N+1 mpf_t floats with variable precision.
01085     It assumes that each next element needs one bit less.
01086     Initializes all of them.
01087     We take some extra space, to have one limb overflow everywhere.
01088     Because the depth of the MZV's can get close to the weight
01089     and each deeper sum goes one higher, we make the tablesize a bit bigger.
01090     This may not be needed if we fiddle with the sum boundaries.
01091 */
01092 #ifdef WITHPTHREADS
01093     int i, Nw, id, totnum;
01094     size_t N;
01095     mpf_t *a;
01096     Nw = AC.DefaultPrecision;
01097     N = (size_t)Nw;
01098     totnum = AM.totalnumberofthreads;
01099     for ( id = 0; id < totnum; id++ ) {
01100         AB[id]->T.mpf_tab1 = (void *)Malloc1((N+2)*sizeof(mpf_t),"mpftab1");
01101         a = (mpf_t *)AB[id]->T.mpf_tab1;
01102         for ( i = 0; i <=Nw; i++ ) {
01103 /*
01104     As explained in the comment above, we could make this variable precision
01105     using mpf_init2.
01106 */
01107             mpf_init(a[i]);
01108         }
01109         AB[id]->T.mpf_tab2 = (void *)Malloc1((N+2)*sizeof(mpf_t),"mpftab2");
01110         a = (mpf_t *)AB[id]->T.mpf_tab2;
01111         for ( i = 0; i <=Nw; i++ ) {
01112             mpf_init(a[i]);
01113         }
01114     }
01115 #else
01116     int i, Nw;
01117     size_t N;
01118     Nw = AC.DefaultPrecision;
01119     N = (size_t)Nw;
01120     AT.mpf_tab1 = (void *)Malloc1((N+2)*sizeof(mpf_t),"mpftab1");
01121     for ( i = 0; i <= Nw; i++ ) {
```

```
01122 /*
01123     As explained in the comment above, we could make this variable precision
01124     using mpf_init2.
01125 */
01126         mpf_init(mpftab1[i]);
01127     }
01128     AT.mpf_tab2 = (void *)Malloc1((N+2)*sizeof(mpf_t),"mpftab2");
01129     for ( i = 0; i <= Nw; i++ ) {
01130         mpf_init(mpftab2[i]);
01131     }
01132 #endif
01133     AS.delta_1 = (void *)Malloc1(sizeof(mpf_t),"delta1");
01134     mpf_init(mpfdelta1);
01135     SimpleDelta(mpfdelta1,1); /* this can speed up things. delta1 = ln(2) */
01136 }
01137
01138 /*
01139         #] SetupMZVTables :
01140         #[ SetupMPFTables :
01141
01142         Allocates the aux variables
01143 */
01144
01145 void SetupMPFTables(void)
01146 {
01147 #ifdef WITHPTHREADS
01148     int id, totnum;
01149     mpf_t *a;
01150 #ifdef WITHSORTBOTS
01151     totnum = MaX(2*AM.totalnumberofthreads-3,AM.totalnumberofthreads);
01152 #endif
01153     for ( id = 0; id < totnum; id++ ) {
01154 /*
01155         We work here with a[0] etc because the aux1 etc contain B which
01156         in the current routine would be AB[0] only
01157 */
01158         AB[id]->T.aux_ = (void *)Malloc1(sizeof(mpf_t)*8,"AB[id]->T.aux_");
01159         a = (mpf_t *)AB[id]->T.aux_;
01160         mpf_inits(a[0],a[1],a[2],a[3],a[4],a[5],a[6],a[7],(mpf_ptr)0);
01161         if ( AB[id]->T.indi1 ) M_free(AB[id]->T.indi1,"indi1");
01162         AB[id]->T.indi1 = (WORD *)Malloc1(sizeof(WORD)*AC.MaxWeight*2,"indi1");
01163         AB[id]->T.indi2 = AB[id]->T.indi1 + AC.MaxWeight;
01164     }
01165 #else
01166     AT.aux_ = (void *)Malloc1(sizeof(mpf_t)*8,"AT.aux_");
01167     mpf_inits(aux1,aux2,aux3,aux4,aux5,auxjm,auxjjm,auxsum,(mpf_ptr)0);
01168     if ( AT.indi1 ) M_free(AT.indi1,"indi1");
01169     AT.indi1 = (WORD *)Malloc1(sizeof(WORD)*AC.MaxWeight*2,"indi1");
01170     AT.indi2 = AT.indi1 + AC.MaxWeight;
01171 #endif
01172 }
01173
01174 /*
01175         #] SetupMPFTables :
01176         #[ ClearMZVTables :
01177 */
01178
01179 void ClearMZVTables(void)
01180 {
01181 #ifdef WITHPTHREADS
01182     int i, id, totnum;
01183     mpf_t *a;
01184     totnum = AM.totalnumberofthreads;
01185     for ( id = 0; id < totnum; id++ ) {
01186         if ( AB[id]->T.mpf_tab1 ) {
01187 /*
01188             We work here with a[0] etc because the aux1, mpftab1 etc contain B
01189             which in the current routine would be AB[0] only
01190 */
01191             a = (mpf_t *)AB[id]->T.mpf_tab1;
01192             for ( i = 0; i <=AC.DefaultPrecision; i++ ) {
01193                 mpf_clear(a[i]);
01194             }
01195             M_free(AB[id]->T.mpf_tab1,"mpftab1");
01196             AB[id]->T.mpf_tab1 = 0;
01197         }
01198         if ( AB[id]->T.mpf_tab2 ) {
01199             a = (mpf_t *)AB[id]->T.mpf_tab2;
01200             for ( i = 0; i <=AC.DefaultPrecision; i++ ) {
01201                 mpf_clear(a[i]);
01202             }
01203             M_free(AB[id]->T.mpf_tab2,"mpftab2");
01204             AB[id]->T.mpf_tab2 = 0;
01205         }
01206     }
01207 #ifdef WITHSORTBOTS
01208     totnum = MaX(2*AM.totalnumberofthreads-3,AM.totalnumberofthreads);
```

```
01209 #endif
01210     for ( id = 0; id < totnum; id++ ) {
01211         if ( AB[id]->T.aux_ ) {
01212             a = (mpf_t *)AB[id]->T.aux_;
01213             mpf_clears(a[0],a[1],a[2],a[3],a[4],a[5],a[6],a[7],(mpf_ptr)0);
01214             M_free(AB[id]->T.aux_,"AB[id]->T.aux_");
01215             AB[id]->T.aux_ = 0;
01216         }
01217         if ( AB[id]->T.indi1 ) { M_free(AB[id]->T.indi1,"indi1"); AB[id]->T.indi1 = 0; }
01218     }
01219 #else
01220     int i;
01221     if ( AT.mpf_tab1 ) {
01222         for ( i = 0; i <= AC.DefaultPrecision; i++ ) {
01223             mpf_clear(mpftab1[i]);
01224         }
01225         M_free(AT.mpf_tab1,"mpftab1");
01226         AT.mpf_tab1 = 0;
01227     }
01228     if ( AT.mpf_tab2 ) {
01229         for ( i = 0; i <= AC.DefaultPrecision; i++ ) {
01230             mpf_clear(mpftab2[i]);
01231         }
01232         M_free(AT.mpf_tab2,"mpftab2");
01233         AT.mpf_tab2 = 0;
01234     }
01235     if ( AT.aux_ ) {
01236         mpf_clears(aux1,aux2,aux3,aux4,aux5,auxjm,auxjjm,auxsum,(mpf_ptr)0);
01237         M_free(AT.aux_,"AT.aux_");
01238         AT.aux_ = 0;
01239     }
01240     if ( AT.indi1 ) { M_free(AT.indi1,"indi1"); AT.indi1 = 0; }
01241 #endif
01242     if ( AO.floatspace ) { M_free(AO.floatspace,"floatspace"); AO.floatspace = 0;
01243         AO.floatsize = 0; }
01244     if ( AS.delta_1 ) {
01245         mpf_clear(mpfdelta1);
01246         M_free(AS.delta_1,"delta1");
01247         AS.delta_1 = 0;
01248     }
01249 }
01250
01251 /*
01252         #] ClearMZVTables :
01253         #[ CoToFloat :
01254 */
01255
01256 int CoToFloat(UBYTE *s)
01257 {
01258     GETIDENTITY
01259     if ( AT.aux_ == 0 ) {
01260         MesPrint("&Illegal attempt to convert to float_ without activating floating point numbers.");
01261         MesPrint("&Forgotten %#startfloat instruction?");
01262         return(1);
01263     }
01264     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
01265     if ( *s ) {
01266         MesPrint("&Illegal argument(s) in ToFloat statement: '%s'",s);
01267         return(1);
01268     }
01269     Add2Com(TYPETOFLOAT);
01270     return(0);
01271 }
01272
01273 /*
01274         #] CoToFloat :
01275         #[ CoToRat :
01276 */
01277
01278 int CoToRat(UBYTE *s)
01279 {
01280     GETIDENTITY
01281     if ( AT.aux_ == 0 ) {
01282         MesPrint("&Illegal attempt to convert from float_ without activating floating point
      numbers.");
01283         MesPrint("&Forgotten %#startfloat instruction?");
01284         return(1);
01285     }
01286     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
01287     if ( *s ) {
01288         MesPrint("&Illegal argument(s) in ToRat statement: '%s'",s);
01289         return(1);
01290     }
01291     Add2Com(TYPETORAT);
01292     return(0);
01293 }
01294
```

```
01295 /*
01296         #] CoToRat :
01297         #[ CoStrictRounding :
01298
01299         Syntax: StrictRounding [precision][base]
01300         - precision: number of digits to round to (optional)
01301         - base: 'd' for decimal (base 10) or 'b' for binary (base 2)
01302
01303         If no arguments are provided, uses default precision with binary base.
01304 */
01305 int CoStrictRounding(UBYTE *s)
01306 {
01307     GETIDENTITY
01308     WORD x;
01309     int base;
01310     if ( AT.aux_ == 0 ) {
01311         MesPrint("&Illegal attempt for strict rounding without activating floating point numbers.");
01312         MesPrint("&Forgotten %#startfloat instruction?");
01313         return(1);
01314     }
01315     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
01316     if ( *s == 0 ) {
01317         /* No subkey, which means round to default precision */
01318         x = AC.DefaultPrecision - AC.MaxWeight - 1;
01319         base = 2;
01320     }
01321     else if ( FG.cTable[*s] == 1 ) { /* number */
01322         ParseNumber(x,s)
01323         if ( tolower(*s) == 'd' ) { base = 10; s++; }      /* decimal base */
01324         else if ( tolower(*s) == 'b' ){ base = 2; s++; }  /* binary base */
01325         else goto IllPar;  /* invalid base specification */
01326     }
01327     else {
01328         goto IllPar;
01329     }
01330     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
01331
01332     /* Check for invalid arguments */
01333     if ( *s ) {
01334 IllPar:
01335         MesPrint("&Illegal argument(s) in StrictRounding statement: '%s'",s);
01336         return(1);
01337     }
01338     Add4Com(TYPESTRICTROUNDING,x,base);
01339     return(0);
01340 }
01341 /*
01342         #] CoStrictRounding :
01343         #[ CoChop :
01344
01345         LHS notation of the chop statement:
01346             TYPECHOP, length, FLOATFUN, ...
01347         where FLOATFUN, ... represents the threshold of the chop statement in
01348         the notation of a float_ function with its arguments.
01349
01350 */
01351
01352 int CoChop(UBYTE *s)
01353 {
01354     GETIDENTITY
01355     UBYTE *ss, c;
01356     WORD *w, *OldWork;
01357     int spec, pow = 1;
01358     unsigned long x;
01359     if ( AT.aux_ == 0 ) {
01360         MesPrint("&Illegal attempt to chop a float_ without activating floating point numbers.");
01361         MesPrint("&Forgotten %#startfloat instruction?");
01362         return(1);
01363     }
01364     if ( *s == 0 ) {
01365         MesPrint("&Chop needs a number (float, rational or power) as an argument.");
01366         return(1);
01367     }
01368     /* Create TYPECHOP header */
01369     w = OldWork = AT.WorkPointer;
01370     *w++ = TYPECHOP;
01371     w++;
01372
01373     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
01374
01375     /*
01376         The argument of chop can be
01377         1: a floating-point number
01378         2: an integer, rational number or power
01379      */
01380     if ( FG.cTable[*s] == 1 || *s == '.' ) {
01381         /* 1: Attempt to parse as floating-point number */
```

```
01382         ss = CheckFloat(s, &spec);
01383         if ( ss > s ) {
01384             /* CheckFloat found a valid float */
01385             AT.WorkPointer = w;
01386             /*
01387                 Reads the floating point number and outputs it at AT.WorkPointer as if it were a
    float_
01388                 function with its arguments.
01389              */
01390             ReadFloat((SBYTE *)s);
01391             s = ss;
01392             w += w[1];
01393         }
01394         else {
01395             /* 2: CheckFloat didn't find a float, we now try for rationals and powers */
01396             /* Parse the integer part (numerator for rationals) */
01397             if ( FG.cTable[*s] == 1 ) {
01398                 ParseNumber(x,s)
01399                 mpf_set_ui(aux1,x);
01400             }
01401             while ( *s == ' ' || *s == '\t' ) s++;
01402             /* Check for rational number or power*/
01403             if ( *s == '/' || *s == '^' ) {
01404                 c = *s; s++;
01405                 while ( *s == ' ' || *s == '\t' ) s++;
01406                 if ( *s == '-' ) { s++; pow = -1; } /* negative power */
01407                 /* Parse the denominator or power */
01408                 if ( FG.cTable[*s] == 1 ) {
01409                     ParseNumber(x,s)
01410                     if ( c == '/' ) { /* rational */
01411                         if ( x == 0 ) {
01412                             MesPrint("&Division by zero in chop statement.");
01413                             return(1);
01414                         }
01415                         /* Perform the division */
01416                         mpf_div_ui(aux1, aux1,x);
01417                     }
01418                     else { /* Power */
01419                         mpf_pow_ui(aux1,aux1,x);
01420                         if ( pow == -1 ) {
01421                             mpf_ui_div(aux1,(unsigned long) 1, aux1);
01422                         }
01423                     }
01424                 }
01425             }
01426             /* Put aux1 in the notation of a float_ function */
01427             PackFloat(w, aux1);
01428             w += w[1];
01429         }
01430     }
01431     if ( *s ) {
01432         MesPrint("&Illegal argument(s) in Chop statement: '%s'.",s);
01433         return(1);
01434     }
01435     AT.WorkPointer = OldWork;
01436     AT.WorkPointer[1] = w - AT.WorkPointer;  /* Set total length */
01437     AddNtoL(AT.WorkPointer[1],AT.WorkPointer); /* Add the LHS to the compiler buffer */
01438     return(0);
01439 }
01440
01441 /*
01442         #] CoChop :
01443         #[ ToFloat :
01444
01445         Converts the coefficient to floating point if it is still a rat.
01446 */
01447
01448 int ToFloat(PHEAD WORD *term, WORD level)
01449 {
01450     WORD *t, *tstop, nsize, nsign = 3;
01451     t = term+*term;
01452     nsize = ABS(t[-1]);
01453     tstop = t - nsize;
01454     if ( t[-1] < 0 ) { nsign = -nsign; }
01455     if ( nsize == 3 && t[-2] == 1 && t[-3] == 1 ) { /* Could be float */
01456         t = term + 1;
01457         while ( t < tstop ) {
01458             if ( ( *t == FLOATFUN ) && ( t+t[1] == tstop ) ) {
01459                 return(Generator(BHEAD term,level));
01460             }
01461             t += t[1];
01462         }
01463     }
01464     RatToFloat(aux4,(UWORD *)tstop,nsize);
01465     PackFloat(tstop,aux4);
01466     tstop += tstop[1];
01467     *tstop++ = 1; *tstop++ = 1; *tstop++ = nsign;
```

```
01468     *term = tstop - term;
01469     AT.WorkPointer = tstop;
01470     return(Generator(BHEAD term,level));
01471 }
01472
01473 /*
01474         #] ToFloat :
01475         #[ ToRat :
01476
01477         Tries to convert the coefficient to rational if it is still a float.
01478 */
01479
01480 int ToRat(PHEAD WORD *term, WORD level)
01481 {
01482     WORD *tstop, *t, nsize, nsign, ncoef;
01483 /*
01484     1: find the float which should be at the end.
01485 */
01486     tstop = term + *term; nsize = ABS(tstop[-1]);
01487     nsign = tstop[-1] < 0 ? -1: 1; tstop -= nsize;
01488     t = term+1;
01489     while ( t < tstop ) {
01490         if ( *t == FLOATFUN && t + t[1] == tstop && TestFloat(t) &&
01491         nsize == 3 && tstop[0] == 1 && tstop[1] == 1 ) break;
01492         t += t[1];
01493     }
01494     if ( t < tstop ) {
01495 /*
01496         Now t points at the float_ function and everything is correct.
01497         The result can go straight over the float_ function.
01498 */
01499         UnpackFloat(aux4,t);
01500         // If aux4 is zero, the term vanishes
01501         if ( mpf_sgn(aux4) == 0 ) return(0);
01502         if ( FloatToRat((UWORD *)t,&ncoef,aux4) == 0 ) {
01503             // Check if the resulting rational is zero
01504             if ( t[0] == 0 && t[1] == 1 && ncoef == 3 ) return(0);
01505             t += ABS(ncoef);
01506             t[-1] = ncoef*nsign;
01507             *term = t - term;
01508         }
01509     }
01510     return(Generator(BHEAD term,level));
01511 }
01512
01513 /*
01514         #] ToRat :
01515         #[ StrictRounding :
01516
01517         Rounds floating point numbers to a specified precision
01518         in a given base (decimal or binary).
01519 */
01520 int StrictRounding(PHEAD WORD *term, WORD level, WORD prec, WORD base) {
01521     WORD *t,*tstop;
01522     int sign,size,maxprec = AC.DefaultPrecision-AC.MaxWeight-1;
01523     /* maxprec is in bits */
01524     if ( base == 2 && prec > maxprec ) {
01525         prec = maxprec;
01526     }
01527     if ( base == 10 && prec > (int)(maxprec*log10(2.0)) ) {
01528         prec = maxprec*log10(2.0);
01529     }
01530     /* Find the float which should be at the end. */
01531     tstop = term + *term; size = ABS(tstop[-1]);
01532     sign = tstop[-1] < 0 ? -1: 1; tstop -= size;
01533     t = term+1;
01534     while ( t < tstop ) {
01535         if ( *t == FLOATFUN && t + t[1] == tstop && TestFloat(t) &&
01536         size == 3 && tstop[0] == 1 && tstop[1] == 1) {
01537             break;
01538         }
01539         t += t[1];
01540     }
01541     if ( t < tstop ) {
01542 /*
01543         Now t points at the float_ function and everything is correct.
01544         The result can go straight over the float_ function.
01545 */
01546         char *s;
01547         mp_exp_t exp;
01548         /* Extract the floating point value */
01549         UnpackFloat(aux4,t);
01550         /* Convert to string:
01551            - Format as MeN with M the mantissa and N the exponent
01552            - the generated string by mpf_get_str is the fraction/mantissa with
01553            an implicit radix point immediately to the left of the first digit.
01554            The applicable exponent is written in exp. */
```

```
01555         s = (char *)AO.floatspace;
01556         *s++ = '.';
01557         mpf_get_str(s,&exp, base, prec, aux4);
01558         while ( *s != 0 ) s++;
01559         *s++ = 'e';
01560         snprintf(s,AO.floatsize-(s-(char *)AO.floatspace),"%ld",exp);
01561         /* Negative base values are used to specify that the exponent is in decimal */
01562         mpf_set_str(aux4,(char *)AO.floatspace,-base);
01563         /* Pack the rounded floating point value back into the term */
01564         PackFloat(t,aux4);
01565         t+=t[1];
01566         *t++ = 1; *t++ = 1; *t++ = 3*sign;
01567         *term = t - term;
01568     }
01569     return(Generator(BHEAD term,level));
01570 }
01571 /*
01572         #] StrictRounding :
01573         #[ Chop :
01574 
01575     Removes terms with a floating point number smaller than a given threshold.
01576 
01577     Search for a FLOATFUN and compares its absolute value against the threshold
01578     specified in the chop statement. This threshold can be obtained from the
01579     LHS of the chop statement in the compiler buffer.
01580 
01581  */
01582 int Chop(PHEAD WORD *term, WORD level)
01583 {
01584     WORD *tstop, *t, nsize, *threshold;
01585     CBUF *C = cbuf+AM.rbufnum;
01586     /* Find the float which should be at the end. */
01587     tstop = term + *term;
01588     nsize = ABS(tstop[-1]); tstop -= nsize;
01589     t = term+1;
01590     while ( t < tstop ) {
01591         if ( *t == FLOATFUN && t + t[1] == tstop && TestFloat(t) &&
01592         nsize == 3 && tstop[0] == 1 && tstop[1] == 1 ) break;
01593         t += t[1];
01594     }
01595     if ( t < tstop ) {
01596         /* Get threshold value from compiler buffer */
01597         threshold = C->lhs[level];
01598         threshold += 2;  /* Skip TYPECHOP header */
01599         UnpackFloat(aux5, threshold);
01600 
01601         /* Extract float and compute its absolute value */
01602         UnpackFloat(aux4, t);
01603         mpf_abs(aux4, aux4);
01604 
01605         /* Remove if < threshold */
01606         if ( mpf_cmp(aux4, aux5) < 0 ) return(0);
01607     }
01608     return(Generator(BHEAD term,level));
01609 }
01610 
01611 /*
01612         #] Chop :
01613     #] Float Routines :
01614     #[ Sorting :
01615 
01616         We need a method to see whether two terms need addition that could
01617         involve floating point numbers.
01618         1: if PolyWise is active, we do not need anything, because
01619            Poly(Rat)Fun and floating point are mutually exclusive.
01620         2: This means that there should be only interference in AddCoef.
01621            and the AddRat parts in MergePatches, MasterMerge, SortBotMerge
01622            and PF_GetLoser.
01623         The compare routine(s) should recognise the float_ and give off
01624         a code (SortFloatMode) inside the AT struct:
01625         0: no float_
01626         1: float_ in term1 only
01627         2: float_ in term2 only
01628         3: float_ in both terms
01629         Be careful: we have several compare routines, because various codes
01630         use their own routines and we just set a variable with its address.
01631         Currently we have Compare1, CompareSymbols and CompareHSymbols.
01632         Only Compare1 should be active for this. The others should make sure
01633         that the proper variable is always zero.
01634         Remember: the sign of the coefficient is in the last word of the term.
01635 
01636         We need two routines:
01637         1: AddWithFloat for SplitMerge which sorts by pointer.
01638         2: MergeWithFloat for MergePatches etc which keeps terms as much
01639            as possible in their location.
01640         The routines start out the same, because AT.SortFloatMode, set in
01641         Compare1, tells more or less what should be done.
```

```
01642         The difference is in where we leave the result.
01643
01644         In the future we may want to add a feature that makes the result
01645         zero if the sum comes within a certain distance of the numerical
01646         accuracy, like for instance 5% of the number of bits.
01647
01648         #[ AddWithFloat :
01649 */
01650
01651 int AddWithFloat(PHEAD WORD **ps1, WORD **ps2)
01652 {
01653     GETBIDENTITY
01654     SORTING *S = AT.SS;
01655     WORD *coef1, *coef2, size1, size2, *fun1, *fun2, *fun3;
01656     WORD *s1,*s2,sign3,j,jj, *t1, *t2, i;
01657     s1 = *ps1;
01658     s2 = *ps2;
01659     coef1 = s1+*s1; size1 = coef1[-1]; coef1 -= ABS(coef1[-1]);
01660     coef2 = s2+*s2; size2 = coef2[-1]; coef2 -= ABS(coef2[-1]);
01661     if ( AT.SortFloatMode == 3 ) {
01662         fun1 = s1+1; while ( fun1 < coef1 && fun1[0] != FLOATFUN ) fun1 += fun1[1];
01663         fun2 = s2+1; while ( fun2 < coef2 && fun2[0] != FLOATFUN ) fun2 += fun2[1];
01664         UnpackFloat(aux1,fun1);
01665         if ( size1 < 0 ) mpf_neg(aux1,aux1);
01666         UnpackFloat(aux2,fun2);
01667         if ( size2 < 0 ) mpf_neg(aux2,aux2);
01668     }
01669     else if ( AT.SortFloatMode == 1 ) {
01670         fun1 = s1+1; while ( fun1 < coef1 && fun1[0] != FLOATFUN ) fun1 += fun1[1];
01671         UnpackFloat(aux1,fun1);
01672         if ( size1 < 0 ) mpf_neg(aux1,aux1);
01673         fun2 = coef2;
01674         RatToFloat(aux2,(UWORD *)coef2,size2);
01675     }
01676     else if ( AT.SortFloatMode == 2 ) {
01677         fun2 = s2+1; while ( fun2 < coef2 && fun2[0] != FLOATFUN ) fun2 += fun2[1];
01678         fun1 = coef1;
01679         RatToFloat(aux1,(UWORD *)coef1,size1);
01680         UnpackFloat(aux2,fun2);
01681         if ( size2 < 0 ) mpf_neg(aux2,aux2);
01682     }
01683     else {
01684 /* INTERNAL_ERROR_EXCL_START */
01685         MLOCK(ErrorMessageLock);
01686         MesPrint("!>Illegal value %d for AT.SortFloatMode in AddWithFloat.",AT.SortFloatMode);
01687         MUNLOCK(ErrorMessageLock);
01688         Terminate(-1);
01689         return(0);
01690 /* INTERNAL_ERROR_EXCL_STOP */
01691     }
01692     mpf_add(aux3,aux1,aux2);
01693     sign3 = mpf_sgn(aux3);
01694     if ( sign3 < 0 ) mpf_neg(aux3,aux3);
01695     fun3 = TermMalloc("AddWithFloat");
01696     PackFloat(fun3,aux3);
01697 /*
01698     Now we have to calculate where the sumterm fits.
01699     If we are sloppy, we can be faster, but run the risk to need the
01700     extension space, even when it is not needed. At slightly lower speed
01701     (ie first creating the result in scribble space) we are better off.
01702     This is why we use TermMalloc.
01703
01704     The new term will have a rational coefficient of 1,1,+-3.
01705     The size will be (fun1 or fun2 - term) + fun3 +3;
01706 */
01707     if ( AT.SortFloatMode == 3 ) {
01708         if ( fun1[1] == fun3[1]  ) {
01709 Over1:
01710             i = fun3[1]; t1 = fun1; t2 = fun3; NCOPY(t1,t2,i);
01711             *t1++ = 1; *t1++ = 1; *t1++ = sign3 < 0 ? -3: 3;
01712             *s1 = t1-s1; goto Finished;
01713         }
01714         else if ( fun2[1] == fun3[1]  ) {
01715 Over2:
01716             i = fun3[1]; t1 = fun2; t2 = fun3; NCOPY(t1,t2,i);
01717             *t1++ = 1; *t1++ = 1; *t1++ = sign3 < 0 ? -3: 3;
01718             *s2 = t1-s2; *ps1 = s2; goto Finished;
01719         }
01720         else if ( fun1[1] >= fun3[1]  ) goto Over1;
01721         else if ( fun2[1] >= fun3[1]  ) goto Over2;
01722     }
01723     else if ( AT.SortFloatMode == 1 ) {
01724         if ( fun1[1] >= fun3[1]  ) goto Over1;
01725         else if ( fun3[1]+3 <= ABS(size2) ) goto Over2;
01726     }
01727     else if ( AT.SortFloatMode == 2 ) {
01728         if ( fun2[1] >= fun3[1]  ) goto Over2;
```

```
01729         else if ( fun3[1]+3 <= ABS(size1) ) goto Over1;
01730     }
01731 /*
01732     Does not fit. Go to extension space.
01733 */
01734     jj = fun1-s1;
01735     j = jj+fun3[1]+3; /* space needed */
01736     if ( (S->sFill + j) >= S->sTop2 ) {
01737         GarbHand();
01738         s1 = *ps1; /* new position of the term after the garbage collection */
01739         fun1 = s1+jj;
01740     }
01741     t1 = S->sFill;
01742     for ( i = 0; i < jj; i++ ) *t1++ = s1[i];
01743     i = fun3[1]; s1 = fun3; NCOPY(t1,s1,i);
01744     *t1++ = 1; *t1++ = 1; *t1++ = sign3 < 0 ? -3: 3;
01745     *ps1 = S->sFill;
01746     **ps1 = t1-*ps1;
01747     S->sFill = t1;
01748 Finished:
01749     *ps2 = 0;
01750     TermFree(fun3,"AddWithFloat");
01751     AT.SortFloatMode = 0;
01752     if ( **ps1 > AM.MaxTer/((LONG)(sizeof(WORD))) ) {
01753         MLOCK(ErrorMessageLock);
01754         MesPrint("Term too complex after addition in sort. MaxTermSize = %10l",
01755         AM.MaxTer/sizeof(WORD));
01756         MUNLOCK(ErrorMessageLock);
01757         Terminate(-1);
01758     }
01759     if ( **ps1 > S->verbMaxTermSize ) S->verbMaxTermSize = **ps1;
01760     return(1);
01761 }
01762
01763 /*
01764         #] AddWithFloat :
01765         #[ MergeWithFloat :
01766
01767         Note that we always park the result on top of term1.
01768         This makes life easier, because the original AddRat in MergePatches
01769         does this as well.
01770 */
01771
01772 int MergeWithFloat(PHEAD WORD **interm1, WORD **interm2)
01773 {
01774     GETBIDENTITY
01775     WORD *coef1, *coef2, size1, size2, *fun1, *fun2, *fun3, *tt;
01776     WORD sign3,jj, *t1, *t2, i, *term1 = *interm1, *term2 = *interm2;
01777     int retval = 0;
01778     coef1 = term1+*term1; size1 = coef1[-1]; coef1 -= ABS(size1);
01779     coef2 = term2+*term2; size2 = coef2[-1]; coef2 -= ABS(size2);
01780     if ( AT.SortFloatMode == 3 ) {
01781         fun1 = term1+1; while ( fun1 < coef1 && fun1[0] != FLOATFUN ) fun1 += fun1[1];
01782         fun2 = term2+1; while ( fun2 < coef2 && fun2[0] != FLOATFUN ) fun2 += fun2[1];
01783         UnpackFloat(aux1,fun1);
01784         if ( size1 < 0 ) mpf_neg(aux1,aux1);
01785         UnpackFloat(aux2,fun2);
01786         if ( size2 < 0 ) mpf_neg(aux2,aux2);
01787     }
01788     else if ( AT.SortFloatMode == 1 ) {
01789         fun1 = term1+1; while ( fun1 < coef1 && fun1[0] != FLOATFUN ) fun1 += fun1[1];
01790         UnpackFloat(aux1,fun1);
01791         if ( size1 < 0 ) mpf_neg(aux1,aux1);
01792         fun2 = coef2;
01793         RatToFloat(aux2,(UWORD *)coef2,size2);
01794     }
01795     else if ( AT.SortFloatMode == 2 ) {
01796         fun2 = term2+1; while ( fun2 < coef2 && fun2[0] != FLOATFUN ) fun2 += fun2[1];
01797         fun1 = coef1;
01798         RatToFloat(aux1,(UWORD *)coef1,size1);
01799         UnpackFloat(aux2,fun2);
01800         if ( size2 < 0 ) mpf_neg(aux2,aux2);
01801     }
01802     else {
01803 /* INTERNAL_ERROR_EXCL_START */
01804         MLOCK(ErrorMessageLock);
01805         MesPrint("!>Illegal value %d for AT.SortFloatMode in MergeWithFloat.",AT.SortFloatMode);
01806         MUNLOCK(ErrorMessageLock);
01807         Terminate(-1);
01808         return(0);
01809 /* INTERNAL_ERROR_EXCL_STOP */
01810     }
01811     mpf_add(aux3,aux1,aux2);
01812     sign3 = mpf_sgn(aux3);
01813 /*
01814     Now check whether we can park the result on top of one of the input terms.
01815 */
```

```
01816     if ( sign3 < 0 ) mpf_neg(aux3,aux3);
01817     fun3 = TermMalloc("MergeWithFloat");
01818     PackFloat(fun3,aux3);
01819     if ( AT.SortFloatMode == 3 ) {
01820         if ( fun1[1] + ABS(size1) == fun3[1] + 3 ) {
01821 OnTopOf1:   t1 = fun3; t2 = fun1;
01822             for ( i = 0; i < fun3[1]; i++ ) *t2++ = *t1++;
01823             *t2++ = 1; *t2++ = 1; *t2++ = sign3 < 0 ? -3: 3;
01824             retval = 1;
01825         }
01826         else if ( fun1[1] + ABS(size1) > fun3[1] + 3 ) {
01827 Shift1:     t2 = term1 + *term1; tt = t2;
01828             *--t2 = sign3 < 0 ? -3: 3; *--t2 = 1; *--t2 = 1;
01829             t1 = fun3 + fun3[1]; for ( i = 0; i < fun3[1]; i++ ) *--t2 = *--t1;
01830             t1 = fun1;
01831             while ( t1 > term1 ) *--t2 = *--t1;
01832             *t2 = tt-t2; term1 = t2;
01833             retval = 1;
01834         }
01835         else { /* Here we have to move term1 to the left to make room. */
01836             jj = fun3[1]-fun1[1]+3-ABS(size1); /* This is positive */
01837 Over1:      t2 = term1-jj; t1 = term1;
01838             while ( t1 < fun1 ) *t2++ = *t1++;
01839             term1 -= jj;
01840             *term1 += jj;
01841             for ( i = 0; i < fun3[1]; i++ ) *t2++ = fun3[i];
01842             *t2++ = 1; *t2++ = 1;  *t2++ = sign3 < 0 ? -3: 3;
01843             retval = 1;
01844         }
01845     }
01846     else if ( AT.SortFloatMode == 1 ) {
01847         if ( fun1[1] + ABS(size1) == fun3[1] + 3 ) goto OnTopOf1;
01848         else if ( fun1[1] + ABS(size1) > fun3[1] + 3 ) goto Shift1;
01849         else {
01850             jj = fun3[1]-fun1[1]+3-ABS(size1); /* This is positive */
01851             goto Over1;
01852         }
01853     }
01854     else { /* Can only be 2, based on previous tests */
01855         if ( fun3[1] + 3 == ABS(size1) ) goto OnTopOf1;
01856         else if ( fun3[1] + 3 < ABS(size1) ) goto Shift1;
01857         else {
01858             jj = fun3[1]+3-ABS(size1); /* This is positive */
01859             goto Over1;
01860         }
01861     }
01862     *interm1 = term1;
01863     TermFree(fun3,"MergeWithFloat");
01864     AT.SortFloatMode = 0;
01865     return(retval);
01866 }
01867
01868 /*
01869         #] MergeWithFloat :
01870     #] Sorting :
01871     #[ MZV :
01872
01873         The functions here allow the arbitrary precision calculation
01874         of MZV's and Euler sums.
01875         They are called when the functions mzv_ and/or euler_ are used
01876         and the evaluate statement is applied.
01877         The output is in a float_ function.
01878         The expand statement tries to express the functions in terms of a basis.
01879         The bases are the 'standard basis for the euler sums and the
01880         pushdown bases from the datamine, unless otherwise specified.
01881
01882         #[ SimpleDelta :
01883 */
01884
01885 void SimpleDelta(mpf_t sum, int m)
01886 {
01887     long s = 1;
01888     long prec = AC.DefaultPrecision;
01889     unsigned long xprec = (unsigned long)prec, x;
01890     int j, jmax, n;
01891     mpf_t jm,jjm;
01892     mpf_init(jm); mpf_init(jjm);
01893     if ( m < 0 ) { s = -1; m = -m; }
01894
01895 /*
01896     We will loop until 1/2^j/j^m is smaller than the default precision.
01897     Just running to prec is however overkill, specially when m is large.
01898     We try to estimate a better value.
01899     jmax = prec - ((log2(prec)-1)*m).
01900     Hence we need the leading bit of prec.
01901     We are still overshooting a bit.
01902 */
```

```
01903     n = 0; x = xprec;
01904     while ( x ) { x >>= 1; n++; }
01905 /*
01906     We have now n = floor(log2(x))+1.
01907 */
01908     n--;
01909     jmax = (int)((int)xprec - (n-1)*m);
01910 /*
01911     For small prec and large m, the estimate can be wrong and even be negative,
01912     so we increase jmax until jmax + m*log2(jmax) > prec
01913 */
01914     if ( jmax < 0 ) jmax = 1;
01915     do {
01916         n = 0;
01917         x = (unsigned long)jmax;
01918         while (x) { x >>= 1; n++; }
01919         n--; // floor(log2(jmax))
01920         jmax++;
01921     } while ( jmax + m * n <= prec );
01922     mpf_set_ui(sum,0);
01923     for ( j = 1; j <= jmax; j++ ) {
01924 #ifdef WITHCUTOFF
01925         xprec--;
01926         mpf_set_prec_raw(jm,xprec);
01927         mpf_set_prec_raw(jjm,xprec);
01928 #endif
01929         mpf_set_ui(jm,1L);
01930         mpf_div_ui(jjm,jm,(unsigned long)j);
01931         mpf_pow_ui(jm,jjm,(unsigned long)m);
01932         mpf_div_2exp(jjm,jm,(unsigned long)j);
01933         if ( s == -1 && j%2 == 1 ) mpf_sub(sum,sum,jjm);
01934         else                       mpf_add(sum,sum,jjm);
01935     }
01936     mpf_clear(jjm); mpf_clear(jm);
01937 }
01938
01939 /*
01940         #] SimpleDelta :
01941         #[ SimpleDeltaC :
01942 */
01943
01944 void SimpleDeltaC(mpf_t sum, int m)
01945 {
01946     long s = 1;
01947     long prec = AC.DefaultPrecision;
01948     unsigned long xprec = (unsigned long)prec, x;
01949     int j, jmax, n;
01950     mpf_t jm,jjm;
01951     mpf_init(jm); mpf_init(jjm);
01952     if ( m < 0 ) { s = -1; m = -m; }
01953 /*
01954     We will loop until 1/2^j/j^m is smaller than the default precision.
01955     Just running to prec is however overkill, specially when m is large.
01956     We try to estimate a better value.
01957     jmax = prec - ((log2(prec)-1)*m).
01958     Hence we need the leading bit of prec.
01959     We are still overshooting a bit.
01960 */
01961     n = 0; x = xprec;
01962     while ( x ) { x >>= 1; n++; }
01963 /*
01964     We have now n = floor(log2(x))+1.
01965 */
01966     n--;
01967     jmax = (int)((int)xprec - (n-1)*m);
01968 /*
01969     For small prec and large m, the estimate can be wrong and even be negative,
01970     so we increase jmax until jmax + m*log2(jmax) > prec
01971 */
01972     if ( jmax < 0 ) jmax = 1;
01973     do {
01974         n = 0;
01975         x = (unsigned long)jmax;
01976         while (x) { x >>= 1; n++; }
01977         n--; // floor(log2(jmax))
01978         jmax++;
01979     } while ( jmax + m * n <= prec );
01980     if ( s < 0 ) jmax /= 2;
01981     mpf_set_si(sum,0L);
01982     for ( j = 1; j <= jmax; j++ ) {
01983 #ifdef WITHCUTOFF
01984         xprec--;
01985         mpf_set_prec_raw(jm,xprec);
01986         mpf_set_prec_raw(jjm,xprec);
01987 #endif
01988         mpf_set_ui(jm,1L);
01989         mpf_div_ui(jjm,jm,(unsigned long)j);
```

```
01990         mpf_pow_ui(jm,jjm,(unsigned long)m);
01991         if ( s == -1 ) mpf_div_2exp(jjm,jm,2*(unsigned long)j);
01992         else           mpf_div_2exp(jjm,jm,(unsigned long)j);
01993         mpf_add(sum,sum,jjm);
01994     }
01995     mpf_clear(jjm); mpf_clear(jm);
01996 }
01997
01998 /*
01999         #] SimpleDeltaC :
02000         #[ SingleTable :
02001 */
02002
02003 void SingleTable(mpf_t *tabl, int N, int m, int pow)
02004 {
02005 /*
02006     Creates a table T(1,...,N) with
02007         T(i) = sum_(j,i,N,[sign_(j)]/2^j/j^m)
02008     To make this table we have two options:
02009     1: run the sum backwards with the potential problem that the
02010        precision is difficult to manage.
02011     2: run the sum forwards. Take sum_(j,1,i-1,...) and later subtract.
02012     When doing Euler sums we may need also 1/4^j
02013     pow: 1 -> 1/2^j
02014          2 -> 1/4^j
02015 */
02016     GETIDENTITY
02017     long s = 1;
02018     int j;
02019 #ifdef WITHCUTOFF
02020     long prec = mpf_get_default_prec();
02021 #endif
02022     mpf_t jm,jjm;
02023     mpf_init(jm); mpf_init(jjm);
02024     if ( pow < 1 || pow > 2 ) {
02025 /* INTERNAL_ERROR_EXCL_START */
02026         MLOCK(ErrorMessageLock);
02027         MesPrint("!>Wrong parameter pow in SingleTable: %d\n",pow);
02028         MUNLOCK(ErrorMessageLock);
02029         Terminate(-1);
02030 /* INTERNAL_ERROR_EXCL_STOP */
02031     }
02032     if ( m < 0 ) { m = -m; s = -1; }
02033     mpf_set_si(auxsum,0L);
02034     for ( j = N; j > 0; j-- ) {
02035 #ifdef WITHCUTOFF
02036         mpf_set_prec_raw(jm,prec-j);
02037         mpf_set_prec_raw(jjm,prec-j);
02038 #endif
02039         mpf_set_ui(jm,1L);
02040         mpf_div_ui(jjm,jm,(unsigned long)j);
02041         mpf_pow_ui(jm,jjm,(unsigned long)m);
02042         mpf_div_2exp(jjm,jm,pow*(unsigned long)j);
02043         if ( pow == 2 ) mpf_add(auxsum,auxsum,jjm);
02044         else if ( s == -1 && j%2 == 1 ) mpf_sub(auxsum,auxsum,jjm);
02045         else                        mpf_add(auxsum,auxsum,jjm);
02046 /*
02047         And now copy auxsum to tablelement j
02048 */
02049         mpf_set(tabl[j],auxsum);
02050     }
02051     mpf_clear(jjm); mpf_clear(jm);
02052 }
02053
02054 /*
02055         #] SingleTable :
02056         #[ DoubleTable :
02057 */
02058
02059 void DoubleTable(mpf_t *tabout, mpf_t *tabin, int N, int m, int pow)
02060 {
02061     GETIDENTITY
02062     long s = 1;
02063 #ifdef WITHCUTOFF
02064     long prec = mpf_get_default_prec();
02065 #endif
02066     int j;
02067     mpf_t jm,jjm;
02068     mpf_init(jm); mpf_init(jjm);
02069     if ( pow < -1 || pow > 2 ) {
02070 /* INTERNAL_ERROR_EXCL_START */
02071         MLOCK(ErrorMessageLock);
02072         MesPrint("!>Wrong parameter pow in DoubleTable: %d\n",pow);
02073         MUNLOCK(ErrorMessageLock);
02074         Terminate(-1);
02075 /* INTERNAL_ERROR_EXCL_STOP */
02076     }
```

```
02077     if ( m < 0 ) { m = -m; s = -1; }
02078     mpf_set_ui(auxsum,0L);
02079     for ( j = N; j > 0; j-- ) {
02080 #ifdef WITHCUTOFF
02081         mpf_set_prec_raw(jm,prec-j);
02082         mpf_set_prec_raw(jjm,prec-j);
02083 #endif
02084         mpf_set_ui(jm,1L);
02085         mpf_div_ui(jjm,jm,(unsigned long)j);
02086         mpf_pow_ui(jm,jjm,(unsigned long)m);
02087         if ( pow == -1 ) {
02088             mpf_mul_2exp(jjm,jm,(unsigned long)j);
02089             mpf_mul(jm,jjm,tabin[j+1]);
02090         }
02091         else if ( pow > 0 ) {
02092             mpf_div_2exp(jjm,jm,pow*(unsigned long)j);
02093             mpf_mul(jm,jjm,tabin[j+1]);
02094         }
02095         else {
02096             mpf_mul(jm,jm,tabin[j+1]);
02097         }
02098         if ( pow == 2 ) mpf_add(auxsum,auxsum,jm);
02099         else if ( s == -1 && j%2 == 1 ) mpf_sub(auxsum,auxsum,jm);
02100         else                        mpf_add(auxsum,auxsum,jm);
02101 /*
02102         And now copy auxsum to tablelement j
02103 */
02104         mpf_set(tabout[j],auxsum);
02105     }
02106     mpf_clear(jjm); mpf_clear(jm);
02107 }
02108
02109 /*
02110         #] DoubleTable :
02111         #[ EndTable :
02112 */
02113
02114 void EndTable(mpf_t sum, mpf_t *tabin, int N, int m, int pow)
02115 {
02116     long s = 1;
02117 #ifdef WITHCUTOFF
02118     long prec = mpf_get_default_prec();
02119 #endif
02120     int j;
02121     mpf_t jm,jjm;
02122     mpf_init(jm); mpf_init(jjm);
02123     if ( pow < -1 || pow > 2 ) {
02124 /* INTERNAL_ERROR_EXCL_START */
02125         MLOCK(ErrorMessageLock);
02126         MesPrint("!>Wrong parameter pow in EndTable: %d\n",pow);
02127         MUNLOCK(ErrorMessageLock);
02128         Terminate(-1);
02129 /* INTERNAL_ERROR_EXCL_STOP */
02130     }
02131     if ( m < 0 ) { m = -m; s = -1; }
02132     mpf_set_si(sum,0L);
02133     for ( j = N; j > 0; j-- ) {
02134 #ifdef WITHCUTOFF
02135         mpf_set_prec_raw(jm,prec-j);
02136         mpf_set_prec_raw(jjm,prec-j);
02137 #endif
02138         mpf_set_ui(jm,1L);
02139         mpf_div_ui(jjm,jm,(unsigned long)j);
02140         mpf_pow_ui(jm,jjm,(unsigned long)m);
02141         if ( pow == -1 ) {
02142             mpf_mul_2exp(jjm,jm,(unsigned long)j);
02143             mpf_mul(jm,jjm,tabin[j+1]);
02144         }
02145         else if ( pow > 0 ) {
02146             mpf_div_2exp(jjm,jm,pow*(unsigned long)j);
02147             mpf_mul(jm,jjm,tabin[j+1]);
02148         }
02149         else {
02150             mpf_mul(jm,jm,tabin[j+1]);
02151         }
02152         if ( s == -1 && j%2 == 1 ) mpf_sub(sum,sum,jm);
02153         else                       mpf_add(sum,sum,jm);
02154     }
02155     mpf_clear(jjm); mpf_clear(jm);
02156 }
02157
02158 /*
02159         #] EndTable :
02160         #[ deltaMZV :
02161 */
02162
02163 void deltaMZV(mpf_t result, WORD *indexes, int depth)
```

```
02164 {
02165     GETIDENTITY
02166 /*
02167     Because all sums go roughly like 1/2^j we need about depth steps.
02168 */
02169 /*  MesPrint("deltaMZV(%a)",depth,indexes); */
02170     if ( depth == 1 ) {
02171         if ( indexes[0] == 1 ) {
02172             mpf_set(result,mpfdelta1);
02173             return;
02174         }
02175         SimpleDelta(result,indexes[0]);
02176     }
02177     else if ( depth == 2 ) {
02178         if ( indexes[0] == 1 && indexes[1] == 1 ) {
02179             mpf_pow_ui(result,mpfdelta1,2);
02180             mpf_div_2exp(result,result,1);
02181         }
02182         else {
02183             SingleTable(mpftab1,AC.DefaultPrecision-AC.MaxWeight+1,indexes[0],1);
02184             EndTable(result,mpftab1,AC.DefaultPrecision-AC.MaxWeight,indexes[1],0);
02185         };
02186     }
02187     else if ( depth > 2 ) {
02188         mpf_t *mpftab3;
02189         int d;
02190 /*
02191         Check first whether this is a power of delta1 = ln(2)
02192 */
02193         for ( d = 0; d < depth; d++ ) {
02194             if ( indexes[d] != 1 ) break;
02195         }
02196         if ( d == depth ) { /* divide by fac_(depth) */
02197             mpf_pow_ui(result,mpfdelta1,depth);
02198             for ( d = 2; d <= depth; d++ ) {
02199                 mpf_div_ui(result,result,(unsigned long)d);
02200             }
02201         }
02202         else {
02203             d = depth-1;
02204             SingleTable(mpftab1,AC.DefaultPrecision-AC.MaxWeight+d,*indexes,1);
02205             d--; indexes++;
02206             for(;;) {
02207                 DoubleTable(mpftab2,mpftab1,AC.DefaultPrecision-AC.MaxWeight+d,*indexes,0);
02208                 d--; indexes++;
02209                 if ( d == 0 ) {
02210                     EndTable(result,mpftab2,AC.DefaultPrecision-AC.MaxWeight,*indexes,0);
02211                     break;
02212                 }
02213                 mpftab3 = (mpf_t *)AT.mpf_tab1; AT.mpf_tab1 = AT.mpf_tab2;
02214                 AT.mpf_tab2 = (void *)mpftab3;
02215             }
02216         }
02217     }
02218     else if ( depth == 0 ) {
02219         mpf_set_ui(result,1L);
02220     }
02221 }
02222
02223 /*
02224         #] deltaMZV :
02225         #[ deltaEuler :
02226
02227         Regular Euler delta with - signs, but everywhere 1/2^j
02228 */
02229
02230 void deltaEuler(mpf_t result, WORD *indexes, int depth)
02231 {
02232     GETIDENTITY
02233     int m;
02234 #ifdef DEBUG
02235     int i;
02236     printf(" deltaEuler(");
02237     for ( i = 0; i < depth; i++ ) {
02238         if ( i != 0 ) printf(",");
02239         printf("%d",indexes[i]);
02240     }
02241     printf(") = ");
02242 #endif
02243     if ( depth == 1 ) {
02244         SimpleDelta(result,indexes[0]);
02245     }
02246     else if ( depth == 2 ) {
02247         SingleTable(mpftab1,AC.DefaultPrecision-AC.MaxWeight+1,indexes[0],1);
02248         m = indexes[1]; if ( indexes[0] < 0 ) m = -m;
02249         EndTable(result,mpftab1,AC.DefaultPrecision-AC.MaxWeight,m,0);
02250     }
```

```
02251     else if ( depth > 2 ) {
02252         int d;
02253         mpf_t *mpftab3;
02254         d = depth-1;
02255         SingleTable(mpftab1,AC.DefaultPrecision-AC.MaxWeight+d,*indexes,1);
02256         d--; indexes++;
02257         m = *indexes; if ( indexes[-1] < 0 ) m = -m;
02258         for(;;) {
02259             DoubleTable(mpftab2,mpftab1,AC.DefaultPrecision-AC.MaxWeight+d,m,0);
02260             d--; indexes++;
02261             m = *indexes; if ( indexes[-1] < 0 ) m = -m;
02262             if ( d == 0 ) {
02263                 EndTable(result,mpftab2,AC.DefaultPrecision-AC.MaxWeight,m,0);
02264                 break;
02265             }
02266             mpftab3 = (mpf_t *)AT.mpf_tab1; AT.mpf_tab1 = AT.mpf_tab2;
02267             AT.mpf_tab2 = (void *)mpftab3;
02268         }
02269     }
02270     else if ( depth == 0 ) {
02271         mpf_set_ui(result,1L);
02272     }
02273 #ifdef DEBUG
02274     gmp_printf("%.*Fe\n",40,result);
02275 #endif
02276 }
02277
02278 /*
02279         #] deltaEuler :
02280         #[ deltaEulerC :
02281
02282         Conjugate Euler delta without - signs, but possibly 1/4^j
02283         When there is a - in the string we have 1/4.
02284 */
02285
02286 void deltaEulerC(mpf_t result, WORD *indexes, int depth)
02287 {
02288     GETIDENTITY
02289     int m;
02290 #ifdef DEBUG
02291     int i;
02292     printf(" deltaEulerC(");
02293     for ( i = 0; i < depth; i++ ) {
02294         if ( i != 0 ) printf(",");
02295         printf("%d",indexes[i]);
02296     }
02297     printf(") = ");
02298 #endif
02299     mpf_set_ui(result,0);
02300     if ( depth == 1 ) {
02301         if ( indexes[0] == 0 ) {
02302             MLOCK(ErrorMessageLock);
02303             MesPrint("Illegal index in depth=1 deltaEulerC: %d\n",indexes[0]);
02304             MUNLOCK(ErrorMessageLock);
02305             Terminate(-1);
02306         }
02307         if ( indexes[0] < 0 ) SimpleDeltaC(result,indexes[0]);
02308         else                  SimpleDelta(result,indexes[0]);
02309     }
02310     else if ( depth == 2 ) {
02311         int par;
02312         m = indexes[0];
02313         if ( m < 0 ) SingleTable(mpftab1,AC.DefaultPrecision-AC.MaxWeight+depth,-m,2);
02314         else         SingleTable(mpftab1,AC.DefaultPrecision-AC.MaxWeight+depth, m,1);
02315         m = indexes[1];
02316         if ( m < 0 ) { m = -m; par = indexes[0] < 0 ? 0: 1; }
02317         else { par = indexes[0] < 0 ? -1: 0; }
02318         EndTable(result,mpftab1,AC.DefaultPrecision-AC.MaxWeight,m,par);
02319     }
02320     else if ( depth > 2 ) {
02321         int d, par;
02322         mpf_t *mpftab3;
02323         d = depth-1;
02324         m = indexes[0];
02325         ZeroTable(mpftab1,AC.DefaultPrecision+1);
02326         if ( m < 0 ) SingleTable(mpftab1,AC.DefaultPrecision-AC.MaxWeight+d+1,-m,2);
02327         else         SingleTable(mpftab1,AC.DefaultPrecision-AC.MaxWeight+d+1, m,1);
02328         d--; indexes++; m = indexes[0];
02329         if ( m < 0 ) { m = -m; par = indexes[-1] < 0 ? 0: 1; }
02330         else { par = indexes[-1] < 0 ? -1: 0; }
02331         for(;;) {
02332             ZeroTable(mpftab2,AC.DefaultPrecision-AC.MaxWeight+d);
02333             DoubleTable(mpftab2,mpftab1,AC.DefaultPrecision-AC.MaxWeight+d,m,par);
02334             d--; indexes++; m = indexes[0];
02335             if ( m < 0 ) { m = -m; par = indexes[-1] < 0 ? 0: 1; }
02336             else { par = indexes[-1] < 0 ? -1: 0; }
02337             if ( d == 0 ) {
```

```
02338                 mpf_set_ui(result,0);
02339                 EndTable(result,mpftab2,AC.DefaultPrecision-AC.MaxWeight,m,par);
02340                 break;
02341             }
02342             mpftab3 = (mpf_t *)AT.mpf_tab1; AT.mpf_tab1 = AT.mpf_tab2;
02343             AT.mpf_tab2 = (void *)mpftab3;
02344         }
02345     }
02346     else if ( depth == 0 ) {
02347         mpf_set_ui(result,1L);
02348     }
02349 #ifdef DEBUG
02350     gmp_printf("%.*Fe\n",40,result);
02351 #endif
02352 }
02353
02354 /*
02355         #] deltaEulerC :
02356         #[ CalculateMZVhalf :
02357
02358         This routine is mainly for support when large numbers of
02359         MZV's have to be calculated at the same time.
02360 */
02361
02362 void CalculateMZVhalf(mpf_t result, WORD *indexes, int depth)
02363 {
02364     int i;
02365     if ( depth < 0 ) {
02366         MLOCK(ErrorMessageLock);
02367         MesPrint("Illegal depth in CalculateMZVhalf: %d",depth);
02368         MUNLOCK(ErrorMessageLock);
02369         Terminate(-1);
02370     }
02371     for ( i = 0; i < depth; i++ ) {
02372         if ( indexes[i] <= 0 ) {
02373             MLOCK(ErrorMessageLock);
02374             MesPrint("Illegal index[%d] in CalculateMZVhalf: %d",i,indexes[i]);
02375             MUNLOCK(ErrorMessageLock);
02376             Terminate(-1);
02377         }
02378     }
02379     deltaMZV(result,indexes,depth);
02380 }
02381
02382 /*
02383         #] CalculateMZVhalf :
02384         #[ CalculateMZV :
02385 */
02386
02387 void CalculateMZV(mpf_t result, WORD *indexes, int depth)
02388 {
02389     GETIDENTITY
02390     int num1, num2 = 0, i, j = 0;
02391     if ( depth < 0 ) {
02392         MLOCK(ErrorMessageLock);
02393         MesPrint("Illegal depth in CalculateMZV: %d",depth);
02394         MUNLOCK(ErrorMessageLock);
02395         Terminate(-1);
02396     }
02397     if ( indexes[0] == 1 ) {
02398         MLOCK(ErrorMessageLock);
02399         MesPrint("Divergent MZV in CalculateMZV");
02400         MUNLOCK(ErrorMessageLock);
02401         Terminate(-1);
02402     }
02403 /*  MesPrint("calculateMZV(%a)",depth,indexes); */
02404     for ( i = 0; i < depth; i++ ) {
02405         if ( indexes[i] <= 0 ) {
02406             MLOCK(ErrorMessageLock);
02407             MesPrint("Illegal index[%d] in CalculateMZV: %d",i,indexes[i]);
02408             MUNLOCK(ErrorMessageLock);
02409             Terminate(-1);
02410         }
02411         AT.indi1[i] = indexes[i];
02412     }
02413     num1 = depth; i = 0;
02414     deltaMZV(result,indexes,depth);
02415     for(;;) {
02416         if ( AT.indi1[0] > 1 ) {
02417             AT.indi1[0] -= 1;
02418             for ( j = num2; j > 0; j-- ) AT.indi2[j] = AT.indi2[j-1];
02419             AT.indi2[0] = 1; num2++;
02420         }
02421         else {
02422             AT.indi2[0] += 1;
02423             for ( j = 1; j < num1; j++ ) AT.indi1[j-1] = AT.indi1[j];
02424             num1--;
```

```
02425             if ( num1 == 0 ) break;
02426         }
02427         deltaMZV(aux1,AT.indi1,num1);
02428         deltaMZV(aux2,AT.indi2,num2);
02429         mpf_mul(aux3,aux1,aux2);
02430         mpf_add(result,result,aux3);
02431     }
02432     deltaMZV(aux3,AT.indi2,num2);
02433     mpf_add(result,result,aux3);
02434 }
02435
02436 /*
02437         #] CalculateMZV :
02438         #[ CalculateEuler :
02439
02440         There is a problem of notation here.
02441         Basically there are two notations.
02442         1: Z(m1,m2,m3) = sum_(j3,1,inf,sign_(m3)/j3^abs_(m3)*
02443                          sum_(j2,j3+1,inf,sign_(m2)/j2^abs_(m2)*
02444                          sum_(j1,j2+1,inf,sign_(m1)/j1^abs_(m1))))
02445            See Broadhurst,1996
02446         2: L(m1,m2,m3) = sum_(j1,1,inf,sign_(m1)*
02447                          sum_(j2,1,inf,sign_(m2)*
02448                          sum_(j3,1,inf,sign_(m3)
02449                             /j3^abs_(m3)
02450                             /(j2+j3)^abs_(m2)
02451                             /(j1+j2+j3)^abs_(m1) )))
02452            See Borwein, Bradley, Broadhurst and Lisonek, 1999
02453         The L corresponds with the H of the datamine up to sign_(m1*m2*m3)
02454         The algorithm follows 2, but the more common notation is 1.
02455         Hence we start with a conversion.
02456 */
02457
02458 void CalculateEuler(mpf_t result, WORD *Zindexes, int depth)
02459 {
02460     GETIDENTITY
02461     int s1, num1, num2, i, j;
02462
02463     WORD *indexes = AT.WorkPointer;
02464     for ( i = 0; i < depth; i++ ) indexes[i] = Zindexes[i];
02465     for ( i = 0; i < depth-1; i++ ) {
02466         if ( Zindexes[i] < 0 ) {
02467             for ( j = i+1; j < depth; j++ ) indexes[j] = -indexes[j];
02468         }
02469     }
02470
02471     if ( depth < 0 ) {
02472         MLOCK(ErrorMessageLock);
02473         MesPrint("Illegal depth in CalculateEuler: %d\n",depth);
02474         MUNLOCK(ErrorMessageLock);
02475         Terminate(-1);
02476     }
02477     if ( indexes[0] == 1 ) {
02478         MLOCK(ErrorMessageLock);
02479         MesPrint("Divergent Euler sum in CalculateEuler\n");
02480         MUNLOCK(ErrorMessageLock);
02481         Terminate(-1);
02482     }
02483     for ( i = 0, j = 0; i < depth; i++ ) {
02484         if ( indexes[i] == 0 ) {
02485             MLOCK(ErrorMessageLock);
02486             MesPrint("Illegal index[%d] in CalculateEuler: %d\n",i,indexes[i]);
02487             MUNLOCK(ErrorMessageLock);
02488             Terminate(-1);
02489         }
02490         if ( indexes[i] < 0 ) j = 1;
02491         AT.indi1[i] = indexes[i];
02492     }
02493     if ( j == 0 ) {
02494         CalculateMZV(result,indexes,depth);
02495         return;
02496     }
02497     num1 = depth; AT.indi2[0] = 0; num2 = 0;
02498     deltaEuler(result,AT.indi1,depth);
02499     s1 = 0;
02500     for(;;) {
02501         s1++;
02502         if ( AT.indi1[0] > 1 ) {
02503             AT.indi1[0] -= 1;
02504             for ( j = num2; j > 0; j-- ) AT.indi2[j] = AT.indi2[j-1];
02505             AT.indi2[0] = 1; num2++;
02506         }
02507         else if ( AT.indi1[0] < -1 ) {
02508             AT.indi1[0] += 1;
02509             for ( j = num2; j > 0; j-- ) AT.indi2[j] = AT.indi2[j-1];
02510             AT.indi2[0] = 1; num2++;
02511         }
```

```
02512         else if ( AT.indi1[0] == -1 ) {
02513             for ( j = num2; j > 0; j-- ) AT.indi2[j] = AT.indi2[j-1];
02514             AT.indi2[0] = -1; num2++;
02515             for ( j = 1; j < num1; j++ ) AT.indi1[j-1] = AT.indi1[j];
02516             num1--;
02517         }
02518         else {
02519             AT.indi2[0] = AT.indi2[0] < 0 ? AT.indi2[0]-1: AT.indi2[0]+1;
02520             for ( j = 1; j < num1; j++ ) AT.indi1[j-1] = AT.indi1[j];
02521             num1--;
02522         }
02523         if ( num1 == 0 ) break;
02524         deltaEuler(aux1,AT.indi1,num1);
02525         deltaEulerC(aux2,AT.indi2,num2);
02526         mpf_mul(aux3,aux1,aux2);
02527         if ( (depth+num1+num2+s1)%2 == 0 ) mpf_add(result,result,aux3);
02528         else                               mpf_sub(result,result,aux3);
02529     }
02530     deltaEulerC(aux3,AT.indi2,num2);
02531     if ( (depth+num2+s1)%2 == 0 ) mpf_add(result,result,aux3);
02532     else                          mpf_sub(result,result,aux3);
02533 }
02534
02535 /*
02536         #] CalculateEuler :
02537         #[ ExpandMZV :
02538 */
02539 /* UNFINISHED_FEATURE_EXCL_START */
02540 int ExpandMZV(WORD *term, WORD level)
02541 {
02542     DUMMYUSE(term);
02543     DUMMYUSE(level);
02544     return(0);
02545 }
02546 /* UNFINISHED_FEATURE_EXCL_STOP */
02547 /*
02548         #] ExpandMZV :
02549         #[ ExpandEuler :
02550 */
02551 /* UNFINISHED_FEATURE_EXCL_START */
02552 int ExpandEuler(WORD *term, WORD level)
02553 {
02554     DUMMYUSE(term);
02555     DUMMYUSE(level);
02556     return(0);
02557 }
02558 /* UNFINISHED_FEATURE_EXCL_STOP */
02559 /*
02560         #] ExpandEuler :
02561         #[ EvaluateEuler :
02562
02563     There are various steps here:
02564     1: locate an Euler function.
02565     2: evaluate it into a float.
02566     3: convert the coefficient to a float and multiply.
02567     4: Put the new term together.
02568     5: Restart to see whether there are more Euler functions.
02569     The compiler should check that there is no conflict between
02570     the evaluate command and a potential polyfun or polyratfun.
02571     Yet, when reading in expressions there can be a conflict.
02572     Hence the Normalize routine should check as well.
02573
02574     par == MZV: MZV
02575     par == EULER: Euler
02576     par == MZVHALF: MZV sums in 1/2 rather than 1. -> deltaMZV.
02577     par == ALLMZVFUNCTIONS: all of the above.
02578 */
02579
02580 int EvaluateEuler(PHEAD WORD *term, WORD level, WORD par)
02581 {
02582     WORD *t, *tstop, *tt, *tnext, sumweight, depth, first = 1, *newterm, i;
02583     WORD *oldworkpointer = AT.WorkPointer, nsize, *indexes;
02584     int retval;
02585     tstop = term + *term; tstop -= ABS(tstop[-1]);
02586     if ( AT.WorkPointer < term+*term ) AT.WorkPointer = term + *term;
02587 /*
02588     Step 1. We also need to verify that the argument we find is legal
02589 */
02590     t = term+1;
02591     while ( t < tstop ) {
02592         if ( ( *t == par ) || ( ( par == ALLMZVFUNCTIONS ) && (
02593             *t == MZV || *t == EULER || *t == MZVHALF ) ) ) {
02594     /* all arguments should be small integers */
02595             indexes = AT.WorkPointer;
02596             sumweight = 0; depth = 0;
02597             tnext = t + t[1]; tt = t + FUNHEAD;
02598             while ( tt < tnext ) {
```

```
02599                 if ( *tt != -SNUMBER ) goto nextfun;
02600                 if ( tt[1] == 0 ) goto nextfun;
02601                 indexes[depth] = tt[1];
02602                 sumweight += ABS(tt[1]); depth++;
02603                 tt += 2;
02604             }
02605             /* euler sum without arguments, i.e. mzv_(), euler_() or mzvhalf_() */
02606             if ( depth == 0) goto nextfun;
02607             if ( sumweight > AC.MaxWeight ) {
02608                 MLOCK(ErrorMessageLock);
02609                 MesPrint("Error: Weight of Euler/MZV sum greater than %d.",AC.MaxWeight);
02610                 MesPrint("Please increase the maximum weight in %#startfloat.");
02611                 MUNLOCK(ErrorMessageLock);
02612                 Terminate(-1);
02613             }
02614 /*
02615             Step 2: evaluate:
02616 */
02617             AT.WorkPointer += depth;
02618             if ( first ) {
02619                 if ( *t == MZV ) CalculateMZV(aux4,indexes,depth);
02620                 else if ( *t == EULER ) CalculateEuler(aux4,indexes,depth);
02621                 else if ( *t == MZVHALF ) CalculateMZVhalf(aux4,indexes,depth);
02622                 first = 0;
02623             }
02624             else {
02625                 if ( *t == MZV ) CalculateMZV(aux5,indexes,depth);
02626                 else if ( *t == EULER ) CalculateEuler(aux5,indexes,depth);
02627                 else if ( *t == MZVHALF ) CalculateMZVhalf(aux5,indexes,depth);
02628                 mpf_mul(aux4,aux4,aux5);
02629             }
02630             *t = 0;
02631         }
02632 nextfun:
02633         t += t[1];
02634     }
02635     if ( first == 1 ) return(Generator(BHEAD term,level));
02636 /*
02637     Step 3:
02638     Now the regular coefficient, if it is not 1/1.
02639     We have two cases: size +- 3, or bigger.
02640 */
02641     nsize = term[*term-1];
02642     if ( nsize < 0 ) {
02643         mpf_neg(aux4,aux4);
02644         nsize = -nsize;
02645     }
02646     if ( nsize == 3 ) {
02647         if ( tstop[0] != 1 ) {
02648             mpf_mul_ui(aux4,aux4,(ULONG)((UWORD)tstop[0]));
02649         }
02650         if ( tstop[1] != 1 ) {
02651             mpf_div_ui(aux4,aux4,(ULONG)((UWORD)tstop[1]));
02652         }
02653     }
02654     else {
02655         RatToFloat(aux5,(UWORD *)tstop,nsize);
02656         mpf_mul(aux4,aux4,aux5);
02657     }
02658 /*
02659     Now we have to locate possible other float_ functions.
02660 */
02661     t = term+1;
02662     while ( t < tstop ) {
02663         if ( *t == FLOATFUN ) {
02664             UnpackFloat(aux5,t);
02665             mpf_mul(aux4,aux4,aux5);
02666         }
02667         t += t[1];
02668     }
02669 /*
02670     Now we should compose the new term in the WorkSpace.
02671 */
02672     t = term+1;
02673     newterm = AT.WorkPointer;
02674     tt = newterm+1;
02675     while ( t < tstop ) {
02676         if ( *t == 0 || *t == FLOATFUN ) t += t[1];
02677         else {
02678             i = t[1]; NCOPY(tt,t,i);
02679         }
02680     }
02681     PackFloat(tt,aux4);
02682     tt += tt[1];
02683     *tt++ = 1; *tt++ = 1; *tt++ = 3;
02684     *newterm = tt-newterm;
02685     AT.WorkPointer = tt;
```

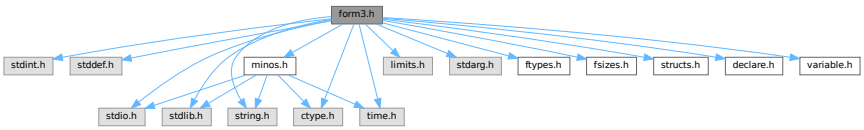
```
02686     retval = Generator(BHEAD newterm,level);
02687     AT.WorkPointer = oldworkpointer;
02688     return(retval);
02689 }
02690 
02691 /*
02692         #] EvaluateEuler :
02693     #] MZV :
02694     #[ Functions :
02695         #[ CoEvaluate :
02696 
02697         Current subkeys: mzv_, euler_ and sqrt_.
02698 */
02699 
02700 int CoEvaluate(UBYTE *s)
02701 {
02702     GETIDENTITY
02703     UBYTE *subkey, c;
02704     WORD numfun, type;
02705     int error = 0;
02706     if ( AT.aux_ == 0 ) {
02707         MesPrint("&Illegal attempt to evaluate a function without activating floating point
      numbers.");
02708         MesPrint("&Forgotten %#startfloat instruction?");
02709         return(1);
02710     }
02711     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
02712     if ( *s == 0 ) {
02713 /*
02714         No subkey, which means all functions.
02715 */
02716         Add3Com(TYPEEVALUATE,ALLFUNCTIONS);
02717 /*
02718         The MZV, EULER and MZVHALF are done separately
02719 */
02720         Add3Com(TYPEEVALUATE,ALLMZVFUNCTIONS);
02721         return(0);
02722     }
02723     while ( *s ) {
02724     subkey = s;
02725     while ( FG.cTable[*s] == 0 ) s++;
02726       if ( *s == '2' ) s++; /* cases li2_ and atan2_ */
02727       if ( *s == '_' ) s++;
02728       c = *s; *s = 0;
02729 /*
02730         We still need provisions for pi_ and possibly other constants.
02731 */
02732       if ( ( ( type = GetName(AC.varnames,subkey,&numfun,NOAUTO) ) != CFUNCTION )
02733             || ( functions[numfun].spec != 0 ) ) {
02734 
02735         if ( type == CSYMBOL ) {
02736             Add4Com(TYPEEVALUATE,SYMBOL,numfun);
02737             break;
02738         }
02739 /*
02740             This cannot work.
02741 */
02742         MesPrint("&%s should be a built in function that can be evaluated numerically.",s);
02743         return(1);
02744       }
02745       else {
02746         switch ( numfun+FUNCTION ) {
02747             case MZV:
02748             case EULER:
02749             case MZVHALF:
02750 /*
02751             The following functions are treated in evaluate.c
02752 */
02753             case SQRTFUNCTION:
02754             case LNFUNCTION:
02755             case EXPFUNCTION:
02756             case SINFUNCTION:
02757             case COSFUNCTION:
02758             case TANFUNCTION:
02759             case ASINFUNCTION:
02760             case ACOSFUNCTION:
02761             case ATANFUNCTION:
02762             case ATAN2FUNCTION:
02763             case SINHFUNCTION:
02764             case COSHFUNCTION:
02765             case TANHFUNCTION:
02766             case ASINHFUNCTION:
02767             case ACOSHFUNCTION:
02768             case ATANHFUNCTION:
02769             case LI2FUNCTION:
02770             case LINFUNCTION:
02771             case AGMFUNCTION:
```

```
02772             case GAMMAFUN:
02773 /*
02774             At a later stage we can add more functions from mpfr here
02775                 mpfr_(number,arg(s))
02776 */
02777                 Add3Com(TYPEEVALUATE,numfun+FUNCTION);
02778                 break;
02779             default:
02780                 MesPrint("&%s should be a built in function that can be evaluated numerically.",s);
02781                 error = 1;
02782                 break;
02783           }
02784       }
02785       *s = c;
02786       while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
02787     }
02788     return(error);
02789 }
02790
02791 /*
02792         #] CoEvaluate :
02793     #] Functions :
02794 */
```

4.49 form3.h File Reference

```
#include <stdint.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <limits.h>
#include <stdarg.h>
#include <time.h>
#include "ftypes.h"
#include "fsizes.h"
#include "minos.h"
#include "structs.h"
#include "declare.h"
#include "variable.h"
```

Include dependency graph for form3.h:



Macros

- #define MAJORVERSION X
- #define MINORVERSION Y
- #define PATCHVERSION Z
- #define PRODUCTIONDATE "DD-MM-YYYY"
- #define inline
- #define NDEBUG
- #define STATIC_ASSERT(condition) STATIC_ASSERT__1(condition,__LINE__)
- #define STATIC_ASSERT__1(X, L) STATIC_ASSERT__2(X,L)

- #define STATIC_ASSERT__2(X, L) STATIC_ASSERT__3(X,L)
- #define STATIC_ASSERT__3(X, L) typedef char static_assertion_failed_##L[(!!(X))*2-1]
- #define BITSINWORD 32
- #define BITSINLONG 64
- #define WORD_MIN_VALUE INT32_MIN
- #define WORD_MAX_VALUE INT32_MAX
- #define LONG_MIN_VALUE INT64_MIN
- #define LONG_MAX_VALUE INT64_MAX
- #define TOPBITONLY ((ULONG)1 << (BITSINWORD - 1)) /* 0x00008000UL */
- #define TOPLONGBITONLY ((ULONG)1 << (BITSINLONG - 1)) /* 0x80000000UL */
- #define SPECMASK ((UWORD)1 << (BITSINWORD - 1)) /* 0x8000U */
- #define WILDMASK ((UWORD)1 << (BITSINWORD - 2)) /* 0x4000U */
- #define WORDMASK ((ULONG)FULLMAX - 1) /* 0x0000FFFFUL */
- #define AWORDMASK (WORDMASK << BITSINWORD) /* 0xFFFF0000UL */
- #define FULLMAX ((LONG)1 << BITSINWORD) /* 0x00010000L */
- #define MAXPOSITIVE ((LONG)(TOPBITONLY - 1)) /* 0x00007FFFL */
- #define MAXLONG ((LONG)(TOPLONGBITONLY - 1)) /* 0x7FFFFFFFL */
- #define MAXPOSITIVE2 (MAXPOSITIVE / 2) /* 0x00003FFFL */
- #define MAXPOSITIVE4 (MAXPOSITIVE / 4) /* 0x00001FFFL */
- #define form_alignof(type) offsetof(struct { char c_; type x_; }, x_)
- #define WITHSORTBOTS
- #define FILES FILE
- #define Uopen(x, y) fopen(x,y)
- #define Uflush(x) fflush(x)
- #define Uclose(x) fclose(x)
- #define Uread(x, y, z, u) fread(x,y,z,u)
- #define Uwrite(x, y, z, u) fwrite(x,y,z,u)
- #define Usetbuf(x, y) setbuf(x,y)
- #define Useek(x, y, z) fseek(x,y,z)
- #define Utell(x) ftell(x)
- #define Ugetpos(x, y) fgetpos(x,y)
- #define Usetpos(x, y) fsetpos(x,y)
- #define Usync(x) fflush(x)
- #define Utruncate(x) _chsize(_fileno(x),0)
- #define Ustdout stdout
- #define MAX_OPEN_FILES FOPEN_MAX
- #define bzero(b, len) (memset((b), 0, (len)), (void)0)
- #define GetPID() ((LONG)GetCurrentProcessId())

Typedefs

- typedef int32_t WORD
- typedef int64_t LONG
- typedef uint32_t UWORD
- typedef uint64_t ULONG
- typedef signed char SBYTE
- typedef unsigned char UBYTE
- typedef unsigned int UINT
- typedef ULONG RLONG
- typedef int64_t MLONG
- typedef char BOOL

Functions

- **STATIC_ASSERT** (sizeof(WORD) ∗8==BITSINWORD)
- **STATIC_ASSERT** (sizeof(LONG) ∗8==BITSINLONG)
- **STATIC_ASSERT** (sizeof(LONG) >=sizeof(int ∗))
- **STATIC_ASSERT** (sizeof(int ∗) >=sizeof(int))
- **STATIC_ASSERT** (sizeof(int) >=sizeof(WORD))
- **STATIC_ASSERT** (sizeof(WORD) >=sizeof(char))
- **STATIC_ASSERT** (sizeof(char)==1)
- **STATIC_ASSERT** (sizeof(off_t) >=sizeof(LONG))

4.49.1 Detailed Description

Contains critical defines for the compilation process Also contains the inclusion of all necessary header files. There are also some system dependencies concerning file functions.

Definition in file form3.h.

4.49.2 Macro Definition Documentation

4.49.2.1 MAJORVERSION

```
#define MAJORVERSION X
```

Definition at line 48 of file form3.h.

4.49.2.2 MINORVERSION

```
#define MINORVERSION Y
```

Definition at line 49 of file form3.h.

4.49.2.3 PATCHVERSION

```
#define PATCHVERSION Z
```

Definition at line 50 of file form3.h.

4.49.2.4 PRODUCTIONDATE

```
#define PRODUCTIONDATE "DD-MM-YYYY"
```

Definition at line 55 of file form3.h.

4.49.2.5 inline

```
#define inline
```

Definition at line 122 of file form3.h.

4.49.2.6 NDEBUG

```
#define NDEBUG
```

Definition at line 149 of file form3.h.

4.49.2.7 STATIC_ASSERT

```
#define STATIC_ASSERT(
              condition ) STATIC_ASSERT__1(condition,__LINE__)
```

Definition at line 160 of file form3.h.

4.49.2.8 STATIC_ASSERT__1

```
#define STATIC_ASSERT__1(
              X,
              L ) STATIC_ASSERT__2(X,L)
```

Definition at line 161 of file form3.h.

4.49.2.9 STATIC_ASSERT__2

```
#define STATIC_ASSERT__2(
              X,
              L ) STATIC_ASSERT__3(X,L)
```

Definition at line 162 of file form3.h.

4.49.2.10 STATIC_ASSERT__3

```
#define STATIC_ASSERT__3(
              X,
              L )  typedef char static_assertion_failed_##L[(!!(X))*2-1]
```

Definition at line 163 of file form3.h.

4.49.2.11 BITSINWORD

```
#define BITSINWORD 32
```

Definition at line 202 of file form3.h.

4.49.2.12 BITSINLONG

```
#define BITSINLONG 64
```

Definition at line 203 of file form3.h.

4.49.2.13 WORD_MIN_VALUE

```
#define WORD_MIN_VALUE INT32_MIN
```

Definition at line 204 of file form3.h.

4.49.2.14 WORD_MAX_VALUE

```
#define WORD_MAX_VALUE INT32_MAX
```

Definition at line 205 of file form3.h.

4.49.2.15 LONG_MIN_VALUE

```
#define LONG_MIN_VALUE INT64_MIN
```

Definition at line 206 of file form3.h.

4.49.2.16 LONG_MAX_VALUE

```
#define LONG_MAX_VALUE INT64_MAX
```

Definition at line 207 of file form3.h.

4.49.2.17 TOPBITONLY

```
#define TOPBITONLY ((ULONG)1 << (BITSINWORD - 1)) /* 0x00008000UL */
```

Definition at line 244 of file form3.h.

4.49.2.18 TOPLONGBITONLY

```
#define TOPLONGBITONLY ((ULONG)1 << (BITSINLONG - 1)) /* 0x80000000UL */
```

Definition at line 245 of file form3.h.

4.49.2.19 SPECMASK

```
#define SPECMASK ((UWORD)1 << (BITSINWORD - 1)) /* 0x8000U */
```

Definition at line 246 of file form3.h.

4.49.2.20 WILDMASK

```
#define WILDMASK ((UWORD)1 << (BITSINWORD - 2)) /* 0x4000U */
```

Definition at line 247 of file form3.h.

4.49.2.21 WORDMASK

```
#define WORDMASK ((ULONG)FULLMAX - 1) /* 0x0000FFFFUL */
```

Definition at line 248 of file form3.h.

4.49.2.22 AWORDMASK

```
#define AWORDMASK (WORDMASK << BITSINWORD) /* 0xFFFF0000UL */
```

Definition at line 249 of file form3.h.

4.49.2.23 FULLMAX

```
#define FULLMAX ((LONG)1 << BITSINWORD) /* 0x00010000L */
```

Definition at line 250 of file form3.h.

4.49.2.24 MAXPOSITIVE

```
#define MAXPOSITIVE ((LONG)(TOPBITONLY - 1)) /* 0x00007FFFL */
```

Definition at line 251 of file form3.h.

4.49.2.25 MAXLONG

```
#define MAXLONG ((LONG)(TOPLONGBITONLY - 1)) /* 0x7FFFFFFFL */
```

Definition at line 252 of file form3.h.

4.49.2.26 MAXPOSITIVE2

```
#define MAXPOSITIVE2 (MAXPOSITIVE / 2) /* 0x00003FFFL */
```

Definition at line 253 of file form3.h.

4.49.2.27 MAXPOSITIVE4

```
#define MAXPOSITIVE4 (MAXPOSITIVE / 4) /* 0x00001FFFL */
```

Definition at line 254 of file form3.h.

4.49.2.28 form_alignof

```
#define form_alignof(
              type ) offsetof(struct { char c_; type x_; }, x_)
```

Definition at line 270 of file form3.h.

4.49.2.29 WITHSORTBOTS

```
#define WITHSORTBOTS
```

Definition at line 290 of file form3.h.

4.49.2.30 FILES

```
#define FILES FILE
```

Definition at line 371 of file form3.h.

4.49.2.31 Uopen

```
#define Uopen(
              x,
              y ) fopen(x,y)
```

Definition at line 372 of file form3.h.

4.49.2.32 Uflush

```
#define Uflush(
              x ) fflush(x)
```

Definition at line 373 of file form3.h.

4.49.2.33 Uclose

```
#define Uclose(
              x ) fclose(x)
```

Definition at line 374 of file form3.h.

4.49.2.34 Uread

```
#define Uread(
              x,
              y,
              z,
              u ) fread(x,y,z,u)
```

Definition at line 375 of file form3.h.

4.49.2.35 Uwrite

```
#define Uwrite(
              x,
              y,
              z,
              u ) fwrite(x,y,z,u)
```

Definition at line 376 of file form3.h.

4.49.2.36 Usetbuf

```
#define Usetbuf(
              x,
              y ) setbuf(x,y)
```

Definition at line 377 of file form3.h.

4.49.2.37 Useek

```
#define Useek(
              x,
              y,
              z ) fseek(x,y,z)
```

Definition at line 378 of file form3.h.

4.49.2.38 Utell

```
#define Utell(
              x ) ftell(x)
```

Definition at line 379 of file form3.h.

4.49.2.39 Ugetpos

```
#define Ugetpos(
              x,
              y ) fgetpos(x,y)
```

Definition at line 380 of file form3.h.

4.49.2.40 Usetpos

```
#define Usetpos(
              x,
              y ) fsetpos(x,y)
```

Definition at line 381 of file form3.h.

4.49.2.41 Usync

```
#define Usync(
              x ) fflush(x)
```

Definition at line 382 of file form3.h.

4.49.2.42 Utruncate

```
#define Utruncate(
              x ) _chsize(_fileno(x),0)
```

Definition at line 383 of file form3.h.

4.49.2.43 Ustdout

```
#define Ustdout stdout
```

Definition at line 384 of file form3.h.

4.49.2.44 MAX_OPEN_FILES

```
#define MAX_OPEN_FILES FOPEN_MAX
```

Definition at line 385 of file form3.h.

4.49.2.45 bzero

```
#define bzero(
              b,
              len ) (memset((b), 0, (len)), (void)0)
```

Definition at line 386 of file form3.h.

4.49.2.46 GetPID

```
#define GetPID( ) ((LONG)GetCurrentProcessId())
```

Definition at line 387 of file form3.h.

4.49.3 Typedef Documentation

4.49.3.1 WORD

```
typedef int32_t WORD
```

Definition at line 198 of file form3.h.

4.49.3.2 LONG

```
typedef int64_t LONG
```

Definition at line 199 of file form3.h.

4.49.3.3 UWORD

```
typedef uint32_t UWORD
```

Definition at line 200 of file form3.h.

4.49.3.4 ULONG

```
typedef uint64_t ULONG
```

Definition at line 201 of file form3.h.

4.49.3.5 SBYTE

```
typedef signed char SBYTE
```

Definition at line 233 of file form3.h.

4.49.3.6 UBYTE

```
typedef unsigned char UBYTE
```

Definition at line 234 of file form3.h.

4.49.3.7 UINT

```
typedef unsigned int UINT
```

Definition at line 235 of file form3.h.

4.49.3.8 RLONG

```
typedef ULONG RLONG
```

Definition at line 236 of file form3.h.

4.49.3.9 MLONG

```
typedef int64_t MLONG
```

Definition at line 237 of file form3.h.

4.49.3.10 BOOL

```
typedef char BOOL
```

Definition at line 242 of file form3.h.

4.50 form3.h

Go to the documentation of this file.

```
00001
00008 /* #[ License : */
00009 /*
00010  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00011  *   When using this file you are requested to refer to the publication
00012  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013  *   This is considered a matter of courtesy as the development was paid
00014  *   for by FOM the Dutch physics granting agency and we would like to
00015  *   be able to track its scientific use to convince FOM of its value
00016  *   for the community.
00017  *
00018  *   This file is part of FORM.
00019  *
00020  *   FORM is free software: you can redistribute it and/or modify it under the
00021  *   terms of the GNU General Public License as published by the Free Software
00022  *   Foundation, either version 3 of the License, or (at your option) any later
00023  *   version.
00024  *
00025  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00027  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00028  *   details.
00029  *
00030  *   You should have received a copy of the GNU General Public License along
00031  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00032  */
00033 /* #] License : */
00034
00035 #ifndef __FORM3H__
00036 #define __FORM3H__
00037
00038 #ifdef HAVE_CONFIG_H
00039
00040 #ifndef CONFIG_H_INCLUDED
00041 #define CONFIG_H_INCLUDED
00042 #include <config.h>
00043 #endif
00044
00045 #else  /* HAVE_CONFIG_H */
00046
00047 // This information is now only determined by scripts/git-version-gen.sh
00048 #define MAJORVERSION X
00049 #define MINORVERSION Y
00050 #define PATCHVERSION Z
00051
00052 #ifdef __DATE__
00053 #define PRODUCTIONDATE __DATE__
00054 #else
00055 #define PRODUCTIONDATE "DD-MM-YYYY"
00056 #endif
00057
00058 #undef BETAVERSION
00059 /*#define BETAVERSION*/
00060
00061 #ifdef LINUX32
00062 #define UNIX
00063 #define LINUX
00064 #define _FILE_OFFSET_BITS 64
00065 #define WITHZLIB
00066 #define WITHGMP
00067 #define WITHPOSIXCLOCK
00068 #endif
00069
00070 #ifdef LINUX64
00071 #define UNIX
00072 #define LINUX
00073 #define WITHZLIB
00074 #define WITHGMP
00075 #define WITHPOSIXCLOCK
```

```
00076 #define WITHFLOAT
00077 #endif
00078
00079 #ifdef APPLE32
00080 #define UNIX
00081 #define _FILE_OFFSET_BITS 64
00082 #define WITHZLIB
00083 #endif
00084
00085 #ifdef APPLE64
00086 #define UNIX
00087 #define WITHZLIB
00088 #define WITHGMP
00089 #define WITHPOSIXCLOCK
00090 #define WITHFLOAT
00091 #define HAVE_UNORDERED_MAP
00092 #define HAVE_UNORDERED_SET
00093 #endif
00094
00095 #ifdef CYGWIN32
00096 #define UNIX
00097 #endif
00098
00099 #ifdef _MSC_VER
00100 #define WINDOWS
00101 #define _CRT_SECURE_NO_WARNINGS
00102 #endif
00103
00104 /*
00105  * We must not define WITHPOSIXCLOCK in compiling the sequential FORM or ParFORM.
00106  */
00107 #if !defined(WITHPTHREADS) && defined(WITHPOSIXCLOCK)
00108 #undef WITHPOSIXCLOCK
00109 #endif
00110
00111 #if !defined(__cplusplus) && !defined(inline)
00112 #if (defined(__STDC_VERSION__) && __STDC_VERSION__ >= 199901L) || (defined(__GNUC__) &&
      !defined(__STRICT_ANSI__))
00113 /* "inline" is available. */
00114 #elif defined(__GNUC__)
00115 /* GNU C compiler has "__inline__". */
00116 #define inline __inline__
00117 #elif defined(_MSC_VER)
00118 /* Microsoft C compiler has "__inline". */
00119 #define inline __inline
00120 #else
00121 /* Inline functions may be not supported. Define "inline" to be empty. */
00122 #define inline
00123 #endif
00124 #endif
00125
00126 #endif  /* HAVE_CONFIG_H */
00127
00128 /* Workaround for MSVC. */
00129 #if defined(_MSC_VER)
00130 /*
00131  * Old versions of MSVC didn't support C99 function `snprintf', which is used
00132  * in poly.cc. On the other hand, macroizing `snprintf' gives a fatal error
00133  * with MSVC >= 2015.
00134  */
00135 #if _MSC_VER < 1900
00136 #define snprintf _snprintf
00137 #endif
00138 #endif
00139
00140 /*
00141  * Translate our dialect "DEBUGGING" to the standard "NDEBUG".
00142  */
00143 #ifdef DEBUGGING
00144 #ifdef NDEBUG
00145 #undef NDEBUG
00146 #endif
00147 #else
00148 #ifndef NDEBUG
00149 #define NDEBUG
00150 #endif
00151 #endif
00152
00153 /*
00154  * STATIC_ASSERT(condition) will fail to be compiled if the given
00155  * condition is false.
00156  */
00157 /*
00158 #define STATIC_ASSERT(condition)
00159 */
00160 #define STATIC_ASSERT(condition) STATIC_ASSERT__1(condition,__LINE__)
00161 #define STATIC_ASSERT__1(X,L) STATIC_ASSERT__2(X,L)
```

```
00162 #define STATIC_ASSERT__2(X,L) STATIC_ASSERT__3(X,L)
00163 #define STATIC_ASSERT__3(X,L) \
00164     typedef char static_assertion_failed_##L[(!!(X))*2-1]
00165
00166 /*
00167  * UNIX or WINDOWS must be defined.
00168  */
00169 #if defined(UNIX)
00170 #define mBSD
00171 #define ANSI
00172 #elif defined(WINDOWS)
00173 #define ANSI
00174 #define WIN32_LEAN_AND_MEAN
00175 #include <windows.h>
00176 #include <io.h>
00177 #include <fcntl.h>
00178 /* Undefine/rename conflicted symbols. */
00179 #undef MAXLONG  /* WinNT.h */
00180 #define WORD FORM_WORD  /* WinDef.h */
00181 #define LONG FORM_LONG /* WinNT.h */
00182 #define ULONG FORM_ULONG  /* WinDef.h */
00183 #define BOOL FORM_BOOL  /* WinDef.h */
00184 #undef CreateFile  /* WinBase.h */
00185 #undef CopyFile  /* WinBase.h */
00186 #define OpenFile FORM_OpenFile  /* WinBase.h */
00187 #define ReOpenFile FORM_ReOpenFile  /* WinBase.h */
00188 #define ReadFile FORM_ReadFile  /* WinBase.h */
00189 #define WriteFile FORM_WriteFile  /* WinBase.h */
00190 #define DeleteObject FORM_DeleteObject  /* WinGDI.h */
00191 #else
00192 #error UNIX or WINDOWS must be defined!
00193 #endif
00194
00195 #include <stdint.h>
00196
00197 #if UINTPTR_MAX == UINT64_MAX
00198     typedef int32_t WORD;
00199     typedef int64_t LONG;
00200     typedef uint32_t UWORD;
00201     typedef uint64_t ULONG;
00202     #define BITSINWORD 32
00203     #define BITSINLONG 64
00204     #define WORD_MIN_VALUE INT32_MIN
00205     #define WORD_MAX_VALUE INT32_MAX
00206     #define LONG_MIN_VALUE INT64_MIN
00207     #define LONG_MAX_VALUE INT64_MAX
00208 #elif UINTPTR_MAX == UINT32_MAX
00209     typedef int16_t WORD;
00210     typedef int32_t LONG;
00211     typedef uint16_t UWORD;
00212     typedef uint32_t ULONG;
00213     #define BITSINWORD 16
00214     #define BITSINLONG 32
00215     #define WORD_MIN_VALUE INT16_MIN
00216     #define WORD_MAX_VALUE INT16_MAX
00217     #define LONG_MIN_VALUE INT32_MIN
00218     #define LONG_MAX_VALUE INT32_MAX
00219 #else
00220     #error Can not detect if this is a 32-bit or 64-bit platform.
00221 #endif
00222
00223 STATIC_ASSERT(sizeof(WORD) * 8 == BITSINWORD);
00224 STATIC_ASSERT(sizeof(LONG) * 8 == BITSINLONG);
00225 STATIC_ASSERT(sizeof(WORD) * 2 == sizeof(LONG));
00226 STATIC_ASSERT(sizeof(LONG) >= sizeof(int *));
00227 STATIC_ASSERT(sizeof(LONG) >= sizeof(int *));
00228 STATIC_ASSERT(sizeof(int *) >= sizeof(int));
00229 STATIC_ASSERT(sizeof(int) >= sizeof(WORD));
00230 STATIC_ASSERT(sizeof(WORD) >= sizeof(char));
00231 STATIC_ASSERT(sizeof(char) == 1);
00232
00233 typedef signed char SBYTE;
00234 typedef unsigned char UBYTE;
00235 typedef unsigned int UINT;
00236 typedef ULONG RLONG;  /* Used in reken.c. */
00237 typedef int64_t MLONG;  /* See commentary in minos.h. */
00238 /*
00239  * NOTE: we don't use the standard _Bool (or C++ bool) because its size is
00240  * implementation-dependent.
00241  */
00242 typedef char BOOL;
00243                                                  /* E.g. in 32-bits */
00244 #define TOPBITONLY     ((ULONG)1 « (BITSINWORD - 1))  /* 0x00008000UL */
00245 #define TOPLONGBITONLY ((ULONG)1 « (BITSINLONG - 1))  /* 0x80000000UL */
00246 #define SPECMASK       ((UWORD)1 « (BITSINWORD - 1))  /*     0x8000U  */
00247 #define WILDMASK       ((UWORD)1 « (BITSINWORD - 2))  /*     0x4000U  */
00248 #define WORDMASK       ((ULONG)FULLMAX - 1)            /* 0x0000FFFFUL */
```

```
00249 #define AWORDMASK       (WORDMASK « BITSINWORD)         /* 0xFFFF0000UL */
00250 #define FULLMAX         ((LONG)1 « BITSINWORD)          /* 0x00010000L  */
00251 #define MAXPOSITIVE     ((LONG)(TOPBITONLY - 1))         /* 0x00007FFFL  */
00252 #define MAXLONG         ((LONG)(TOPLONGBITONLY - 1))     /* 0x7FFFFFFFL  */
00253 #define MAXPOSITIVE2    (MAXPOSITIVE / 2)                /* 0x00003FFFL  */
00254 #define MAXPOSITIVE4    (MAXPOSITIVE / 4)                /* 0x00001FFFL  */
00255
00256 /*
00257  * form_alignof(type) returns the number of bytes used in the alignment of
00258  * the type.
00259  */
00260 #if !defined(form_alignof)
00261 #if defined(__GNUC__)
00262 /* GNU C compiler has "__alignof__". */
00263 #define form_alignof(type) __alignof__(type)
00264 #elif defined(_MSC_VER)
00265 /* Microsoft C compiler has "__alignof". */
00266 #define form_alignof(type) __alignof(type)
00267 #elif !defined(__cplusplus)
00268 /* Generic case in C. */
00269 #include <stddef.h>
00270 #define form_alignof(type) offsetof(struct { char c_; type x_; }, x_)
00271 #else
00272 /* Generic case in C++, at least works with a POD struct. */
00273 #include <cstddef>
00274 namespace alignof_impl_ {
00275 template<typename T> struct calc {
00276     struct X { char c_; T x_; };
00277     enum { value = offsetof(X, x_) };
00278 };
00279 }
00280 #define form_alignof(type) alignof_impl_::calc<type>::value
00281 #endif
00282 #endif
00283
00284
00285 /*
00286 #define WITHPCOUNTER
00287 #define DEBUGGINGLOCKS
00288 #define WITHSTATS
00289 */
00290 #define WITHSORTBOTS
00291
00292 #include <stdio.h>
00293 #include <stdlib.h>
00294 #include <string.h>
00295 #include <ctype.h>
00296 #include <limits.h>
00297 #include <stdarg.h>
00298 #include <time.h>
00299 #ifdef WINDOWS
00300 #include "fwin.h"
00301 #endif
00302 #ifdef UNIX
00303 #include <unistd.h>
00304 #include <fcntl.h>
00305 #include <sys/file.h>
00306 #include "unix.h"
00307 #endif
00308 #ifdef WITHZLIB
00309 #ifdef WITHZSTD
00310 #include <zstd_zlibwrapper.h>
00311 #else
00312 #include <zlib.h>
00313 #endif
00314 #endif
00315 #ifdef WITHPTHREADS
00316 #include <pthread.h>
00317 #endif
00318
00319 /*
00320     PARALLELCODE indicates code that is common for TFORM and ParFORM but
00321     should not be there for sequential FORM.
00322 */
00323 #if defined(WITHMPI) || defined(WITHPTHREADS)
00324 #define PARALLELCODE
00325 #endif
00326
00327 #include "ftypes.h"
00328 #include "fsizes.h"
00329 #include "minos.h"
00330 #include "structs.h"
00331 #include "declare.h"
00332 #include "variable.h"
00333
00334 STATIC_ASSERT(sizeof(off_t) >= sizeof(LONG));
00335
```

```
00336 /*
00337  * The interface to file routines for UNIX or non-UNIX (Windows).
00338  */
00339 #ifdef UNIX
00340
00341 #define UFILES
00342 typedef struct FiLeS {
00343     int descriptor;
00344 } FILES;
00345 extern FILES *Uopen(char *,char *);
00346 extern int    Uclose(FILES *);
00347 extern size_t Uread(char *,size_t,size_t,FILES *);
00348 extern size_t Uwrite(char *,size_t,size_t,FILES *);
00349 extern int    Useek(FILES *,off_t,int);
00350 extern off_t  Utell(FILES *);
00351 extern void   Uflush(FILES *);
00352 extern int    Ugetpos(FILES *,fpos_t *);
00353 extern int    Usetpos(FILES *,fpos_t *);
00354 extern void   Usetbuf(FILES *,char *);
00355 #define Usync(f) fsync(f->descriptor)
00356 #define Utruncate(f) { \
00357     if ( ftruncate(f->descriptor, 0) ) { \
00358         MLOCK(ErrorMessageLock); \
00359         MesPrint("Utruncate failed"); \
00360         MUNLOCK(ErrorMessageLock); \
00361         /* Calling Terminate() here may cause an infinite loop due to CleanUpSort(). */ \
00362         /* Terminate(-1); */ \
00363     } \
00364 }
00365 extern FILES *Ustdout;
00366 #define MAX_OPEN_FILES getdtablesize()
00367 #define GetPID() ((LONG)getpid())
00368
00369 #else  /* UNIX */
00370
00371 #define FILES FILE
00372 #define Uopen(x,y) fopen(x,y)
00373 #define Uflush(x) fflush(x)
00374 #define Uclose(x) fclose(x)
00375 #define Uread(x,y,z,u) fread(x,y,z,u)
00376 #define Uwrite(x,y,z,u) fwrite(x,y,z,u)
00377 #define Usetbuf(x,y) setbuf(x,y)
00378 #define Useek(x,y,z) fseek(x,y,z)
00379 #define Utell(x) ftell(x)
00380 #define Ugetpos(x,y) fgetpos(x,y)
00381 #define Usetpos(x,y) fsetpos(x,y)
00382 #define Usync(x) fflush(x)
00383 #define Utruncate(x) _chsize(_fileno(x),0)
00384 #define Ustdout stdout
00385 #define MAX_OPEN_FILES FOPEN_MAX
00386 #define bzero(b,len) (memset((b), 0, (len)), (void)0)
00387 #define GetPID() ((LONG)GetCurrentProcessId())
00388
00389 #endif  /* UNIX */
00390
00391 /*
00392  * Some system may implement the POSIX "environ" variable as a macro, e.g.,
00393  * https://github.com/mingw-w64/mingw-w64/blob/v10.0.0/mingw-w64-headers/crt/stdlib.h#L704
00394  * which breaks the definition of DoShattering() in diagrams.c that uses
00395  * "environ" as a formal parameter. Because FORM doesn't use the POSIX "environ"
00396  * or its variant anyway, we can just undefine it.
00397  */
00398 #ifdef environ
00399 #undef environ
00400 #endif
00401
00402 #ifdef WITHMPI
00403 #include "parallel.h"
00404 #endif
00405
00406 #endif  /*  __FORM3H__ */
```

4.51 fsizes.h File Reference

This graph shows which files directly or indirectly include this file:

Macros

- #define MAXPRENAMESIZE 128
- #define MAXENAME 16
- #define MAXSAVEFUNCTION 16384
- #define MAXPARLEVEL 100
- #define MAXNUMBERSIZE 200
- #define MAXREPEAT 100
- #define NORMSIZE 1000
- #define INITNODESIZE 10
- #define INITNAMESIZE 100
- #define NUMFIXED 128
- #define MAXNEST 100
- #define MAXMATCH 30
- #define MAXIF 20
- #define SIZEFACS 640L
- #define NUMFACS 50
- #define MAXLOOPS 30
- #define MAXLABELS 20
- #define COMMERCIALSIZE 24
- #define MAXFLAGS 16
- #define COMPRESSBUFFER 90000
- #define FORTRANCONTINUATIONLINES 15
- #define MAXLEVELS 2000
- #define MAXLHS 400
- #define MAXWILDC 100
- #define NUMTABLEENTRIES 1000
- #define COMPILERBUFFER 20000
- #define MAXPATCHES 256
- #define MAXFPATCHES 256
- #define SORTIOSIZE 200000L
- #define SSMALLBUFFER 2560016L
- #define SSMALLOVERFLOW 3840032L
- #define STERMSSMALL 10000L
- #define SLARGEBUFFER 26880512L
- #define SMAXPATCHES 64
- #define SMAXFPATCHES 64
- #define SSORTIOSIZE 32768L
- #define SPECTATORSIZE 1048576L
- #define MAXFLEVELS 30
- #define COMPINC 2
- #define MAXNUMSIZE 10
- #define MAXBRACKETBUFFERSIZE 200000
- #define SFHSIZE 40
- #define DEFAULTPROCESSBUCKETSIZE 1000
- #define SHMWINSIZE 65536L
- #define TABLEEXTENSION 6
- #define GZIPDEFAULT 6
- #define DEFAULTTHREADS 0
- #define DEFAULTTHREADBUCKETSIZE 500
- #define DEFAULTTHREADLOADBALANCING 1
- #define THREADSCRATCHSIZE 100000L
- #define THREADSCRATCHOUTSIZE 2500000L
- #define NUMSTORECACHES 4
- #define SIZESTORECACHE 32768

- #define INDENTSPACE 3
- #define MULTIINDENTSPACE 1
- #define MAXMULTIBRACKETLEVELS 25
- #define FBUFFERSIZE 1026
- #define NPAIR1 38
- #define NPAIR2 89
- #define MAXLINELENGTH 256
- #define MINALLOC 32
- #define JUMPRATIO 4
- #define MAXPARTICLES 20
- #define MAXCOUPLINGS 20
- #define NUMOPTIONS 20
- #define MAXLEGS 20

4.51.1 Detailed Description

The definition the default values for certain buffer sizes etc.

Definition in file fsizes.h.

4.51.2 Macro Definition Documentation

4.51.2.1 MAXPRENAMESIZE

```
#define MAXPRENAMESIZE 128
```

Definition at line 36 of file fsizes.h.

4.51.2.2 MAXENAME

```
#define MAXENAME 16
```

Definition at line 73 of file fsizes.h.

4.51.2.3 MAXSAVEFUNCTION

```
#define MAXSAVEFUNCTION 16384
```

Definition at line 74 of file fsizes.h.

4.51.2.4 MAXPARLEVEL

```
#define MAXPARLEVEL 100
```

Definition at line 76 of file fsizes.h.

4.51.2.5 MAXNUMBERSIZE

```
#define MAXNUMBERSIZE 200
```

Definition at line 77 of file fsizes.h.

4.51.2.6 MAXREPEAT

```
#define MAXREPEAT 100
```

Definition at line 79 of file fsizes.h.

4.51.2.7 NORMSIZE

```
#define NORMSIZE 1000
```

Definition at line 80 of file fsizes.h.

4.51.2.8 INITNODESIZE

```
#define INITNODESIZE 10
```

Definition at line 82 of file fsizes.h.

4.51.2.9 INITNAMESIZE

```
#define INITNAMESIZE 100
```

Definition at line 83 of file fsizes.h.

4.51.2.10 NUMFIXED

```
#define NUMFIXED 128
```

Definition at line 85 of file fsizes.h.

4.51.2.11 MAXNEST

```
#define MAXNEST 100
```

Definition at line 86 of file fsizes.h.

4.51.2.12 MAXMATCH

```
#define MAXMATCH 30
```

Definition at line 87 of file fsizes.h.

4.51.2.13 MAXIF

```
#define MAXIF 20
```

Definition at line 88 of file fsizes.h.

4.51.2.14 SIZEFACS

```
#define SIZEFACS 640L
```

Definition at line 89 of file fsizes.h.

4.51.2.15 NUMFACS

```
#define NUMFACS 50
```

Definition at line 90 of file fsizes.h.

4.51.2.16 MAXLOOPS

```
#define MAXLOOPS 30
```

Definition at line 91 of file fsizes.h.

4.51.2.17 MAXLABELS

```
#define MAXLABELS 20
```

Definition at line 92 of file fsizes.h.

4.51.2.18 COMMERCIALSIZE

```
#define COMMERCIALSIZE 24
```

Definition at line 93 of file fsizes.h.

4.51.2.19 MAXFLAGS

```
#define MAXFLAGS 16
```

Definition at line 94 of file fsizes.h.

4.51.2.20 COMPRESSBUFFER

```
#define COMPRESSBUFFER 90000
```

Definition at line 99 of file fsizes.h.

4.51.2.21 FORTRANCONTINUATIONLINES

```
#define FORTRANCONTINUATIONLINES 15
```

Definition at line 100 of file fsizes.h.

4.51.2.22 MAXLEVELS

```
#define MAXLEVELS 2000
```

Definition at line 101 of file fsizes.h.

4.51.2.23 MAXLHS

```
#define MAXLHS 400
```

Definition at line 102 of file fsizes.h.

4.51.2.24 MAXWILDC

```
#define MAXWILDC 100
```

Definition at line 103 of file fsizes.h.

4.51.2.25 NUMTABLEENTRIES

```
#define NUMTABLEENTRIES 1000
```

Definition at line 104 of file fsizes.h.

4.51.2.26 COMPILERBUFFER

```
#define COMPILERBUFFER 20000
```

Definition at line 105 of file fsizes.h.

4.51.2.27 MAXPATCHES

```
#define MAXPATCHES 256
```

Definition at line 131 of file fsizes.h.

4.51.2.28 MAXFPATCHES

```
#define MAXFPATCHES 256
```

Definition at line 132 of file fsizes.h.

4.51.2.29 SORTIOSIZE

```
#define SORTIOSIZE 200000L
```

Definition at line 133 of file fsizes.h.

4.51.2.30 SSMALLBUFFER

```
#define SSMALLBUFFER 2560016L
```

Definition at line 135 of file fsizes.h.

4.51.2.31 SSMALLOVERFLOW

```
#define SSMALLOVERFLOW 3840032L
```

Definition at line 136 of file fsizes.h.

4.51.2.32 STERMSSMALL

```
#define STERMSSMALL 10000L
```

Definition at line 137 of file fsizes.h.

4.51.2.33 SLARGEBUFFER

```
#define SLARGEBUFFER 26880512L
```

Definition at line 138 of file fsizes.h.

4.51.2.34 SMAXPATCHES

```
#define SMAXPATCHES 64
```

Definition at line 139 of file fsizes.h.

4.51.2.35 SMAXFPATCHES

```
#define SMAXFPATCHES 64
```

Definition at line 140 of file fsizes.h.

4.51.2.36 SSORTIOSIZE

```
#define SSORTIOSIZE 32768L
```

Definition at line 141 of file fsizes.h.

4.51.2.37 SPECTATORSIZE

```
#define SPECTATORSIZE 1048576L
```

Definition at line 143 of file fsizes.h.

4.51.2.38 MAXFLEVELS

```
#define MAXFLEVELS 30
```

Definition at line 145 of file fsizes.h.

4.51.2.39 COMPINC

```
#define COMPINC 2
```

Definition at line 147 of file fsizes.h.

4.51.2.40 MAXNUMSIZE

```
#define MAXNUMSIZE 10
```

Definition at line 149 of file fsizes.h.

4.51.2.41 MAXBRACKETBUFFERSIZE

```
#define MAXBRACKETBUFFERSIZE 200000
```

Definition at line 151 of file fsizes.h.

4.51.2.42 SFHSIZE

```
#define SFHSIZE 40
```

Definition at line 153 of file fsizes.h.

4.51.2.43 DEFAULTPROCESSBUCKETSIZE

```
#define DEFAULTPROCESSBUCKETSIZE 1000
```

Definition at line 155 of file fsizes.h.

4.51.2.44 SHMWINSIZE

```
#define SHMWINSIZE 65536L
```

Definition at line 156 of file fsizes.h.

4.51.2.45 TABLEEXTENSION

```
#define TABLEEXTENSION 6
```

Definition at line 158 of file fsizes.h.

4.51.2.46 GZIPDEFAULT

```
#define GZIPDEFAULT 6
```

Definition at line 160 of file fsizes.h.

4.51.2.47 DEFAULTTHREADS

```
#define DEFAULTTHREADS 0
```

Definition at line 161 of file fsizes.h.

4.51.2.48 DEFAULTTHREADBUCKETSIZE

```
#define DEFAULTTHREADBUCKETSIZE 500
```

Definition at line 162 of file fsizes.h.

4.51.2.49 DEFAULTTHREADLOADBALANCING

```
#define DEFAULTTHREADLOADBALANCING 1
```

Definition at line 163 of file fsizes.h.

4.51.2.50 THREADSCRATCHSIZE

```
#define THREADSCRATCHSIZE 100000L
```

Definition at line 164 of file fsizes.h.

4.51.2.51 THREADSCRATCHOUTSIZE

```
#define THREADSCRATCHOUTSIZE 2500000L
```

Definition at line 165 of file fsizes.h.

4.51.2.52 NUMSTORECACHES

```
#define NUMSTORECACHES 4
```

Definition at line 177 of file fsizes.h.

4.51.2.53 SIZESTORECACHE

```
#define SIZESTORECACHE 32768
```

Definition at line 178 of file fsizes.h.

4.51.2.54 INDENTSPACE

```
#define INDENTSPACE 3
```

Definition at line 180 of file fsizes.h.

4.51.2.55 MULTIINDENTSPACE

```
#define MULTIINDENTSPACE 1
```

Definition at line 182 of file fsizes.h.

4.51.2.56 MAXMULTIBRACKETLEVELS

```
#define MAXMULTIBRACKETLEVELS 25
```

Definition at line 183 of file fsizes.h.

4.51.2.57 FBUFFERSIZE

```
#define FBUFFERSIZE 1026
```

Definition at line 185 of file fsizes.h.

4.51.2.58 NPAIR1

```
#define NPAIR1 38
```

Definition at line 189 of file fsizes.h.

4.51.2.59 NPAIR2

```
#define NPAIR2 89
```

Definition at line 190 of file fsizes.h.

4.51.2.60 MAXLINELENGTH

```
#define MAXLINELENGTH 256
```

Definition at line 192 of file fsizes.h.

4.51.2.61 MINALLOC

```
#define MINALLOC 32
```

Definition at line 193 of file fsizes.h.

4.51.2.62 JUMPRATIO

```
#define JUMPRATIO 4
```

Definition at line 195 of file fsizes.h.

4.51.2.63 MAXPARTICLES

```
#define MAXPARTICLES 20
```

Definition at line 199 of file fsizes.h.

4.51.2.64 MAXCOUPLINGS

```
#define MAXCOUPLINGS 20
```

Definition at line 200 of file fsizes.h.

4.51.2.65 NUMOPTIONS

```
#define NUMOPTIONS 20
```

Definition at line 201 of file fsizes.h.

4.51.2.66 MAXLEGS

```
#define MAXLEGS 20
```

Definition at line 202 of file fsizes.h.

4.52 fsizes.h

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032
00033 /*
00034     First the fixed variables
00035 */
00036 #define MAXPRENAMESIZE 128
00037 /*
00038     The following variables are default sizes. They can be changed
00039     into values read from the setup file
00040
00041     Remark (21-dec-2008 JV): WILDOFFSET*3 should be larger than WILDMASK!!!!
00042         old value was WILDOFFSET 200000100
00043         be careful with old .sav files!!!
00044 */
00045 #if BITSINWORD == 32
00046     #define MAXPOWER 500000000
00047     #define MAXVARIABLES 200000050
00048     #define MAXDOLLARVARIABLES 1000000000L
00049     #define WILDOFFSET 400000100
00050     #define MAXINNAMETREE 2000000000
00051     #define MAXDUMMIES 100000000
00052     #define WORKBUFFER 40000000
00053     #define MAXTER 40000
00054     #define HALFMAX 0x10000
00055     #define MAXSUBEXPRESSIONS 0x1FFFFFF
00056     #define MAXFILESTREAMSIZE 1024
00057 #elif BITSINWORD == 16
00058     #define MAXPOWER 10000
00059     #define MAXVARIABLES 8050
00060     #define MAXDOLLARVARIABLES 32000
00061     #define WILDOFFSET 6100
00062     #define MAXINNAMETREE 32768
00063     #define MAXDUMMIES 1000
00064     #define WORKBUFFER 10000000
00065     #define MAXTER 10000
00066     #define HALFMAX 0x100
00067     #define MAXSUBEXPRESSIONS 0x3FFF
00068     #define MAXFILESTREAMSIZE 1048576
00069 #else
00070     #error Only 64-bit and 32-bit platforms are supported.
00071 #endif
00072
00073 #define MAXENAME 16
00074 #define MAXSAVEFUNCTION 16384
00075
00076 #define MAXPARLEVEL 100
00077 #define MAXNUMBERSIZE 200
00078
00079 #define MAXREPEAT 100
00080 #define NORMSIZE 1000
00081
00082 #define INITNODESIZE 10
00083 #define INITNAMESIZE 100
00084
00085 #define NUMFIXED 128
00086 #define MAXNEST 100
```

```
00087 #define MAXMATCH 30
00088 #define MAXIF 20
00089 #define SIZEFACS 640L
00090 #define NUMFACS 50
00091 #define MAXLOOPS 30
00092 #define MAXLABELS 20
00093 #define COMMERCIALSIZE 24
00094 #define MAXFLAGS 16
00095 /*
00096     The next quantities should still be eliminated from the program
00097     This should be together with changes in setfile!
00098 */
00099 #define COMPRESSBUFFER 90000
00100 #define FORTRANCONTINUATIONLINES 15
00101 #define MAXLEVELS 2000
00102 #define MAXLHS 400
00103 #define MAXWILDC 100
00104 #define NUMTABLEENTRIES 1000
00105 #define COMPILERBUFFER 20000
00106
00107 #if BITSINWORD == 32
00108     #ifdef WITHPTHREADS
00109         #define SMALLBUFFER   300000000L
00110         #define SMALLOVERFLOW 600000000L
00111         #define TERMSSMALL      3000000L
00112         #define LARGEBUFFER  1500000000L
00113         #define SCRATCHSIZE   500000000L
00114     #else
00115         #define SMALLBUFFER   150000000L
00116         #define SMALLOVERFLOW 300000000L
00117         #define TERMSSMALL      2000000L
00118         #define LARGEBUFFER   800000000L
00119         #define SCRATCHSIZE   500000000L
00120     #endif
00121 #elif BITSINWORD == 16
00122     #define SMALLBUFFER   10000000L
00123     #define SMALLOVERFLOW 20000000L
00124     #define TERMSSMALL      100000L
00125     #define LARGEBUFFER   50000000L
00126     #define SCRATCHSIZE   50000000L
00127 #else
00128     #error Only 64-bit and 32-bit platforms are supported.
00129 #endif
00130
00131 #define MAXPATCHES 256
00132 #define MAXFPATCHES 256
00133 #define SORTIOSIZE 200000L
00134
00135 #define SSMALLBUFFER 2560016L
00136 #define SSMALLOVERFLOW 3840032L
00137 #define STERMSSMALL 10000L
00138 #define SLARGEBUFFER 26880512L
00139 #define SMAXPATCHES 64
00140 #define SMAXFPATCHES 64
00141 #define SSORTIOSIZE 32768L
00142
00143 #define SPECTATORSIZE 1048576L
00144
00145 #define MAXFLEVELS 30
00146
00147 #define COMPINC 2
00148
00149 #define MAXNUMSIZE 10
00150
00151 #define MAXBRACKETBUFFERSIZE 200000
00152
00153 #define SFHSIZE 40
00154
00155 #define DEFAULTPROCESSBUCKETSIZE 1000
00156 #define SHMWINSIZE     65536L
00157
00158 #define TABLEEXTENSION 6
00159
00160 #define GZIPDEFAULT 6
00161 #define DEFAULTTHREADS 0
00162 #define DEFAULTTHREADBUCKETSIZE 500
00163 #define DEFAULTTHREADLOADBALANCING 1
00164 #define THREADSCRATCHSIZE 100000L
00165 #define THREADSCRATCHOUTSIZE 2500000L
00166
00167 #if BITSINWORD == 32
00168     #define MAXTABLECOMBUF 1000000000000L
00169     #define MAXCOMBUFRHS 1000000000L
00170 #elif BITSINWORD == 16
00171     #define MAXTABLECOMBUF 1000000L
00172     #define MAXCOMBUFRHS 32500L
00173 #else
```

```
00174     #error Only 64-bit and 32-bit platforms are supported.
00175 #endif
00176
00177 #define NUMSTORECACHES 4
00178 #define SIZESTORECACHE 32768
00179
00180 #define INDENTSPACE 3
00181
00182 #define MULTIINDENTSPACE 1
00183 #define MAXMULTIBRACKETLEVELS 25
00184
00185 #define FBUFFERSIZE 1026
00186 /*
00187     For the random number generator (see commentary there)
00188 */
00189 #define NPAIR1 38
00190 #define NPAIR2 89
00191
00192 #define MAXLINELENGTH 256
00193 #define MINALLOC 32
00194
00195 #define JUMPRATIO 4
00196 /*
00197     Note: MAXCOUPLINGS should be at least MAXPARTICLES/2+1
00198 */
00199 #define MAXPARTICLES 20
00200 #define MAXCOUPLINGS 20
00201 #define NUMOPTIONS 20
00202 #define MAXLEGS 20
00203
00204 #ifdef WITHFLOAT
00205 #define MAXWEIGHT 0
00206 #define DEFAULTPRECISION 1000
00207 #endif
```

4.53 ftypes.h File Reference

This graph shows which files directly or indirectly include this file:

Macros

- #define WITHOUTERROR 0
- #define WITHERROR 1
- #define FILESTREAM 0
- #define PREVARSTREAM 1
- #define PREREADSTREAM 2
- #define PIPESTREAM 3
- #define PRECALCSTREAM 4
- #define DOLLARSTREAM 5
- #define PREREADSTREAM2 6
- #define EXTERNALCHANNELSTREAM 7
- #define PREREADSTREAM3 8
- #define REVERSEFILESTREAM 9
- #define INPUTSTREAM 10
- #define ENDOFSTREAM 0xFF
- #define ENDOFINPUT 0xFF
- #define SUBROUTINEFILE 0
- #define PROCEDUREFILE 1
- #define HEADERFILE 2
- #define SETUPFILE 3

- #define TABLEBASEFILE 4
- #define FIRSTMODULE -1
- #define GLOBALMODULE 0
- #define SORTMODULE 1
- #define STOREMODULE 2
- #define CLEARMODULE 3
- #define ENDMODULE 4
- #define POLYFUN 0
- #define NOPARALLEL_DOLLAR 0x0001
- #define NOPARALLEL_RHS 0x0002
- #define NOPARALLEL_CONVPOLY 0x0004
- #define NOPARALLEL_SPECTATOR 0x0008
- #define NOPARALLEL_USER 0x0010
- #define NOPARALLEL_TBLDOLLAR 0x0100
- #define NOPARALLEL_NPROC 0x0200
- #define PARALLELFLAG 0x0000
- #define PRENOACTION 0
- #define PRERAISEAFTER 1
- #define PRELOWERAFTER 2
- #define WITHSEMICOLON 0
- #define WITHOUTSEMICOLON 1
- #define MODULEINSTACK 8
- #define EXECUTINGIF 0
- #define LOOKINGFORELSE 1
- #define LOOKINGFORENDIF 2
- #define NEWSTATEMENT 1
- #define OLDSTATEMENT 0
- #define EXECUTINGPRESWITCH 0
- #define SEARCHINGPRECASE 1
- #define SEARCHINGPREENDSWITCH 2
- #define PREPROONLY 1
- #define DUMPTOCOMPILER 2
- #define DUMPOUTTERMS 4
- #define DUMPINTERMS 8
- #define DUMPTOSORT 16
- #define DUMPTOPARALLEL 32
- #define THREADSDEBUG 64
- #define ERROROUT 0
- #define INPUTOUT 1
- #define STATSOUT 2
- #define EXPRSOUT 3
- #define WRITEOUT 4
- #define EXTERNALCHANNELOUT 5
- #define NUMERICALLOOP 0
- #define LISTEDLOOP 1
- #define ONEEXPRESSION 2
- #define PRETYPENONE 0
- #define PRETYPEIF 1
- #define PRETYPEDO 2
- #define PRETYPEPROCEDURE 3
- #define PRETYPESWITCH 4
- #define PRETYPEINSIDE 5
- #define DECLARATION 1
- #define SPECIFICATION 2
- #define DEFINITION 3

- #define STATEMENT 4
- #define TOOUTPUT 5
- #define ATENDOFMODULE 6
- #define MIXED2 8 /∗ unused ∗/
- #define MIXED 9
- #define NOAUTO 0
- #define PARTEST 1 /∗ parentheses test ∗/
- #define WITHAUTO 2 /∗ auto-declaration ∗/
- #define ALLVARIABLES -1
- #define SYMBOLONLY 1
- #define INDEXONLY 2
- #define VECTORONLY 4
- #define FUNCTIONONLY 8
- #define SETONLY 16
- #define EXPRESSIONONLY 32

Defines: compiler types

Type of variable found by the compiler.

- #define CDELETE -1
- #define ANYTYPE -1
- #define CSYMBOL 0
- #define CINDEX 1
- #define CVECTOR 2
- #define CFUNCTION 3
- #define CSET 4
- #define CEXPRESSION 5
- #define CDOTPRODUCT 6
- #define CNUMBER 7
- #define CSUBEXP 8
- #define CDELTA 9
- #define CDOLLAR 10
- #define CDUBIOUS 11
- #define CRANGE 12
- #define CMODEL 13
- #define CVECTOR1 21
- #define CDOUBLEDOT 22
- #define TSYMBOL -1
- #define TINDEX -2
- #define TVECTOR -3
- #define TFUNCTION -4
- #define TSET -5
- #define TEXPRESSION -6
- #define TDOTPRODUCT -7
- #define TNUMBER -8
- #define TSUBEXP -9
- #define TDELTA -10
- #define TDOLLAR -11
- #define TDUBIOUS -12
- #define LPARENTHESIS -13
- #define RPARENTHESIS -14
- #define TWILDCARD -15
- #define TWILDARG -16
- #define TDOT -17
- #define LBRACE -18
- #define RBRACE -19
- #define TCOMMA -20
- #define TFUNOPEN -21
- #define TFUNCLOSE -22
- #define TMULTIPLY -23
- #define TDIVIDE -24

- #define TPOWER -25
- #define TPLUS -26
- #define TMINUS -27
- #define TNOT -28
- #define TENDOFIT -29
- #define TSETOPEN -30
- #define TSETCLOSE -31
- #define TGENINDEX -32
- #define TCONJUGATE -33
- #define LRPARENTHESES -34
- #define TNUMBER1 -35
- #define TPOWER1 -36
- #define TEMPTY -37
- #define TSETNUM -38
- #define TSGAMMA -39
- #define TSETDOL -40
- #define TFLOAT -41
- #define TYPEISFUN 0
- #define TYPEISSUB 1
- #define TYPEISMYSTERY -1
- #define LHSIDEX 2
- #define LHSIDE 1
- #define RHSIDE 0
- #define FORTRANMODE 1
- #define REDUCEMODE 2
- #define MAPLEMODE 3
- #define MATHEMATICAMODE 4
- #define CMODE 5
- #define VORTRANMODE 6
- #define PFORTRANMODE 7
- #define DOUBLEFORTRANMODE 33
- #define DOUBLEPRECISIONFLAG 32
- #define NODOUBLEMASK 31
- #define QUADRUPLEFORTRANMODE 65
- #define QUADRUPLEPRECISIONFLAG 64
- #define ALLINTEGERDOUBLE 128
- #define NOQUADMASK 63
- #define NORMALFORMAT 0
- #define NOSPACEFORMAT 1
- #define ISNOTFORTRAN90 0
- #define ISFORTRAN90 1
- #define ALSOREVERSE 1
- #define CHISHOLM 2
- #define NOTRICK 16
- #define SORTLOWFIRST 0
- #define SORTHIGHFIRST 1
- #define SORTPOWERFIRST 2
- #define SORTANTIPOWER 3
- #define STATSSPLITMERGE 0
- #define STATSMERGETOFILE 1
- #define STATSPOSTSORT 2
- #define NOCHECKLOGTYPE 0
- #define CHECKLOGTYPE 1
- #define NMIN4SHIFT 4
- #define SYMBOL 1
- #define DOTPRODUCT 2
- #define VECTOR 3
- #define INDEX 4
- #define EXPRESSION 5
- #define SUBEXPRESSION 6
- #define DOLLAREXPRESSION 7
- #define SETSET 8
- #define ARGWILD 9
- #define MINVECTOR 10

- #define SETEXP 11
- #define DOLLAREXPR2 12
- #define HAAKJE0 9
- #define FUNCTION 20
- #define TMPPOLYFUN 14
- #define ARGFIELD 15
- #define SNUMBER 16
- #define LNUMBER 17
- #define HAAKJE 18
- #define DELTA 19
- #define EXPONENT 20
- #define DENOMINATOR 21
- #define SETFUNCTION 22
- #define GAMMA 23
- #define GAMMAI 24
- #define GAMMAFIVE 25
- #define GAMMASIX 26
- #define GAMMASEVEN 27
- #define SUMF1 28
- #define SUMF2 29
- #define DUMFUN 30
- #define REPLACEMENT 31
- #define REVERSEFUNCTION 32
- #define DISTRIBUTION 33
- #define DELTA3 34
- #define DUMMYFUN 35
- #define DUMMYTEN 36
- #define LEVICIVITA 37
- #define FACTORIAL 38
- #define INVERSEFACTORIAL 39
- #define BINOMIAL 40
- #define NUMARGSFUN 41
- #define SIGNFUN 42
- #define MODFUNCTION 43
- #define MOD2FUNCTION 44
- #define MINFUNCTION 45
- #define MAXFUNCTION 46
- #define ABSFUNCTION 47
- #define SIGFUNCTION 48
- #define INTFUNCTION 49
- #define THETA 50
- #define THETA2 51
- #define DELTA2 52
- #define DELTAP 53
- #define BERNOULLIFUNCTION 54
- #define COUNTFUNCTION 55
- #define MATCHFUNCTION 56
- #define PATTERNFUNCTION 57
- #define TERMFUNCTION 58
- #define CONJUGATION 59
- #define ROOTFUNCTION 60
- #define TABLEFUNCTION 61
- #define FIRSTBRACKET 62
- #define TERMSINEXPR 63
- #define NUMTERMSFUN 64
- #define GCDFUNCTION 65
- #define DIVFUNCTION 66
- #define REMFUNCTION 67
- #define MAXPOWEROF 68
- #define MINPOWEROF 69
- #define TABLESTUB 70
- #define FACTORIN 71
- #define TERMSINBRACKET 72
- #define WILDARGFUN 73

- #define SQRTFUNCTION 74
- #define LNFUNCTION 75
- #define SINFUNCTION 76
- #define COSFUNCTION 77
- #define TANFUNCTION 78
- #define ASINFUNCTION 79
- #define ACOSFUNCTION 80
- #define ATANFUNCTION 81
- #define ATAN2FUNCTION 82
- #define SINHFUNCTION 83
- #define COSHFUNCTION 84
- #define TANHFUNCTION 85
- #define ASINHFUNCTION 86
- #define ACOSHFUNCTION 87
- #define ATANHFUNCTION 88
- #define LI2FUNCTION 89
- #define LINFUNCTION 90
- #define EXTRASYMFUN 91
- #define RANDOMFUNCTION 92
- #define RANPERM 93
- #define NUMFACTORS 94
- #define FIRSTTERM 95
- #define CONTENTTERM 96
- #define PRIMENUMBER 97
- #define EXTEUCLIDEAN 98
- #define MAKERATIONAL 99
- #define INVERSEFUNCTION 100
- #define IDFUNCTION 101
- #define PUTFIRST 102
- #define PERMUTATIONS 103
- #define PARTITIONS 104
- #define MULFUNCTION 105
- #define MOEBIUS 106
- #define TOPOLOGIES 107
- #define DIAGRAMS 108
- #define TOPO 109
- #define NODEFUNCTION 110
- #define EDGE 111
- #define SIZEOFFUNCTION 112
- #define BLOCK 113
- #define ONEPI 114
- #define PHI 115
- #define FLOATFUN 116
- #define TOFLOAT 117
- #define TORAT 118
- #define MZV 119
- #define EULER 120
- #define MZVHALF 121
- #define AGMFUNCTION 122
- #define GAMMAFUN 123
- #define EXPFUNCTION 124
- #define HPLFUNCTION 125
- #define MPLFUNCTION 126
- #define MAXBUILTINFUNCTION 126
- #define FIRSTUSERFUNCTION 150
- #define ALLFUNCTIONS -1
- #define ALLMZVFUNCTIONS -2
- #define ISYMBOL 0
- #define PISYMBOL 1
- #define COEFFSYMBOL 2
- #define NUMERATORSYMBOL 3
- #define DENOMINATORSYMBOL 4
- #define WILDARGSYMBOL 5
- #define DIMENSIONSYMBOL 6

- #define FACTORSYMBOL 7
- #define SEPARATESYMBOL 8
- #define EESYMBOL 9
- #define EMSYMBOL 10
- #define BUILTINSYMBOLS 11
- #define FIRSTUSERSYMBOL 20
- #define BUILTININDICES 1
- #define BUILTINVECTORS 1
- #define BUILTINDOLLARS 1
- #define WILDARGVECTOR 0
- #define WILDARGINDEX 0
- #define TYPEEXPRESSION 0
- #define TYPEIDNEW 1
- #define TYPEIDOLD 2
- #define TYPEOPERATION 3
- #define TYPEREPEAT 4
- #define TYPEENDREPEAT 5
- #define TYPECOUNT 20
- #define TYPEMULT 21
- #define TYPEGOTO 22
- #define TYPEDISCARD 23
- #define TYPEIF 24
- #define TYPEELSE 25
- #define TYPEELIF 26
- #define TYPEENDIF 27
- #define TYPESUM 28
- #define TYPECHISHOLM 29
- #define TYPEREVERSE 30
- #define TYPEARG 31
- #define TYPENORM 32
- #define TYPENORM2 33
- #define TYPENORM3 34
- #define TYPEEXIT 35
- #define TYPESETEXIT 36
- #define TYPEPRINT 37
- #define TYPEFPRINT 38
- #define TYPEREDEFPRE 39
- #define TYPESPLITARG 40
- #define TYPESPLITARG2 41
- #define TYPEFACTARG 42
- #define TYPEFACTARG2 43
- #define TYPETRY 44
- #define TYPEASSIGN 45
- #define TYPERENUMBER 46
- #define TYPESUMFIX 47
- #define TYPEFINDLOOP 48
- #define TYPEUNRAVEL 49
- #define TYPEADJUSTBOUNDS 50
- #define TYPEINSIDE 51
- #define TYPETERM 52
- #define TYPESORT 53
- #define TYPEDETCURDUM 54
- #define TYPEINEXPRESSION 55
- #define TYPESPLITFIRSTARG 56
- #define TYPESPLITLASTARG 57
- #define TYPEMERGE 58
- #define TYPETESTUSE 59
- #define TYPEAPPLY 60
- #define TYPEAPPLYRESET 61
- #define TYPECHAININ 62
- #define TYPECHAINOUT 63
- #define TYPENORM4 64
- #define TYPEFACTOR 65
- #define TYPEARGIMPLODE 66

- #define TYPEARGEXPLODE 67
- #define TYPEDENOMINATORS 68
- #define TYPESTUFFLE 69
- #define TYPEDROPCOEFFICIENT 70
- #define TYPETRANSFORM 71
- #define TYPETOPOLYNOMIAL 72
- #define TYPEFROMPOLYNOMIAL 73
- #define TYPEDOLOOP 74
- #define TYPEENDDOLOOP 75
- #define TYPEDROPSYMBOLS 76
- #define TYPEPUTINSIDE 77
- #define TYPETOSPECTATOR 78
- #define TYPEARGTOEXTRASYMBOL 79
- #define TYPECANONICALIZE 80
- #define TYPESWITCH 81
- #define TYPEENDSWITCH 82
- #define TYPESETUSERFLAG 83
- #define TYPECLEARUSERFLAG 84
- #define TYPEALLLOOPS 85
- #define TYPEALLPATHS 86
- #define TAKETRACE 1
- #define CONTRACT 2
- #define RATIO 3
- #define SYMMETRIZE 4
- #define TENVEC 5
- #define SUMNUM1 6
- #define SUMNUM2 7
- #define WILDDUMMY 0
- #define SYMTONUM 1
- #define SYMTOSYM 2
- #define SYMTOSUB 3
- #define VECTOMIN 4
- #define VECTOVEC 5
- #define VECTOSUB 6
- #define INDTOIND 7
- #define INDTOSUB 8
- #define FUNTOFUN 9
- #define ARGTOARG 10
- #define ARLTOARL 11
- #define EXPTOEXP 12
- #define FROMBRAC 13
- #define FROMSET 14
- #define SETTONUM 15
- #define WILDCARDS 16
- #define SETNUMBER 17
- #define LOADDOLLAR 18
- #define NUMTONUM 20
- #define NUMTOSYM 21
- #define NUMTOIND 22
- #define NUMTOSUB 23
- #define EATTENSOR 0x2000
- #define CLEANFLAG 0
- #define DIRTYFLAG 1
- #define DIRTYSYMFLAG 2
- #define MUSTCLEANPRF 4
- #define SUBTERMUSED1 8
- #define SUBTERMUSED2 16
- #define ALLDIRTY (DIRTYFLAG|DIRTYSYMFLAG)
- #define ARGHEAD 2
- #define FUNHEAD 3
- #define SUBEXPSIZE 5
- #define EXPRHEAD 5
- #define TYPEARGHEADSIZE 6
- #define SKIP 1

- #define DROP 2
- #define HIDE 3
- #define UNHIDE 4
- #define INTOHIDE 5
- #define LOCALEXPRESSION 0
- #define SKIPLEXPRESSION 1
- #define DROPLEXPRESSION 2
- #define DROPPEDEXPRESSION 3
- #define GLOBALEXPRESSION 4
- #define SKIPGEXPRESSION 5
- #define DROPGEXPRESSION 6
- #define STOREDEXPRESSION 8
- #define HIDDENLEXPRESSION 9
- #define HIDDENGEXPRESSION 13
- #define INCEXPRESSION 9
- #define HIDELEXPRESSION 10
- #define HIDEGEXPRESSION 14
- #define DROPHLEXPRESSION 11
- #define DROPHGEXPRESSION 15
- #define UNHIDELEXPRESSION 12
- #define UNHIDEGEXPRESSION 16
- #define INTOHIDELEXPRESSION 17
- #define INTOHIDEGEXPRESSION 18
- #define SPECTATOREXPRESSION 19
- #define DROPSPECTATOREXPRESSION 20
- #define SKIPUNHIDELEXPRESSION 21
- #define SKIPUNHIDEGEXPRESSION 22
- #define PRINTOFF 0
- #define PRINTON 1
- #define PRINTCONTENTS 2
- #define PRINTCONTENT 3
- #define PRINTLFILE 4
- #define PRINTONETERM 8
- #define PRINTONEFUNCTION 16
- #define PRINTALL 32
- #define REGULAREXPRESSION -1
- #define REDEFINEDEXPRESSION -2
- #define NEWLYDEFINEDEXPRESSION -3

Defines: function specs

Function specifications.

- #define VERTEXFUNCTION -1
- #define GENERALFUNCTION 0
- #define TENSORFUNCTION 2
- #define GAMMAFUNCTION 4
- #define POS_ 0 /∗ integer > 0 ∗/
- #define POS0_ 1 /∗ integer >= 0 ∗/
- #define NEG_ 2 /∗ integer < 0 ∗/
- #define NEG0_ 3 /∗ integer <= 0 ∗/
- #define EVEN_ 4 /∗ integer (even) ∗/
- #define ODD_ 5 /∗ integer (odd) ∗/
- #define Z_ 6 /∗ integer ∗/
- #define SYMBOL_ 7 /∗ symbol only ∗/
- #define FIXED_ 8 /∗ fixed index ∗/
- #define INDEX_ 9 /∗ index only ∗/
- #define Q_ 10 /∗ rational ∗/
- #define DUMMYINDEX_ 11 /∗ dummy index only ∗/
- #define VECTOR_ 12 /∗ vector only ∗/
- #define GAMMA1 0
- #define GAMMA5 -1
- #define GAMMA6 -2
- #define GAMMA7 -3

- #define FUNNYVEC -4
- #define FUNNYWILD -5
- #define SUMMEDIND -6
- #define NOINDEX -7
- #define FUNNYDOLLAR -8
- #define EMPTYINDEX -9
- #define MINSPEC -10
- #define USEDFLAG 2
- #define DUMMYFLAG 1
- #define MAINSORT 0
- #define FUNCTIONSORT 1
- #define SUBSORT 2
- #define FLOATMODE 1
- #define RATIONALMODE 0
- #define NUMSPECSETS 10
- #define ISZERO 1
- #define ISUNMODIFIED 2
- #define ISCOMPRESSED 4
- #define ISINRHS 8
- #define ISFACTORIZED 16
- #define TOBEFACTORED 32
- #define TOBEUNFACTORED 64
- #define KEEPZERO 128
- #define VARTYPENONE 0
- #define VARTYPECOMPLEX 1
- #define VARTYPEIMAGINARY 2
- #define VARTYPEROOTOFUNITY 4
- #define VARTYPEMINUS 8
- #define CYCLESYMMETRIC 1
- #define RCYCLESYMMETRIC 2
- #define SYMMETRIC 3
- #define ANTISYMMETRIC 4
- #define REVERSEORDER 256
- #define SUBMULTI 1
- #define SUBONCE 2
- #define SUBONLY 3
- #define SUBMANY 4
- #define SUBVECTOR 5
- #define SUBSELECT 6
- #define SUBALL 7
- #define SUBMASK 15
- #define SUBDISORDER 16
- #define SUBAFTER 32
- #define SUBAFTERNOT 64
- #define IDHEAD 6
- #define DOLLARFLAG 1
- #define NORMALIZEFLAG 2
- #define GIDENT 1
- #define GFIVE 4
- #define GPLUS 3
- #define GMINUS 2
- #define LONGNUMBER 1
- #define MATCH 2
- #define COEFFI 3
- #define SUBEXPR 4
- #define MULTIPLEOF 5
- #define IFDOLLAR 6
- #define IFEXPRESSION 7
- #define IFDOLLAREXTRA 8
- #define IFISFACTORIZED 9
- #define IFOCCURS 10
- #define IFUSERFLAG 11
- #define IFFLOATNUMBER 12
- #define GREATER 0

- #define GREATEREQUAL 1
- #define LESS 2
- #define LESSEQUAL 3
- #define EQUAL 4
- #define NOTEQUAL 5
- #define ORCOND 6
- #define ANDCOND 7
- #define DUMMY 1
- #define SORT 1
- #define STORE 2
- #define END 3
- #define GLOBAL 4
- #define CLEAR 5
- #define VECTBIT 1
- #define DOTPBIT 2
- #define FUNBIT 4
- #define SETBIT 8
- #define EXTRAPARAMETER 0x4000
- #define GENCOMMUTE 0
- #define GENNONCOMMUTE 0x2000
- #define NAMENOTFOUND -9
- #define DOLUNDEFINED 0
- #define DOLNUMBER 1
- #define DOLARGUMENT 2
- #define DOLSUBTERM 3
- #define DOLTERMS 4
- #define DOLWILDARGS 5
- #define DOLINDEX 6
- #define DOLZERO 7
- #define FINDLOOP 0
- #define REPLACELOOP 1
- #define NOFUNPOWERS 0
- #define COMFUNPOWERS 1
- #define ALLFUNPOWERS 2
- #define PROPERORDERFLAG 0
- #define REGULAR 0
- #define FINISH 1
- #define POLYADD 1
- #define POLYSUB 2
- #define POLYMUL 3
- #define POLYDIV 4
- #define POLYREM 5
- #define POLYGCD 6
- #define POLYINTFAC 7
- #define POLYNORM 8
- #define MODNONE 0
- #define MODSUM 1
- #define MODMAX 2
- #define MODMIN 3
- #define MODLOCAL 4
- #define ELEMENTUSED 1
- #define ELEMENTLOADED 2
- #define POSNEG 0x1
- #define INVERSETABLE 0x2
- #define COEFFICIENTSONLY 0x4
- #define ALSOPOWERS 0x8
- #define ALSOFUNARGS 0x10
- #define ALSODOLLARS 0x20
- #define NOINVERSES 0x40
- #define POSITIVEONLY 0
- #define UNPACK 0x80
- #define NOUNPACK 0
- #define FROMFUNCTION 0x100
- #define VARNAMES 0

- #define AUTONAMES 1
- #define EXPRNAMES 2
- #define DOLLARNAMES 3
- #define DUMMYBUFFER 1
- #define ALLARGS 1
- #define NUMARG 2
- #define ARGRANGE 3
- #define MAKEARGS 4
- #define MAXRANGEINDICATOR 4
- #define REPLACEARG 5
- #define ENCODEARG 6
- #define DECODEARG 7
- #define IMPLODEARG 8
- #define EXPLODEARG 9
- #define PERMUTEARG 10
- #define REVERSEARG 11
- #define CYCLEARG 12
- #define ISLYNDON 13
- #define ISLYNDONR 14
- #define TOLYNDON 15
- #define TOLYNDONR 16
- #define ADDARG 17
- #define MULTIPLYARG 18
- #define DROPARG 19
- #define SELECTARG 20
- #define DEDUPARG 21
- #define ZTOHARG 22
- #define HTOZARG 23
- #define BASECODE 1
- #define YESLYNDON 1
- #define NOLYNDON 2
- #define TOPOLYNOMIALFLAG 1
- #define FACTARGFLAG 2
- #define OLDFACTARG 1
- #define NEWFACTARG 0
- #define FROMMODULEOPTION 0
- #define FROMPOINTINSTRUCTION 1
- #define EXTRASYMBOL 0
- #define REGULARSYMBOL 1
- #define EXPRESSIONNUMBER 2
- #define O_NONE 0
- #define O_CSE 1
- #define O_CSEGREEDY 2
- #define O_GREEDY 3
- #define O_OCCURRENCE 0
- #define O_MCTS 1
- #define O_SIMULATED_ANNEALING 2
- #define O_FORWARD 0
- #define O_BACKWARD 1
- #define O_FORWARDORBACKWARD 2
- #define O_FORWARDANDBACKWARD 3
- #define OPTHEAD 3
- #define DOALL 1
- #define ONLYFUNCTIONS 2
- #define INUSE 1
- #define COULDCOMMUTE 2
- #define DOESNOTCOMMUTE 4
- #define DICT_NONUMBERS 0
- #define DICT_INTEGERONLY 1
- #define DICT_RATIONALONLY 2
- #define DICT_ALLNUMBERS 3
- #define DICT_NOVARIABLES 0
- #define DICT_DOVARIABLES 1
- #define DICT_NOSPECIALS 0

- #define DICT_DOSPECIALS 1
- #define DICT_NOFUNWITHARGS 0
- #define DICT_DOFUNWITHARGS 1
- #define DICT_NOTINDOLLARS 0
- #define DICT_INDOLLARS 1
- #define DICT_INTEGERNUMBER 1
- #define DICT_RATIONALNUMBER 2
- #define DICT_SYMBOL 3
- #define DICT_VECTOR 4
- #define DICT_INDEX 5
- #define DICT_FUNCTION 6
- #define DICT_FUNCTION_WITH_ARGUMENTS 7
- #define DICT_SPECIALCHARACTER 8
- #define DICT_RANGE 9
- #define READSPECTATORFLAG 3
- #define GLOBALSPECTATORFLAG 1
- #define ORDEREDSET 1
- #define DENSETABLE 1
- #define SPARSETABLE 0
- #define TOPOLOGIESONLY 1
- #define WITHOUTNODES 2
- #define WITHEDGES 4
- #define WITHBLOCKS 8
- #define WITHONEPISETS 16
- #define WITHSYMMETRIZEI 32
- #define WITHSYMMETRIZEF 64
- #define CHECKEXTERN 128
- #define ONEPARTI 256
- #define ONEPARTR 512
- #define ONSHELL 1024
- #define OFFSHELL 2048
- #define NOSIGMA 4096
- #define SIGMA 8192
- #define NOSNAIL 16384
- #define SNAIL 32768
- #define NOTADPOLE 65536
- #define TADPOLE 131072
- #define SIMPLE 262144
- #define NOTSIMPLE 524288
- #define BIPART 1048576
- #define NONBIPART 2097152
- #define CYCLI 4194304
- #define CYCLR 8388608
- #define FLOOP 16777216
- #define NOTFLOOP 33554432

Typedefs

- typedef void(∗ PVFUNWP) (WORD ∗)
- typedef void(∗ PVFUNV) (void)
- typedef int(∗ CFUN) (void)
- typedef int(∗ TFUN) (UBYTE ∗)
- typedef int(∗ TFUN1) (UBYTE ∗, int)

4.53.1 Detailed Description

Contains the definitions of many internal codes Rather than using numbers directly we do this by defines, making it much easier to change things. Changing things is sometimes also a good way of testing the code.

Definition in file ftypes.h.

4.53.2 Macro Definition Documentation

4.53.2.1 WITHOUTERROR

```
#define WITHOUTERROR 0
```

The next macros were introduced when TFORM was programmed. In the case of workers, each worker may need some private data. These can in principle be accessed by some posix calls but that is unnecessarily slow. The passing of a pointer to the complete data struct with private data will be much faster. And anyway, there would have to be a macro that either makes the posix call (TFORM) or doesn't (FORM). The solution by having macro's that either pass the pointer (TFORM) or don't pass it (FORM) is seen as the best solution.

In the declarations and the calling of the functions we have to use the PHEAD or the BHEAD macro, respectively, if the pointer is to be passed. These macro's contain the comma as well. Hence we need special macro's if there are no other arguments. These are called PHEAD0 and BHEAD0.

Definition at line 51 of file ftypes.h.

4.53.2.2 WITHERROR

```
#define WITHERROR 1
```

Definition at line 52 of file ftypes.h.

4.53.2.3 FILESTREAM

```
#define FILESTREAM 0
```

Definition at line 58 of file ftypes.h.

4.53.2.4 PREVARSTREAM

```
#define PREVARSTREAM 1
```

Definition at line 59 of file ftypes.h.

4.53.2.5 PREREADSTREAM

```
#define PREREADSTREAM 2
```

Definition at line 60 of file ftypes.h.

4.53.2.6 PIPESTREAM

```
#define PIPESTREAM 3
```

Definition at line 61 of file ftypes.h.

4.53.2.7 PRECALCSTREAM

```
#define PRECALCSTREAM 4
```

Definition at line 62 of file ftypes.h.

4.53.2.8 DOLLARSTREAM

```
#define DOLLARSTREAM 5
```

Definition at line 63 of file ftypes.h.

4.53.2.9 PREREADSTREAM2

```
#define PREREADSTREAM2 6
```

Definition at line 64 of file ftypes.h.

4.53.2.10 EXTERNALCHANNELSTREAM

```
#define EXTERNALCHANNELSTREAM 7
```

Definition at line 65 of file ftypes.h.

4.53.2.11 PREREADSTREAM3

```
#define PREREADSTREAM3 8
```

Definition at line 66 of file ftypes.h.

4.53.2.12 REVERSEFILESTREAM

```
#define REVERSEFILESTREAM 9
```

Definition at line 67 of file ftypes.h.

4.53.2.13 INPUTSTREAM

```
#define INPUTSTREAM 10
```

Definition at line 68 of file ftypes.h.

4.53.2.14 ENDOFSTREAM

```
#define ENDOFSTREAM 0xFF
```

Definition at line 70 of file ftypes.h.

4.53.2.15 ENDOFINPUT

```
#define ENDOFINPUT 0xFF
```

Definition at line 71 of file ftypes.h.

4.53.2.16 SUBROUTINEFILE

```
#define SUBROUTINEFILE 0
```

Definition at line 77 of file ftypes.h.

4.53.2.17 PROCEDUREFILE

```
#define PROCEDUREFILE 1
```

Definition at line 78 of file ftypes.h.

4.53.2.18 HEADERFILE

```
#define HEADERFILE 2
```

Definition at line 79 of file ftypes.h.

4.53.2.19 SETUPFILE

```
#define SETUPFILE 3
```

Definition at line 80 of file ftypes.h.

4.53.2.20 TABLEBASEFILE

```
#define TABLEBASEFILE 4
```

Definition at line 81 of file ftypes.h.

4.53.2.21 FIRSTMODULE

```
#define FIRSTMODULE -1
```

Definition at line 87 of file ftypes.h.

4.53.2.22 GLOBALMODULE

```
#define GLOBALMODULE 0
```

Definition at line 88 of file ftypes.h.

4.53.2.23 SORTMODULE

```
#define SORTMODULE 1
```

Definition at line 89 of file ftypes.h.

4.53.2.24 STOREMODULE

```
#define STOREMODULE 2
```

Definition at line 90 of file ftypes.h.

4.53.2.25 CLEARMODULE

```
#define CLEARMODULE 3
```

Definition at line 91 of file ftypes.h.

4.53.2.26 ENDMODULE

```
#define ENDMODULE 4
```

Definition at line 92 of file ftypes.h.

4.53.2.27 POLYFUN

```
#define POLYFUN 0
```

Definition at line 94 of file ftypes.h.

4.53.2.28 NOPARALLEL_DOLLAR

```
#define NOPARALLEL_DOLLAR 0x0001
```

Definition at line 96 of file ftypes.h.

4.53.2.29 NOPARALLEL_RHS

```
#define NOPARALLEL_RHS 0x0002
```

Definition at line 97 of file ftypes.h.

4.53.2.30 NOPARALLEL_CONVPOLY

```
#define NOPARALLEL_CONVPOLY 0x0004
```

Definition at line 98 of file ftypes.h.

4.53.2.31 NOPARALLEL_SPECTATOR

```
#define NOPARALLEL_SPECTATOR 0x0008
```

Definition at line 99 of file ftypes.h.

4.53.2.32 NOPARALLEL_USER

```
#define NOPARALLEL_USER 0x0010
```

Definition at line 100 of file ftypes.h.

4.53.2.33 NOPARALLEL_TBLDOLLAR

```
#define NOPARALLEL_TBLDOLLAR 0x0100
```

Definition at line 101 of file ftypes.h.

4.53.2.34 NOPARALLEL_NPROC

```
#define NOPARALLEL_NPROC 0x0200
```

Definition at line 102 of file ftypes.h.

4.53.2.35 PARALLELFLAG

```
#define PARALLELFLAG 0x0000
```

Definition at line 103 of file ftypes.h.

4.53.2.36 PRENOACTION

```
#define PRENOACTION 0
```

Definition at line 105 of file ftypes.h.

4.53.2.37 PRERAISEAFTER

```
#define PRERAISEAFTER 1
```

Definition at line 106 of file ftypes.h.

4.53.2.38 PRELOWERAFTER

```
#define PRELOWERAFTER 2
```

Definition at line 107 of file ftypes.h.

4.53.2.39 WITHSEMICOLON

```
#define WITHSEMICOLON 0
```

Definition at line 114 of file ftypes.h.

4.53.2.40 WITHOUTSEMICOLON

```
#define WITHOUTSEMICOLON 1
```

Definition at line 115 of file ftypes.h.

4.53.2.41 MODULEINSTACK

```
#define MODULEINSTACK 8
```

Definition at line 116 of file ftypes.h.

4.53.2.42 EXECUTINGIF

```
#define EXECUTINGIF 0
```

Definition at line 117 of file ftypes.h.

4.53.2.43 LOOKINGFORELSE

```
#define LOOKINGFORELSE 1
```

Definition at line 118 of file ftypes.h.

4.53.2.44 LOOKINGFORENDIF

```
#define LOOKINGFORENDIF 2
```

Definition at line 119 of file ftypes.h.

4.53.2.45 NEWSTATEMENT

```
#define NEWSTATEMENT 1
```

Definition at line 120 of file ftypes.h.

4.53.2.46 OLDSTATEMENT

```
#define OLDSTATEMENT 0
```

Definition at line 121 of file ftypes.h.

4.53.2.47 EXECUTINGPRESWITCH

```
#define EXECUTINGPRESWITCH 0
```

Definition at line 123 of file ftypes.h.

4.53.2.48 SEARCHINGPRECASE

```
#define SEARCHINGPRECASE 1
```

Definition at line 124 of file ftypes.h.

4.53.2.49 SEARCHINGPREENDSWITCH

```
#define SEARCHINGPREENDSWITCH 2
```

Definition at line 125 of file ftypes.h.

4.53.2.50 PREPROONLY

```
#define PREPROONLY 1
```

Definition at line 127 of file ftypes.h.

4.53.2.51 DUMPTOCOMPILER

```
#define DUMPTOCOMPILER 2
```

Definition at line 128 of file ftypes.h.

4.53.2.52 DUMPOUTTERMS

```
#define DUMPOUTTERMS 4
```

Definition at line 129 of file ftypes.h.

4.53.2.53 DUMPINTERMS

```
#define DUMPINTERMS 8
```

Definition at line 130 of file ftypes.h.

4.53.2.54 DUMPTOSORT

```
#define DUMPTOSORT 16
```

Definition at line 131 of file ftypes.h.

4.53.2.55 DUMPTOPARALLEL

```
#define DUMPTOPARALLEL 32
```

Definition at line 132 of file ftypes.h.

4.53.2.56 THREADSDEBUG

```
#define THREADSDEBUG 64
```

Definition at line 133 of file ftypes.h.

4.53.2.57 ERROROUT

```
#define ERROROUT 0
```

Definition at line 135 of file ftypes.h.

4.53.2.58 INPUTOUT

```
#define INPUTOUT 1
```

Definition at line 136 of file ftypes.h.

4.53.2.59 STATSOUT

```
#define STATSOUT 2
```

Definition at line 137 of file ftypes.h.

4.53.2.60 EXPRSOUT

```
#define EXPRSOUT 3
```

Definition at line 138 of file ftypes.h.

4.53.2.61 WRITEOUT

```
#define WRITEOUT 4
```

Definition at line 139 of file ftypes.h.

4.53.2.62 EXTERNALCHANNELOUT

```
#define EXTERNALCHANNELOUT 5
```

Definition at line 141 of file ftypes.h.

4.53.2.63 NUMERICALLOOP

```
#define NUMERICALLOOP 0
```

Definition at line 143 of file ftypes.h.

4.53.2.64 LISTEDLOOP

```
#define LISTEDLOOP 1
```

Definition at line 144 of file ftypes.h.

4.53.2.65 ONEEXPRESSION

```
#define ONEEXPRESSION 2
```

Definition at line 145 of file ftypes.h.

4.53.2.66 PRETYPENONE

```
#define PRETYPENONE 0
```

Definition at line 147 of file ftypes.h.

4.53.2.67 PRETYPEIF

```
#define PRETYPEIF 1
```

Definition at line 148 of file ftypes.h.

4.53.2.68 PRETYPEDO

```
#define PRETYPEDO 2
```

Definition at line 149 of file ftypes.h.

4.53.2.69 PRETYPEPROCEDURE

```
#define PRETYPEPROCEDURE 3
```

Definition at line 150 of file ftypes.h.

4.53.2.70 PRETYPESWITCH

```
#define PRETYPESWITCH 4
```

Definition at line 151 of file ftypes.h.

4.53.2.71 PRETYPEINSIDE

```
#define PRETYPEINSIDE 5
```

Definition at line 152 of file ftypes.h.

4.53.2.72 DECLARATION

```
#define DECLARATION 1
```

Definition at line 158 of file ftypes.h.

4.53.2.73 SPECIFICATION

```
#define SPECIFICATION 2
```

Definition at line 159 of file ftypes.h.

4.53.2.74 DEFINITION

```
#define DEFINITION 3
```

Definition at line 160 of file ftypes.h.

4.53.2.75 STATEMENT

```
#define STATEMENT 4
```

Definition at line 161 of file ftypes.h.

4.53.2.76 TOOUTPUT

```
#define TOOUTPUT 5
```

Definition at line 162 of file ftypes.h.

4.53.2.77 ATENDOFMODULE

```
#define ATENDOFMODULE 6
```

Definition at line 163 of file ftypes.h.

4.53.2.78 MIXED2

```
#define MIXED2 8 /* unused */
```

Definition at line 164 of file ftypes.h.

4.53.2.79 MIXED

```
#define MIXED 9
```

Definition at line 165 of file ftypes.h.

4.53.2.80 NOAUTO

```
#define NOAUTO 0
```

Definition at line 181 of file ftypes.h.

4.53.2.81 PARTEST

```
#define PARTEST 1 /* parentheses test */
```

Definition at line 182 of file ftypes.h.

4.53.2.82 WITHAUTO

```
#define WITHAUTO 2 /* auto-declaration */
```

Definition at line 183 of file ftypes.h.

4.53.2.83 ALLVARIABLES

```
#define ALLVARIABLES -1
```

Definition at line 185 of file ftypes.h.

4.53.2.84 SYMBOLONLY

```
#define SYMBOLONLY 1
```

Definition at line 186 of file ftypes.h.

4.53.2.85 INDEXONLY

```
#define INDEXONLY 2
```

Definition at line 187 of file ftypes.h.

4.53.2.86 VECTORONLY

```
#define VECTORONLY 4
```

Definition at line 188 of file ftypes.h.

4.53.2.87 FUNCTIONONLY

```
#define FUNCTIONONLY 8
```

Definition at line 189 of file ftypes.h.

4.53.2.88 SETONLY

```
#define SETONLY 16
```

Definition at line 190 of file ftypes.h.

4.53.2.89 EXPRESSIONONLY

```
#define EXPRESSIONONLY 32
```

Definition at line 191 of file ftypes.h.

4.53.2.90 CDELETE

```
#define CDELETE -1
```

Definition at line 202 of file ftypes.h.

4.53.2.91 ANYTYPE

```
#define ANYTYPE -1
```

Definition at line 203 of file ftypes.h.

4.53.2.92 CSYMBOL

```
#define CSYMBOL 0
```

Definition at line 204 of file ftypes.h.

4.53.2.93 CINDEX

```
#define CINDEX 1
```

Definition at line 205 of file ftypes.h.

4.53.2.94 CVECTOR

```
#define CVECTOR 2
```

Definition at line 206 of file ftypes.h.

4.53.2.95 CFUNCTION

```
#define CFUNCTION 3
```

Definition at line 207 of file ftypes.h.

4.53.2.96 CSET

```
#define CSET 4
```

Definition at line 208 of file ftypes.h.

4.53.2.97 CEXPRESSION

```
#define CEXPRESSION 5
```

Definition at line 209 of file ftypes.h.

4.53.2.98 CDOTPRODUCT

```
#define CDOTPRODUCT 6
```

Definition at line 210 of file ftypes.h.

4.53.2.99 CNUMBER

```
#define CNUMBER 7
```

Definition at line 211 of file ftypes.h.

4.53.2.100 CSUBEXP

```
#define CSUBEXP 8
```

Definition at line 212 of file ftypes.h.

4.53.2.101 CDELTA

```
#define CDELTA 9
```

Definition at line 213 of file ftypes.h.

4.53.2.102 CDOLLAR

```
#define CDOLLAR 10
```

Definition at line 214 of file ftypes.h.

4.53.2.103 CDUBIOUS

```
#define CDUBIOUS 11
```

Definition at line 215 of file ftypes.h.

4.53.2.104 CRANGE

```
#define CRANGE 12
```

Definition at line 216 of file ftypes.h.

4.53.2.105 CMODEL

```
#define CMODEL 13
```

Definition at line 217 of file ftypes.h.

4.53.2.106 CVECTOR1

```
#define CVECTOR1 21
```

Definition at line 218 of file ftypes.h.

4.53.2.107 CDOUBLEDOT

```
#define CDOUBLEDOT 22
```

Definition at line 219 of file ftypes.h.

4.53.2.108 TSYMBOL

```
#define TSYMBOL -1
```

Definition at line 227 of file ftypes.h.

4.53.2.109 TINDEX

```
#define TINDEX -2
```

Definition at line 228 of file ftypes.h.

4.53.2.110 TVECTOR

```
#define TVECTOR -3
```

Definition at line 229 of file ftypes.h.

4.53.2.111 TFUNCTION

```
#define TFUNCTION -4
```

Definition at line 230 of file ftypes.h.

4.53.2.112 TSET

```
#define TSET -5
```

Definition at line 231 of file ftypes.h.

4.53.2.113 TEXPRESSION

```
#define TEXPRESSION -6
```

Definition at line 232 of file ftypes.h.

4.53.2.114 TDOTPRODUCT

```
#define TDOTPRODUCT -7
```

Definition at line 233 of file ftypes.h.

4.53.2.115 TNUMBER

```
#define TNUMBER -8
```

Definition at line 234 of file ftypes.h.

4.53.2.116 TSUBEXP

```
#define TSUBEXP -9
```

Definition at line 235 of file ftypes.h.

4.53.2.117 TDELTA

```
#define TDELTA -10
```

Definition at line 236 of file ftypes.h.

4.53.2.118 TDOLLAR

```
#define TDOLLAR -11
```

Definition at line 237 of file ftypes.h.

4.53.2.119 TDUBIOUS

```
#define TDUBIOUS -12
```

Definition at line 238 of file ftypes.h.

4.53.2.120 LPARENTHESIS

```
#define LPARENTHESIS -13
```

Definition at line 239 of file ftypes.h.

4.53.2.121 RPARENTHESIS

```
#define RPARENTHESIS -14
```

Definition at line 240 of file ftypes.h.

4.53.2.122 TWILDCARD

```
#define TWILDCARD -15
```

Definition at line 241 of file ftypes.h.

4.53.2.123 TWILDARG

```
#define TWILDARG -16
```

Definition at line 242 of file ftypes.h.

4.53.2.124 TDOT

```
#define TDOT -17
```

Definition at line 243 of file ftypes.h.

4.53.2.125 LBRACE

```
#define LBRACE -18
```

Definition at line 244 of file ftypes.h.

4.53.2.126 RBRACE

```
#define RBRACE -19
```

Definition at line 245 of file ftypes.h.

4.53.2.127 TCOMMA

```
#define TCOMMA -20
```

Definition at line 246 of file ftypes.h.

4.53.2.128 TFUNOPEN

```
#define TFUNOPEN -21
```

Definition at line 247 of file ftypes.h.

4.53.2.129 TFUNCLOSE

```
#define TFUNCLOSE -22
```

Definition at line 248 of file ftypes.h.

4.53.2.130 TMULTIPLY

```
#define TMULTIPLY -23
```

Definition at line 249 of file ftypes.h.

4.53.2.131 TDIVIDE

```
#define TDIVIDE -24
```

Definition at line 250 of file ftypes.h.

4.53.2.132 TPOWER

```
#define TPOWER -25
```

Definition at line 251 of file ftypes.h.

4.53.2.133 TPLUS

```
#define TPLUS -26
```

Definition at line 252 of file ftypes.h.

4.53.2.134 TMINUS

```
#define TMINUS -27
```

Definition at line 253 of file ftypes.h.

4.53.2.135 TNOT

```
#define TNOT -28
```

Definition at line 254 of file ftypes.h.

4.53.2.136 TENDOFIT

```
#define TENDOFIT -29
```

Definition at line 255 of file ftypes.h.

4.53.2.137 TSETOPEN

```
#define TSETOPEN -30
```

Definition at line 256 of file ftypes.h.

4.53.2.138 TSETCLOSE

```
#define TSETCLOSE -31
```

Definition at line 257 of file ftypes.h.

4.53.2.139 TGENINDEX

```
#define TGENINDEX -32
```

Definition at line 258 of file ftypes.h.

4.53.2.140 TCONJUGATE

```
#define TCONJUGATE -33
```

Definition at line 259 of file ftypes.h.

4.53.2.141 LRPARENTHESES

```
#define LRPARENTHESES -34
```

Definition at line 260 of file ftypes.h.

4.53.2.142 TNUMBER1

```
#define TNUMBER1 -35
```

Definition at line 261 of file ftypes.h.

4.53.2.143 TPOWER1

```
#define TPOWER1 -36
```

Definition at line 262 of file ftypes.h.

4.53.2.144 TEMPTY

```
#define TEMPTY -37
```

Definition at line 263 of file ftypes.h.

4.53.2.145 TSETNUM

```
#define TSETNUM -38
```

Definition at line 264 of file ftypes.h.

4.53.2.146 TSGAMMA

```
#define TSGAMMA -39
```

Definition at line 265 of file ftypes.h.

4.53.2.147 TSETDOL

```
#define TSETDOL -40
```

Definition at line 266 of file ftypes.h.

4.53.2.148 TFLOAT

```
#define TFLOAT -41
```

Definition at line 267 of file ftypes.h.

4.53.2.149 TYPEISFUN

```
#define TYPEISFUN 0
```

Definition at line 269 of file ftypes.h.

4.53.2.150 TYPEISSUB

```
#define TYPEISSUB 1
```

Definition at line 270 of file ftypes.h.

4.53.2.151 TYPEISMYSTERY

```
#define TYPEISMYSTERY -1
```

Definition at line 271 of file ftypes.h.

4.53.2.152 LHSIDEX

```
#define LHSIDEX 2
```

Definition at line 273 of file ftypes.h.

4.53.2.153 LHSIDE

```
#define LHSIDE 1
```

Definition at line 274 of file ftypes.h.

4.53.2.154 RHSIDE

```
#define RHSIDE 0
```

Definition at line 275 of file ftypes.h.

4.53.2.155 FORTRANMODE

```
#define FORTRANMODE 1
```

Definition at line 281 of file ftypes.h.

4.53.2.156 REDUCEMODE

```
#define REDUCEMODE 2
```

Definition at line 282 of file ftypes.h.

4.53.2.157 MAPLEMODE

```
#define MAPLEMODE 3
```

Definition at line 283 of file ftypes.h.

4.53.2.158 MATHEMATICAMODE

```
#define MATHEMATICAMODE 4
```

Definition at line 284 of file ftypes.h.

4.53.2.159 CMODE

```
#define CMODE 5
```

Definition at line 285 of file ftypes.h.

4.53.2.160 VORTRANMODE

```
#define VORTRANMODE 6
```

Definition at line 286 of file ftypes.h.

4.53.2.161 PFORTRANMODE

```
#define PFORTRANMODE 7
```

Definition at line 287 of file ftypes.h.

4.53.2.162 DOUBLEFORTRANMODE

```
#define DOUBLEFORTRANMODE 33
```

Definition at line 288 of file ftypes.h.

4.53.2.163 DOUBLEPRECISIONFLAG

```
#define DOUBLEPRECISIONFLAG 32
```

Definition at line 289 of file ftypes.h.

4.53.2.164 NODOUBLEMASK

```
#define NODOUBLEMASK 31
```

Definition at line 290 of file ftypes.h.

4.53.2.165 QUADRUPLEFORTRANMODE

```
#define QUADRUPLEFORTRANMODE 65
```

Definition at line 291 of file ftypes.h.

4.53.2.166 QUADRUPLEPRECISIONFLAG

```
#define QUADRUPLEPRECISIONFLAG 64
```

Definition at line 292 of file ftypes.h.

4.53.2.167 ALLINTEGERDOUBLE

```
#define ALLINTEGERDOUBLE 128
```

Definition at line 293 of file ftypes.h.

4.53.2.168 NOQUADMASK

```
#define NOQUADMASK 63
```

Definition at line 294 of file ftypes.h.

4.53.2.169 NORMALFORMAT

```
#define NORMALFORMAT 0
```

Definition at line 295 of file ftypes.h.

4.53.2.170 NOSPACEFORMAT

```
#define NOSPACEFORMAT 1
```

Definition at line 296 of file ftypes.h.

4.53.2.171 ISNOTFORTRAN90

```
#define ISNOTFORTRAN90 0
```

Definition at line 298 of file ftypes.h.

4.53.2.172 ISFORTRAN90

```
#define ISFORTRAN90 1
```

Definition at line 299 of file ftypes.h.

4.53.2.173 ALSOREVERSE

```
#define ALSOREVERSE 1
```

Definition at line 301 of file ftypes.h.

4.53.2.174 CHISHOLM

```
#define CHISHOLM 2
```

Definition at line 302 of file ftypes.h.

4.53.2.175 NOTRICK

```
#define NOTRICK 16
```

Definition at line 303 of file ftypes.h.

4.53.2.176 SORTLOWFIRST

```
#define SORTLOWFIRST 0
```

Definition at line 305 of file ftypes.h.

4.53.2.177 SORTHIGHFIRST

```
#define SORTHIGHFIRST 1
```

Definition at line 306 of file ftypes.h.

4.53.2.178 SORTPOWERFIRST

```
#define SORTPOWERFIRST 2
```

Definition at line 307 of file ftypes.h.

4.53.2.179 SORTANTIPOWER

```
#define SORTANTIPOWER 3
```

Definition at line 308 of file ftypes.h.

4.53.2.180 STATSSPLITMERGE

```
#define STATSSPLITMERGE 0
```

Definition at line 314 of file ftypes.h.

4.53.2.181 STATSMERGETOFILE

```
#define STATSMERGETOFILE 1
```

Definition at line 315 of file ftypes.h.

4.53.2.182 STATSPOSTSORT

```
#define STATSPOSTSORT 2
```

Definition at line 316 of file ftypes.h.

4.53.2.183 NOCHECKLOGTYPE

```
#define NOCHECKLOGTYPE 0
```

Definition at line 319 of file ftypes.h.

4.53.2.184 CHECKLOGTYPE

```
#define CHECKLOGTYPE 1
```

Definition at line 320 of file ftypes.h.

4.53.2.185 NMIN4SHIFT

```
#define NMIN4SHIFT 4
```

Definition at line 322 of file ftypes.h.

4.53.2.186 SYMBOL

```
#define SYMBOL 1
```

Definition at line 336 of file ftypes.h.

4.53.2.187 DOTPRODUCT

```
#define DOTPRODUCT 2
```

Definition at line 337 of file ftypes.h.

4.53.2.188 VECTOR

```
#define VECTOR 3
```

Definition at line 338 of file ftypes.h.

4.53.2.189 INDEX

```
#define INDEX 4
```

Definition at line 339 of file ftypes.h.

4.53.2.190 EXPRESSION

```
#define EXPRESSION 5
```

Definition at line 340 of file ftypes.h.

4.53.2.191 SUBEXPRESSION

```
#define SUBEXPRESSION 6
```

Definition at line 341 of file ftypes.h.

4.53.2.192 DOLLAREXPRESSION

```
#define DOLLAREXPRESSION 7
```

Definition at line 342 of file ftypes.h.

4.53.2.193 SETSET

```
#define SETSET 8
```

Definition at line 343 of file ftypes.h.

4.53.2.194 ARGWILD

```
#define ARGWILD 9
```

Definition at line 344 of file ftypes.h.

4.53.2.195 MINVECTOR

```
#define MINVECTOR 10
```

Definition at line 345 of file ftypes.h.

4.53.2.196 SETEXP

```
#define SETEXP 11
```

Definition at line 346 of file ftypes.h.

4.53.2.197 DOLLAREXPR2

```
#define DOLLAREXPR2 12
```

Definition at line 347 of file ftypes.h.

4.53.2.198 HAAKJE0

```
#define HAAKJE0 9
```

Definition at line 348 of file ftypes.h.

4.53.2.199 FUNCTION

```
#define FUNCTION 20
```

Definition at line 349 of file ftypes.h.

4.53.2.200 TMPPOLYFUN

```
#define TMPPOLYFUN 14
```

Definition at line 351 of file ftypes.h.

4.53.2.201 ARGFIELD

```
#define ARGFIELD 15
```

Definition at line 352 of file ftypes.h.

4.53.2.202 SNUMBER

```
#define SNUMBER 16
```

Definition at line 353 of file ftypes.h.

4.53.2.203 LNUMBER

```
#define LNUMBER 17
```

Definition at line 354 of file ftypes.h.

4.53.2.204 HAAKJE

```
#define HAAKJE 18
```

Definition at line 355 of file ftypes.h.

4.53.2.205 DELTA

```
#define DELTA 19
```

Definition at line 356 of file ftypes.h.

4.53.2.206 EXPONENT

```
#define EXPONENT 20
```

Definition at line 357 of file ftypes.h.

4.53.2.207 DENOMINATOR

```
#define DENOMINATOR 21
```

Definition at line 358 of file ftypes.h.

4.53.2.208 SETFUNCTION

```
#define SETFUNCTION 22
```

Definition at line 359 of file ftypes.h.

4.53.2.209 GAMMA

```
#define GAMMA 23
```

Definition at line 360 of file ftypes.h.

4.53.2.210 GAMMAI

```
#define GAMMAI 24
```

Definition at line 361 of file ftypes.h.

4.53.2.211 GAMMAFIVE

```
#define GAMMAFIVE 25
```

Definition at line 362 of file ftypes.h.

4.53.2.212 GAMMASIX

```
#define GAMMASIX 26
```

Definition at line 363 of file ftypes.h.

4.53.2.213 GAMMASEVEN

```
#define GAMMASEVEN 27
```

Definition at line 364 of file ftypes.h.

4.53.2.214 SUMF1

```
#define SUMF1 28
```

Definition at line 365 of file ftypes.h.

4.53.2.215 SUMF2

```
#define SUMF2 29
```

Definition at line 366 of file ftypes.h.

4.53.2.216 DUMFUN

```
#define DUMFUN 30
```

Definition at line 367 of file ftypes.h.

4.53.2.217 REPLACEMENT

```
#define REPLACEMENT 31
```

Definition at line 368 of file ftypes.h.

4.53.2.218 REVERSEFUNCTION

```
#define REVERSEFUNCTION 32
```

Definition at line 369 of file ftypes.h.

4.53.2.219 DISTRIBUTION

```
#define DISTRIBUTION 33
```

Definition at line 370 of file ftypes.h.

4.53.2.220 DELTA3

```
#define DELTA3 34
```

Definition at line 371 of file ftypes.h.

4.53.2.221 DUMMYFUN

```
#define DUMMYFUN 35
```

Definition at line 372 of file ftypes.h.

4.53.2.222 DUMMYTEN

```
#define DUMMYTEN 36
```

Definition at line 373 of file ftypes.h.

4.53.2.223 LEVICIVITA

```
#define LEVICIVITA 37
```

Definition at line 374 of file ftypes.h.

4.53.2.224 FACTORIAL

```
#define FACTORIAL 38
```

Definition at line 375 of file ftypes.h.

4.53.2.225 INVERSEFACTORIAL

```
#define INVERSEFACTORIAL 39
```

Definition at line 376 of file ftypes.h.

4.53.2.226 BINOMIAL

```
#define BINOMIAL 40
```

Definition at line 377 of file ftypes.h.

4.53.2.227 NUMARGSFUN

```
#define NUMARGSFUN 41
```

Definition at line 378 of file ftypes.h.

4.53.2.228 SIGNFUN

```
#define SIGNFUN 42
```

Definition at line 379 of file ftypes.h.

4.53.2.229 MODFUNCTION

```
#define MODFUNCTION 43
```

Definition at line 380 of file ftypes.h.

4.53.2.230 MOD2FUNCTION

```
#define MOD2FUNCTION 44
```

Definition at line 381 of file ftypes.h.

4.53.2.231 MINFUNCTION

```
#define MINFUNCTION 45
```

Definition at line 382 of file ftypes.h.

4.53.2.232 MAXFUNCTION

```
#define MAXFUNCTION 46
```

Definition at line 383 of file ftypes.h.

4.53.2.233 ABSFUNCTION

```
#define ABSFUNCTION 47
```

Definition at line 384 of file ftypes.h.

4.53.2.234 SIGFUNCTION

```
#define SIGFUNCTION 48
```

Definition at line 385 of file ftypes.h.

4.53.2.235 INTFUNCTION

```
#define INTFUNCTION 49
```

Definition at line 386 of file ftypes.h.

4.53.2.236 THETA

```
#define THETA 50
```

Definition at line 387 of file ftypes.h.

4.53.2.237 THETA2

```
#define THETA2 51
```

Definition at line 388 of file ftypes.h.

4.53.2.238 DELTA2

```
#define DELTA2 52
```

Definition at line 389 of file ftypes.h.

4.53.2.239 DELTAP

```
#define DELTAP 53
```

Definition at line 390 of file ftypes.h.

4.53.2.240 BERNOULLIFUNCTION

```
#define BERNOULLIFUNCTION 54
```

Definition at line 391 of file ftypes.h.

4.53.2.241 COUNTFUNCTION

```
#define COUNTFUNCTION 55
```

Definition at line 392 of file ftypes.h.

4.53.2.242 MATCHFUNCTION

```
#define MATCHFUNCTION 56
```

Definition at line 393 of file ftypes.h.

4.53.2.243 PATTERNFUNCTION

```
#define PATTERNFUNCTION 57
```

Definition at line 394 of file ftypes.h.

4.53.2.244 TERMFUNCTION

```
#define TERMFUNCTION 58
```

Definition at line 395 of file ftypes.h.

4.53.2.245 CONJUGATION

```
#define CONJUGATION 59
```

Definition at line 396 of file ftypes.h.

4.53.2.246 ROOTFUNCTION

```
#define ROOTFUNCTION 60
```

Definition at line 397 of file ftypes.h.

4.53.2.247 TABLEFUNCTION

```
#define TABLEFUNCTION 61
```

Definition at line 398 of file ftypes.h.

4.53.2.248 FIRSTBRACKET

```
#define FIRSTBRACKET 62
```

Definition at line 399 of file ftypes.h.

4.53.2.249 TERMSINEXPR

```
#define TERMSINEXPR 63
```

Definition at line 400 of file ftypes.h.

4.53.2.250 NUMTERMSFUN

```
#define NUMTERMSFUN 64
```

Definition at line 401 of file ftypes.h.

4.53.2.251 GCDFUNCTION

```
#define GCDFUNCTION 65
```

Definition at line 402 of file ftypes.h.

4.53.2.252 DIVFUNCTION

```
#define DIVFUNCTION 66
```

Definition at line 403 of file ftypes.h.

4.53.2.253 REMFUNCTION

```
#define REMFUNCTION 67
```

Definition at line 404 of file ftypes.h.

4.53.2.254 MAXPOWEROF

```
#define MAXPOWEROF 68
```

Definition at line 405 of file ftypes.h.

4.53.2.255 MINPOWEROF

```
#define MINPOWEROF 69
```

Definition at line 406 of file ftypes.h.

4.53.2.256 TABLESTUB

```
#define TABLESTUB 70
```

Definition at line 407 of file ftypes.h.

4.53.2.257 FACTORIN

```
#define FACTORIN 71
```

Definition at line 408 of file ftypes.h.

4.53.2.258 TERMSINBRACKET

```
#define TERMSINBRACKET 72
```

Definition at line 409 of file ftypes.h.

4.53.2.259 WILDARGFUN

```
#define WILDARGFUN 73
```

Definition at line 410 of file ftypes.h.

4.53.2.260 SQRTFUNCTION

```
#define SQRTFUNCTION 74
```

Definition at line 418 of file ftypes.h.

4.53.2.261 LNFUNCTION

```
#define LNFUNCTION 75
```

Definition at line 419 of file ftypes.h.

4.53.2.262 SINFUNCTION

```
#define SINFUNCTION 76
```

Definition at line 420 of file ftypes.h.

4.53.2.263 COSFUNCTION

```
#define COSFUNCTION 77
```

Definition at line 421 of file ftypes.h.

4.53.2.264 TANFUNCTION

```
#define TANFUNCTION 78
```

Definition at line 422 of file ftypes.h.

4.53.2.265 ASINFUNCTION

```
#define ASINFUNCTION 79
```

Definition at line 423 of file ftypes.h.

4.53.2.266 ACOSFUNCTION

```
#define ACOSFUNCTION 80
```

Definition at line 424 of file ftypes.h.

4.53.2.267 ATANFUNCTION

```
#define ATANFUNCTION 81
```

Definition at line 425 of file ftypes.h.

4.53.2.268 ATAN2FUNCTION

```
#define ATAN2FUNCTION 82
```

Definition at line 426 of file ftypes.h.

4.53.2.269 SINHFUNCTION

```
#define SINHFUNCTION 83
```

Definition at line 427 of file ftypes.h.

4.53.2.270 COSHFUNCTION

```
#define COSHFUNCTION 84
```

Definition at line 428 of file ftypes.h.

4.53.2.271 TANHFUNCTION

```
#define TANHFUNCTION 85
```

Definition at line 429 of file ftypes.h.

4.53.2.272 ASINHFUNCTION

```
#define ASINHFUNCTION 86
```

Definition at line 430 of file ftypes.h.

4.53.2.273 ACOSHFUNCTION

```
#define ACOSHFUNCTION 87
```

Definition at line 431 of file ftypes.h.

4.53.2.274 ATANHFUNCTION

```
#define ATANHFUNCTION 88
```

Definition at line 432 of file ftypes.h.

4.53.2.275 LI2FUNCTION

```
#define LI2FUNCTION 89
```

Definition at line 433 of file ftypes.h.

4.53.2.276 LINFUNCTION

```
#define LINFUNCTION 90
```

Definition at line 434 of file ftypes.h.

4.53.2.277 EXTRASYMFUN

```
#define EXTRASYMFUN 91
```

Definition at line 436 of file ftypes.h.

4.53.2.278 RANDOMFUNCTION

```
#define RANDOMFUNCTION 92
```

Definition at line 437 of file ftypes.h.

4.53.2.279 RANPERM

```
#define RANPERM 93
```

Definition at line 438 of file ftypes.h.

4.53.2.280 NUMFACTORS

```
#define NUMFACTORS 94
```

Definition at line 439 of file ftypes.h.

4.53.2.281 FIRSTTERM

```
#define FIRSTTERM 95
```

Definition at line 440 of file ftypes.h.

4.53.2.282 CONTENTTERM

```
#define CONTENTTERM 96
```

Definition at line 441 of file ftypes.h.

4.53.2.283 PRIMENUMBER

```
#define PRIMENUMBER 97
```

Definition at line 442 of file ftypes.h.

4.53.2.284 EXTEUCLIDEAN

```
#define EXTEUCLIDEAN 98
```

Definition at line 443 of file ftypes.h.

4.53.2.285 MAKERATIONAL

```
#define MAKERATIONAL 99
```

Definition at line 444 of file ftypes.h.

4.53.2.286 INVERSEFUNCTION

```
#define INVERSEFUNCTION 100
```

Definition at line 445 of file ftypes.h.

4.53.2.287 IDFUNCTION

```
#define IDFUNCTION 101
```

Definition at line 446 of file ftypes.h.

4.53.2.288 PUTFIRST

```
#define PUTFIRST 102
```

Definition at line 447 of file ftypes.h.

4.53.2.289 PERMUTATIONS

```
#define PERMUTATIONS 103
```

Definition at line 448 of file ftypes.h.

4.53.2.290 PARTITIONS

```
#define PARTITIONS 104
```

Definition at line 449 of file ftypes.h.

4.53.2.291 MULFUNCTION

```
#define MULFUNCTION 105
```

Definition at line 450 of file ftypes.h.

4.53.2.292 MOEBIUS

```
#define MOEBIUS 106
```

Definition at line 451 of file ftypes.h.

4.53.2.293 TOPOLOGIES

```
#define TOPOLOGIES 107
```

Definition at line 452 of file ftypes.h.

4.53.2.294 DIAGRAMS

```
#define DIAGRAMS 108
```

Definition at line 453 of file ftypes.h.

4.53.2.295 TOPO

```
#define TOPO 109
```

Definition at line 454 of file ftypes.h.

4.53.2.296 NODEFUNCTION

```
#define NODEFUNCTION 110
```

Definition at line 455 of file ftypes.h.

4.53.2.297 EDGE

```
#define EDGE 111
```

Definition at line 456 of file ftypes.h.

4.53.2.298 SIZEOFFUNCTION

```
#define SIZEOFFUNCTION 112
```

Definition at line 457 of file ftypes.h.

4.53.2.299 BLOCK

```
#define BLOCK 113
```

Definition at line 458 of file ftypes.h.

4.53.2.300 ONEPI

```
#define ONEPI 114
```

Definition at line 459 of file ftypes.h.

4.53.2.301 PHI

```
#define PHI 115
```

Definition at line 460 of file ftypes.h.

4.53.2.302 FLOATFUN

```
#define FLOATFUN 116
```

Definition at line 461 of file ftypes.h.

4.53.2.303 TOFLOAT

```
#define TOFLOAT 117
```

Definition at line 462 of file ftypes.h.

4.53.2.304 TORAT

```
#define TORAT 118
```

Definition at line 463 of file ftypes.h.

4.53.2.305 MZV

```
#define MZV 119
```

Definition at line 464 of file ftypes.h.

4.53.2.306 EULER

```
#define EULER 120
```

Definition at line 465 of file ftypes.h.

4.53.2.307 MZVHALF

```
#define MZVHALF 121
```

Definition at line 466 of file ftypes.h.

4.53.2.308 AGMFUNCTION

```
#define AGMFUNCTION 122
```

Definition at line 467 of file ftypes.h.

4.53.2.309 GAMMAFUN

```
#define GAMMAFUN 123
```

Definition at line 468 of file ftypes.h.

4.53.2.310 EXPFUNCTION

```
#define EXPFUNCTION 124
```

Definition at line 469 of file ftypes.h.

4.53.2.311 HPLFUNCTION

```
#define HPLFUNCTION 125
```

Definition at line 470 of file ftypes.h.

4.53.2.312 MPLFUNCTION

```
#define MPLFUNCTION 126
```

Definition at line 471 of file ftypes.h.

4.53.2.313 MAXBUILTINFUNCTION

```
#define MAXBUILTINFUNCTION 126
```

Definition at line 472 of file ftypes.h.

4.53.2.314 FIRSTUSERFUNCTION

```
#define FIRSTUSERFUNCTION 150
```

Definition at line 474 of file ftypes.h.

4.53.2.315 ALLFUNCTIONS

```
#define ALLFUNCTIONS -1
```

Definition at line 476 of file ftypes.h.

4.53.2.316 ALLMZVFUNCTIONS

```
#define ALLMZVFUNCTIONS -2
```

Definition at line 477 of file ftypes.h.

4.53.2.317 ISYMBOL

```
#define ISYMBOL 0
```

Definition at line 485 of file ftypes.h.

4.53.2.318 PISYMBOL

```
#define PISYMBOL 1
```

Definition at line 486 of file ftypes.h.

4.53.2.319 COEFFSYMBOL

```
#define COEFFSYMBOL 2
```

Definition at line 487 of file ftypes.h.

4.53.2.320 NUMERATORSYMBOL

```
#define NUMERATORSYMBOL 3
```

Definition at line 488 of file ftypes.h.

4.53.2.321 DENOMINATORSYMBOL

```
#define DENOMINATORSYMBOL 4
```

Definition at line 489 of file ftypes.h.

4.53.2.322 WILDARGSYMBOL

```
#define WILDARGSYMBOL 5
```

Definition at line 490 of file ftypes.h.

4.53.2.323 DIMENSIONSYMBOL

```
#define DIMENSIONSYMBOL 6
```

Definition at line 491 of file ftypes.h.

4.53.2.324 FACTORSYMBOL

```
#define FACTORSYMBOL 7
```

Definition at line 492 of file ftypes.h.

4.53.2.325 SEPARATESYMBOL

```
#define SEPARATESYMBOL 8
```

Definition at line 493 of file ftypes.h.

4.53.2.326 EESYMBOL

```
#define EESYMBOL 9
```

Definition at line 494 of file ftypes.h.

4.53.2.327 EMSYMBOL

```
#define EMSYMBOL 10
```

Definition at line 495 of file ftypes.h.

4.53.2.328 BUILTINSYMBOLS

```
#define BUILTINSYMBOLS 11
```

Definition at line 497 of file ftypes.h.

4.53.2.329 FIRSTUSERSYMBOL

```
#define FIRSTUSERSYMBOL 20
```

Definition at line 498 of file ftypes.h.

4.53.2.330 BUILTININDICES

```
#define BUILTININDICES 1
```

Definition at line 500 of file ftypes.h.

4.53.2.331 BUILTINVECTORS

```
#define BUILTINVECTORS 1
```

Definition at line 501 of file ftypes.h.

4.53.2.332 BUILTINDOLLARS

```
#define BUILTINDOLLARS 1
```

Definition at line 502 of file ftypes.h.

4.53.2.333 WILDARGVECTOR

```
#define WILDARGVECTOR 0
```

Definition at line 504 of file ftypes.h.

4.53.2.334 WILDARGINDEX

```
#define WILDARGINDEX 0
```

Definition at line 505 of file ftypes.h.

4.53.2.335 TYPEEXPRESSION

```
#define TYPEEXPRESSION 0
```

Definition at line 516 of file ftypes.h.

4.53.2.336 TYPEIDNEW

```
#define TYPEIDNEW 1
```

Definition at line 517 of file ftypes.h.

4.53.2.337 TYPEIDOLD

```
#define TYPEIDOLD 2
```

Definition at line 518 of file ftypes.h.

4.53.2.338 TYPEOPERATION

```
#define TYPEOPERATION 3
```

Definition at line 519 of file ftypes.h.

4.53.2.339 TYPEREPEAT

```
#define TYPEREPEAT 4
```

Definition at line 520 of file ftypes.h.

4.53.2.340 TYPEENDREPEAT

```
#define TYPEENDREPEAT 5
```

Definition at line 521 of file ftypes.h.

4.53.2.341 TYPECOUNT

```
#define TYPECOUNT 20
```

Definition at line 525 of file ftypes.h.

4.53.2.342 TYPEMULT

```
#define TYPEMULT 21
```

Definition at line 526 of file ftypes.h.

4.53.2.343 TYPEGOTO

```
#define TYPEGOTO 22
```

Definition at line 527 of file ftypes.h.

4.53.2.344 TYPEDISCARD

```
#define TYPEDISCARD 23
```

Definition at line 528 of file ftypes.h.

4.53.2.345 TYPEIF

```
#define TYPEIF 24
```

Definition at line 529 of file ftypes.h.

4.53.2.346 TYPEELSE

```
#define TYPEELSE 25
```

Definition at line 530 of file ftypes.h.

4.53.2.347 TYPEELIF

```
#define TYPEELIF 26
```

Definition at line 531 of file ftypes.h.

4.53.2.348 TYPEENDIF

```
#define TYPEENDIF 27
```

Definition at line 532 of file ftypes.h.

4.53.2.349 TYPESUM

```
#define TYPESUM 28
```

Definition at line 533 of file ftypes.h.

4.53.2.350 TYPECHISHOLM

```
#define TYPECHISHOLM 29
```

Definition at line 534 of file ftypes.h.

4.53.2.351 TYPEREVERSE

```
#define TYPEREVERSE 30
```

Definition at line 535 of file ftypes.h.

4.53.2.352 TYPEARG

```
#define TYPEARG 31
```

Definition at line 536 of file ftypes.h.

4.53.2.353 TYPENORM

```
#define TYPENORM 32
```

Definition at line 537 of file ftypes.h.

4.53.2.354 TYPENORM2

```
#define TYPENORM2 33
```

Definition at line 538 of file ftypes.h.

4.53.2.355 TYPENORM3

```
#define TYPENORM3 34
```

Definition at line 539 of file ftypes.h.

4.53.2.356 TYPEEXIT

```
#define TYPEEXIT 35
```

Definition at line 540 of file ftypes.h.

4.53.2.357 TYPESETEXIT

```
#define TYPESETEXIT 36
```

Definition at line 541 of file ftypes.h.

4.53.2.358 TYPEPRINT

```
#define TYPEPRINT 37
```

Definition at line 542 of file ftypes.h.

4.53.2.359 TYPEFPRINT

```
#define TYPEFPRINT 38
```

Definition at line 543 of file ftypes.h.

4.53.2.360 TYPEREDEFPRE

```
#define TYPEREDEFPRE 39
```

Definition at line 544 of file ftypes.h.

4.53.2.361 TYPESPLITARG

```
#define TYPESPLITARG 40
```

Definition at line 545 of file ftypes.h.

4.53.2.362 TYPESPLITARG2

```
#define TYPESPLITARG2 41
```

Definition at line 546 of file ftypes.h.

4.53.2.363 TYPEFACTARG

```
#define TYPEFACTARG 42
```

Definition at line 547 of file ftypes.h.

4.53.2.364 TYPEFACTARG2

```
#define TYPEFACTARG2 43
```

Definition at line 548 of file ftypes.h.

4.53.2.365 TYPETRY

```
#define TYPETRY 44
```

Definition at line 549 of file ftypes.h.

4.53.2.366 TYPEASSIGN

```
#define TYPEASSIGN 45
```

Definition at line 550 of file ftypes.h.

4.53.2.367 TYPERENUMBER

```
#define TYPERENUMBER 46
```

Definition at line 551 of file ftypes.h.

4.53.2.368 TYPESUMFIX

```
#define TYPESUMFIX 47
```

Definition at line 552 of file ftypes.h.

4.53.2.369 TYPEFINDLOOP

```
#define TYPEFINDLOOP 48
```

Definition at line 553 of file ftypes.h.

4.53.2.370 TYPEUNRAVEL

```
#define TYPEUNRAVEL 49
```

Definition at line 554 of file ftypes.h.

4.53.2.371 TYPEADJUSTBOUNDS

```
#define TYPEADJUSTBOUNDS 50
```

Definition at line 555 of file ftypes.h.

4.53.2.372 TYPEINSIDE

```
#define TYPEINSIDE 51
```

Definition at line 556 of file ftypes.h.

4.53.2.373 TYPETERM

```
#define TYPETERM 52
```

Definition at line 557 of file ftypes.h.

4.53.2.374 TYPESORT

```
#define TYPESORT 53
```

Definition at line 558 of file ftypes.h.

4.53.2.375 TYPEDETCURDUM

```
#define TYPEDETCURDUM 54
```

Definition at line 559 of file ftypes.h.

4.53.2.376 TYPEINEXPRESSION

```
#define TYPEINEXPRESSION 55
```

Definition at line 560 of file ftypes.h.

4.53.2.377 TYPESPLITFIRSTARG

```
#define TYPESPLITFIRSTARG 56
```

Definition at line 561 of file ftypes.h.

4.53.2.378 TYPESPLITLASTARG

```
#define TYPESPLITLASTARG 57
```

Definition at line 562 of file ftypes.h.

4.53.2.379 TYPEMERGE

```
#define TYPEMERGE 58
```

Definition at line 563 of file ftypes.h.

4.53.2.380 TYPETESTUSE

```
#define TYPETESTUSE 59
```

Definition at line 564 of file ftypes.h.

4.53.2.381 TYPEAPPLY

```
#define TYPEAPPLY 60
```

Definition at line 565 of file ftypes.h.

4.53.2.382 TYPEAPPLYRESET

```
#define TYPEAPPLYRESET 61
```

Definition at line 566 of file ftypes.h.

4.53.2.383 TYPECHAININ

```
#define TYPECHAININ 62
```

Definition at line 567 of file ftypes.h.

4.53.2.384 TYPECHAINOUT

```
#define TYPECHAINOUT 63
```

Definition at line 568 of file ftypes.h.

4.53.2.385 TYPENORM4

```
#define TYPENORM4 64
```

Definition at line 569 of file ftypes.h.

4.53.2.386 TYPEFACTOR

```
#define TYPEFACTOR 65
```

Definition at line 570 of file ftypes.h.

4.53.2.387 TYPEARGIMPLODE

```
#define TYPEARGIMPLODE 66
```

Definition at line 571 of file ftypes.h.

4.53.2.388 TYPEARGEXPLODE

```
#define TYPEARGEXPLODE 67
```

Definition at line 572 of file ftypes.h.

4.53.2.389 TYPEDENOMINATORS

```
#define TYPEDENOMINATORS 68
```

Definition at line 573 of file ftypes.h.

4.53.2.390 TYPESTUFFLE

```
#define TYPESTUFFLE 69
```

Definition at line 574 of file ftypes.h.

4.53.2.391 TYPEDROPCOEFFICIENT

```
#define TYPEDROPCOEFFICIENT 70
```

Definition at line 575 of file ftypes.h.

4.53.2.392 TYPETRANSFORM

```
#define TYPETRANSFORM 71
```

Definition at line 576 of file ftypes.h.

4.53.2.393 TYPETOPOLYNOMIAL

```
#define TYPETOPOLYNOMIAL 72
```

Definition at line 577 of file ftypes.h.

4.53.2.394 TYPEFROMPOLYNOMIAL

```
#define TYPEFROMPOLYNOMIAL 73
```

Definition at line 578 of file ftypes.h.

4.53.2.395 TYPEDOLOOP

```
#define TYPEDOLOOP 74
```

Definition at line 579 of file ftypes.h.

4.53.2.396 TYPEENDDOLOOP

```
#define TYPEENDDOLOOP 75
```

Definition at line 580 of file ftypes.h.

4.53.2.397 TYPEDROPSYMBOLS

```
#define TYPEDROPSYMBOLS 76
```

Definition at line 581 of file ftypes.h.

4.53.2.398 TYPEPUTINSIDE

```
#define TYPEPUTINSIDE 77
```

Definition at line 582 of file ftypes.h.

4.53.2.399 TYPETOSPECTATOR

```
#define TYPETOSPECTATOR 78
```

Definition at line 583 of file ftypes.h.

4.53.2.400 TYPEARGTOEXTRASYMBOL

```
#define TYPEARGTOEXTRASYMBOL 79
```

Definition at line 584 of file ftypes.h.

4.53.2.401 TYPECANONICALIZE

```
#define TYPECANONICALIZE 80
```

Definition at line 585 of file ftypes.h.

4.53.2.402 TYPESWITCH

```
#define TYPESWITCH 81
```

Definition at line 586 of file ftypes.h.

4.53.2.403 TYPEENDSWITCH

```
#define TYPEENDSWITCH 82
```

Definition at line 587 of file ftypes.h.

4.53.2.404 TYPESETUSERFLAG

```
#define TYPESETUSERFLAG 83
```

Definition at line 588 of file ftypes.h.

4.53.2.405 TYPECLEARUSERFLAG

```
#define TYPECLEARUSERFLAG 84
```

Definition at line 589 of file ftypes.h.

4.53.2.406 TYPEALLLOOPS

```
#define TYPEALLLOOPS 85
```

Definition at line 590 of file ftypes.h.

4.53.2.407 TYPEALLPATHS

```
#define TYPEALLPATHS 86
```

Definition at line 591 of file ftypes.h.

4.53.2.408 TAKETRACE

```
#define TAKETRACE 1
```

Definition at line 605 of file ftypes.h.

4.53.2.409 CONTRACT

```
#define CONTRACT 2
```

Definition at line 606 of file ftypes.h.

4.53.2.410 RATIO

```
#define RATIO 3
```

Definition at line 607 of file ftypes.h.

4.53.2.411 SYMMETRIZE

```
#define SYMMETRIZE 4
```

Definition at line 608 of file ftypes.h.

4.53.2.412 TENVEC

```
#define TENVEC 5
```

Definition at line 609 of file ftypes.h.

4.53.2.413 SUMNUM1

```
#define SUMNUM1 6
```

Definition at line 610 of file ftypes.h.

4.53.2.414 SUMNUM2

```
#define SUMNUM2 7
```

Definition at line 611 of file ftypes.h.

4.53.2.415 WILDDUMMY

```
#define WILDDUMMY 0
```

Definition at line 617 of file ftypes.h.

4.53.2.416 SYMTONUM

```
#define SYMTONUM 1
```

Definition at line 618 of file ftypes.h.

4.53.2.417 SYMTOSYM

```
#define SYMTOSYM 2
```

Definition at line 619 of file ftypes.h.

4.53.2.418 SYMTOSUB

```
#define SYMTOSUB 3
```

Definition at line 620 of file ftypes.h.

4.53.2.419 VECTOMIN

```
#define VECTOMIN 4
```

Definition at line 621 of file ftypes.h.

4.53.2.420 VECTOVEC

```
#define VECTOVEC 5
```

Definition at line 622 of file ftypes.h.

4.53.2.421 VECTOSUB

```
#define VECTOSUB 6
```

Definition at line 623 of file ftypes.h.

4.53.2.422 INDTOIND

```
#define INDTOIND 7
```

Definition at line 624 of file ftypes.h.

4.53.2.423 INDTOSUB

```
#define INDTOSUB 8
```

Definition at line 625 of file ftypes.h.

4.53.2.424 FUNTOFUN

```
#define FUNTOFUN 9
```

Definition at line 626 of file ftypes.h.

4.53.2.425 ARGTOARG

```
#define ARGTOARG 10
```

Definition at line 627 of file ftypes.h.

4.53.2.426 ARLTOARL

```
#define ARLTOARL 11
```

Definition at line 628 of file ftypes.h.

4.53.2.427 EXPTOEXP

```
#define EXPTOEXP 12
```

Definition at line 629 of file ftypes.h.

4.53.2.428 FROMBRAC

```
#define FROMBRAC 13
```

Definition at line 630 of file ftypes.h.

4.53.2.429 FROMSET

```
#define FROMSET 14
```

Definition at line 631 of file ftypes.h.

4.53.2.430 SETTONUM

```
#define SETTONUM 15
```

Definition at line 632 of file ftypes.h.

4.53.2.431 WILDCARDS

```
#define WILDCARDS 16
```

Definition at line 633 of file ftypes.h.

4.53.2.432 SETNUMBER

```
#define SETNUMBER 17
```

Definition at line 634 of file ftypes.h.

4.53.2.433 LOADDOLLAR

```
#define LOADDOLLAR 18
```

Definition at line 635 of file ftypes.h.

4.53.2.434 NUMTONUM

```
#define NUMTONUM 20
```

Definition at line 639 of file ftypes.h.

4.53.2.435 NUMTOSYM

```
#define NUMTOSYM 21
```

Definition at line 640 of file ftypes.h.

4.53.2.436 NUMTOIND

```
#define NUMTOIND 22
```

Definition at line 641 of file ftypes.h.

4.53.2.437 NUMTOSUB

```
#define NUMTOSUB 23
```

Definition at line 642 of file ftypes.h.

4.53.2.438 EATTENSOR

```
#define EATTENSOR 0x2000
```

Definition at line 646 of file ftypes.h.

4.53.2.439 CLEANFLAG

```
#define CLEANFLAG 0
```

Definition at line 652 of file ftypes.h.

4.53.2.440 DIRTYFLAG

```
#define DIRTYFLAG 1
```

Definition at line 653 of file ftypes.h.

4.53.2.441 DIRTYSYMFLAG

```
#define DIRTYSYMFLAG 2
```

Definition at line 654 of file ftypes.h.

4.53.2.442 MUSTCLEANPRF

```
#define MUSTCLEANPRF 4
```

Definition at line 655 of file ftypes.h.

4.53.2.443 SUBTERMUSED1

```
#define SUBTERMUSED1 8
```

Definition at line 656 of file ftypes.h.

4.53.2.444 SUBTERMUSED2

```
#define SUBTERMUSED2 16
```

Definition at line 657 of file ftypes.h.

4.53.2.445 ALLDIRTY

```
#define ALLDIRTY (DIRTYFLAG|DIRTYSYMFLAG)
```

Definition at line 658 of file ftypes.h.

4.53.2.446 ARGHEAD

```
#define ARGHEAD 2
```

Definition at line 660 of file ftypes.h.

4.53.2.447 FUNHEAD

```
#define FUNHEAD 3
```

Definition at line 661 of file ftypes.h.

4.53.2.448 SUBEXPSIZE

```
#define SUBEXPSIZE 5
```

Definition at line 662 of file ftypes.h.

4.53.2.449 EXPRHEAD

```
#define EXPRHEAD 5
```

Definition at line 663 of file ftypes.h.

4.53.2.450 TYPEARGHEADSIZE

```
#define TYPEARGHEADSIZE 6
```

Definition at line 664 of file ftypes.h.

4.53.2.451 SKIP

```
#define SKIP 1
```

Definition at line 671 of file ftypes.h.

4.53.2.452 DROP

```
#define DROP 2
```

Definition at line 672 of file ftypes.h.

4.53.2.453 HIDE

```
#define HIDE 3
```

Definition at line 673 of file ftypes.h.

4.53.2.454 UNHIDE

```
#define UNHIDE 4
```

Definition at line 674 of file ftypes.h.

4.53.2.455 INTOHIDE

```
#define INTOHIDE 5
```

Definition at line 675 of file ftypes.h.

4.53.2.456 LOCALEXPRESSION

```
#define LOCALEXPRESSION 0
```

Definition at line 681 of file ftypes.h.

4.53.2.457 SKIPLEXPRESSION

```
#define SKIPLEXPRESSION 1
```

Definition at line 682 of file ftypes.h.

4.53.2.458 DROPLEXPRESSION

```
#define DROPLEXPRESSION 2
```

Definition at line 683 of file ftypes.h.

4.53.2.459 DROPPEDEXPRESSION

```
#define DROPPEDEXPRESSION 3
```

Definition at line 684 of file ftypes.h.

4.53.2.460 GLOBALEXPRESSION

```
#define GLOBALEXPRESSION 4
```

Definition at line 685 of file ftypes.h.

4.53.2.461 SKIPGEXPRESSION

```
#define SKIPGEXPRESSION 5
```

Definition at line 686 of file ftypes.h.

4.53.2.462 DROPGEXPRESSION

```
#define DROPGEXPRESSION 6
```

Definition at line 687 of file ftypes.h.

4.53.2.463 STOREDEXPRESSION

```
#define STOREDEXPRESSION 8
```

Definition at line 688 of file ftypes.h.

4.53.2.464 HIDDENLEXPRESSION

```
#define HIDDENLEXPRESSION 9
```

Definition at line 689 of file ftypes.h.

4.53.2.465 HIDDENGEXPRESSION

```
#define HIDDENGEXPRESSION 13
```

Definition at line 690 of file ftypes.h.

4.53.2.466 INCEXPRESSION

```
#define INCEXPRESSION 9
```

Definition at line 691 of file ftypes.h.

4.53.2.467 HIDELEXPRESSION

```
#define HIDELEXPRESSION 10
```

Definition at line 692 of file ftypes.h.

4.53.2.468 HIDEGEXPRESSION

```
#define HIDEGEXPRESSION 14
```

Definition at line 693 of file ftypes.h.

4.53.2.469 DROPHLEXPRESSION

```
#define DROPHLEXPRESSION 11
```

Definition at line 694 of file ftypes.h.

4.53.2.470 DROPHGEXPRESSION

```
#define DROPHGEXPRESSION 15
```

Definition at line 695 of file ftypes.h.

4.53.2.471 UNHIDELEXPRESSION

```
#define UNHIDELEXPRESSION 12
```

Definition at line 696 of file ftypes.h.

4.53.2.472 UNHIDEGEXPRESSION

```
#define UNHIDEGEXPRESSION 16
```

Definition at line 697 of file ftypes.h.

4.53.2.473 INTOHIDELEXPRESSION

```
#define INTOHIDELEXPRESSION 17
```

Definition at line 698 of file ftypes.h.

4.53.2.474 INTOHIDEGEXPRESSION

```
#define INTOHIDEGEXPRESSION 18
```

Definition at line 699 of file ftypes.h.

4.53.2.475 SPECTATOREXPRESSION

```
#define SPECTATOREXPRESSION 19
```

Definition at line 700 of file ftypes.h.

4.53.2.476 DROPSPECTATOREXPRESSION

```
#define DROPSPECTATOREXPRESSION 20
```

Definition at line 701 of file ftypes.h.

4.53.2.477 SKIPUNHIDELEXPRESSION

```
#define SKIPUNHIDELEXPRESSION 21
```

Definition at line 702 of file ftypes.h.

4.53.2.478 SKIPUNHIDEGEXPRESSION

```
#define SKIPUNHIDEGEXPRESSION 22
```

Definition at line 703 of file ftypes.h.

4.53.2.479 PRINTOFF

```
#define PRINTOFF 0
```

Definition at line 705 of file ftypes.h.

4.53.2.480 PRINTON

```
#define PRINTON 1
```

Definition at line 706 of file ftypes.h.

4.53.2.481 PRINTCONTENTS

```
#define PRINTCONTENTS 2
```

Definition at line 707 of file ftypes.h.

4.53.2.482 PRINTCONTENT

```
#define PRINTCONTENT 3
```

Definition at line 708 of file ftypes.h.

4.53.2.483 PRINTLFILE

```
#define PRINTLFILE 4
```

Definition at line 709 of file ftypes.h.

4.53.2.484 PRINTONETERM

```
#define PRINTONETERM 8
```

Definition at line 710 of file ftypes.h.

4.53.2.485 PRINTONEFUNCTION

```
#define PRINTONEFUNCTION 16
```

Definition at line 711 of file ftypes.h.

4.53.2.486 PRINTALL

```
#define PRINTALL 32
```

Definition at line 712 of file ftypes.h.

4.53.2.487 REGULAREXPRESSION

```
#define REGULAREXPRESSION -1
```

Definition at line 718 of file ftypes.h.

4.53.2.488 REDEFINEDEXPRESSION

```
#define REDEFINEDEXPRESSION -2
```

Definition at line 719 of file ftypes.h.

4.53.2.489 NEWLYDEFINEDEXPRESSION

```
#define NEWLYDEFINEDEXPRESSION -3
```

Definition at line 720 of file ftypes.h.

4.53.2.490 VERTEXFUNCTION

```
#define VERTEXFUNCTION -1
```

Definition at line 729 of file ftypes.h.

4.53.2.491 GENERALFUNCTION

```
#define GENERALFUNCTION 0
```

Definition at line 730 of file ftypes.h.

4.53.2.492 TENSORFUNCTION

```
#define TENSORFUNCTION 2
```

Definition at line 731 of file ftypes.h.

4.53.2.493 GAMMAFUNCTION

```
#define GAMMAFUNCTION 4
```

Definition at line 732 of file ftypes.h.

4.53.2.494 POS_

```
#define POS_ 0 /* integer > 0 */
```

Definition at line 739 of file ftypes.h.

4.53.2.495 POS0_

```
#define POS0_ 1 /* integer >= 0 */
```

Definition at line 740 of file ftypes.h.

4.53.2.496 NEG_

```
#define NEG_ 2 /* integer < 0 */
```

Definition at line 741 of file ftypes.h.

4.53.2.497 NEG0_

```
#define NEG0_ 3 /* integer <= 0 */
```

Definition at line 742 of file ftypes.h.

4.53.2.498 EVEN_

```
#define EVEN_ 4 /* integer (even) */
```

Definition at line 743 of file ftypes.h.

4.53.2.499 ODD_

```
#define ODD_ 5 /* integer (odd) */
```

Definition at line 744 of file ftypes.h.

4.53.2.500 Z_

```
#define Z_ 6 /* integer */
```

Definition at line 745 of file ftypes.h.

4.53.2.501 SYMBOL_

```
#define SYMBOL_ 7 /* symbol only */
```

Definition at line 746 of file ftypes.h.

4.53.2.502 FIXED_

```
#define FIXED_ 8 /* fixed index */
```

Definition at line 747 of file ftypes.h.

4.53.2.503 INDEX_

```
#define INDEX_ 9 /* index only */
```

Definition at line 748 of file ftypes.h.

4.53.2.504 Q_

```
#define Q_ 10 /* rational */
```

Definition at line 749 of file ftypes.h.

4.53.2.505 DUMMYINDEX_

```
#define DUMMYINDEX_ 11 /* dummy index only */
```

Definition at line 750 of file ftypes.h.

4.53.2.506 VECTOR_

```
#define VECTOR_ 12 /* vector only */
```

Definition at line 751 of file ftypes.h.

4.53.2.507 GAMMA1

```
#define GAMMA1 0
```

Definition at line 757 of file ftypes.h.

4.53.2.508 GAMMA5

```
#define GAMMA5 -1
```

Definition at line 758 of file ftypes.h.

4.53.2.509 GAMMA6

```
#define GAMMA6 -2
```

Definition at line 759 of file ftypes.h.

4.53.2.510 GAMMA7

```
#define GAMMA7 -3
```

Definition at line 760 of file ftypes.h.

4.53.2.511 FUNNYVEC

```
#define FUNNYVEC -4
```

Definition at line 761 of file ftypes.h.

4.53.2.512 FUNNYWILD

```
#define FUNNYWILD -5
```

Definition at line 762 of file ftypes.h.

4.53.2.513 SUMMEDIND

```
#define SUMMEDIND -6
```

Definition at line 763 of file ftypes.h.

4.53.2.514 NOINDEX

```
#define NOINDEX -7
```

Definition at line 764 of file ftypes.h.

4.53.2.515 FUNNYDOLLAR

```
#define FUNNYDOLLAR -8
```

Definition at line 765 of file ftypes.h.

4.53.2.516 EMPTYINDEX

```
#define EMPTYINDEX -9
```

Definition at line 766 of file ftypes.h.

4.53.2.517 MINSPEC

```
#define MINSPEC -10
```

Definition at line 772 of file ftypes.h.

4.53.2.518 USEDFLAG

```
#define USEDFLAG 2
```

Definition at line 774 of file ftypes.h.

4.53.2.519 DUMMYFLAG

```
#define DUMMYFLAG 1
```

Definition at line 775 of file ftypes.h.

4.53.2.520 MAINSORT

```
#define MAINSORT 0
```

Definition at line 777 of file ftypes.h.

4.53.2.521 FUNCTIONSORT

```
#define FUNCTIONSORT 1
```

Definition at line 778 of file ftypes.h.

4.53.2.522 SUBSORT

```
#define SUBSORT 2
```

Definition at line 779 of file ftypes.h.

4.53.2.523 FLOATMODE

```
#define FLOATMODE 1
```

Definition at line 781 of file ftypes.h.

4.53.2.524 RATIONALMODE

```
#define RATIONALMODE 0
```

Definition at line 782 of file ftypes.h.

4.53.2.525 NUMSPECSETS

```
#define NUMSPECSETS 10
```

Definition at line 784 of file ftypes.h.

4.53.2.526 ISZERO

```
#define ISZERO 1
```

Definition at line 786 of file ftypes.h.

4.53.2.527 ISUNMODIFIED

```
#define ISUNMODIFIED 2
```

Definition at line 787 of file ftypes.h.

4.53.2.528 ISCOMPRESSED

```
#define ISCOMPRESSED 4
```

Definition at line 788 of file ftypes.h.

4.53.2.529 ISINRHS

```
#define ISINRHS 8
```

Definition at line 789 of file ftypes.h.

4.53.2.530 ISFACTORIZED

```
#define ISFACTORIZED 16
```

Definition at line 790 of file ftypes.h.

4.53.2.531 TOBEFACTORED

```
#define TOBEFACTORED 32
```

Definition at line 791 of file ftypes.h.

4.53.2.532 TOBEUNFACTORED

```
#define TOBEUNFACTORED 64
```

Definition at line 792 of file ftypes.h.

4.53.2.533 KEEPZERO

```
#define KEEPZERO 128
```

Definition at line 793 of file ftypes.h.

4.53.2.534 VARTYPENONE

```
#define VARTYPENONE 0
```

Definition at line 795 of file ftypes.h.

4.53.2.535 VARTYPECOMPLEX

```
#define VARTYPECOMPLEX 1
```

Definition at line 796 of file ftypes.h.

4.53.2.536 VARTYPEIMAGINARY

```
#define VARTYPEIMAGINARY 2
```

Definition at line 797 of file ftypes.h.

4.53.2.537 VARTYPEROOTOFUNITY

```
#define VARTYPEROOTOFUNITY 4
```

Definition at line 798 of file ftypes.h.

4.53.2.538 VARTYPEMINUS

```
#define VARTYPEMINUS 8
```

Definition at line 799 of file ftypes.h.

4.53.2.539 CYCLESYMMETRIC

```
#define CYCLESYMMETRIC 1
```

Definition at line 800 of file ftypes.h.

4.53.2.540 RCYCLESYMMETRIC

```
#define RCYCLESYMMETRIC 2
```

Definition at line 801 of file ftypes.h.

4.53.2.541 SYMMETRIC

```
#define SYMMETRIC 3
```

Definition at line 802 of file ftypes.h.

4.53.2.542 ANTISYMMETRIC

```
#define ANTISYMMETRIC 4
```

Definition at line 803 of file ftypes.h.

4.53.2.543 REVERSEORDER

```
#define REVERSEORDER 256
```

Definition at line 804 of file ftypes.h.

4.53.2.544 SUBMULTI

```
#define SUBMULTI 1
```

Definition at line 810 of file ftypes.h.

4.53.2.545 SUBONCE

```
#define SUBONCE 2
```

Definition at line 811 of file ftypes.h.

4.53.2.546 SUBONLY

```
#define SUBONLY 3
```

Definition at line 812 of file ftypes.h.

4.53.2.547 SUBMANY

```
#define SUBMANY 4
```

Definition at line 813 of file ftypes.h.

4.53.2.548 SUBVECTOR

```
#define SUBVECTOR 5
```

Definition at line 814 of file ftypes.h.

4.53.2.549 SUBSELECT

```
#define SUBSELECT 6
```

Definition at line 815 of file ftypes.h.

4.53.2.550 SUBALL

```
#define SUBALL 7
```

Definition at line 816 of file ftypes.h.

4.53.2.551 SUBMASK

```
#define SUBMASK 15
```

Definition at line 817 of file ftypes.h.

4.53.2.552 SUBDISORDER

```
#define SUBDISORDER 16
```

Definition at line 818 of file ftypes.h.

4.53.2.553 SUBAFTER

```
#define SUBAFTER 32
```

Definition at line 819 of file ftypes.h.

4.53.2.554 SUBAFTERNOT

```
#define SUBAFTERNOT 64
```

Definition at line 820 of file ftypes.h.

4.53.2.555 IDHEAD

```
#define IDHEAD 6
```

Definition at line 822 of file ftypes.h.

4.53.2.556 DOLLARFLAG

```
#define DOLLARFLAG 1
```

Definition at line 824 of file ftypes.h.

4.53.2.557 NORMALIZEFLAG

```
#define NORMALIZEFLAG 2
```

Definition at line 825 of file ftypes.h.

4.53.2.558 GIDENT

```
#define GIDENT 1
```

Definition at line 827 of file ftypes.h.

4.53.2.559 GFIVE

```
#define GFIVE 4
```

Definition at line 828 of file ftypes.h.

4.53.2.560 GPLUS

```
#define GPLUS 3
```

Definition at line 829 of file ftypes.h.

4.53.2.561 GMINUS

```
#define GMINUS 2
```

Definition at line 830 of file ftypes.h.

4.53.2.562 LONGNUMBER

```
#define LONGNUMBER 1
```

Definition at line 836 of file ftypes.h.

4.53.2.563 MATCH

```
#define MATCH 2
```

Definition at line 837 of file ftypes.h.

4.53.2.564 COEFFI

```
#define COEFFI 3
```

Definition at line 838 of file ftypes.h.

4.53.2.565 SUBEXPR

```
#define SUBEXPR 4
```

Definition at line 839 of file ftypes.h.

4.53.2.566 MULTIPLEOF

```
#define MULTIPLEOF 5
```

Definition at line 840 of file ftypes.h.

4.53.2.567 IFDOLLAR

```
#define IFDOLLAR 6
```

Definition at line 841 of file ftypes.h.

4.53.2.568 IFEXPRESSION

```
#define IFEXPRESSION 7
```

Definition at line 842 of file ftypes.h.

4.53.2.569 IFDOLLAREXTRA

```
#define IFDOLLAREXTRA 8
```

Definition at line 843 of file ftypes.h.

4.53.2.570 IFISFACTORIZED

```
#define IFISFACTORIZED 9
```

Definition at line 844 of file ftypes.h.

4.53.2.571 IFOCCURS

```
#define IFOCCURS 10
```

Definition at line 845 of file ftypes.h.

4.53.2.572 IFUSERFLAG

```
#define IFUSERFLAG 11
```

Definition at line 846 of file ftypes.h.

4.53.2.573 IFFLOATNUMBER

```
#define IFFLOATNUMBER 12
```

Definition at line 847 of file ftypes.h.

4.53.2.574 GREATER

```
#define GREATER 0
```

Definition at line 848 of file ftypes.h.

4.53.2.575 GREATEREQUAL

```
#define GREATEREQUAL 1
```

Definition at line 849 of file ftypes.h.

4.53.2.576 LESS

```
#define LESS 2
```

Definition at line 850 of file ftypes.h.

4.53.2.577 LESSEQUAL

```
#define LESSEQUAL 3
```

Definition at line 851 of file ftypes.h.

4.53.2.578 EQUAL

```
#define EQUAL 4
```

Definition at line 852 of file ftypes.h.

4.53.2.579 NOTEQUAL

```
#define NOTEQUAL 5
```

Definition at line 853 of file ftypes.h.

4.53.2.580 ORCOND

```
#define ORCOND 6
```

Definition at line 854 of file ftypes.h.

4.53.2.581 ANDCOND

```
#define ANDCOND 7
```

Definition at line 855 of file ftypes.h.

4.53.2.582 DUMMY

```
#define DUMMY 1
```

Definition at line 856 of file ftypes.h.

4.53.2.583 SORT

```
#define SORT 1
```

Definition at line 857 of file ftypes.h.

4.53.2.584 STORE

```
#define STORE 2
```

Definition at line 858 of file ftypes.h.

4.53.2.585 END

```
#define END 3
```

Definition at line 859 of file ftypes.h.

4.53.2.586 GLOBAL

```
#define GLOBAL 4
```

Definition at line 860 of file ftypes.h.

4.53.2.587 CLEAR

```
#define CLEAR 5
```

Definition at line 861 of file ftypes.h.

4.53.2.588 VECTBIT

```
#define VECTBIT 1
```

Definition at line 863 of file ftypes.h.

4.53.2.589 DOTPBIT

```
#define DOTPBIT 2
```

Definition at line 864 of file ftypes.h.

4.53.2.590 FUNBIT

```
#define FUNBIT 4
```

Definition at line 865 of file ftypes.h.

4.53.2.591 SETBIT

```
#define SETBIT 8
```

Definition at line 866 of file ftypes.h.

4.53.2.592 EXTRAPARAMETER

```
#define EXTRAPARAMETER 0x4000
```

Definition at line 868 of file ftypes.h.

4.53.2.593 GENCOMMUTE

```
#define GENCOMMUTE 0
```

Definition at line 869 of file ftypes.h.

4.53.2.594 GENNONCOMMUTE

```
#define GENNONCOMMUTE 0x2000
```

Definition at line 870 of file ftypes.h.

4.53.2.595 NAMENOTFOUND

```
#define NAMENOTFOUND -9
```

Definition at line 872 of file ftypes.h.

4.53.2.596 DOLUNDEFINED

```
#define DOLUNDEFINED 0
```

Definition at line 878 of file ftypes.h.

4.53.2.597 DOLNUMBER

```
#define DOLNUMBER 1
```

Definition at line 879 of file ftypes.h.

4.53.2.598 DOLARGUMENT

```
#define DOLARGUMENT 2
```

Definition at line 880 of file ftypes.h.

4.53.2.599 DOLSUBTERM

```
#define DOLSUBTERM 3
```

Definition at line 881 of file ftypes.h.

4.53.2.600 DOLTERMS

```
#define DOLTERMS 4
```

Definition at line 882 of file ftypes.h.

4.53.2.601 DOLWILDARGS

```
#define DOLWILDARGS 5
```

Definition at line 883 of file ftypes.h.

4.53.2.602 DOLINDEX

```
#define DOLINDEX 6
```

Definition at line 884 of file ftypes.h.

4.53.2.603 DOLZERO

```
#define DOLZERO 7
```

Definition at line 885 of file ftypes.h.

4.53.2.604 FINDLOOP

```
#define FINDLOOP 0
```

Definition at line 887 of file ftypes.h.

4.53.2.605 REPLACELOOP

```
#define REPLACELOOP 1
```

Definition at line 888 of file ftypes.h.

4.53.2.606 NOFUNPOWERS

```
#define NOFUNPOWERS 0
```

Definition at line 890 of file ftypes.h.

4.53.2.607 COMFUNPOWERS

```
#define COMFUNPOWERS 1
```

Definition at line 891 of file ftypes.h.

4.53.2.608 ALLFUNPOWERS

```
#define ALLFUNPOWERS 2
```

Definition at line 892 of file ftypes.h.

4.53.2.609 PROPERORDERFLAG

```
#define PROPERORDERFLAG 0
```

Definition at line 894 of file ftypes.h.

4.53.2.610 REGULAR

```
#define REGULAR 0
```

Definition at line 896 of file ftypes.h.

4.53.2.611 FINISH

```
#define FINISH 1
```

Definition at line 897 of file ftypes.h.

4.53.2.612 POLYADD

```
#define POLYADD 1
```

Definition at line 899 of file ftypes.h.

4.53.2.613 POLYSUB

```
#define POLYSUB 2
```

Definition at line 900 of file ftypes.h.

4.53.2.614 POLYMUL

```
#define POLYMUL 3
```

Definition at line 901 of file ftypes.h.

4.53.2.615 POLYDIV

```
#define POLYDIV 4
```

Definition at line 902 of file ftypes.h.

4.53.2.616 POLYREM

```
#define POLYREM 5
```

Definition at line 903 of file ftypes.h.

4.53.2.617 POLYGCD

```
#define POLYGCD 6
```

Definition at line 904 of file ftypes.h.

4.53.2.618 POLYINTFAC

```
#define POLYINTFAC 7
```

Definition at line 905 of file ftypes.h.

4.53.2.619 POLYNORM

```
#define POLYNORM 8
```

Definition at line 906 of file ftypes.h.

4.53.2.620 MODNONE

```
#define MODNONE 0
```

Definition at line 908 of file ftypes.h.

4.53.2.621 MODSUM

```
#define MODSUM 1
```

Definition at line 909 of file ftypes.h.

4.53.2.622 MODMAX

```
#define MODMAX 2
```

Definition at line 910 of file ftypes.h.

4.53.2.623 MODMIN

```
#define MODMIN 3
```

Definition at line 911 of file ftypes.h.

4.53.2.624 MODLOCAL

```
#define MODLOCAL 4
```

Definition at line 912 of file ftypes.h.

4.53.2.625 ELEMENTUSED

```
#define ELEMENTUSED 1
```

Definition at line 914 of file ftypes.h.

4.53.2.626 ELEMENTLOADED

```
#define ELEMENTLOADED 2
```

Definition at line 915 of file ftypes.h.

4.53.2.627 POSNEG

```
#define POSNEG 0x1
```

Definition at line 920 of file ftypes.h.

4.53.2.628 INVERSETABLE

```
#define INVERSETABLE 0x2
```

Definition at line 921 of file ftypes.h.

4.53.2.629 COEFFICIENTSONLY

```
#define COEFFICIENTSONLY 0x4
```

Definition at line 922 of file ftypes.h.

4.53.2.630 ALSOPOWERS

```
#define ALSOPOWERS 0x8
```

Definition at line 923 of file ftypes.h.

4.53.2.631 ALSOFUNARGS

```
#define ALSOFUNARGS 0x10
```

Definition at line 924 of file ftypes.h.

4.53.2.632 ALSODOLLARS

```
#define ALSODOLLARS 0x20
```

Definition at line 925 of file ftypes.h.

4.53.2.633 NOINVERSES

```
#define NOINVERSES 0x40
```

Definition at line 926 of file ftypes.h.

4.53.2.634 POSITIVEONLY

```
#define POSITIVEONLY 0
```

Definition at line 928 of file ftypes.h.

4.53.2.635 UNPACK

```
#define UNPACK 0x80
```

Definition at line 929 of file ftypes.h.

4.53.2.636 NOUNPACK

```
#define NOUNPACK 0
```

Definition at line 930 of file ftypes.h.

4.53.2.637 FROMFUNCTION

```
#define FROMFUNCTION 0x100
```

Definition at line 931 of file ftypes.h.

4.53.2.638 VARNAMES

```
#define VARNAMES 0
```

Definition at line 933 of file ftypes.h.

4.53.2.639 AUTONAMES

```
#define AUTONAMES 1
```

Definition at line 934 of file ftypes.h.

4.53.2.640 EXPRNAMES

```
#define EXPRNAMES 2
```

Definition at line 935 of file ftypes.h.

4.53.2.641 DOLLARNAMES

```
#define DOLLARNAMES 3
```

Definition at line 936 of file ftypes.h.

4.53.2.642 DUMMYBUFFER

```
#define DUMMYBUFFER 1
```

Definition at line 1008 of file ftypes.h.

4.53.2.643 ALLARGS

```
#define ALLARGS 1
```

Definition at line 1010 of file ftypes.h.

4.53.2.644 NUMARG

```
#define NUMARG 2
```

Definition at line 1011 of file ftypes.h.

4.53.2.645 ARGRANGE

```
#define ARGRANGE 3
```

Definition at line 1012 of file ftypes.h.

4.53.2.646 MAKEARGS

```
#define MAKEARGS 4
```

Definition at line 1013 of file ftypes.h.

4.53.2.647 MAXRANGEINDICATOR

```
#define MAXRANGEINDICATOR 4
```

Definition at line 1014 of file ftypes.h.

4.53.2.648 REPLACEARG

```
#define REPLACEARG 5
```

Definition at line 1015 of file ftypes.h.

4.53.2.649 ENCODEARG

```
#define ENCODEARG 6
```

Definition at line 1016 of file ftypes.h.

4.53.2.650 DECODEARG

```
#define DECODEARG 7
```

Definition at line 1017 of file ftypes.h.

4.53.2.651 IMPLODEARG

```
#define IMPLODEARG 8
```

Definition at line 1018 of file ftypes.h.

4.53.2.652 EXPLODEARG

```
#define EXPLODEARG 9
```

Definition at line 1019 of file ftypes.h.

4.53.2.653 PERMUTEARG

```
#define PERMUTEARG 10
```

Definition at line 1020 of file ftypes.h.

4.53.2.654 REVERSEARG

```
#define REVERSEARG 11
```

Definition at line 1021 of file ftypes.h.

4.53.2.655 CYCLEARG

```
#define CYCLEARG 12
```

Definition at line 1022 of file ftypes.h.

4.53.2.656 ISLYNDON

```
#define ISLYNDON 13
```

Definition at line 1023 of file ftypes.h.

4.53.2.657 ISLYNDONR

```
#define ISLYNDONR 14
```

Definition at line 1024 of file ftypes.h.

4.53.2.658 TOLYNDON

```
#define TOLYNDON 15
```

Definition at line 1025 of file ftypes.h.

4.53.2.659 TOLYNDONR

```
#define TOLYNDONR 16
```

Definition at line 1026 of file ftypes.h.

4.53.2.660 ADDARG

```
#define ADDARG 17
```

Definition at line 1027 of file ftypes.h.

4.53.2.661 MULTIPLYARG

```
#define MULTIPLYARG 18
```

Definition at line 1028 of file ftypes.h.

4.53.2.662 DROPARG

```
#define DROPARG 19
```

Definition at line 1029 of file ftypes.h.

4.53.2.663 SELECTARG

```
#define SELECTARG 20
```

Definition at line 1030 of file ftypes.h.

4.53.2.664 DEDUPARG

```
#define DEDUPARG 21
```

Definition at line 1031 of file ftypes.h.

4.53.2.665 ZTOHARG

```
#define ZTOHARG 22
```

Definition at line 1032 of file ftypes.h.

4.53.2.666 HTOZARG

```
#define HTOZARG 23
```

Definition at line 1033 of file ftypes.h.

4.53.2.667 BASECODE

```
#define BASECODE 1
```

Definition at line 1041 of file ftypes.h.

4.53.2.668 YESLYNDON

```
#define YESLYNDON 1
```

Definition at line 1042 of file ftypes.h.

4.53.2.669 NOLYNDON

```
#define NOLYNDON 2
```

Definition at line 1043 of file ftypes.h.

4.53.2.670 TOPOLYNOMIALFLAG

```
#define TOPOLYNOMIALFLAG 1
```

Definition at line 1045 of file ftypes.h.

4.53.2.671 FACTARGFLAG

```
#define FACTARGFLAG 2
```

Definition at line 1046 of file ftypes.h.

4.53.2.672 OLDFACTARG

```
#define OLDFACTARG 1
```

Definition at line 1048 of file ftypes.h.

4.53.2.673 NEWFACTARG

```
#define NEWFACTARG 0
```

Definition at line 1049 of file ftypes.h.

4.53.2.674 FROMMODULEOPTION

```
#define FROMMODULEOPTION 0
```

Definition at line 1051 of file ftypes.h.

4.53.2.675 FROMPOINTINSTRUCTION

```
#define FROMPOINTINSTRUCTION 1
```

Definition at line 1052 of file ftypes.h.

4.53.2.676 EXTRASYMBOL

```
#define EXTRASYMBOL 0
```

Definition at line 1054 of file ftypes.h.

4.53.2.677 REGULARSYMBOL

```
#define REGULARSYMBOL 1
```

Definition at line 1055 of file ftypes.h.

4.53.2.678 EXPRESSIONNUMBER

```
#define EXPRESSIONNUMBER 2
```

Definition at line 1056 of file ftypes.h.

4.53.2.679 O_NONE

```
#define O_NONE 0
```

Definition at line 1058 of file ftypes.h.

4.53.2.680 O_CSE

```
#define O_CSE 1
```

Definition at line 1059 of file ftypes.h.

4.53.2.681 O_CSEGREEDY

```
#define O_CSEGREEDY 2
```

Definition at line 1060 of file ftypes.h.

4.53.2.682 O_GREEDY

```
#define O_GREEDY 3
```

Definition at line 1061 of file ftypes.h.

4.53.2.683 O_OCCURRENCE

```
#define O_OCCURRENCE 0
```

Definition at line 1063 of file ftypes.h.

4.53.2.684 O_MCTS

```
#define O_MCTS 1
```

Definition at line 1064 of file ftypes.h.

4.53.2.685 O_SIMULATED_ANNEALING

```
#define O_SIMULATED_ANNEALING 2
```

Definition at line 1065 of file ftypes.h.

4.53.2.686 O_FORWARD

```
#define O_FORWARD 0
```

Definition at line 1067 of file ftypes.h.

4.53.2.687 O_BACKWARD

```
#define O_BACKWARD 1
```

Definition at line 1068 of file ftypes.h.

4.53.2.688 O_FORWARDORBACKWARD

```
#define O_FORWARDORBACKWARD 2
```

Definition at line 1069 of file ftypes.h.

4.53.2.689 O_FORWARDANDBACKWARD

```
#define O_FORWARDANDBACKWARD 3
```

Definition at line 1070 of file ftypes.h.

4.53.2.690 OPTHEAD

```
#define OPTHEAD 3
```

Definition at line 1072 of file ftypes.h.

4.53.2.691 DOALL

```
#define DOALL 1
```

Definition at line 1073 of file ftypes.h.

4.53.2.692 ONLYFUNCTIONS

```
#define ONLYFUNCTIONS 2
```

Definition at line 1074 of file ftypes.h.

4.53.2.693 INUSE

```
#define INUSE 1
```

Definition at line 1076 of file ftypes.h.

4.53.2.694 COULDCOMMUTE

```
#define COULDCOMMUTE 2
```

Definition at line 1077 of file ftypes.h.

4.53.2.695 DOESNOTCOMMUTE

```
#define DOESNOTCOMMUTE 4
```

Definition at line 1078 of file ftypes.h.

4.53.2.696 DICT_NONUMBERS

```
#define DICT_NONUMBERS 0
```

Definition at line 1080 of file ftypes.h.

4.53.2.697 DICT_INTEGERONLY

```
#define DICT_INTEGERONLY 1
```

Definition at line 1081 of file ftypes.h.

4.53.2.698 DICT_RATIONALONLY

```
#define DICT_RATIONALONLY 2
```

Definition at line 1082 of file ftypes.h.

4.53.2.699 DICT_ALLNUMBERS

```
#define DICT_ALLNUMBERS 3
```

Definition at line 1083 of file ftypes.h.

4.53.2.700 DICT_NOVARIABLES

```
#define DICT_NOVARIABLES 0
```

Definition at line 1084 of file ftypes.h.

4.53.2.701 DICT_DOVARIABLES

```
#define DICT_DOVARIABLES 1
```

Definition at line 1085 of file ftypes.h.

4.53.2.702 DICT_NOSPECIALS

```
#define DICT_NOSPECIALS 0
```

Definition at line 1086 of file ftypes.h.

4.53.2.703 DICT_DOSPECIALS

```
#define DICT_DOSPECIALS 1
```

Definition at line 1087 of file ftypes.h.

4.53.2.704 DICT_NOFUNWITHARGS

```
#define DICT_NOFUNWITHARGS 0
```

Definition at line 1088 of file ftypes.h.

4.53.2.705 DICT_DOFUNWITHARGS

```
#define DICT_DOFUNWITHARGS 1
```

Definition at line 1089 of file ftypes.h.

4.53.2.706 DICT_NOTINDOLLARS

```
#define DICT_NOTINDOLLARS 0
```

Definition at line 1090 of file ftypes.h.

4.53.2.707 DICT_INDOLLARS

```
#define DICT_INDOLLARS 1
```

Definition at line 1091 of file ftypes.h.

4.53.2.708 DICT_INTEGERNUMBER

```
#define DICT_INTEGERNUMBER 1
```

Definition at line 1093 of file ftypes.h.

4.53.2.709 DICT_RATIONALNUMBER

```
#define DICT_RATIONALNUMBER 2
```

Definition at line 1094 of file ftypes.h.

4.53.2.710 DICT_SYMBOL

```
#define DICT_SYMBOL 3
```

Definition at line 1095 of file ftypes.h.

4.53.2.711 DICT_VECTOR

```
#define DICT_VECTOR 4
```

Definition at line 1096 of file ftypes.h.

4.53.2.712 DICT_INDEX

```
#define DICT_INDEX 5
```

Definition at line 1097 of file ftypes.h.

4.53.2.713 DICT_FUNCTION

```
#define DICT_FUNCTION 6
```

Definition at line 1098 of file ftypes.h.

4.53.2.714 DICT_FUNCTION_WITH_ARGUMENTS

```
#define DICT_FUNCTION_WITH_ARGUMENTS 7
```

Definition at line 1099 of file ftypes.h.

4.53.2.715 DICT_SPECIALCHARACTER

```
#define DICT_SPECIALCHARACTER 8
```

Definition at line 1100 of file ftypes.h.

4.53.2.716 DICT_RANGE

```
#define DICT_RANGE 9
```

Definition at line 1101 of file ftypes.h.

4.53.2.717 READSPECTATORFLAG

```
#define READSPECTATORFLAG 3
```

Definition at line 1103 of file ftypes.h.

4.53.2.718 GLOBALSPECTATORFLAG

```
#define GLOBALSPECTATORFLAG 1
```

Definition at line 1104 of file ftypes.h.

4.53.2.719 ORDEREDSET

```
#define ORDEREDSET 1
```

Definition at line 1106 of file ftypes.h.

4.53.2.720 DENSETABLE

```
#define DENSETABLE 1
```

Definition at line 1108 of file ftypes.h.

4.53.2.721 SPARSETABLE

```
#define SPARSETABLE 0
```

Definition at line 1109 of file ftypes.h.

4.53.2.722 TOPOLOGIESONLY

```
#define TOPOLOGIESONLY 1
```

Definition at line 1115 of file ftypes.h.

4.53.2.723 WITHOUTNODES

```
#define WITHOUTNODES 2
```

Definition at line 1116 of file ftypes.h.

4.53.2.724 WITHEDGES

```
#define WITHEDGES 4
```

Definition at line 1117 of file ftypes.h.

4.53.2.725 WITHBLOCKS

```
#define WITHBLOCKS 8
```

Definition at line 1118 of file ftypes.h.

4.53.2.726 WITHONEPISETS

```
#define WITHONEPISETS 16
```

Definition at line 1119 of file ftypes.h.

4.53.2.727 WITHSYMMETRIZEI

```
#define WITHSYMMETRIZEI 32
```

Definition at line 1120 of file ftypes.h.

4.53.2.728 WITHSYMMETRIZEF

```
#define WITHSYMMETRIZEF 64
```

Definition at line 1121 of file ftypes.h.

4.53.2.729 CHECKEXTERN

```
#define CHECKEXTERN 128
```

Definition at line 1123 of file ftypes.h.

4.53.2.730 ONEPARTI

```
#define ONEPARTI 256
```

Definition at line 1125 of file ftypes.h.

4.53.2.731 ONEPARTR

```
#define ONEPARTR 512
```

Definition at line 1126 of file ftypes.h.

4.53.2.732 ONSHELL

```
#define ONSHELL 1024
```

Definition at line 1127 of file ftypes.h.

4.53.2.733 OFFSHELL

```
#define OFFSHELL 2048
```

Definition at line 1128 of file ftypes.h.

4.53.2.734 NOSIGMA

```
#define NOSIGMA 4096
```

Definition at line 1129 of file ftypes.h.

4.53.2.735 SIGMA

```
#define SIGMA 8192
```

Definition at line 1130 of file ftypes.h.

4.53.2.736 NOSNAIL

```
#define NOSNAIL 16384
```

Definition at line 1131 of file ftypes.h.

4.53.2.737 SNAIL

```
#define SNAIL 32768
```

Definition at line 1132 of file ftypes.h.

4.53.2.738 NOTADPOLE

```
#define NOTADPOLE 65536
```

Definition at line 1133 of file ftypes.h.

4.53.2.739 TADPOLE

```
#define TADPOLE 131072
```

Definition at line 1134 of file ftypes.h.

4.53.2.740 SIMPLE

```
#define SIMPLE 262144
```

Definition at line 1135 of file ftypes.h.

4.53.2.741 NOTSIMPLE

```
#define NOTSIMPLE 524288
```

Definition at line 1136 of file ftypes.h.

4.53.2.742 BIPART

```
#define BIPART 1048576
```

Definition at line 1137 of file ftypes.h.

4.53.2.743 NONBIPART

```
#define NONBIPART 2097152
```

Definition at line 1138 of file ftypes.h.

4.53.2.744 CYCLI

```
#define CYCLI 4194304
```

Definition at line 1139 of file ftypes.h.

4.53.2.745 CYCLR

```
#define CYCLR 8388608
```

Definition at line 1140 of file ftypes.h.

4.53.2.746 FLOOP

```
#define FLOOP 16777216
```

Definition at line 1141 of file ftypes.h.

4.53.2.747 NOTFLOOP

```
#define NOTFLOOP 33554432
```

Definition at line 1142 of file ftypes.h.

4.53.3 Typedef Documentation

4.53.3.1 PVFUNWP

```
typedef void(* PVFUNWP) (WORD *)
```

Definition at line 171 of file ftypes.h.

4.53.3.2 PVFUNV

```
typedef void(* PVFUNV) (void)
```

Definition at line 172 of file ftypes.h.

4.53.3.3 CFUN

```
typedef int(* CFUN) (void)
```

Definition at line 173 of file ftypes.h.

4.53.3.4 TFUN

```
typedef int(* TFUN) (UBYTE *)
```

Definition at line 174 of file ftypes.h.

4.53.3.5 TFUN1

```
typedef int(* TFUN1) (UBYTE *, int)
```

Definition at line 175 of file ftypes.h.

4.54 ftypes.h

Go to the documentation of this file.

```
00001
00009 /* #[ License : */
00010 /*
00011  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00012  *   When using this file you are requested to refer to the publication
00013  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00014  *   This is considered a matter of courtesy as the development was paid
00015  *   for by FOM the Dutch physics granting agency and we would like to
00016  *   be able to track its scientific use to convince FOM of its value
00017  *   for the community.
00018  *
00019  *   This file is part of FORM.
00020  *
00021  *   FORM is free software: you can redistribute it and/or modify it under the
00022  *   terms of the GNU General Public License as published by the Free Software
00023  *   Foundation, either version 3 of the License, or (at your option) any later
00024  *   version.
00025  *
00026  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00027  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00028  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00029  *   details.
00030  *
00031  *   You should have received a copy of the GNU General Public License along
00032  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00033  */
00034 /* #] License : */
00035
00051 #define WITHOUTERROR 0
00052 #define WITHERROR 1
00053
00054 /*
00055     The various streams. (look also in tools.c)
00056 */
00057
00058 #define FILESTREAM 0
00059 #define PREVARSTREAM 1
00060 #define PREREADSTREAM 2
00061 #define PIPESTREAM 3
00062 #define PRECALCSTREAM 4
00063 #define DOLLARSTREAM 5
00064 #define PREREADSTREAM2 6
00065 #define EXTERNALCHANNELSTREAM 7
00066 #define PREREADSTREAM3 8
00067 #define REVERSEFILESTREAM 9
00068 #define INPUTSTREAM 10
```

```
00069
00070 #define ENDOFSTREAM 0xFF
00071 #define ENDOFINPUT 0xFF
00072
00073 /*
00074     Types of files
00075 */
00076
00077 #define SUBROUTINEFILE 0
00078 #define PROCEDUREFILE 1
00079 #define HEADERFILE 2
00080 #define SETUPFILE 3
00081 #define TABLEBASEFILE 4
00082
00083 /*
00084     Types of modules
00085 */
00086
00087 #define FIRSTMODULE -1
00088 #define GLOBALMODULE 0
00089 #define SORTMODULE 1
00090 #define STOREMODULE 2
00091 #define CLEARMODULE 3
00092 #define ENDMODULE 4
00093
00094 #define POLYFUN 0
00095
00096 #define NOPARALLEL_DOLLAR       0x0001
00097 #define NOPARALLEL_RHS          0x0002
00098 #define NOPARALLEL_CONVPOLY     0x0004
00099 #define NOPARALLEL_SPECTATOR    0x0008
00100 #define NOPARALLEL_USER         0x0010
00101 #define NOPARALLEL_TBLDOLLAR    0x0100
00102 #define NOPARALLEL_NPROC        0x0200
00103 #define PARALLELFLAG            0x0000
00104
00105 #define PRENOACTION 0
00106 #define PRERAISEAFTER 1
00107 #define PRELOWERAFTER 2
00108 /*
00109 #define ELIUMOD 1
00110 #define ELIZMOD 2
00111 #define SKIUMOD 3
00112 #define SKIZMOD 4
00113 */
00114 #define WITHSEMICOLON 0
00115 #define WITHOUTSEMICOLON 1
00116 #define MODULEINSTACK 8
00117 #define EXECUTINGIF 0
00118 #define LOOKINGFORELSE 1
00119 #define LOOKINGFORENDIF 2
00120 #define NEWSTATEMENT 1
00121 #define OLDSTATEMENT 0
00122
00123 #define EXECUTINGPRESWITCH 0
00124 #define SEARCHINGPRECASE 1
00125 #define SEARCHINGPREENDSWITCH 2
00126
00127 #define PREPROONLY 1
00128 #define DUMPTOCOMPILER 2
00129 #define DUMPOUTTERMS 4
00130 #define DUMPINTERMS 8
00131 #define DUMPTOSORT 16
00132 #define DUMPTOPARALLEL 32
00133 #define THREADSDEBUG 64
00134
00135 #define ERROROUT 0
00136 #define INPUTOUT 1
00137 #define STATSOUT 2
00138 #define EXPRSOUT 3
00139 #define WRITEOUT 4
00140
00141 #define EXTERNALCHANNELOUT 5
00142
00143 #define NUMERICALLOOP 0
00144 #define LISTEDLOOP 1
00145 #define ONEEXPRESSION 2
00146
00147 #define PRETYPENONE 0
00148 #define PRETYPEIF 1
00149 #define PRETYPEDO 2
00150 #define PRETYPEPROCEDURE 3
00151 #define PRETYPESWITCH 4
00152 #define PRETYPEINSIDE 5
00153
00154 /*
00155     Type of statement. Used to make sure that the statements are in proper order
```

```
00156 */
00157
00158 #define DECLARATION   1
00159 #define SPECIFICATION 2
00160 #define DEFINITION    3
00161 #define STATEMENT     4
00162 #define TOOUTPUT      5
00163 #define ATENDOFMODULE 6
00164 #define MIXED2        8  /* unused */
00165 #define MIXED         9
00166
00167 /*
00168     The typedefs are to allow the compilers to do better error checking.
00169 */
00170
00171 typedef void (*PVFUNWP)(WORD *);
00172 typedef void (*PVFUNV)(void);
00173 typedef int (*CFUN)(void);
00174 typedef int (*TFUN)(UBYTE *);
00175 typedef int (*TFUN1)(UBYTE *,int);
00176
00177 /*
00178     Flags used in the compiler's KEYWORD tables.
00179 */
00180
00181 #define NOAUTO 0
00182 #define PARTEST 1   /* parentheses test */
00183 #define WITHAUTO 2  /* auto-declaration */
00184
00185 #define ALLVARIABLES -1
00186 #define SYMBOLONLY 1
00187 #define INDEXONLY 2
00188 #define VECTORONLY 4
00189 #define FUNCTIONONLY 8
00190 #define SETONLY 16
00191 #define EXPRESSIONONLY 32
00192
00193
00201
00202 #define CDELETE -1
00203 #define ANYTYPE -1
00204 #define CSYMBOL 0
00205 #define CINDEX 1
00206 #define CVECTOR 2
00207 #define CFUNCTION 3
00208 #define CSET 4
00209 #define CEXPRESSION 5
00210 #define CDOTPRODUCT 6
00211 #define CNUMBER 7
00212 #define CSUBEXP 8
00213 #define CDELTA 9
00214 #define CDOLLAR 10
00215 #define CDUBIOUS 11
00216 #define CRANGE 12
00217 #define CMODEL 13
00218 #define CVECTOR1 21
00219 #define CDOUBLEDOT 22
00220
00223 /*
00224     Types of tokens in the tokenizer.
00225 */
00226
00227 #define TSYMBOL -1
00228 #define TINDEX -2
00229 #define TVECTOR -3
00230 #define TFUNCTION -4
00231 #define TSET -5
00232 #define TEXPRESSION -6
00233 #define TDOTPRODUCT -7
00234 #define TNUMBER -8
00235 #define TSUBEXP -9
00236 #define TDELTA -10
00237 #define TDOLLAR -11
00238 #define TDUBIOUS -12
00239 #define LPARENTHESIS -13
00240 #define RPARENTHESIS -14
00241 #define TWILDCARD -15
00242 #define TWILDARG -16
00243 #define TDOT -17
00244 #define LBRACE -18
00245 #define RBRACE -19
00246 #define TCOMMA -20
00247 #define TFUNOPEN -21
00248 #define TFUNCLOSE -22
00249 #define TMULTIPLY -23
00250 #define TDIVIDE -24
00251 #define TPOWER -25
```

```
00252 #define TPLUS -26
00253 #define TMINUS -27
00254 #define TNOT -28
00255 #define TENDOFIT -29
00256 #define TSETOPEN -30
00257 #define TSETCLOSE -31
00258 #define TGENINDEX -32
00259 #define TCONJUGATE -33
00260 #define LRPARENTHESES -34
00261 #define TNUMBER1 -35
00262 #define TPOWER1 -36
00263 #define TEMPTY -37
00264 #define TSETNUM -38
00265 #define TSGAMMA -39
00266 #define TSETDOL -40
00267 #define TFLOAT -41
00268
00269 #define TYPEISFUN 0
00270 #define TYPEISSUB 1
00271 #define TYPEISMYSTERY -1
00272
00273 #define LHSIDEX 2
00274 #define LHSIDE 1
00275 #define RHSIDE 0
00276
00277 /*
00278     Output modes
00279 */
00280
00281 #define FORTRANMODE 1
00282 #define REDUCEMODE 2
00283 #define MAPLEMODE 3
00284 #define MATHEMATICAMODE 4
00285 #define CMODE 5
00286 #define VORTRANMODE 6
00287 #define PFORTRANMODE 7
00288 #define DOUBLEFORTRANMODE 33
00289 #define DOUBLEPRECISIONFLAG 32
00290 #define NODOUBLEMASK 31
00291 #define QUADRUPLEFORTRANMODE 65
00292 #define QUADRUPLEPRECISIONFLAG 64
00293 #define ALLINTEGERDOUBLE 128
00294 #define NOQUADMASK 63
00295 #define NORMALFORMAT 0
00296 #define NOSPACEFORMAT 1
00297
00298 #define ISNOTFORTRAN90 0
00299 #define ISFORTRAN90 1
00300
00301 #define ALSOREVERSE 1
00302 #define CHISHOLM 2
00303 #define NOTRICK 16
00304
00305 #define SORTLOWFIRST 0
00306 #define SORTHIGHFIRST 1
00307 #define SORTPOWERFIRST 2
00308 #define SORTANTIPOWER 3
00309
00310 /* These control the output of WriteStats. The different sort
00311  * types have differently formatted stats. These variables should
00312  * not have arbitrary values since they are also used as array
00313  * indices by WriteStats. */
00314 #define STATSSPLITMERGE 0
00315 #define STATSMERGETOFILE 1
00316 #define STATSPOSTSORT 2
00317 /* CHECKLOGTYPE implies that statistics will not be printed in
00318  * the log file if AM.LogType = 1 (when running with -ll or -F). */
00319 #define NOCHECKLOGTYPE 0
00320 #define CHECKLOGTYPE 1
00321
00322 #define NMIN4SHIFT 4
00323 /*
00324     The next are the main codes.
00325     Note: SETSET is not allowed to be 4*n+1
00326     We use those codes in CoIdExpression for function information
00327     after the pattern. Because SETSET also stands there we have to
00328     be careful!!
00329     Don't forget MAXBUILTINFUNCTION when adding codes!
00330     The object FUNCTION is at the start of the functions that are in regular
00331     notation. Anything below it is in special notation.
00332
00333     Remark: HAAKJE0 is for compression purposes and should only occur
00334     at moments that ARGWILD cannot occur.
00335 */
00336 #define SYMBOL 1
00337 #define DOTPRODUCT 2
00338 #define VECTOR 3
```

```
00339 #define INDEX 4
00340 #define EXPRESSION 5
00341 #define SUBEXPRESSION 6
00342 #define DOLLAREXPRESSION 7
00343 #define SETSET 8
00344 #define ARGWILD 9
00345 #define MINVECTOR 10
00346 #define SETEXP 11
00347 #define DOLLAREXPR2 12
00348 #define HAAKJE0 9
00349 #define FUNCTION 20
00350
00351 #define TMPPOLYFUN 14
00352 #define ARGFIELD 15
00353 #define SNUMBER 16
00354 #define LNUMBER 17
00355 #define HAAKJE 18
00356 #define DELTA 19
00357 #define EXPONENT 20
00358 #define DENOMINATOR 21
00359 #define SETFUNCTION 22
00360 #define GAMMA 23
00361 #define GAMMAI 24
00362 #define GAMMAFIVE 25
00363 #define GAMMASIX 26
00364 #define GAMMASEVEN 27
00365 #define SUMF1 28
00366 #define SUMF2 29
00367 #define DUMFUN 30
00368 #define REPLACEMENT 31
00369 #define REVERSEFUNCTION 32
00370 #define DISTRIBUTION 33
00371 #define DELTA3 34
00372 #define DUMMYFUN 35
00373 #define DUMMYTEN 36
00374 #define LEVICIVITA 37
00375 #define FACTORIAL 38
00376 #define INVERSEFACTORIAL 39
00377 #define BINOMIAL 40
00378 #define NUMARGSFUN 41
00379 #define SIGNFUN 42
00380 #define MODFUNCTION 43
00381 #define MOD2FUNCTION 44
00382 #define MINFUNCTION 45
00383 #define MAXFUNCTION 46
00384 #define ABSFUNCTION 47
00385 #define SIGFUNCTION 48
00386 #define INTFUNCTION 49
00387 #define THETA 50
00388 #define THETA2 51
00389 #define DELTA2 52
00390 #define DELTAP 53
00391 #define BERNOULLIFUNCTION 54
00392 #define COUNTFUNCTION 55
00393 #define MATCHFUNCTION 56
00394 #define PATTERNFUNCTION 57
00395 #define TERMFUNCTION 58
00396 #define CONJUGATION 59
00397 #define ROOTFUNCTION 60
00398 #define TABLEFUNCTION 61
00399 #define FIRSTBRACKET 62
00400 #define TERMSINEXPR 63
00401 #define NUMTERMSFUN 64
00402 #define GCDFUNCTION 65
00403 #define DIVFUNCTION 66
00404 #define REMFUNCTION 67
00405 #define MAXPOWEROF 68
00406 #define MINPOWEROF 69
00407 #define TABLESTUB 70
00408 #define FACTORIN 71
00409 #define TERMSINBRACKET 72
00410 #define WILDARGFUN 73
00411 /*
00412     In the past we would add new functions here and raise the numbers
00413     on the special reserved names. This is impractical in the light of
00414     the .sav files. The .sav files need a mechanism that contains the
00415     value of MAXBUILTINFUNCTION at the moment of writing. This allows
00416     form corrections if this value has changed in the mean time.
00417 */
00418 #define SQRTFUNCTION 74
00419 #define LNFUNCTION 75
00420 #define SINFUNCTION 76
00421 #define COSFUNCTION 77
00422 #define TANFUNCTION 78
00423 #define ASINFUNCTION 79
00424 #define ACOSFUNCTION 80
00425 #define ATANFUNCTION 81
```

```
00426 #define ATAN2FUNCTION 82
00427 #define SINHFUNCTION 83
00428 #define COSHFUNCTION 84
00429 #define TANHFUNCTION 85
00430 #define ASINHFUNCTION 86
00431 #define ACOSHFUNCTION 87
00432 #define ATANHFUNCTION 88
00433 #define LI2FUNCTION 89
00434 #define LINFUNCTION 90
00435
00436 #define EXTRASYMFUN 91
00437 #define RANDOMFUNCTION 92
00438 #define RANPERM 93
00439 #define NUMFACTORS 94
00440 #define FIRSTTERM 95
00441 #define CONTENTTERM 96
00442 #define PRIMENUMBER 97
00443 #define EXTEUCLIDEAN 98
00444 #define MAKERATIONAL 99
00445 #define INVERSEFUNCTION 100
00446 #define IDFUNCTION 101
00447 #define PUTFIRST 102
00448 #define PERMUTATIONS 103
00449 #define PARTITIONS 104
00450 #define MULFUNCTION 105
00451 #define MOEBIUS 106
00452 #define TOPOLOGIES 107
00453 #define DIAGRAMS 108
00454 #define TOPO 109
00455 #define NODEFUNCTION 110
00456 #define EDGE 111
00457 #define SIZEOFFUNCTION 112
00458 #define BLOCK 113
00459 #define ONEPI 114
00460 #define PHI 115
00461 #define FLOATFUN 116
00462 #define TOFLOAT 117
00463 #define TORAT 118
00464 #define MZV 119
00465 #define EULER 120
00466 #define MZVHALF 121
00467 #define AGMFUNCTION 122
00468 #define GAMMAFUN 123
00469 #define EXPFUNCTION 124
00470 #define HPLFUNCTION 125
00471 #define MPLFUNCTION 126
00472 #define MAXBUILTINFUNCTION 126
00473
00474 #define FIRSTUSERFUNCTION 150
00475
00476 #define ALLFUNCTIONS -1
00477 #define ALLMZVFUNCTIONS -2
00478
00479 /*
00480     Note: if we add a builtin table we have to look also inside names.c
00481     in the routine Globalize because there we assume there does not exist
00482     such an object
00483 */
00484
00485 #define ISYMBOL 0
00486 #define PISYMBOL 1
00487 #define COEFFSYMBOL 2
00488 #define NUMERATORSYMBOL 3
00489 #define DENOMINATORSYMBOL 4
00490 #define WILDARGSYMBOL 5
00491 #define DIMENSIONSYMBOL 6
00492 #define FACTORSYMBOL 7
00493 #define SEPARATESYMBOL 8
00494 #define EESYMBOL 9
00495 #define EMSYMBOL 10
00496
00497 #define BUILTINSYMBOLS 11
00498 #define FIRSTUSERSYMBOL 20
00499
00500 #define BUILTININDICES 1
00501 #define BUILTINVECTORS 1
00502 #define BUILTINDOLLARS 1
00503
00504 #define WILDARGVECTOR 0
00505 #define WILDARGINDEX 0
00506
00507 /*
00508     The objects that have a name that starts with TYPE are codes of statements
00509     made by the compiler. Each statement starts with such a code, followed by
00510     its size. For how most of these statements are used can be seen in the
00511     Generator function in the file proces.c
00512     TYPEOPERATION is an anachronism that remains used only for the statements
```

```
00513     that are executed in the file opera.c (like traces and contractions).
00514 */
00515
00516 #define TYPEEXPRESSION 0
00517 #define TYPEIDNEW 1
00518 #define TYPEIDOLD 2
00519 #define TYPEOPERATION 3
00520 #define TYPEREPEAT 4
00521 #define TYPEENDREPEAT 5
00522 /*
00523     The next counts must be higher than the ones before
00524 */
00525 #define TYPECOUNT 20
00526 #define TYPEMULT 21
00527 #define TYPEGOTO 22
00528 #define TYPEDISCARD 23
00529 #define TYPEIF 24
00530 #define TYPEELSE 25
00531 #define TYPEELIF 26
00532 #define TYPEENDIF 27
00533 #define TYPESUM 28
00534 #define TYPECHISHOLM 29
00535 #define TYPEREVERSE 30
00536 #define TYPEARG 31
00537 #define TYPENORM 32
00538 #define TYPENORM2 33
00539 #define TYPENORM3 34
00540 #define TYPEEXIT 35
00541 #define TYPESETEXIT 36
00542 #define TYPEPRINT 37
00543 #define TYPEFPRINT 38
00544 #define TYPEREDEFPRE 39
00545 #define TYPESPLITARG 40
00546 #define TYPESPLITARG2 41
00547 #define TYPEFACTARG 42
00548 #define TYPEFACTARG2 43
00549 #define TYPETRY 44
00550 #define TYPEASSIGN 45
00551 #define TYPERENUMBER 46
00552 #define TYPESUMFIX 47
00553 #define TYPEFINDLOOP 48
00554 #define TYPEUNRAVEL 49
00555 #define TYPEADJUSTBOUNDS 50
00556 #define TYPEINSIDE 51
00557 #define TYPETERM 52
00558 #define TYPESORT 53
00559 #define TYPEDETCURDUM 54
00560 #define TYPEINEXPRESSION 55
00561 #define TYPESPLITFIRSTARG 56
00562 #define TYPESPLITLASTARG 57
00563 #define TYPEMERGE 58
00564 #define TYPETESTUSE 59
00565 #define TYPEAPPLY 60
00566 #define TYPEAPPLYRESET 61
00567 #define TYPECHAININ 62
00568 #define TYPECHAINOUT 63
00569 #define TYPENORM4 64
00570 #define TYPEFACTOR 65
00571 #define TYPEARGIMPLODE 66
00572 #define TYPEARGEXPLODE 67
00573 #define TYPEDENOMINATORS 68
00574 #define TYPESTUFFLE 69
00575 #define TYPEDROPCOEFFICIENT 70
00576 #define TYPETRANSFORM 71
00577 #define TYPETOPOLYNOMIAL 72
00578 #define TYPEFROMPOLYNOMIAL 73
00579 #define TYPEDOLOOP 74
00580 #define TYPEENDDOLOOP 75
00581 #define TYPEDROPSYMBOLS 76
00582 #define TYPEPUTINSIDE 77
00583 #define TYPETOSPECTATOR 78
00584 #define TYPEARGTOEXTRASYMBOL 79
00585 #define TYPECANONICALIZE 80
00586 #define TYPESWITCH 81
00587 #define TYPEENDSWITCH 82
00588 #define TYPESETUSERFLAG 83
00589 #define TYPECLEARUSERFLAG 84
00590 #define TYPEALLLOOPS 85
00591 #define TYPEALLPATHS 86
00592 #ifdef WITHFLOAT
00593 #define TYPEEVALUATE 87
00594 #define TYPEEXPAND 88
00595 #define TYPETOFLOAT 89
00596 #define TYPETORAT 90
00597 #define TYPESTRICTROUNDING 91
00598 #define TYPECHOP 92
00599 #endif
```

```
00600
00601 /*
00602     The codes for the 'operations' that are part of TYPEOPERATION.
00603 */
00604
00605 #define TAKETRACE 1
00606 #define CONTRACT 2
00607 #define RATIO 3
00608 #define SYMMETRIZE 4
00609 #define TENVEC 5
00610 #define SUMNUM1 6
00611 #define SUMNUM2 7
00612
00613 /*
00614     The various types of wildcards.
00615 */
00616
00617 #define WILDDUMMY 0
00618 #define SYMTONUM 1
00619 #define SYMTOSYM 2
00620 #define SYMTOSUB 3
00621 #define VECTOMIN 4
00622 #define VECTOVEC 5
00623 #define VECTOSUB 6
00624 #define INDTOIND 7
00625 #define INDTOSUB 8
00626 #define FUNTOFUN 9
00627 #define ARGTOARG 10
00628 #define ARLTOARL 11
00629 #define EXPTOEXP 12
00630 #define FROMBRAC 13
00631 #define FROMSET 14
00632 #define SETTONUM 15
00633 #define WILDCARDS 16
00634 #define SETNUMBER 17
00635 #define LOADDOLLAR 18
00636 /*
00637     Some new types of wildcards that hold only for function arguments.
00638 */
00639 #define NUMTONUM 20
00640 #define NUMTOSYM 21
00641 #define NUMTOIND 22
00642 #define NUMTOSUB 23
00643 /*
00644     Flag or-ed with ARGTOARG to give the number of arguments.
00645 */
00646 #define EATTENSOR 0x2000
00647
00648 /*
00649     Dirty flags (introduced when functions got a field with a dirty flag)
00650 */
00651
00652 #define CLEANFLAG 0
00653 #define DIRTYFLAG 1
00654 #define DIRTYSYMFLAG 2
00655 #define MUSTCLEANPRF 4
00656 #define SUBTERMUSED1 8
00657 #define SUBTERMUSED2 16
00658 #define ALLDIRTY (DIRTYFLAG|DIRTYSYMFLAG)
00659
00660 #define ARGHEAD 2
00661 #define FUNHEAD 3
00662 #define SUBEXPSIZE 5
00663 #define EXPRHEAD 5
00664 #define TYPEARGHEADSIZE 6
00665
00666 /*
00667     Actions to be taken with expressions. They are marked with these objects
00668     during compilation.
00669 */
00670
00671 #define SKIP 1
00672 #define DROP 2
00673 #define HIDE 3
00674 #define UNHIDE 4
00675 #define INTOHIDE 5
00676
00677 /*
00678     Types of expressions
00679 */
00680
00681 #define LOCALEXPRESSION 0
00682 #define SKIPLEXPRESSION 1
00683 #define DROPLEXPRESSION 2
00684 #define DROPPEDEXPRESSION 3
00685 #define GLOBALEXPRESSION 4
00686 #define SKIPGEXPRESSION 5
```

```
00687 #define DROPGEXPRESSION 6
00688 #define STOREDEXPRESSION 8
00689 #define HIDDENLEXPRESSION 9
00690 #define HIDDENGEXPRESSION 13
00691 #define INCEXPRESSION 9
00692 #define HIDELEXPRESSION 10
00693 #define HIDEGEXPRESSION 14
00694 #define DROPHLEXPRESSION 11
00695 #define DROPHGEXPRESSION 15
00696 #define UNHIDELEXPRESSION 12
00697 #define UNHIDEGEXPRESSION 16
00698 #define INTOHIDELEXPRESSION 17
00699 #define INTOHIDEGEXPRESSION 18
00700 #define SPECTATOREXPRESSION 19
00701 #define DROPSPECTATOREXPRESSION 20
00702 #define SKIPUNHIDELEXPRESSION 21
00703 #define SKIPUNHIDEGEXPRESSION 22
00704
00705 #define PRINTOFF 0
00706 #define PRINTON 1
00707 #define PRINTCONTENTS 2
00708 #define PRINTCONTENT 3
00709 #define PRINTLFILE 4
00710 #define PRINTONETERM 8
00711 #define PRINTONEFUNCTION 16
00712 #define PRINTALL 32
00713
00714 /*
00715     Special codes for the replace variable in the EXPRESSIONS struct
00716 */
00717
00718 #define REGULAREXPRESSION -1
00719 #define REDEFINEDEXPRESSION -2
00720 #define NEWLYDEFINEDEXPRESSION -3
00721
00729 #define VERTEXFUNCTION -1
00730 #define GENERALFUNCTION 0
00731 #define TENSORFUNCTION 2
00732 #define GAMMAFUNCTION 4
00735 /*
00736     Special sets
00737 */
00738
00739 #define POS_         0   /* integer > 0 */
00740 #define POS0_        1   /* integer >= 0 */
00741 #define NEG_         2   /* integer < 0 */
00742 #define NEG0_        3   /* integer <= 0 */
00743 #define EVEN_        4   /* integer (even) */
00744 #define ODD_         5   /* integer (odd) */
00745 #define Z_           6   /* integer */
00746 #define SYMBOL_      7   /* symbol only */
00747 #define FIXED_       8   /* fixed index */
00748 #define INDEX_       9   /* index only */
00749 #define Q_          10   /* rational */
00750 #define DUMMYINDEX_ 11   /* dummy index only */
00751 #define VECTOR_     12   /* vector only */
00752
00753 /*
00754     Special indices.
00755 */
00756
00757 #define GAMMA1 0
00758 #define GAMMA5 -1
00759 #define GAMMA6 -2
00760 #define GAMMA7 -3
00761 #define FUNNYVEC -4
00762 #define FUNNYWILD -5
00763 #define SUMMEDIND -6
00764 #define NOINDEX -7
00765 #define FUNNYDOLLAR -8
00766 #define EMPTYINDEX -9
00767
00768 /*
00769     The next one should be less than all of the above special indices.
00770 */
00771
00772 #define MINSPEC -10
00773
00774 #define USEDFLAG 2
00775 #define DUMMYFLAG 1
00776
00777 #define MAINSORT 0
00778 #define FUNCTIONSORT 1
00779 #define SUBSORT 2
00780
00781 #define FLOATMODE 1
00782 #define RATIONALMODE 0
```

```
00783
00784 #define NUMSPECSETS 10
00785
00786 #define ISZERO 1
00787 #define ISUNMODIFIED 2
00788 #define ISCOMPRESSED 4
00789 #define ISINRHS 8
00790 #define ISFACTORIZED 16
00791 #define TOBEFACTORED 32
00792 #define TOBEUNFACTORED 64
00793 #define KEEPZERO 128
00794
00795 #define VARTYPENONE 0
00796 #define VARTYPECOMPLEX 1
00797 #define VARTYPEIMAGINARY 2
00798 #define VARTYPEROOTOFUNITY 4
00799 #define VARTYPEMINUS 8
00800 #define CYCLESYMMETRIC 1
00801 #define RCYCLESYMMETRIC 2
00802 #define SYMMETRIC 3
00803 #define ANTISYMMETRIC 4
00804 #define REVERSEORDER 256
00805
00806 /*
00807     Types of id statements (substitutions)
00808 */
00809
00810 #define SUBMULTI 1
00811 #define SUBONCE 2
00812 #define SUBONLY 3
00813 #define SUBMANY 4
00814 #define SUBVECTOR 5
00815 #define SUBSELECT 6
00816 #define SUBALL 7
00817 #define SUBMASK 15
00818 #define SUBDISORDER 16
00819 #define SUBAFTER 32
00820 #define SUBAFTERNOT 64
00821
00822 #define IDHEAD 6
00823
00824 #define DOLLARFLAG 1
00825 #define NORMALIZEFLAG 2
00826
00827 #define GIDENT 1
00828 #define GFIVE 4
00829 #define GPLUS 3
00830 #define GMINUS 2
00831
00832 /*
00833     Types of objects inside an if clause.
00834 */
00835
00836 #define LONGNUMBER 1
00837 #define MATCH 2
00838 #define COEFFI 3
00839 #define SUBEXPR 4
00840 #define MULTIPLEOF 5
00841 #define IFDOLLAR 6
00842 #define IFEXPRESSION 7
00843 #define IFDOLLAREXTRA 8
00844 #define IFISFACTORIZED 9
00845 #define IFOCCURS 10
00846 #define IFUSERFLAG 11
00847 #define IFFLOATNUMBER 12
00848 #define GREATER 0
00849 #define GREATEREQUAL 1
00850 #define LESS 2
00851 #define LESSEQUAL 3
00852 #define EQUAL 4
00853 #define NOTEQUAL 5
00854 #define ORCOND 6
00855 #define ANDCOND 7
00856 #define DUMMY 1
00857 #define SORT 1
00858 #define STORE 2
00859 #define END 3
00860 #define GLOBAL 4
00861 #define CLEAR 5
00862
00863 #define VECTBIT 1
00864 #define DOTPBIT 2
00865 #define FUNBIT  4
00866 #define SETBIT  8
00867
00868 #define EXTRAPARAMETER 0x4000
00869 #define GENCOMMUTE 0
```

```
00870 #define GENNONCOMMUTE 0x2000
00871
00872 #define NAMENOTFOUND -9
00873
00874 /*
00875     Types of dollar expressions.
00876 */
00877
00878 #define DOLUNDEFINED 0
00879 #define DOLNUMBER 1
00880 #define DOLARGUMENT 2
00881 #define DOLSUBTERM 3
00882 #define DOLTERMS 4
00883 #define DOLWILDARGS 5
00884 #define DOLINDEX 6
00885 #define DOLZERO 7
00886
00887 #define FINDLOOP 0
00888 #define REPLACELOOP 1
00889
00890 #define NOFUNPOWERS 0
00891 #define COMFUNPOWERS 1
00892 #define ALLFUNPOWERS 2
00893
00894 #define PROPERORDERFLAG 0
00895
00896 #define REGULAR 0
00897 #define FINISH 1
00898
00899 #define POLYADD 1
00900 #define POLYSUB 2
00901 #define POLYMUL 3
00902 #define POLYDIV 4
00903 #define POLYREM 5
00904 #define POLYGCD 6
00905 #define POLYINTFAC 7
00906 #define POLYNORM 8
00907
00908 #define MODNONE 0
00909 #define MODSUM 1
00910 #define MODMAX 2
00911 #define MODMIN 3
00912 #define MODLOCAL 4
00913
00914 #define ELEMENTUSED 1
00915 #define ELEMENTLOADED 2
00916 /*
00917     Variables for the modulus statement, flags in AC.modmode
00918     For explanation, see CoModulus
00919 */
00920 #define POSNEG 0x1
00921 #define INVERSETABLE 0x2
00922 #define COEFFICIENTSONLY 0x4
00923 #define ALSOPOWERS 0x8
00924 #define ALSOFUNARGS 0x10
00925 #define ALSODOLLARS 0x20
00926 #define NOINVERSES 0x40
00927
00928 #define POSITIVEONLY 0
00929 #define UNPACK 0x80
00930 #define NOUNPACK 0
00931 #define FROMFUNCTION 0x100
00932
00933 #define VARNAMES 0
00934 #define AUTONAMES 1
00935 #define EXPRNAMES 2
00936 #define DOLLARNAMES 3
00937
00938 #ifdef WITHPTHREADS
00939 /*
00940     Signals that the workers have to react to
00941 */
00942
00943 #define TERMINATETHREAD -1
00944 #define STARTNEWEXPRESSION 1
00945 #define LOWESTLEVELGENERATION 2
00946 #define FINISHEXPRESSION 3
00947 #define CLEANUPEXPRESSION 4
00948 #define HIGHERLEVELGENERATION 5
00949 #define STARTNEWMODULE 6
00950 #define CLAIMOUTPUT 7
00951 #define FINISHEXPRESSION2 8
00952 #define INISORTBOT 7
00953 #define RUNSORTBOT 8
00954 #define DOONEEXPRESSION 9
00955 #define DOBRACKETS 10
00956 #define CLEARCLOCK 11
```

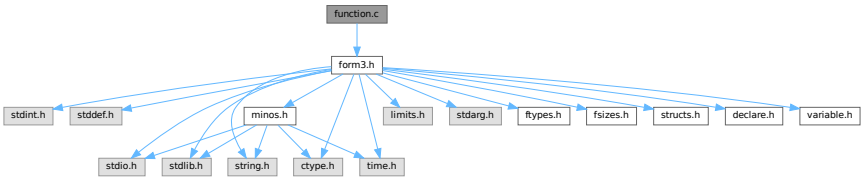
```
00957 #define MCTSEXPANDTREE 12
00958 #define OPTIMIZEEXPRESSION 13
00959
00960 #define MASTERBUFFERISFULL 1
00961
00962 /*
00963     Bucket states
00964 */
00965
00966 #define BUCKETFREE 1
00967 #define BUCKETINUSE 0
00968 #define BUCKETCOMINGFREE 2
00969 #define BUCKETFILLED -1
00970 #define BUCKETATEND -2
00971 #define BUCKETTERMINATED 3
00972 #define BUCKETRELEASED 4
00973
00974 #define BUCKETDOINGTERM 1
00975 #define BUCKETASSIGNED -1
00976 #define BUCKETTOBERELEASED -2
00977 #define BUCKETPREPARINGTERM 0
00978
00979 #define BUCKETDOINGTERMS 0
00980 #define BUCKETDOINGBRACKET 1
00981
00982 /*
00983     Sortblock config
00984 */
00985 // The number of blocks in the ring-buffer for data transfer between
00986 // workers or sortbots and the master thread or other sortbots. With
00987 // sortbots, each thread has half of this number.
00988 // The merging algorithm requires at least 4 blocks, so this must be
00989 // at least 8, with sortbots.
00990 #define NUMBEROFBLOCKSINSORT 8
00991 // The minimum number of terms which must fit in a block, i.e. they
00992 // must be at least this number times MaxTermSize.
00993 #define MINIMUMNUMBEROFTERMS 1
00994 // The minimum number of terms which will be written in a block,
00995 // before potentially yielding the block to a waiting reading thread.
00996 #define MINWRITENUMBEROFTERMS 1
00997
00998 // The minimum number of terms which should fit in a bucket
00999 #define BUCKETMINTERMS 1
01000
01001 #endif
01002
01003 /*
01004     The next variable is because there is some use of cbufnum that is
01005     probably irrelevant. We use here DUMMYBUFNUM instead of AC.cbufnum
01006     just in case we run into trouble later.
01007 */
01008 #define DUMMYBUFFER 1
01009
01010 #define ALLARGS      1
01011 #define NUMARG       2
01012 #define ARGRANGE     3
01013 #define MAKEARGS     4
01014 #define MAXRANGEINDICATOR 4
01015 #define REPLACEARG   5
01016 #define ENCODEARG    6
01017 #define DECODEARG    7
01018 #define IMPLODEARG   8
01019 #define EXPLODEARG   9
01020 #define PERMUTEARG  10
01021 #define REVERSEARG  11
01022 #define CYCLEARG    12
01023 #define ISLYNDON    13
01024 #define ISLYNDONR   14
01025 #define TOLYNDON    15
01026 #define TOLYNDONR   16
01027 #define ADDARG      17
01028 #define MULTIPLYARG 18
01029 #define DROPARG     19
01030 #define SELECTARG   20
01031 #define DEDUPARG    21
01032 #define ZTOHARG     22
01033 #define HTOZARG     23
01034 /*
01035 #define ZTOSARG     24
01036 #define STOZARG     25
01037 #define HTOSARG     26
01038 #define STOHARG     27
01039 */
01040
01041 #define BASECODE 1
01042 #define YESLYNDON 1
01043 #define NOLYNDON 2
```

```
01044
01045 #define TOPOLYNOMIALFLAG 1
01046 #define FACTARGFLAG 2
01047
01048 #define OLDFACTARG 1
01049 #define NEWFACTARG 0
01050
01051 #define FROMMODULEOPTION 0
01052 #define FROMPOINTINSTRUCTION 1
01053
01054 #define EXTRASYMBOL 0
01055 #define REGULARSYMBOL 1
01056 #define EXPRESSIONNUMBER 2
01057
01058 #define O_NONE 0
01059 #define O_CSE 1
01060 #define O_CSEGREEDY 2
01061 #define O_GREEDY 3
01062
01063 #define O_OCCURRENCE 0
01064 #define O_MCTS 1
01065 #define O_SIMULATED_ANNEALING 2
01066
01067 #define O_FORWARD 0
01068 #define O_BACKWARD 1
01069 #define O_FORWARDORBACKWARD 2
01070 #define O_FORWARDANDBACKWARD 3
01071
01072 #define OPTHEAD 3
01073 #define DOALL 1
01074 #define ONLYFUNCTIONS 2
01075
01076 #define INUSE 1
01077 #define COULDCOMMUTE 2
01078 #define DOESNOTCOMMUTE 4
01079
01080 #define DICT_NONUMBERS 0
01081 #define DICT_INTEGERONLY 1
01082 #define DICT_RATIONALONLY 2
01083 #define DICT_ALLNUMBERS 3
01084 #define DICT_NOVARIABLES 0
01085 #define DICT_DOVARIABLES 1
01086 #define DICT_NOSPECIALS 0
01087 #define DICT_DOSPECIALS 1
01088 #define DICT_NOFUNWITHARGS 0
01089 #define DICT_DOFUNWITHARGS 1
01090 #define DICT_NOTINDOLLARS 0
01091 #define DICT_INDOLLARS 1
01092
01093 #define DICT_INTEGERNUMBER 1
01094 #define DICT_RATIONALNUMBER 2
01095 #define DICT_SYMBOL 3
01096 #define DICT_VECTOR 4
01097 #define DICT_INDEX 5
01098 #define DICT_FUNCTION 6
01099 #define DICT_FUNCTION_WITH_ARGUMENTS 7
01100 #define DICT_SPECIALCHARACTER 8
01101 #define DICT_RANGE 9
01102
01103 #define READSPECTATORFLAG 3
01104 #define GLOBALSPECTATORFLAG 1
01105
01106 #define ORDEREDSET 1
01107
01108 #define DENSETABLE 1
01109 #define SPARSETABLE 0
01110
01111 // Diagram generator flags. They should be powers of two, since they are added
01112 // to pass to diagrams and masked in diawrap.cc to check the flags.
01113 // We use a "stringified" version of these when defining the FORM preprocessor
01114 // variables, so can't neatly use "1«10" etc here.
01115 #define TOPOLOGIESONLY  1
01116 #define WITHOUTNODES    2
01117 #define WITHEDGES       4
01118 #define WITHBLOCKS      8
01119 #define WITHONEPISETS   16
01120 #define WITHSYMMETRIZEI 32
01121 #define WITHSYMMETRIZEF 64
01122 // This is not an "option" preprocessor var but is set by "external" particle definitions
01123 #define CHECKEXTERN     128
01124 // The "qgraf compatible" filtering flags:
01125 #define ONEPARTI        256
01126 #define ONEPARTR        512
01127 #define ONSHELL         1024
01128 #define OFFSHELL        2048
01129 #define NOSIGMA         4096
01130 #define SIGMA           8192
```

```
01131 #define NOSNAIL         16384
01132 #define SNAIL           32768
01133 #define NOTADPOLE       65536
01134 #define TADPOLE         131072
01135 #define SIMPLE          262144
01136 #define NOTSIMPLE       524288
01137 #define BIPART          1048576
01138 #define NONBIPART       2097152
01139 #define CYCLI           4194304
01140 #define CYCLR           8388608
01141 #define FLOOP           16777216
01142 #define NOTFLOOP        33554432
01143
```

4.55 function.c File Reference

`#include "form3.h"`
Include dependency graph for function.c:



Functions

- int MakeDirty (WORD *term, WORD *x, WORD par)
- void MarkDirty (WORD *term, WORD flags)
- void PolyFunDirty (PHEAD WORD *term)
- void PolyFunClean (PHEAD WORD *term)
- WORD Symmetrize (PHEAD WORD *func, WORD *Lijst, WORD ngroups, WORD gsize, WORD type)
- int CompGroup (PHEAD WORD type, WORD **args, WORD *a1, WORD *a2, WORD num)
- int FullSymmetrize (PHEAD WORD *fun, int type)
- int SymGen (PHEAD WORD *term, WORD *params, WORD num, WORD level)
- int SymFind (PHEAD WORD *term, WORD *params)
- int ChainIn (PHEAD WORD *term, WORD funnum)
- int ChainOut (PHEAD WORD *term, WORD funnum)
- int MatchFunction (PHEAD WORD *pattern, WORD *interm, WORD *wilds)
- int ScanFunctions (PHEAD WORD *inpat, WORD *inter, WORD par)

4.55.1 Detailed Description

The file with the central routines for the pattern matching of functions and their arguments. The file also contains the routines for the execution of the Symmetrize statement and its variations (like antisymmetrize etc).

Definition in file function.c.

4.55.2 Function Documentation

4.55.2.1 MakeDirty()

```
int MakeDirty (
            WORD * term,
            WORD * x,
            WORD par )
```

Definition at line 51 of file function.c.

4.55.2.2 MarkDirty()

```
void MarkDirty (
            WORD * term,
            WORD flags )
```

Definition at line 97 of file function.c.

4.55.2.3 PolyFunDirty()

```
void PolyFunDirty (
            PHEAD WORD * term )
```

Definition at line 139 of file function.c.

4.55.2.4 PolyFunClean()

```
void PolyFunClean (
            PHEAD WORD * term )
```

Definition at line 174 of file function.c.

4.55.2.5 Symmetrize()

```
WORD Symmetrize (
            PHEAD WORD * func,
            WORD * Lijst,
            WORD ngroups,
            WORD gsize,
            WORD type )
```

Definition at line 213 of file function.c.

4.55.2.6 CompGroup()

```
int CompGroup (
             PHEAD WORD type,
             WORD ** args,
             WORD * a1,
             WORD * a2,
             WORD num )
```

Definition at line 373 of file function.c.

4.55.2.7 FullSymmetrize()

```
int FullSymmetrize (
             PHEAD WORD * fun,
             int type )
```

Definition at line 473 of file function.c.

4.55.2.8 SymGen()

```
int SymGen (
             PHEAD WORD * term,
             WORD * params,
             WORD num,
             WORD level )
```

Definition at line 518 of file function.c.

4.55.2.9 SymFind()

```
int SymFind (
             PHEAD WORD * term,
             WORD * params )
```

Definition at line 633 of file function.c.

4.55.2.10 ChainIn()

```
int ChainIn (
             PHEAD WORD * term,
             WORD funnum )
```

Definition at line 691 of file function.c.

4.55.2.11 ChainOut()

```
int ChainOut (
             PHEAD WORD * term,
             WORD funnum )
```

Definition at line 748 of file function.c.

4.55.2.12 MatchFunction()

```
int MatchFunction (
              PHEAD WORD * pattern,
              WORD * interm,
              WORD * wilds )
```

Definition at line 826 of file function.c.

4.55.2.13 ScanFunctions()

```
int ScanFunctions (
              PHEAD WORD * inpat,
              WORD * inter,
              WORD par )
```

Definition at line 1618 of file function.c.

4.56 function.c

Go to the documentation of this file.

```
00001
00008 /* #[ License : */
00009 /*
00010  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00011  *   When using this file you are requested to refer to the publication
00012  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013  *   This is considered a matter of courtesy as the development was paid
00014  *   for by FOM the Dutch physics granting agency and we would like to
00015  *   be able to track its scientific use to convince FOM of its value
00016  *   for the community.
00017  *
00018  *   This file is part of FORM.
00019  *
00020  *   FORM is free software: you can redistribute it and/or modify it under the
00021  *   terms of the GNU General Public License as published by the Free Software
00022  *   Foundation, either version 3 of the License, or (at your option) any later
00023  *   version.
00024  *
00025  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00027  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00028  *   details.
00029  *
00030  *   You should have received a copy of the GNU General Public License along
00031  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00032  */
00033 /* #] License : */
00034 /*
00035     #[ Includes : function.c
00036 */
00037
00038 #include "form3.h"
00039
00040 /*
00041     #] Includes :
00042     #[ Utilities :
00043         #[ MakeDirty :
00044
00045         Routine finds the function with the address x in it
00046         and mark all arguments that contain x as dirty.
00047         if par == 0 term is a full term, else term is the start of a
00048         function
00049 */
00050
00051 int MakeDirty(WORD *term, WORD *x, WORD par)
00052 {
00053     WORD *next, *n;
00054     if ( !par ) {
00055         next = term; next += *term;
```

```
00056         next -= ABS(next[-1]);
00057         term++;
00058         if ( x < term ) return(0);
00059         if ( x >= next ) return(0);
00060         while ( term < next ) {
00061             n = term + term[1];
00062             if ( x < n ) break;
00063             term = n;
00064         }
00065 /*      next = n; */
00066     }
00067     else {
00068         next = term + term[1];
00069         if ( x < term || x >= next ) return(0);
00070     }
00071     if ( *term < FUNCTION ) return(0);
00072     if ( functions[*term-FUNCTION].spec >= TENSORFUNCTION ) return(0);
00073     term += FUNHEAD;
00074     if ( x < term ) return(0);
00075     next = term; NEXTARG(next)
00076     while ( x >= next ) { term = next; NEXTARG(next) }
00077     if ( *term < 0 ) return(0);
00078     term[1] = 1;
00079     term += ARGHEAD;
00080     if ( x < term ) return(1);
00081     next = term + *term;
00082     while ( x >= next ) { term = next; next += *next; }
00083     MakeDirty(term,x,0);
00084     return(1);
00085 }
00086
00087 /*
00088         #] MakeDirty :
00089         #[ MarkDirty :
00090
00091         Routine marks all functions dirty with the given flags.
00092         Is to be used when there is a possibility that symmetrization
00093         properties of functions may have changed. In that case we play
00094         it safe.
00095 */
00096
00097 void MarkDirty(WORD *term, WORD flags)
00098 {
00099     WORD *t, *r, *m, *tstop;
00100     GETSTOP(term,tstop);
00101     t = term+1;
00102     while ( t < tstop ) {
00103         if ( *t < FUNCTION ) { t += t[1]; continue; }
00104         t[2] |= flags;
00105         if ( *t < FUNCTION+WILDOFFSET && functions[*t-FUNCTION].spec > 0 ) {
00106             t += t[1]; continue;
00107         }
00108         if ( *t >= FUNCTION+WILDOFFSET && functions[*t-FUNCTION-WILDOFFSET].spec > 0 ) {
00109             t += t[1]; continue;
00110         }
00111         r = t + FUNHEAD;
00112         t += t[1];
00113         while ( r < t ) {
00114             if ( *r <= 0 ) {
00115                 if ( *r <= -FUNCTION ) r++;
00116                 else r += 2;
00117                 continue;
00118             }
00119             r[1] |= DIRTYFLAG;
00120             m = r + ARGHEAD;
00121             r += *r;
00122             while ( m < r ) {
00123                 MarkDirty(m,flags);
00124                 m += *m;
00125             }
00126         }
00127     }
00128 }
00129
00130 /*
00131         #] MarkDirty :
00132         #[ PolyFunDirty :
00133
00134         Routine marks the PolyFun or the PolyRatFun dirty.
00135         This is used when there is modular calculus and the modulus
00136         has changed for the current module.
00137 */
00138
00139 void PolyFunDirty(PHEAD WORD *term)
00140 {
00141     GETBIDENTITY
00142     WORD *t, *tstop, *endarg;
```

```
00143     tstop = term + *term;
00144     tstop -= ABS(tstop[-1]);
00145     t = term+1;
00146     while ( t < tstop ) {
00147         if ( *t == AR.PolyFun ) {
00148             if ( AR.PolyFunType == 2 ) t[2] |= MUSTCLEANPRF;
00149             endarg = t + t[1];
00150             t[2] |= DIRTYFLAG;
00151             t += FUNHEAD;
00152             while ( t < endarg ) {
00153                 if ( *t > 0 ) {
00154                     t[1] |= DIRTYFLAG;
00155                 }
00156                 NEXTARG(t);
00157             }
00158         }
00159         else {
00160             t += t[1];
00161         }
00162     }
00163 }
00164
00165 /*
00166         #] PolyFunDirty :
00167         #[ PolyFunClean :
00168
00169         Routine marks the PolyFun or the PolyRatFun clean.
00170         This is used when there is modular calculus and the modulus
00171         has changed for the current module.
00172 */
00173
00174 void PolyFunClean(PHEAD WORD *term)
00175 {
00176     GETBIDENTITY
00177     WORD *t, *tstop;
00178     tstop = term + *term;
00179     tstop -= ABS(tstop[-1]);
00180     t = term+1;
00181     while ( t < tstop ) {
00182         if ( *t == AR.PolyFun ) {
00183             t[2] &= ~MUSTCLEANPRF;
00184         }
00185         t += t[1];
00186     }
00187 }
00188
00189 /*
00190         #] PolyFunClean :
00191         #[ Symmetrize :
00192
00193         (Anti)Symmetrizes the arguments of a function.
00194         Nlist tells of how many arguments are involved.
00195         Nlist == 0      All arguments must be sorted.
00196         Nlist > 0       Arguments mentioned are to be sorted, rest skipped.
00197         type = SYMMETRIC       Full symmetrization
00198         type = ANTISYMMETRIC:  Full symmetrization
00199         type = CYCLESYMMETRIC: Cyclic
00200         type = RCYCLESYMMETRIC:Cyclic or reverse
00201         Return value: OR of:
00202             0 even, 1 odd
00203             2 equal groups
00204             4 there was a permutation.
00205
00206         The information in Lijst tells what grouping is to be applied.
00207         The information is:
00208         ngroups number of groups
00209         gsize size of groups
00210         Lijst[0]....  The groups.
00211 */
00212
00213 WORD Symmetrize(PHEAD WORD *func, WORD *Lijst, WORD ngroups, WORD gsize,
00214                 WORD type)
00215 {
00216     GETBIDENTITY
00217     WORD **args,**arg,nargs;
00218     WORD *to, *r, *fstop;
00219     WORD i, j, k, ff, exch, nexch, neq;
00220     WORD *a1, *a2, *a3;
00221     WORD reverseorder;
00222     if ( ( type & REVERSEORDER ) != 0 ) reverseorder = -1;
00223     else                                reverseorder = 1;
00224     type &= ~REVERSEORDER;
00225
00226     ff = ( *func > FUNCTION ) ? functions[*func-FUNCTION].spec: 0;
00227
00228     if ( 2*func[1] > AN.arglistsize ) {
00229         if ( AN.arglist ) M_free(AN.arglist,"Symmetrize");
```

```
00230         AN.arglistsize = 2*func[1] + 8;
00231         AN.arglist = (WORD **)Malloc1(AN.arglistsize*sizeof(WORD *),"Symmetrize");
00232     }
00233     arg = args = AN.arglist;
00234     to = AT.WorkPointer;
00235     r = func;
00236     fstop = r + r[1];
00237     r += FUNHEAD;
00238     nargs = 0;
00239     while ( r < fstop ) {   /* Make list of arguments */
00240         *arg++ = r;
00241         nargs++;
00242         if ( ff ) {
00243             if ( *r == FUNNYWILD ) r++;
00244             r++;
00245         }
00246         else { NEXTARG(r); }
00247     }
00248     exch = 0;
00249     nexch = 0;
00250     neq = 0;
00251     a1 = Lijst;
00252     if ( type == SYMMETRIC || type == ANTISYMMETRIC ) {
00253     for ( i = 1; i < ngroups; i++ ) {
00254         a3 = a2 = a1 + gsize;
00255         k = reverseorder*CompGroup(BHEAD ff,args,a1,a2,gsize);
00256         if ( k < 0 ) {
00257             j = i-1;
00258             for(;;) {
00259                 for ( k = 0; k < gsize; k++ ) {
00260                     r = args[a1[k]]; args[a1[k]] = args[a2[k]]; args[a2[k]] = r;
00261                 }
00262                 exch ^= 1;
00263                 nexch = 4;
00264                 if ( j <= 0 ) break;
00265                 a1 -= gsize;
00266                 a2 -= gsize;
00267                 k = reverseorder*CompGroup(BHEAD ff,args,a1,a2,gsize);
00268                 if ( k == 0 ) neq = 2;
00269                 if ( k >= 0 ) break;
00270                 j--;
00271             }
00272         }
00273         else if ( k == 0 ) neq = 2;
00274         a1 = a3;
00275     }
00276     }
00277     else if ( type == CYCLESYMMETRIC || type == RCYCLESYMMETRIC ) {
00278         WORD rev = 0, jmin = 0, ii, iimin;
00279 recycle:
00280         for ( j = 1; j < ngroups; j++ ) {
00281             for ( i = 0; i < ngroups; i++ ) {
00282                 iimin = jmin + i;
00283                 if ( iimin >= ngroups ) iimin -= ngroups;
00284                 ii = j + i;
00285                 if ( ii >= ngroups ) ii -= ngroups;
00286                 k = reverseorder*CompGroup(BHEAD ff,args,Lijst+gsize*iimin,Lijst+gsize*ii,gsize);
00287                 if ( k > 0 ) break;
00288                 if ( k < 0 ) { jmin = j; nexch = 4; break; }
00289             }
00290         }
00291         if ( type == RCYCLESYMMETRIC && rev == 0 && ngroups > 1 ) {
00292             for ( j = 0; j < ngroups; j++ ) {
00293                 for ( i = 0; i < ngroups; i++ ) {
00294                     iimin = jmin + i;
00295                     if ( iimin >= ngroups ) iimin -= ngroups;
00296                     ii = j - i;
00297                     if ( ii < 0 ) ii += ngroups;
00298                     k = reverseorder*CompGroup(BHEAD ff,args,Lijst+gsize*iimin,Lijst+gsize*ii,gsize);
00299                     if ( k > 0 ) break;
00300                     if ( k < 0 ) {
00301                         nexch = 4;
00302                         jmin = 0;
00303                         a1 = Lijst;
00304                         a2 = Lijst + gsize * (ngroups-1);
00305                         while ( a2 > a1 ) {
00306                             for ( k = 0; k < gsize; k++ ) {
00307                                 r = args[a1[k]];
00308                                 args[a1[k]] = args[a2[k]];
00309                                 args[a2[k]] = r;
00310                             }
00311                             a1 += gsize; a2 -= gsize;
00312                         }
00313                         rev = 1;
00314                         goto recycle;
00315                     }
00316                 }
```

```
00317             }
00318         }
00319         if ( jmin != 0 ) {
00320             arg = AN.arglist + func[1];
00321             a1 = Lijst + gsize * jmin;
00322             k = gsize * ngroups;
00323             a2 = Lijst + k;
00324             for ( i = 0; i < k; i++ ) {
00325                 if ( a1 >= a2 ) a1 = Lijst;
00326                 *arg++ = args[*a1++];
00327             }
00328             arg = AN.arglist + func[1];
00329             a1 = Lijst;
00330             for ( i = 0; i < k; i++ ) args[*a1++] = *arg++;
00331         }
00332     }
00333     r = func;
00334     i = FUNHEAD;
00335     NCOPY(to,r,i);
00336     for ( i = 0; i < nargs; i++ ) {
00337         if ( ff ) {
00338             if ( *(args[i]) == FUNNYWILD ) {
00339                 *to++ = *(args[i]);
00340                 *to++ = args[i][1];
00341             }
00342             else *to++ = *(args[i]);
00343         }
00344         else if ( ( j = *args[i] ) < 0 ) {
00345             *to++ = j;
00346             if ( j > -FUNCTION ) *to++ = args[i][1];
00347         }
00348         else {
00349             r = args[i];
00350             NCOPY(to,r,j);
00351         }
00352     }
00353     i = func[1];
00354     to = func;
00355     r = AT.WorkPointer;
00356     NCOPY(to,r,i);
00357     return ( exch | nexch | neq );
00358 }
00359
00360 /*
00361         #] Symmetrize :
00362         #[ CompGroup :
00363
00364             Routine compares two groups of arguments
00365             The arguments are in args[a1[i]] and args[a2[i]]
00366             for i = 0 to num
00367             type indicates the type of function.
00368             return value: -1 if there should be an exchange
00369             0 if they are equal
00370             1 if they are OK.
00371 */
00372
00373 int CompGroup(PHEAD WORD type, WORD **args, WORD *a1, WORD *a2, WORD num)
00374 {
00375     GETBIDENTITY
00376     WORD *t1, *t2, i1, i2, n, k;
00377
00378     for ( n = 0; n < num; n++ ) {
00379         t1 = args[a1[n]]; t2 = args[a2[n]];
00380         if ( type >= TENSORFUNCTION ) {
00381             if ( AR.Eside == LHSIDE || AR.Eside == LHSIDEX ) {
00382                 if ( *t1 == FUNNYWILD ) {
00383                     if ( *t2 == FUNNYWILD ) {
00384                         if ( t1[1] < t2[1] ) return(1);
00385                         if ( t1[1] > t2[1] ) return(-1);
00386                     }
00387                     return(-1);
00388                 }
00389                 else if ( *t2 == FUNNYWILD ) {
00390                     return(1);
00391                 }
00392                 else {
00393                     if ( *t1 < *t2 ) return(1);
00394                     if ( *t1 > *t2 ) return(-1);
00395                 }
00396             }
00397             else {
00398                 if ( *t1 < *t2 ) return(1);
00399                 if ( *t1 > *t2 ) return(-1);
00400             }
00401         }
00402         else if ( type == 0 ) {
00403             if ( AC.properorderflag ) {
```

```
00404                 k = CompArg(t1,t2);
00405                 if ( k < 0 ) return(1);
00406                 if ( k > 0 ) return(-1);
00407                 NEXTARG(t1)
00408                 NEXTARG(t2)
00409             }
00410             else {
00411                 if ( *t1 > 0 ) {
00412                     i1 = *t1 - ARGHEAD - 1;
00413                     t1 += ARGHEAD + 1;
00414                     if ( *t2 > 0 ) {
00415                         i2 = *t2 - ARGHEAD - 1;
00416                         t2 += ARGHEAD + 1;
00417                         while ( i1 > 0 && i2 > 0 ) {
00418                             if ( *t1 > *t2 ) return(-1);
00419                             else if ( *t1 < *t2 ) return(1);
00420                             i1--; i2--; t1++; t2++;
00421                         }
00422                         if ( i1 > 0 ) return(-1);
00423                         else if ( i2 > 0 ) return(1);
00424                     }
00425 /*
00426                     This seems to be a bug. Reported by Aneesh Monahar, 28-sep-2005
00427                     else return(1);
00428 */
00429                     else return(-1);
00430                 }
00431                 else if ( *t2 > 0 ) return(1);
00432                 else {
00433                     if ( *t1 != *t2 ) {
00434                         if ( *t1 <= -FUNCTION && *t2 <= -FUNCTION ) {
00435                             if ( *t1 < *t2 ) return(-1);
00436                             return(1);
00437                         }
00438                         else {
00439                             if ( *t1 < *t2 ) return(1);
00440                             return(-1);
00441                         }
00442                     }
00443                     if ( *t1 > -FUNCTION ) {
00444                         if ( t1[1] != t2[1] ) {
00445                             if ( t1[1] < t2[1] ) return(1);
00446                             return(-1);
00447                         }
00448                     }
00449                 }
00450             }
00451         }
00452     }
00453     return(0);
00454 }
00455
00456 /*
00457         #] CompGroup :
00458         #[ FullSymmetrize :
00459
00460         Relay function for Normalize to execute a full symmetrization
00461         of a function fun. It hooks into Symmetrize according to the
00462         calling conventions for it.
00463         type = 0: Symmetrize
00464         type = 1: AntiSymmetrize
00465         type = 2: CycleSymmetrize
00466         type = 3: RCycleSymmetrize
00467         Return values:
00468         bit 0: odd permutation
00469         bit 1: identical arguments
00470         bit 2: there was a permutation.
00471 */
00472
00473 int FullSymmetrize(PHEAD WORD *fun, int type)
00474 {
00475     GETBIDENTITY
00476     WORD *Lijst, count = 0;
00477     WORD *t, *funstop, i;
00478     int retval;
00479
00480     if ( functions[*fun-FUNCTION].spec > 0 ) {
00481         count = fun[1] - FUNHEAD;
00482         for ( i = fun[1]-1; i >= FUNHEAD; i-- ) {
00483             if ( fun[i] == FUNNYWILD ) count--;
00484         }
00485     }
00486     else {
00487         funstop = fun + fun[1];
00488         t = fun + FUNHEAD;
00489         while ( t < funstop ) { count++; NEXTARG(t) }
00490     }
```

```
00491     if ( count < 2 ) {
00492         fun[2] &= ~DIRTYSYMFLAG;
00493         return(0);
00494     }
00495     Lijst = AT.WorkPointer;
00496     for ( i = 0; i < count; i++ ) Lijst[i] = i;
00497     AT.WorkPointer += count;
00498     retval = Symmetrize(BHEAD fun,Lijst,count,1,type);
00499     fun[2] &= ~DIRTYSYMFLAG;
00500     AT.WorkPointer = Lijst;
00501     return(retval);
00502 }
00503
00504 /*
00505         #] FullSymmetrize :
00506         #[ SymGen :
00507
00508         Routine does the outer work in the symmetrization.
00509         It locates the function(s) and loads up the parameters.
00510         It also studies the result.
00511
00512         if params[4] = -1 and no extra -> all
00513                                 extra -> strip groups with elements too large
00514                         0  -> if group with element too large: nofun
00515                         >0 -> must have right number of arguments
00516 */
00517
00518 int SymGen(PHEAD WORD *term, WORD *params, WORD num, WORD level)
00519 {
00520     GETBIDENTITY
00521     WORD *t, *r, *m;
00522     WORD i, j, k, c1, c2, ngroup;
00523     WORD *rstop, Nlist, *inLijst, *Lijst, sign = 1, sumch = 0, count;
00524     DUMMYUSE(num);
00525     c1 = params[3];     /* function number */
00526     c2 = FUNCTION + WILDOFFSET;
00527     Nlist = params[4];
00528     if ( Nlist < 0 ) Nlist = 0;
00529     else Nlist = params[0] - 7;
00530     t = term;
00531     m = t + *t;
00532     m -= ABS(m[-1]);
00533     t++;
00534     while ( t < m ) {
00535         if ( *t == c1 || c1 > c2 ) {    /* Candidate function */
00536             if ( *t >= FUNCTION && functions[*t-FUNCTION].spec
00537             >= TENSORFUNCTION ) {
00538                 count = t[1] - FUNHEAD;
00539             }
00540             else {
00541                 count = 0;
00542                 r = t;
00543                 rstop = t + t[1];
00544                 r += FUNHEAD;
00545                 while ( r < rstop ) { count++; NEXTARG(r) }
00546             }
00547             if ( ( j = params[4] ) > 0 && j != count ) goto NextFun;
00548             if ( j == 0 ) {
00549                 inLijst = params+7;
00550                 for ( i = 0; i < Nlist; i++ )
00551                     if ( inLijst[i] > count-1 ) goto NextFun;
00552             }
00553
00554             if ( Nlist > (params[0] - 7) ) Nlist = params[0] - 7;
00555             Lijst = AT.WorkPointer;
00556             inLijst = params + 7;
00557             ngroup = params[5];
00558             if ( Nlist > 0 && j < 0 ) {
00559                 k = 0;
00560                 for ( i = 0; i < ngroup; i++ ) {
00561                     for ( j = 0; j < params[6]; j++ ) {
00562                         if ( inLijst[j] > count+1 ) {
00563                             inLijst += params[6];
00564                             goto NextGroup;
00565                         }
00566                     }
00567                     j = params[6];
00568                     NCOPY(Lijst,inLijst,j);
00569                     k++;
00570 NextGroup:;
00571                 }
00572                 if ( k <= 1 ) goto NextFun;
00573                 ngroup = k;
00574                 inLijst = AT.WorkPointer;
00575                 AT.WorkPointer = Lijst;
00576                 Lijst = inLijst;
00577             }
```

```
00578             else if ( Nlist == 0 ) {
00579                 for ( i = 0; i < count; i++ ) Lijst[i] = i;
00580                 AT.WorkPointer += count;
00581                 ngroup = count;
00582             }
00583             else {
00584                 for ( i = 0; i < Nlist; i++ ) Lijst[i] = inLijst[i];
00585                 AT.WorkPointer += Nlist;
00586             }
00587             j = Symmetrize(BHEAD t,Lijst,ngroup,params[6],params[2]);
00588             AT.WorkPointer = Lijst;
00589             if ( params[2] == 4 ) { /* antisymmetric */
00590                 if ( ( j & 1 ) != 0 ) sign = -sign;
00591                 if ( ( j & 2 ) != 0 ) return(0); /* equal arguments */
00592             }
00593             if ( ( j & 4 ) != 0 ) sumch++;
00594             t[2] &= ~DIRTYSYMFLAG;
00595         }
00596 NextFun:
00597         t += t[1];
00598     }
00599     if ( sign < 0 ) {
00600         t = term;
00601         t += *t - 1;
00602         *t = -*t;
00603     }
00604     if ( sumch ) {
00605         if ( Normalize(BHEAD term) ) {
00606             MLOCK(ErrorMessageLock);
00607             MesCall("SymGen");
00608             MUNLOCK(ErrorMessageLock);
00609             return(-1);
00610         }
00611         if ( !*term ) return(0);
00612         *AN.RepPoint = 1;
00613         AR.expchanged = 1;
00614         if ( AR.CurDum > AM.IndDum && AR.sLevel <= 0 ) ReNumber(BHEAD term);
00615     }
00616     return(Generator(BHEAD term,level));
00617 }
00618
00619 /*
00620         #] SymGen :
00621         #[ SymFind :
00622
00623         There is a certain amount of double work here, as this routine
00624         finds the function to be treated, while the SymGen routine has
00625         to find it again. Note however that this way things remain
00626         uniform and simple. Moreover this avoids problems with actions
00627         on more than one function simultaneously.
00628         Output in AT.TMout:
00629         Number,sym/anti,fun,lenpar,ngroups,gsize,fields
00630
00631 */
00632
00633 int SymFind(PHEAD WORD *term, WORD *params)
00634 {
00635     GETBIDENTITY
00636     WORD *t, *r, *m;
00637     WORD j, c1, c2, count;
00638     WORD *rstop;
00639     c1 = params[4];     /* function number */
00640     c2 = FUNCTION + WILDOFFSET;
00641     t = term;
00642     m = t + *t;
00643     m -= ABS(m[-1]);
00644     t++;
00645     while ( t < m ) {
00646         if ( *t == c1 || c1 > c2 ) {    /* Candidate function */
00647             if ( *t >= FUNCTION && functions[*t-FUNCTION].spec
00648                 >= TENSORFUNCTION ) { count = t[1] - FUNHEAD; }
00649             else {
00650                 count = 0;
00651                 r = t;
00652                 rstop = t + t[1];
00653                 r += FUNHEAD;
00654                 while ( r < rstop ) { count++; NEXTARG(r) }
00655             }
00656             if ( ( j = params[5] ) > 0 && j != count ) goto NextFun;
00657             if ( j == 0 ) {
00658                 r = params + 8;
00659                 rstop = params + params[1];
00660                 while ( r < rstop ) {
00661                     if ( *r > count + 1 ) goto NextFun;
00662                     r++;
00663                 }
00664             }
```

```
00665
00666             t = AT.TMout;
00667             r = params;
00668             j = r[1] - 1;
00669             *t++ = j;
00670             *t++ = SYMMETRIZE;
00671             r += 3;
00672             j--;
00673             NCOPY(t,r,j);
00674             return(1);
00675         }
00676 NextFun:
00677         t += t[1];
00678     }
00679     return(0);
00680 }
00681
00682 /*
00683         #] SymFind :
00684         #[ ChainIn :
00685
00686         Equivalent to repeat id f(?a)*f(?b) = f(?a,?b);
00687
00688         This one always takes less space.
00689 */
00690
00691 int ChainIn(PHEAD WORD *term, WORD funnum)
00692 {
00693     GETBIDENTITY
00694     WORD *t, *tend, *m, *tt, *ts;
00695     int action, normFlag = 0;
00696     if ( funnum < 0 ) { /* Dollar to be expanded */
00697         funnum = DolToFunction(BHEAD -funnum);
00698         if ( AN.ErrorInDollar || funnum <= 0 ) {
00699             MLOCK(ErrorMessageLock);
00700             MesPrint("Dollar variable does not evaluate to function in ChainIn statement");
00701             MUNLOCK(ErrorMessageLock);
00702             return(-1);
00703         }
00704     }
00705     do {
00706         action = 0;
00707         tend = term+*term;
00708         tend -= ABS(tend[-1]);
00709         t = term+1;
00710         while ( t < tend ) {
00711             if ( *t != funnum ) { t += t[1]; continue; }
00712             m = t;
00713             t += t[1];
00714             tt = t;
00715             if ( t >= tend || *t != funnum ) continue;
00716             action = 1;
00717             normFlag = 1;
00718             while ( t < tend && *t == funnum ) {
00719                 ts = t + t[1];
00720                 t += FUNHEAD;
00721                 while ( t < ts ) *tt++ = *t++;
00722             }
00723             m[1] = tt - m;
00724             ts = term + *term;
00725             while ( t < ts ) *tt++ = *t++;
00726             *term = tt - term;
00727             break;
00728         }
00729     } while ( action );
00730
00731     if ( normFlag ) {
00732         /* We need to check the newly-constructed arguments w.r.t symmetry properties */
00733         MarkDirty(term, DIRTYSYMFLAG);
00734         AT.WorkPointer = term + *term;
00735         Normalize(BHEAD term);
00736     }
00737
00738     return(0);
00739 }
00740
00741 /*
00742         #] ChainIn :
00743         #[ ChainOut :
00744
00745         Equivalent to repeat id f(x1?,x2?,?a) = f(x1)*f(x2,?a);
00746 */
00747
00748 int ChainOut(PHEAD WORD *term, WORD funnum)
00749 {
00750     GETBIDENTITY
00751     WORD *t, *tend, *tt, *ts, *w, *ws;
```

```
00752     int flag = 0, i;
00753     if ( funnum < 0 ) { /* Dollar to be expanded */
00754         funnum = DolToFunction(BHEAD -funnum);
00755         if ( AN.ErrorInDollar || funnum <= 0 ) {
00756             MLOCK(ErrorMessageLock);
00757             MesPrint("Dollar variable does not evaluate to function in ChainOut statement");
00758             MUNLOCK(ErrorMessageLock);
00759             return(-1);
00760         }
00761     }
00762     tend = term+*term;
00763     if ( AT.WorkPointer < tend ) AT.WorkPointer = tend;
00764     tend -= ABS(tend[-1]);
00765     t = term+1; tt = term; w = AT.WorkPointer;
00766     while ( t < tend ) {
00767         if ( *t != funnum || t[1] == FUNHEAD ) { t += t[1]; continue; }
00768         flag = 1;
00769         while ( tt < t ) *w++ = *tt++;
00770         ts = t + t[1];
00771         t += FUNHEAD;
00772         while ( t < ts ) {
00773             ws = w;
00774             for ( i = 0; i < FUNHEAD; i++ ) *w++ = tt[i];
00775             if ( functions[*tt-FUNCTION].spec >= TENSORFUNCTION ) {
00776                 *w++ = *t++;
00777             }
00778             else if ( *t < 0 ) {
00779                 if ( *t <= -FUNCTION ) *w++ = *t++;
00780                 else { *w++ = *t++; *w++ = *t++; }
00781             }
00782             else {
00783                 i = *t; NCOPY(w,t,i);
00784             }
00785             ws[1] = w - ws;
00786         }
00787         tt = t;
00788     }
00789     if ( flag == 1 ) {
00790         ts = term + *term;
00791         while ( tt < ts ) *w++ = *tt++;
00792         *AT.WorkPointer = w - AT.WorkPointer;
00793         t = term; w = AT.WorkPointer; i = *w;
00794         NCOPY(t,w,i)
00795         AT.WorkPointer = term + *term;
00796         Normalize(BHEAD term);
00797     }
00798     return(0);
00799 }
00800
00801 /*
00802         #] ChainOut :
00803     #] Utilities :
00804     #[ Patterns :
00805         #[ MatchFunction :          WORD MatchFunction(pattern,interm,wilds)
00806
00807         The routine assumes that the function numbers are the same.
00808         The contents are compared and a possible wildcard assignment
00809         is made. Note that it may be necessary to use a wildcard
00810         assignment stack to do things right.
00811         The routine can become arbitrarily complicated as there is
00812         no end to the possible wildcarding.
00813         Examples:
00814         -   a:  No wildcarding -> straight match
00815         -   b:  Individual arguments (object -> object)
00816         -   c:  whole arguments (object to subexpression)
00817         -   d:  any argumentlist
00818             e:  part of an argument (object inside subexpression)
00819
00820         The ones with a minus sign in front have been implemented.
00821
00822         Note: the argument wilds allows backtracking when multiple
00823         ?a,?b give a match that later turns out to be useless.
00824 */
00825
00826 int MatchFunction(PHEAD WORD *pattern, WORD *interm, WORD *wilds)
00827 {
00828     GETBIDENTITY
00829     WORD *m, *t, *r, i;
00830     WORD *mstop = 0, *tstop = 0;
00831     WORD *argmstop, *argtstop;
00832     WORD *mtrmstop, *ttrmstop;
00833     WORD *msubstop, *mnextsub;
00834     WORD msizcoef, mcount, tcount, newvalue, j;
00835     WORD *oldm, *oldt;
00836     WORD *OldWork, numofwildarg;
00837     WORD nwstore, tobeeaten, reservevalue = 0, resernum = 0, withwild;
00838     WORD *wildargtaken;
```

```
00839     CBUF *C = cbuf+AT.ebufnum;
00840     int ntwa = AN.NumTotWildArgs;
00841     LONG oldcpointer = C->Pointer - C->Buffer;
00842 #ifdef WITHFLOAT
00843     // Pattern matching against float_ functions is currently disabled.
00844     // To relax this in the future, move this early return down to where gamma
00845     // functions and tensors are handled specially.
00846     if ( *interm == FLOATFUN ) return(0);
00847 #endif
00848 /*
00849     Test first for a straight match
00850 */
00851     AN.RepFunList[AN.RepFunNum+1] = 0;
00852     if ( *wilds == 0 ) {
00853         m = pattern; t = interm;
00854 
00855         if ( *m != *t ) {
00856             if ( *m < (FUNCTION + WILDOFFSET) ) return(0);
00857             if ( *t < FUNCTION ) return(0);
00858             if ( functions[*t-FUNCTION].spec !=
00859             functions[*m-FUNCTION-WILDOFFSET].spec ) return(0);
00860         }
00861         i = m[1];
00862         if ( *m >= (FUNCTION + WILDOFFSET) ) { i--; m++; t++; }
00863         do { if ( *m++ != *t++ ) break; } while ( --i > 0 );
00864         if ( i <= 0 ) {          /* Arguments match */
00865             if ( AN.SignCheck && AN.ExpectedSign ) return(0);
00866             i = *pattern - WILDOFFSET;
00867             if ( i >= FUNCTION ) {
00868                 if ( *interm != GAMMA
00869                 && !CheckWild(BHEAD i,FUNTOFUN,*interm,&newvalue) ) {
00870                     AddWild(BHEAD i,FUNTOFUN,newvalue);
00871                     return(1);
00872                 }
00873                 return(0);
00874             }
00875             else return(1);
00876         }
00877     }
00878 /*
00879     Store the current Wildcard assignments
00880 */
00881     t = wildargtaken = OldWork = AT.WorkPointer;
00882     t += ntwa;
00883     m = AN.WildValue;
00884     nwstore = i = (m[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
00885     if ( i > 0 ) {
00886         r = AT.WildMask;
00887         do {
00888             *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
00889         } while ( --i > 0 );
00890         *t++ = C->numrhs;
00891     }
00892     if ( t >= AT.WorkTop ) {
00893         MLOCK(ErrorMessageLock);
00894         MesWork();
00895         MUNLOCK(ErrorMessageLock);
00896         Terminate(-1);
00897     }
00898     AT.WorkPointer = t;
00899 
00900     if ( *wilds ) {
00901         if ( *wilds == 1 ) goto endoloop;
00902         else               goto enloop;         /* tensors = 2 */
00903     }
00904     m = pattern; t = interm;
00905 /*
00906     Single out the specials
00907 */
00908     if ( *t == GAMMA ) {
00909 /*
00910         #[ GAMMA :
00911 
00912         For the gamma's we need to do two things:
00913         a:  Find that there is a match
00914         b:  Find where the match occurs in the string
00915         This last thing cannot be stored in the current conventions,
00916         but once the wildcard assignments have been made it is much
00917         easier to find it back.
00918         Alternative: replace the function number in the term temporarily
00919         by the offset inside the string. This makes things maybe easier.
00920 */
00921         if ( *m != GAMMA ) goto NoCaseB;
00922         i = t[1] - m[1];
00923         if ( m[1] == FUNHEAD+1 ) {
00924             if ( i ) goto NoCaseB;
00925             if ( m[FUNHEAD] < (AM.OffsetIndex+WILDOFFSET) ||
```

```
00926             t[FUNHEAD] >= (AM.OffsetIndex+WILDOFFSET) ) goto NoCaseB;
00927
00928             if ( CheckWild(BHEAD m[FUNHEAD]-WILDOFFSET,INDTOIND,t[FUNHEAD],&newvalue) ) goto NoCaseB;
00929             AddWild(BHEAD m[FUNHEAD]-WILDOFFSET,INDTOIND,newvalue);
00930
00931             AT.WorkPointer = OldWork;
00932             if ( AN.SignCheck && AN.ExpectedSign ) return(0);
00933             return(1);      /* m was eaten. we have a match! */
00934         }
00935         if ( i < 0 ) goto NoCaseB;  /* Pattern longer than target */
00936         mstop = m + m[1];
00937         tstop = t + t[1];
00938         m += FUNHEAD; t += FUNHEAD;
00939         if ( *m >= (AM.OffsetIndex+WILDOFFSET) && *t < (AM.OffsetIndex+WILDOFFSET) ) {
00940             if ( CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,*t,&newvalue) ) goto NoCaseB;
00941             reservevalue = newvalue;
00942             withwild = 1;
00943             resernum = *m-WILDOFFSET;
00944             AddWild(BHEAD *m-WILDOFFSET,INDTOIND,newvalue);
00945         }
00946         else if ( *m != *t ) goto NoCaseB;
00947         else withwild = 0;
00948         m++; t++;
00949         oldm = m; argtstop = oldt = t;
00950         j = 0;                  /* No wildcard assignments yet */
00951         while ( i >= 0 ) {
00952             if ( *m == *t ) {
00953 WithGamma:      m++; t++;
00954                 if ( m >= mstop ) {
00955                     if ( t < tstop && mstop < AN.patstop ) {
00956                         WORD k;
00957                         mnextsub = pattern + pattern[1];
00958                         k = *mnextsub;
00959                         while ( k == GAMMA && mnextsub[FUNHEAD]
00960                         != pattern[FUNHEAD] ) {
00961                             mnextsub += mnextsub[1];
00962                             if ( mnextsub >= AN.patstop ) goto FullOK;
00963                             k = *mnextsub;
00964                         }
00965                         if ( k >= FUNCTION ) {
00966                             if ( k > (FUNCTION + WILDOFFSET) ) k -= WILDOFFSET;
00967                             if ( functions[k-FUNCTION].commute ) goto NoGamma;
00968                         }
00969                     }
00970 FullOK:             if ( AN.SignCheck && AN.ExpectedSign ) goto NoGamma;
00971                     AN.RepFunList[AN.RepFunNum+1] = WORDDIF(oldt,argtstop);
00972                     return(1);
00973                 }
00974                 if ( t >= tstop ) goto NoCaseB;
00975             }
00976             else if ( *m >= (AM.OffsetIndex+WILDOFFSET)
00977             && *m < (AM.OffsetIndex + (WILDOFFSET<<1)) && ( *t >= 0 ||
00978             *t < MINSPEC ) ) {          /* Wildcard index */
00979                 if ( !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,*t,&newvalue) ) {
00980                     AddWild(BHEAD *m-WILDOFFSET,INDTOIND,newvalue);
00981                     j = 1;
00982                     goto WithGamma;
00983                 }
00984                 else goto NoGamma;
00985             }
00986             else if ( *m < MINSPEC && *m >= (AM.OffsetVector+WILDOFFSET)
00987             && *t < MINSPEC ) {         /* Wildcard vector */
00988                 if ( !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*t,&newvalue) ) {
00989                     AddWild(BHEAD *m-WILDOFFSET,VECTOVEC,newvalue);
00990                     j = 1;
00991                     goto WithGamma;
00992                 }
00993                 else goto NoGamma;
00994             }
00995             else {
00996 NoGamma:
00997                 if ( j ) {      /* Undo wildcards */
00998                     m = AN.WildValue;
00999                     t = OldWork + AN.NumTotWildArgs; r = AT.WildMask; j = nwstore;
01000                     if ( j > 0 ) {
01001                         do {
01002                             *m++ = *t++; *m++ = *t++;
01003                             *m++ = *t++; *m++ = *t++; *r++ = *t++;
01004                         } while ( --j > 0 );
01005                         C->numrhs = *t++;
01006                         C->Pointer = C->Buffer + oldcpointer;
01007                     }
01008                     j = 0;
01009                 }
01010                 m = oldm; t = ++oldt; i--;
01011                 if ( withwild ) {
01012                     AddWild(BHEAD resernum,INDTOIND,reservevalue);
```

```
01013                 }
01014             }
01015         }
01016         goto NoCaseB;
01017 /*
01018         #] GAMMA :
01019         #[ Tensors :
01020 */
01021     }
01022     else if ( *t >= FUNCTION && functions[*t-FUNCTION].spec >= TENSORFUNCTION ) {
01023         mstop = m + m[1];
01024         tstop = t + t[1];
01025         mcount = 0;
01026         m += FUNHEAD;
01027         t += FUNHEAD;
01028         AN.WildArgs = 0;
01029         tcount = WORDDIF(tstop,t);
01030         while ( m < mstop ) {
01031             if ( *m == FUNNYWILD ) { m++; AN.WildArgs++; }
01032             m++; mcount++;
01033         }
01034         tobeeaten = tcount - mcount + AN.WildArgs;
01035         if ( tobeeaten ) {
01036             if ( tobeeaten < 0 || AN.WildArgs == 0 ) {
01037                 AT.WorkPointer = OldWork;
01038                 return(0);  /* Cannot match */
01039             }
01040         }
01041         AT.WildArgTaken[0] = AN.WildEat = tobeeaten;
01042         for ( i = 1; i < AN.WildArgs; i++ ) AT.WildArgTaken[i] = 0;
01043 toploop:
01044         numofwildarg = 0;
01045 
01046         m = pattern; t = interm;
01047         mstop = m + m[1];
01048         if ( *m != *t ) {
01049             i = *m - WILDOFFSET;
01050             if ( CheckWild(BHEAD i,FUNTOFUN,*t,&newvalue) ) goto NoCaseB;
01051             AddWild(BHEAD i,FUNTOFUN,newvalue);
01052         }
01053         m += FUNHEAD;
01054         t += FUNHEAD;
01055         while ( m < mstop ) {
01056 /*
01057             First test for an exact match
01058 */
01059             if ( *m == *t ) { m++; t++; continue; }
01060 /*
01061             No exact match. Try ARGWILD
01062 */
01063             AN.argaddress = t;
01064             if ( *m == FUNNYWILD ) {
01065                 tobeeaten = AT.WildArgTaken[numofwildarg++];
01066                 if ( CheckWild(BHEAD m[1],ARGTOARG|EATTENSOR,tobeeaten,t) ) goto endloop;
01067                 AddWild(BHEAD m[1],ARGTOARG|EATTENSOR,tobeeaten);
01068                 m += 2;
01069                 t += tobeeaten;
01070                 continue;
01071             }
01072 /*
01073             Now the various cases:
01074 */
01075             i = *m;
01076             if ( i < MINSPEC ) {
01077                 if ( *t != i ) {
01078                     if ( *t >= MINSPEC ) goto endloop;
01079                     i -= WILDOFFSET;
01080                     if ( i < AM.OffsetVector ) goto endloop;
01081                     if ( CheckWild(BHEAD i,VECTOVEC,*t,&newvalue) )
01082                         goto endloop;
01083                     AddWild(BHEAD i,VECTOVEC,newvalue);
01084                 }
01085             }
01086             else if ( i >= AM.OffsetIndex ) {           /* Index */
01087                 if ( i < ( AM.OffsetIndex + WILDOFFSET ) ) goto endloop;
01088                 if ( i >= ( AM.OffsetIndex + (WILDOFFSET«1) ) ) {
01089                                                 /* Summed over index */
01090                     goto endloop;               /* For the moment */
01091                 }
01092                 i -= WILDOFFSET;
01093                 if ( CheckWild(BHEAD i,INDTOIND,*t,&newvalue) )
01094                     goto endloop;       /* Assignment not allowed */
01095                 AddWild(BHEAD i,INDTOIND,newvalue);
01096             }
01097             else goto endloop;
01098             m++; t++;
01099         }
```

```
01100         if ( AN.SignCheck && AN.ExpectedSign ) goto endloop;
01101         AT.WorkPointer = OldWork;
01102         if ( AN.WildArgs > 1 ) *wilds = 2;
01103         return(1);      /* m was eaten. we have a match! */
01104
01105 endloop:;
01106 /*
01107     restore the current Wildcard assignments
01108 */
01109         i = nwstore;
01110         if ( i > 0 ) {
01111             m = AN.WildValue;
01112             t = OldWork + ntwa; r = AT.WildMask;
01113             do {
01114                 *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01115             } while ( --i > 0 );
01116             C->numrhs = *t++;
01117             C->Pointer = C->Buffer + oldcpointer;
01118         }
01119 enloop:;
01120         i = AN.WildArgs - 1;
01121         if ( i <= 0 ) {
01122             AT.WorkPointer = OldWork;
01123             return(0);
01124         }
01125         while ( --i >= 0 ) {
01126             if ( AT.WildArgTaken[i] == 0 ) {
01127                 if ( i == 0 ) {
01128                     AT.WorkPointer = OldWork;
01129                     *wilds = 0;
01130                     return(0);
01131                 }
01132             }
01133             else {
01134                 (AT.WildArgTaken[i])--;
01135                 numofwildarg = 0;
01136                 for ( j = 0; j <= i; j++ ) {
01137                     numofwildarg += AT.WildArgTaken[j];
01138                 }
01139                 AT.WildArgTaken[j] = AN.WildEat-numofwildarg;
01140                 for ( j++; j < AN.WildArgs; j++ ) AT.WildArgTaken[j] = 0;
01141                 break;
01142             }
01143         }
01144         goto toploop;
01145 /*
01146         #] Tensors :
01147 */
01148     }
01149 /*
01150     Count the number of arguments. Either equal or an argument wildcard.
01151 */
01152     mstop = m + m[1];
01153     tstop = t + t[1];
01154     mcount = 0; tcount = 0;
01155     m += FUNHEAD; t += FUNHEAD;
01156     while ( t < tstop ) { tcount++; NEXTARG(t) }
01157     AN.WildArgs = 0;
01158     while ( m < mstop ) {
01159         mcount++;
01160         if ( *m == -ARGWILD ) AN.WildArgs++;
01161         NEXTARG(m)
01162     }
01163     tobeeaten = tcount - mcount + AN.WildArgs;
01164     if ( tobeeaten ) {
01165         if ( tobeeaten < 0 || AN.WildArgs == 0 ) {
01166             AT.WorkPointer = OldWork;
01167             return(0);  /* Cannot match */
01168         }
01169     }
01170 /*
01171     Set up the array AT.WildArgTaken for the number of arguments that each
01172     wildarg eats.
01173 */
01174     AT.WildArgTaken[0] = AN.WildEat = tobeeaten;
01175     for ( i = 1; i < AN.WildArgs; i++ ) AT.WildArgTaken[i] = 0;
01176 topofloop:
01177     numofwildarg = 0;
01178 /*
01179     Test for single wildcard object/argument
01180 */
01181     m = pattern; t = interm;
01182     if ( *m != *t ) {
01183         i = *m - WILDOFFSET;
01184         if ( CheckWild(BHEAD i,FUNTOFUN,*t,&newvalue) ) goto NoCaseB;
01185         AddWild(BHEAD i,FUNTOFUN,newvalue);
01186     }
```

```
01187     mstop = m + m[1];
01188 /*  tstop = t + t[1];  */
01189     m += FUNHEAD;
01190     t += FUNHEAD;
01191     while ( m < mstop ) {
01192         argmstop = oldm = m;
01193         argtstop = oldt = t;
01194         NEXTARG(argmstop)
01195         NEXTARG(argtstop)
01196         if ( t == tstop ) { /* This concerns a very rare bug */
01197             if ( *m == -ARGWILD ) goto ArgAll;
01198             goto endofloop;
01199         }
01200         if ( *m < 0 && *t < 0 ) {
01201             if ( *t <= -FUNCTION ) {
01202                 if ( *t == *m ) {}
01203                 else if ( *m <= -FUNCTION-WILDOFFSET
01204                 && functions[-*t-FUNCTION].spec
01205                 == functions[-*m-FUNCTION-WILDOFFSET].spec ) {
01206                     i = -*m - WILDOFFSET;
01207                     if ( CheckWild(BHEAD i,FUNTOFUN,-*t,&newvalue) ) goto endofloop;
01208                     AddWild(BHEAD i,FUNTOFUN,newvalue);
01209                 }
01210                 else if ( *m == -SYMBOL && m[1] >= 2*MAXPOWER ) {
01211                     i = m[1] - 2*MAXPOWER;
01212                     AN.argaddress = AT.FunArg;
01213                     AT.FunArg[ARGHEAD+1] = -*t;
01214                     if ( CheckWild(BHEAD i,SYMTOSUB,1,AN.argaddress) ) goto endofloop;
01215                     AddWild(BHEAD i,SYMTOSUB,0);
01216                 }
01217                 else if ( *m == -ARGWILD ) {
01218 ArgAll:             i = AT.WildArgTaken[numofwildarg++];
01219                     AN.argaddress = t;
01220                     if ( CheckWild(BHEAD m[1],ARGTOARG,i,t) ) goto endofloop;
01221                     AddWild(BHEAD m[1],ARGTOARG,i);
01222 /*                  m += 2; */
01223                     while ( --i >= 0 ) { NEXTARG(t) }
01224                     argtstop = t;
01225                 }
01226                 else goto endofloop;
01227             }
01228             else if ( *t == *m ) {
01229                 if ( t[1] == m[1] ) {}
01230                 else if ( *t == -SYMBOL ) {
01231                     j = SYMTOSYM;
01232 SymAll:
01233                     if ( ( i = m[1] - 2*MAXPOWER ) < 0 ) goto endofloop;
01234                     if ( CheckWild(BHEAD i,j,t[1],&newvalue) ) goto endofloop;
01235                     AddWild(BHEAD i,j,newvalue);
01236                 }
01237                 else if ( *t == -INDEX ) {
01238 IndAll:             i = m[1] - WILDOFFSET;
01239                     if ( i < AM.OffsetIndex || i >= WILDOFFSET+AM.OffsetIndex )
01240                                                         goto endofloop;
01241                                 /* We kill the summed over indices here */
01242                     if ( CheckWild(BHEAD i,INDTOIND,t[1],&newvalue) ) goto endofloop;
01243                     AddWild(BHEAD i,INDTOIND,newvalue);
01244                 }
01245                 else if ( *t == -VECTOR || *t == -MINVECTOR ) {
01246                     i = m[1] - WILDOFFSET;
01247                     if ( i < AM.OffsetVector ) goto endofloop;
01248                     if ( CheckWild(BHEAD i,VECTOVEC,t[1],&newvalue) ) goto endofloop;
01249                     AddWild(BHEAD i,VECTOVEC,newvalue);
01250                 }
01251                 else goto endofloop;
01252             }
01253             else if ( *m == -ARGWILD ) goto ArgAll;
01254             else if ( *m == -INDEX && m[1] >= AM.OffsetIndex+WILDOFFSET
01255             && m[1] < AM.OffsetIndex+(WILDOFFSET<<1) ) {
01256                 if ( *t == -VECTOR ) goto IndAll;
01257                 if ( *t == -SNUMBER && t[1] >= 0 && t[1] < AM.OffsetIndex ) goto IndAll;
01258                 if ( *t == -MINVECTOR ) {
01259                     i = m[1] - WILDOFFSET;
01260                     AN.argaddress = AT.MinVecArg;
01261                     AT.MinVecArg[ARGHEAD+3] = t[1];
01262                     if ( CheckWild(BHEAD i,INDTOSUB,1,AN.argaddress) ) goto endofloop;
01263                     AddWild(BHEAD i,INDTOSUB,(WORD)0);
01264                 }
01265                 else goto endofloop;
01266             }
01267             else if ( *m == -SYMBOL && m[1] >= 2*MAXPOWER && *t == -SNUMBER ) {
01268                 j = SYMTONUM;
01269                 goto SymAll;
01270             }
01271             else if ( *m == -VECTOR && *t == -MINVECTOR &&
01272             ( i = m[1] - WILDOFFSET ) >= AM.OffsetVector ) {
01273 /*
```

```
01274 ================================
01275                 AN.argaddress = AT.MinVecArg;
01276                 AT.MinVecArg[ARGHEAD+3] = t[1];
01277                 if ( CheckWild(BHEAD i,VECTOSUB,1,AN.argaddress) ) goto endofloop;
01278                 AddWild(BHEAD i,VECTOSUB,(WORD)0);
01279 ================================
01280 */
01281                 if ( CheckWild(BHEAD i,VECTOMIN,t[1],&newvalue) ) goto endofloop;
01282                 AddWild(BHEAD i,VECTOMIN,newvalue);
01283
01284             }
01285             else if ( *m == -MINVECTOR && *t == -VECTOR &&
01286             ( i = m[1] - WILDOFFSET ) >= AM.OffsetVector ) {
01287 /*
01288 ================================
01289                 AN.argaddress = AT.MinVecArg;
01290                 AT.MinVecArg[ARGHEAD+3] = t[1];
01291                 if ( CheckWild(BHEAD i,VECTOSUB,1,AN.argaddress) ) goto endofloop;
01292                 AddWild(BHEAD i,VECTOSUB,(WORD)0);
01293 ================================
01294 */
01295                 if ( CheckWild(BHEAD i,VECTOMIN,t[1],&newvalue) ) goto endofloop;
01296                 AddWild(BHEAD i,VECTOMIN,newvalue);
01297             }
01298             else goto endofloop;
01299         }
01300         else if ( *t <= -FUNCTION && *m > 0 ) {
01301             if ( ( m[ARGHEAD]+ARGHEAD == *m ) && m[*m-1] == 3
01302             && m[*m-2] == 1 && m[*m-3] == 1 && m[ARGHEAD+1] >= FUNCTION
01303             && m[ARGHEAD+2] == *m-ARGHEAD-4 ) { /* Check for f(?a) etc */
01304                 WORD *mmmst, *mmm;
01305                 if ( m[ARGHEAD+1] >= FUNCTION+WILDOFFSET ) {
01306 /*                  i = *m - WILDOFFSET; */
01307                     i = m[ARGHEAD+1] - WILDOFFSET;
01308                     if ( CheckWild(BHEAD i,FUNTOFUN,-*t,&newvalue) ) goto endofloop;
01309                     AddWild(BHEAD i,FUNTOFUN,newvalue);
01310                 }
01311                 else if ( m[ARGHEAD+1] != -*t ) goto endofloop;
01312 /*
01313                     Only arguments allowed are ?a etc.
01314 */
01315                 mmmst = m+*m-3;
01316                 mmm = m + ARGHEAD + FUNHEAD + 1;
01317                 while ( mmm < mmmst ) {
01318                     if ( *mmm != -ARGWILD ) goto endofloop;
01319                     i = 0;
01320                     AN.argaddress = t;
01321                     if ( CheckWild(BHEAD mmm[1],ARGTOARG,i,t) ) goto endofloop;
01322                     AddWild(BHEAD mmm[1],ARGTOARG,i);
01323                     mmm += 2;
01324                 }
01325             }
01326             else goto endofloop;
01327         }
01328         else if ( *m < 0 && *t > 0 ) {
01329             if ( *m == -SYMBOL ) {          /* SYMTOSUB */
01330                 if ( m[1] < 2*MAXPOWER ) goto endofloop;
01331                 i = m[1] - 2*MAXPOWER;
01332                 AN.argaddress = t;
01333                 if ( CheckWild(BHEAD i,SYMTOSUB,1,AN.argaddress) ) goto endofloop;
01334                 AddWild(BHEAD i,SYMTOSUB,0);
01335             }
01336             else if ( *m == -VECTOR ) {
01337                 if ( ( i = m[1] - WILDOFFSET ) < AM.OffsetVector )
01338                                                         goto endofloop;
01339                 AN.argaddress = t;
01340                 if ( CheckWild(BHEAD i,VECTOSUB,1,t) ) goto endofloop;
01341                 AddWild(BHEAD i,VECTOSUB,(WORD)0);
01342             }
01343             else if ( *m == -INDEX ) {
01344                 if ( ( i = m[1] - WILDOFFSET ) < AM.OffsetIndex ) goto endofloop;
01345                 if ( i >= AM.OffsetIndex + WILDOFFSET ) goto endofloop;
01346                 AN.argaddress = t;
01347                 if ( CheckWild(BHEAD i,INDTOSUB,1,AN.argaddress) ) goto endofloop;
01348                 AddWild(BHEAD i,INDTOSUB,(WORD)0);
01349             }
01350             else if ( *m == -ARGWILD ) goto ArgAll;
01351             else goto endofloop;
01352         }
01353         else if ( *m > 0 && *t > 0 ) {
01354             WORD ii = *t-*m;
01355             i = *m;
01356             do { if ( *m++ != *t++ ) break; } while ( --i > 0 );
01357             if ( i == 1 && ii == 0 ) {  /* sign difference */
01358                 goto endofloop;
01359             }
01360             else if ( i > 0 ) {
```

```
01361                 WORD *cto, *cfrom, *csav, ci;
01362                 WORD oRepFunNum;
01363                 WORD *oRepFunList;
01364                 WORD *oterstart,*oterstop,*opatstop;
01365                 WORD oExpectedSign;
01366                 WORD wildargs, wildeat;
01367 /*
01368                 Not an exact match here.
01369                 We have to hope that the pattern contains a composite wildcard.
01370 */
01371                 m = oldm; t = oldt;
01372                 m += ARGHEAD; t += ARGHEAD;         /* Point at (first?) term */
01373                 mtrmstop = m + *m;
01374                 ttrmstop = t + *t;
01375                 if ( mtrmstop < argmstop ) goto endofloop;/* More than one term */
01376                 msizcoef = mtrmstop[-1];
01377                 if ( msizcoef < 0 ) msizcoef = -msizcoef;
01378                 msubstop = mtrmstop - msizcoef;
01379                 m++;
01380                 if ( m >= msubstop ) goto endofloop;    /* Only coefficient */
01381 /*
01382                 Here we have a composite term. It can match provided it
01383                 matches the entire argument. This argument must be a
01384                 single term also and the coefficients should match
01385                 (more or less).
01386                 The matching takes:
01387                 1:  Match the functions etc. Nothing can be left.
01388                 2:  Match dotproducts and symbols. ONLY must match
01389                     and nothing may be left.
01390                 For safety it is best to take the term out and put it
01391                 in workspace.
01392 */
01393
01394                 if ( argtstop > ttrmstop ) goto endofloop;
01395                 m--;
01396                 oterstart = AN.terstart;
01397                 oterstop = AN.terstop;
01398                 opatstop = AN.patstop;
01399                 oRepFunList = AN.RepFunList;
01400                 oRepFunNum = AN.RepFunNum;
01401                 AN.RepFunNum = 0;
01402                 AN.RepFunList = AT.WorkPointer;
01403                 AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
01404                 if ( AT.WorkPointer+*t+5 > AT.WorkTop ) {
01405                     MLOCK(ErrorMessageLock);
01406                     MesWork();
01407                     MUNLOCK(ErrorMessageLock);
01408                     return(-1);
01409                 }
01410                 csav = cto = AT.WorkPointer;
01411                 cfrom = t;
01412                 ci = *t;
01413                 while ( --ci >= 0 ) *cto++ = *cfrom++;
01414                 AT.WorkPointer = cto;
01415                 ci = msizcoef;
01416                 cfrom = mtrmstop;
01417                 --ci;
01418                 if ( abs(*--cfrom) != abs(*--cto) ) {
01419                     AT.WorkPointer = csav;
01420                     AN.RepFunList = oRepFunList;
01421                     AN.RepFunNum = oRepFunNum;
01422                     AN.terstart = oterstart;
01423                     AN.terstop = oterstop;
01424                     AN.patstop = opatstop;
01425                     goto endofloop;
01426                 }
01427                 i = (*cfrom != *cto) ? 1 : 0; /* buffer AN.ExpectedSign until we are beyond the goto
      */
01428                 while ( --ci >= 0 ) {
01429                     if ( *--cfrom != *--cto ) {
01430                         AT.WorkPointer = csav;
01431                         AN.RepFunList = oRepFunList;
01432                         AN.RepFunNum = oRepFunNum;
01433                         AN.terstart = oterstart;
01434                         AN.terstop = oterstop;
01435                         AN.patstop = opatstop;
01436                         goto endofloop;
01437                     }
01438                 }
01439                 oExpectedSign =  AN.ExpectedSign; /* buffer AN.ExpectedSign until we are beyond
      FindRest/FindOnly */
01440                 AN.ExpectedSign = i;
01441                 *m -= msizcoef;
01442                 wildargs = AN.WildArgs;
01443                 wildeat = AN.WildEat;
01444                 for ( i = 0; i < wildargs; i++ ) wildargtaken[i] = AT.WildArgTaken[i];
01445                 AN.ForFindOnly = 0; AN.UseFindOnly = 1;
```

```
01446                 AN.nogroundlevel++;
01447                 if ( FindRest(BHEAD csav,m) && ( AN.UsedOtherFind || FindOnly(BHEAD csav,m) ) ) {}
01448                 else {
01449 nomatch:
01450                     *m += msizcoef;
01451                     AT.WorkPointer = csav;
01452                     AN.RepFunList = oRepFunList;
01453                     AN.RepFunNum = oRepFunNum;
01454                     AN.terstart = oterstart;
01455                     AN.terstop = oterstop;
01456                     AN.patstop = opatstop;
01457                     AN.WildArgs = wildargs;
01458                     AN.WildEat = wildeat;
01459                     AN.ExpectedSign = oExpectedSign;
01460                     AN.nogroundlevel--;
01461                     for ( i = 0; i < wildargs; i++ ) AT.WildArgTaken[i] = wildargtaken[i];
01462                     goto endofloop;
01463                 }
01464 /*              if ( *m == 1 || m[1] < FUNCTION || functions[m[1]-FUNCTION].spec >= TENSORFUNCTION ) {
    */
01465                 if ( *m == 1 || m[1] < FUNCTION ) {
01466                     if ( AN.ExpectedSign ) goto nomatch;
01467                 }
01468                 else {
01469                     if ( m[1] > FUNCTION + WILDOFFSET ) {
01470                         if ( functions[m[1]-FUNCTION-WILDOFFSET].spec >= TENSORFUNCTION ) {
01471                             if ( AN.ExpectedSign != AN.RepFunList[AN.RepFunNum-1] ) goto nomatch;
01472                         }
01473                     }
01474                     else {
01475                         if ( AN.ExpectedSign != AN.RepFunList[AN.RepFunNum-1] ) goto nomatch;
01476 /*
01477                         if ( functions[m[1]-FUNCTION].spec >= TENSORFUNCTION ) {
01478                             if ( AN.ExpectedSign != AN.RepFunList[AN.RepFunNum-1] ) goto nomatch;
01479                         }
01480 */
01481                     }
01482                 }
01483                 AN.nogroundlevel--;
01484                 AN.ExpectedSign = oExpectedSign;
01485                 AN.WildArgs = wildargs;
01486                 AN.WildEat = wildeat;
01487                 for ( i = 0; i < wildargs; i++ ) AT.WildArgTaken[i] = wildargtaken[i];
01488                 Substitute(BHEAD csav,m,1);
01489                 cto = csav;
01490                 cfrom = cto + *cto - msizcoef;
01491                 cto++;
01492                 *m += msizcoef;
01493                 AT.WorkPointer = csav;
01494                 AN.RepFunList = oRepFunList;
01495                 AN.RepFunNum = oRepFunNum;
01496                 AN.terstart = oterstart;
01497                 AN.terstop = oterstop;
01498                 AN.patstop = opatstop;
01499                 if ( *cto != SUBEXPRESSION ) goto endofloop;
01500                 cto += cto[1];
01501                 if ( cto < cfrom ) goto endofloop;
01502             }
01503         }
01504         else goto endofloop;
01505
01506         t = argtstop;                       /* Next argument */
01507         m = argmstop;
01508     }
01509     if ( AN.SignCheck && AN.ExpectedSign ) goto endofloop;
01510     AT.WorkPointer = OldWork;
01511     if ( AN.WildArgs > 1 ) *wilds = 1;
01512     if ( AN.SignCheck && AN.ExpectedSign ) return(0);
01513     return(1);      /* m was eaten. we have a match! */
01514
01515 endofloop:;
01516 /*
01517     restore the current Wildcard assignments
01518 */
01519     i = nwstore;
01520     if ( i > 0 ) {
01521         m = AN.WildValue;
01522         t = OldWork + ntwa; r = AT.WildMask;
01523         do {
01524             *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01525         } while ( --i > 0 );
01526         C->numrhs = *t++;
01527         C->Pointer = C->Buffer + oldcpointer;
01528     }
01529
01530 endoloop:;
01531     i = AN.WildArgs-1;
```

```
01532     if ( i <= 0 ) {
01533         AT.WorkPointer = OldWork;
01534         return(0);
01535     }
01536     while ( --i >= 0 ) {
01537         if ( AT.WildArgTaken[i] == 0 ) {
01538             if ( i == 0 ) {
01539                 AT.WorkPointer = OldWork;
01540                 return(0);
01541             }
01542         }
01543         else {
01544             (AT.WildArgTaken[i])--;
01545             numofwildarg = 0;
01546             for ( j = 0; j <= i; j++ ) {
01547                 numofwildarg += AT.WildArgTaken[j];
01548             }
01549             AT.WildArgTaken[j] = AN.WildEat-numofwildarg;
01550 /* ----> bug to be replaced in other source code */
01551             for ( j++; j < AN.WildArgs; j++ ) AT.WildArgTaken[j] = 0;
01552             break;
01553         }
01554     }
01555     goto topofloop;
01556 NoCaseB:
01557 /*
01558     Restore the old Wildcard assignments
01559 */
01560     i = nwstore;
01561     if ( i > 0 ) {
01562         m = AN.WildValue;
01563         t = OldWork + ntwa; r = AT.WildMask;
01564         do {
01565             *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01566         } while ( --i > 0 );
01567         C->numrhs = *t++;
01568         C->Pointer = C->Buffer + oldcpointer;
01569     }
01570     AT.WorkPointer = OldWork;
01571     return(0);      /* no match */
01572 }
01573
01574 /*
01575         #] MatchFunction :
01576         #[ ScanFunctions :          WORD ScanFunctions(inpat,inter,par)
01577
01578         Finds in which functions to look for a match.
01579         inpat is the start of the pattern still to be matched.
01580         inter is the start of the term still to be matched.
01581         par gives information about commutativity.
01582             par = 0: nothing special
01583             par = 1: regular noncommuting function
01584             par = 2: GAMMA function
01585
01586         AN.patstop: end of the functions field in the search pattern
01587         AN.terstop: end of the functions field in the target pattern
01588         AN.terstart: address of entire term;
01589
01590         The actual matching of the functions and their arguments is done
01591         in a number of different routines. Mainly MatchFunction when there
01592         are no symmetry properties.
01593         Also: MatchE
01594               MatchCy
01595               FunMatchSy
01596               FunMatchCy
01597
01598         The main problem here is backtracking, ie continuing with wildcard
01599         possibilities when a first assignment doesn't work.
01600         Important note: this was completely forgotten in the symmetric
01601         functions till 6-jan-2009. As of the moment this still has to
01602         be fixed.  ???????21-mar-2023????? Is this still unfixed?????
01603
01604         Functions inside functions can cause problems when antisymmetric
01605         functions are involved. The sign of the term may be at stake.
01606         At the lowest level this is no problem but in f(-fas(n2,n1)) this
01607         plays a role. Next is when we have a product of functions inside
01608         an argument. The strategy must be that we test the sign only at the
01609         last function. Hence, when inpat+inpat[1] >= AN.patstop.
01610         We might relax that to the last antisymmetric function at a later stage.
01611
01612     New scheme to be implemented for non-commuting objects:
01613     When we are matching a second (or higher) function, any match can only
01614     be directly after the last matched non-commuting function or a commuting
01615     function. This will take care of whatever happens in MatchE etc.
01616 */
01617
01618 int ScanFunctions(PHEAD WORD *inpat, WORD *inter, WORD par)
```

```
01619 {
01620     GETBIDENTITY
01621     WORD i, *m, *t, *r, sym, psym;
01622     WORD *newpat, *newter, *instart, *oinpat = 0, *ointer = 0;
01623     WORD nwstore, offset, *OldWork, SetStop = 0, oRepFunNum = AN.RepFunNum;
01624     WORD wilds, wildargs = 0, wildeat = 0, *wildargtaken;
01625     WORD *Oterfirstcomm = AN.terfirstcomm;
01626     CBUF *C = cbuf+AT.ebufnum;
01627     int ntwa = AN.NumTotWildArgs;
01628     LONG oldcpointer = C->Pointer - C->Buffer;
01629     WORD oldSignCheck = AN.SignCheck;
01630     instart = inter;
01631 /*
01632     Only active for the last function in the pattern.
01633     The actual test on the sign is in MatchFunction or the symmetric functions
01634 */
01635     if ( AN.nogroundlevel ) {
01636         AN.SignCheck = ( inpat + inpat[1] >= AN.patstop ) ? 1 : 0;
01637     }
01638     else {
01639         AN.SignCheck = 0;
01640     }
01641 /*
01642             Store the current Wildcard assignments
01643 */
01644     t = wildargtaken = OldWork = AT.WorkPointer;
01645     t += ntwa;
01646     m = AN.WildValue;
01647     nwstore = i = (m[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
01648     if ( i > 0 ) {
01649         r = AT.WildMask;
01650         do {
01651             *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
01652         } while ( --i > 0 );
01653         *t++ = C->numrhs;
01654     }
01655     if ( t >= AT.WorkTop ) {
01656         MLOCK(ErrorMessageLock);
01657         MesWork();
01658         MUNLOCK(ErrorMessageLock);
01659         Terminate(-1);
01660     }
01661     AT.WorkPointer = t;
01662     do {
01663 #ifndef NEWCOMMUTE
01664 /*
01665         Find an eligible unsubstituted function
01666 */
01667         if ( AN.RepFunNum > 0 ) {
01668 /*
01669             First try a non-commuting function, just after the last
01670             substituted non-commuting function.
01671 */
01672             if ( *inter >= FUNCTION && functions[*inter-FUNCTION].commute ) {
01673                 do {
01674                     offset = WORDDIF(inter,AN.terstart);
01675                     for ( i = 0; i < AN.RepFunNum; i += 2 ) {
01676                         if ( AN.RepFunList[i] >= offset ) break;
01677                     }
01678                     if ( i >= AN.RepFunNum ) break;
01679                     inter += inter[1];
01680                 } while ( inter < AN.terfirstcomm );
01681                 if ( inter < AN.terfirstcomm ) { /* Check that it is directly after */
01682                     for ( i = 0; i < AN.RepFunNum; i += 2 ) {
01683                         if ( functions[AN.terstart[AN.RepFunList[i]]-FUNCTION].commute
01684                         && AN.RepFunList[i]+AN.terstart[AN.RepFunList[i]+1] == offset ) break;
01685                     }
01686                     if ( i < AN.RepFunNum ) goto trythis;
01687                 }
01688                 inter = AN.terfirstcomm;
01689             }
01690 /*
01691             Now try one of the commuting functions
01692 */
01693             while ( inter < AN.terstop ) {
01694                 offset = WORDDIF(inter,AN.terstart);
01695                 for ( i = 0; i < AN.RepFunNum; i += 2 ) {
01696                     if ( AN.RepFunList[i] == offset ) break;
01697                 }
01698                 if ( i >= AN.RepFunNum ) break;
01699                 inter += inter[1];
01700             }
01701             if ( inter >= AN.terstop ) goto Failure;
01702 trythis:;
01703         }
01704         else {
01705 /*
```

```
01706             The first function can be anywhere. We have no problems.
01707 */
01708             offset = WORDDIF(inter,AN.terstart);
01709         }
01710 #else
01711         /* first find an unsubstituted function */
01712         do {
01713             offset = WORDDIF(inter,AN.terstart);
01714             for ( i = 0; i < AN.RepFunNum; i += 2 ) {
01715                 if ( AN.RepFunList[i] == offset ) break;
01716             }
01717             if ( i >= AN.RepFunNum ) break;
01718             inter += inter[1];
01719         } while ( inter < AN.terstop );
01720         if ( inter >= AN.terstop ) goto Failure;
01721 #endif
01722         wilds = 0;
01723         /* We found one */
01724         if ( *inter >= FUNCTION && *inpat >= FUNCTION ) {
01725             if ( *inpat == *inter || *inpat >= FUNCTION + WILDOFFSET ) {
01726 /*
01727                 if ( inter[1] == FUNHEAD ) goto rewild;
01728 */
01729                 if ( functions[*inter-FUNCTION].spec >= TENSORFUNCTION
01730                 && ( *inter == *inpat ||
01731                 functions[*inpat-FUNCTION-WILDOFFSET].spec >= TENSORFUNCTION ) ) {
01732                     sym = functions[*inter-FUNCTION].symmetric & ~REVERSEORDER;
01733                     if ( *inpat == *inter ) psym = sym;
01734                     else psym = functions[*inpat-FUNCTION-WILDOFFSET].symmetric & ~REVERSEORDER;
01735                     if ( sym == ANTISYMMETRIC || sym == SYMMETRIC
01736                     || psym == SYMMETRIC || psym == ANTISYMMETRIC ) {
01737                         if ( sym == ANTISYMMETRIC && psym == SYMMETRIC ) goto rewild;
01738                         if ( sym == SYMMETRIC && psym == ANTISYMMETRIC ) goto rewild;
01739 /*
01740                         Special function call for (anti)symmetric tensors
01741 */
01742                         if ( MatchE(BHEAD inpat,inter,instart,par) ) goto OnSuccess;
01743                     }
01744                     else if ( sym == CYCLESYMMETRIC || sym == RCYCLESYMMETRIC
01745                     || psym == CYCLESYMMETRIC || psym == RCYCLESYMMETRIC ) {
01746 /*
01747                         Special function call for (r)cyclic tensors
01748 */
01749                         if ( MatchCy(BHEAD inpat,inter,instart,par) ) goto OnSuccess;
01750                     }
01751                     else goto rewild;
01752                 }
01753                 else if ( functions[*inter-FUNCTION].spec <= 0
01754                 && ( *inter == *inpat ||
01755                 functions[*inpat-FUNCTION-WILDOFFSET].spec <= 0 ) ) {
01756                     sym = functions[*inter-FUNCTION].symmetric & ~REVERSEORDER;
01757                     if ( *inpat == *inter ) psym = sym;
01758                     else psym = functions[*inpat-FUNCTION-WILDOFFSET].symmetric & ~REVERSEORDER;
01759                     if ( psym == SYMMETRIC || sym == SYMMETRIC
01760 /*
01761                         The next statement was commented out. Why????
01762                         Werkt nog niet. Teken wordt nog niet bijgehouden.
01763                         5-nov-2001
01764 */
01765                         || psym == ANTISYMMETRIC || sym == ANTISYMMETRIC
01766                         ) {
01767                             if ( sym == ANTISYMMETRIC && psym == SYMMETRIC ) goto rewild;
01768                             if ( sym == SYMMETRIC && psym == ANTISYMMETRIC ) goto rewild;
01769                             if ( FunMatchSy(BHEAD inpat,inter,instart,par) ) goto OnSuccess;
01770                         }
01771                         else
01772                             if ( sym == CYCLESYMMETRIC || sym == RCYCLESYMMETRIC
01773                         || psym == CYCLESYMMETRIC || psym == RCYCLESYMMETRIC ) {
01774                             if ( FunMatchCy(BHEAD inpat,inter,instart,par) ) goto OnSuccess;
01775                         }
01776                         else goto rewild;
01777                     }
01778                     else goto rewild;
01779                     AN.terfirstcomm = Oterfirstcomm;
01780                 }
01781                 else if ( par > 0 ) { SetStop = 1; goto maybenext; }
01782             }
01783             else {
01784 rewild:
01785             AN.terfirstcomm = Oterfirstcomm;
01786             if ( *inter != SUBEXPRESSION && MatchFunction(BHEAD inpat,inter,&wilds) ) {
01787                 AN.terfirstcomm = Oterfirstcomm;
01788                 if ( wilds ) {
01789 /*
01790                     Store wildcards to continue in MatchFunction if the current
01791                     wildcards do not work out.
01792 */
```

```
01793                 wildargs = AN.WildArgs;
01794                 wildeat = AN.WildEat;
01795                 for ( i = 0; i < wildargs; i++ ) wildargtaken[i] = AT.WildArgTaken[i];
01796                 oinpat = inpat; ointer = inter;
01797             }
01798             if ( par && *inter == GAMMA && AN.RepFunList[AN.RepFunNum+1] ) {
01799                 SetStop = 1; goto NoMat;
01800             }
01801             if ( par == 2 ) {
01802                 if ( *inter < FUNCTION || functions[*inter-FUNCTION].commute ) {
01803                     goto NoMat;
01804                 }
01805                 par = 1;
01806             }
01807             AN.RepFunList[AN.RepFunNum] = offset;
01808             AN.RepFunNum += 2;
01809             newpat = inpat + inpat[1];
01810             if ( newpat >= AN.patstop ) {
01811                 if ( AN.UseFindOnly == 0 ) {
01812                     if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
01813                         AN.UsedOtherFind = 1;
01814                         goto OnSuccess;
01815                     }
01816                     AN.RepFunNum -= 2;
01817                     goto NoMat;
01818                 }
01819                 goto OnSuccess;
01820             }
01821             if ( *inter < FUNCTION || functions[*inter-FUNCTION].commute ) {
01822                 newter = inter + inter[1];
01823                 if ( newter >= AN.terstop ) goto Failure;
01824                 if ( *inter == GAMMA && inpat[1] <
01825                 inter[1] - AN.RepFunList[AN.RepFunNum-1] ) {
01826                     if ( ScanFunctions(BHEAD newpat,newter,2) ) goto OnSuccess;
01827                     AN.terfirstcomm = Oterfirstcomm;
01828                 }
01829                 else if ( *newter ==  SUBEXPRESSION ) {}
01830                 else if ( functions[*inter-FUNCTION].commute ) {
01831                     if ( ScanFunctions(BHEAD newpat,newter,1) ) goto OnSuccess;
01832                     AN.terfirstcomm = Oterfirstcomm;
01833                     if ( ( *newpat < (FUNCTION+WILDOFFSET)
01834                         && ( functions[*newpat-FUNCTION].commute == 0 ) ) ||
01835                         ( *newpat >= (FUNCTION+WILDOFFSET)
01836                         && ( functions[*newpat-FUNCTION-WILDOFFSET].commute == 0 ) ) ) {
01837                         newter = AN.terfirstcomm;
01838                         if ( newter < AN.terstop && ScanFunctions(BHEAD newpat,newter,1) ) goto
      OnSuccess;
01839                     }
01840                 }
01841                 else {
01842                     if ( ScanFunctions(BHEAD newpat,instart,1) ) goto OnSuccess;
01843                     AN.terfirstcomm = Oterfirstcomm;
01844                 }
01845                 SetStop = par;
01846             }
01847             else {
01848 /*
01849                 Shouldn't this be newpat instead of inpat?????
01850 */
01851                 if ( par && inter > instart && ( ( *newpat < (FUNCTION+WILDOFFSET)
01852                 && functions[*newpat-FUNCTION].commute ) ||
01853                 ( *newpat >= (FUNCTION+WILDOFFSET)
01854                 && functions[*newpat-FUNCTION-WILDOFFSET].commute ) ) ) {
01855                     SetStop = 1;
01856                 }
01857                 else {
01858                     newter = instart;
01859                     if ( ScanFunctions(BHEAD newpat,newter,par) ) goto OnSuccess;
01860                     AN.terfirstcomm = Oterfirstcomm;
01861                 }
01862             }
01863 /*
01864             Restore the old Wildcard assignments
01865 */
01866 NoMat:
01867             i = nwstore;
01868             if ( i > 0 ) {
01869                 m = AN.WildValue;
01870                 t = OldWork + ntwa; r = AT.WildMask;
01871                 do {
01872                     *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01873                 } while ( --i > 0 );
01874                 C->numrhs = *t++;
01875                 C->Pointer = C->Buffer + oldcpointer;
01876             }
01877 /*          AN.RepFunNum -= 2; */
01878             AN.RepFunNum = oRepFunNum;
```

```
01879             if ( wilds ) {
01880                 inter = ointer; inpat = oinpat;
01881                 AN.WildArgs = wildargs;
01882                 AN.WildEat = wildeat;
01883                 for ( i = 0; i < wildargs; i++ ) AT.WildArgTaken[i] = wildargtaken[i];
01884                 goto rewild;
01885             }
01886             if ( SetStop ) break;
01887         }
01888         else if ( par ) {
01889 maybenext:
01890             if ( *inpat < (FUNCTION+WILDOFFSET) ) {
01891                 if ( *inpat < FUNCTION ||
01892                 functions[*inpat-FUNCTION].commute ) break;
01893             }
01894             else {
01895                 if ( functions[*inpat-FUNCTION-WILDOFFSET].commute ) break;
01896             }
01897         }}
01898         inter += inter[1];
01899     } while ( inter < AN.terstop );
01900 Failure:
01901     AN.SignCheck = oldSignCheck;
01902     AT.WorkPointer = OldWork;
01903     return(0);
01904 OnSuccess:
01905     if ( AT.idallflag && AN.nogroundlevel <= 0 ) {
01906         if ( AT.idallmaxnum > 0 && AT.idallnum >= AT.idallmaxnum ) {
01907             AN.terfirstcomm = Oterfirstcomm;
01908             AN.SignCheck = oldSignCheck;
01909             AT.WorkPointer = OldWork;
01910             return(0);
01911         }
01912         SubsInAll(BHEAD0);
01913         AT.idallnum++;
01914         if ( AT.idallmaxnum == 0 || AT.idallnum < AT.idallmaxnum ) goto NoMat;
01915     }
01916     AN.terfirstcomm = Oterfirstcomm;
01917     AN.SignCheck = oldSignCheck;
01918 /*
01919     Now the disorder test
01920 */
01921     if ( AN.DisOrderFlag && AN.RepFunNum >= 4 ) {
01922         WORD k, kk;
01923         for ( i = 2; i < AN.RepFunNum; i += 2 ) {
01924 /*
01925 ------------> We still have to copy the code from Normalize wrt properorderflag
01926 */
01927             m = AN.terstart + AN.RepFunList[i-2];
01928             t = AN.terstart + AN.RepFunList[i];
01929             if ( *m != *t ) {
01930                 if ( *m > *t ) continue;
01931                 goto doesmatch;
01932             }
01933             if ( *m >= FUNCTION && functions[*m-FUNCTION].spec >=
01934                 TENSORFUNCTION ) {
01935                 k = m[1] - FUNHEAD;
01936                 kk = t[1] - FUNHEAD;
01937                 m += FUNHEAD;
01938                 t += FUNHEAD;
01939             }
01940             else {
01941                 k = m[1] - FUNHEAD;
01942                 kk = t[1] - FUNHEAD;
01943                 m += FUNHEAD;
01944                 t += FUNHEAD;
01945             }
01946             while ( k > 0 && kk > 0 ) {
01947                 if ( *m < *t ) goto NextFor;
01948                 else if ( *m++ > *t++ ) goto doesmatch;
01949                 k--; kk--;
01950             }
01951             if ( k > 0 ) goto doesmatch;
01952 NextFor:;
01953         }
01954         SetStop = 1;
01955         goto NoMat;
01956     }
01957 doesmatch:
01958     AT.WorkPointer = OldWork;
01959     return(1);
01960 }
01961
01962 /*
01963         #] ScanFunctions :
01964     #] Patterns :
01965 */
```

4.57 fwin.h File Reference

Macros

- #define LINEFEED '\n'
- #define CARRIAGERETURN 0x0D
- #define WITHRETURN
- #define WITHSYSTEM
- #define P_term(code) exit((int)(code<0?-code:code))
- #define SEPARATOR '\\'
- #define ALTSEPARATOR '/'
- #define PATHSEPARATOR ';'
- #define WITH_ENV

4.57.1 Detailed Description

Settings for Windows computers.

Definition in file fwin.h.

4.57.2 Macro Definition Documentation

4.57.2.1 LINEFEED

```
#define LINEFEED '\n'
```

Definition at line 33 of file fwin.h.

4.57.2.2 CARRIAGERETURN

```
#define CARRIAGERETURN 0x0D
```

Definition at line 34 of file fwin.h.

4.57.2.3 WITHRETURN

```
#define WITHRETURN
```

Definition at line 35 of file fwin.h.

4.57.2.4 WITHSYSTEM

```
#define WITHSYSTEM
```

Definition at line 37 of file fwin.h.

4.57.2.5 P_term

```
#define P_term(
              code ) exit((int)(code<0?-code:code))
```

Definition at line 39 of file fwin.h.

4.57.2.6 SEPARATOR

```
#define SEPARATOR '\\'
```

Definition at line 41 of file fwin.h.

4.57.2.7 ALTSEPARATOR

```
#define ALTSEPARATOR '/'
```

Definition at line 42 of file fwin.h.

4.57.2.8 PATHSEPARATOR

```
#define PATHSEPARATOR ';'
```

Definition at line 43 of file fwin.h.

4.57.2.9 WITH_ENV

```
#define WITH_ENV
```

Definition at line 44 of file fwin.h.

4.58 fwin.h

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
```

```
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032
00033 #define LINEFEED '\n'
00034 #define CARRIAGERETURN 0x0D
00035 #define WITHRETURN
00036
00037 #define WITHSYSTEM
00038
00039 #define P_term(code)    exit((int)(code<0?-code:code))
00040
00041 #define SEPARATOR '\\'
00042 #define ALTSEPARATOR '/'
00043 #define PATHSEPARATOR ';'
00044 #define WITH_ENV
```

4.59 grcc.cc

```
00001 // { ( [
00002 //**************************************************************
00003 // grcc.cc
00004
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 2023-2026 T. Kaneko
00008  *   When using this file you are requested to refer to the publication
00009  *   Comput.Phys.Commun. 92 (1995) 127-152
00010  *
00011  *   This file is part of FORM.
00012  *
00013  *   FORM is free software: you can redistribute it and/or modify it under the
00014  *   terms of the GNU General Public License as published by the Free Software
00015  *   Foundation, either version 3 of the License, or (at your option) any later
00016  *   version.
00017  *
00018  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00019  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00020  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00021  *   details.
00022  *
00023  *   You should have received a copy of the GNU General Public License along
00024  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00025  */
00026 /* #] License : */
00027
00028 #ifndef NOFORM
00029 extern "C" {
00030 #include "form3.h"
00031 }
00032 #endif
00033
00034 #include <iostream>
00035 #include <iomanip>
00036 #include <sstream>
00037 #include <cstdio>
00038 #include <stdlib.h>
00039 #include <string.h>
00040 #include <time.h>
00041
00042 //==============================================================
00043 // new default values of options
00044 // #define OLDDEFAULT
00045
00046 // new function for the output in python format
00047 // #define NEWOUTPY
00048
00049 //==============================================================
00050 // For debugging
00051 //
00052 // #define MONITOR     // in graph elimination
00053 // #define CHECK       // check consistency
00054
00055 //==============================================================
00056 // optimization : best combination depends on process by process
00057 #define SIMPSEL
00058
00059 // optimization : lower and upper bound of deg of target node of connection.
00060 #define MINMAXLEG
00061
00062 // optimization : optimization with respect to 'extonly' particles
```

```
00063 #define OPTEXTONLY
00064
00065 // check unique interaction by name or by code
00066 #define UNIQINTR
00067
00068 // possible extensions of the program
00069 // #define ORBITS
00070
00071 // reverse direction of an edge of agraph when anti-particle flows on it.
00072 // In this case the direction of edges will be diffrent
00073 // from ones of original mgraph.
00074 // #define EDGEPORDER
00075
00076 //--------------------------------------------------------------
00077 #include "grcc.h"
00078
00079 #ifdef GRCC_NAMESPACE
00080 using namespace Grcc;
00081 #endif
00082
00083 #define MAXSTR   1024
00084
00085 //--------------------------------------------------------------
00086 // Macro functions
00087 #define CLWIGHTD(x)    (5*(x))
00088 #define CLWIGHTO(x)    (3*(x)-2)
00089 #define MAXSTRLEN      81
00090
00091 #define MASK(n)        ((1ul) << (n))
00092
00093 #define BOOLSTR(x) ((x) ? "True" : "False")
00094
00095 //--------------------------------------------------------------
00096 // Static variables
00097
00098 static OptDef optDef0[] = {
00099  {"Step",        "Generate particle assigned graphs",               GRCC_AGraph, 0},
00100  {"Outgrf",      "Output to file (out.grf)",                              False, 0},
00101  {"Outgrp",      "Output to file (out.grp)",                              False, 0},
00102  {"OPI",         "Only 1PI graphs",                                       True , 0},
00103  {"NoSelfLoop", "Exclude graphs with loops consist of 1 edge",            True , 0},
00104  {"NoTadpole",  "Exclude graphs with tadpoles (2 edge connected)",        True , 0},
00105  {"No1PtBlock", "Exclude graphs with tadpole blocks (2 node connected)", False, 0},
00106  {"No2PtL1PI",  "Exclude graphs with 2-point subgraphs",                  False, 0},
00107  {"NoExtSelf",  "Exclude graphs with 2-pt subgraphs at ext. particles",   False, 0},
00108  {"NoAdj2PtV",  "Exclude graphs with an edge connecting 2-pt vertices",   False, 0},
00109  {"Block",       "Exclude graphs with more than one block",               False, 0},
00110  {"NoMultiEdge","Exclude graphs with multi-edges",                        False, 0},
00111  {"SymmInitial","Symmetrize initial particles",                           False, 0},
00112  {"SymmFinal",  "Symmetrize final particles",                             False, 0},
00113 };
00114 static OptDef optDef1[] = {
00115  {"Step",        "Generate particle assigned graphs",               GRCC_AGraph, 0},
00116  {"Outgrf",      "Ouput to file (out.grf)",                               False, 0},
00117  {"Outgrp",      "Ouput to file (out.grp)",                               False, 0},
00118  {"OPI",         "Only 1PI graphs",                                       False, 0},
00119  {"NoSelfLoop", "Exclude graphs with loops consist of 1 edge",            False, 0},
00120  {"NoTadpole",  "Exclude graphs with tadpoles (2 edge connected)",        False, 0},
00121  {"No1PtBlock", "Exclude graphs with tadpole blocks (2 node connected)", False, 0},
00122  {"No2PtL1PI",  "Exclude graphs with 2-point subgraphs",                  False, 0},
00123  {"NoExtSelf",  "Exclude graphs with 2-pt subgraphs at ext. particles",   False, 0},
00124  {"NoAdj2PtV",  "Exclude graphs with an edge connecting 2-pt vertices",   False, 0},
00125  {"Block",       "Exclude graphs with more than one block",               False, 0},
00126  {"NoMultiEdge","Exclude graphs with multi-edges",                        False, 0},
00127  {"SymmInitial","Symmetrize initial particles",                           False, 0},
00128  {"SymmFinal",  "Symmetrize final particles",                             False, 0},
00129 };
00130 #ifdef OLDDEFAULT
00131 static OptDef *optDef = &(optDef0[0]);
00132 static int nOptDef = sizeof(optDef0)/sizeof(OptDef);
00133 #else
00134 static OptDef *optDef = &(optDef1[0]);
00135 static int nOptDef = sizeof(optDef1)/sizeof(OptDef);
00136 #endif
00137
00138
00139 static OptQGDef optQGDef[] = {
00140  {"onepi",       "onepr",     "one-particle illreducible"},
00141  {"onshell",     "offshell", "without self-energy part at external particles"},
00142  {"nosigma",     "sigma",     "no 2-point functions"},
00143  {"nosnail",     "snail",     "without snail"},
00144  {"notadpole",   "tadpole",   "without tadpole"},
00145  {"simple",      "notsimple", "without self-loop nor multi-edge"},
00146  {"bipart",      "nonbipart", "only bipartite graph"},
00147  {"cycli",       "cyclr",      "???"},
00148  {"floop",       "",           "without fermion loops of odd length"},
00149 #ifdef GRCC_QGRAF_OPT_TOPOL
```

```
00150  {"topol",      "?",          "topology"},
00151 #endif
00152 };
00153
00154 static int nOptQGDef = sizeof(optQGDef)/sizeof(OptQGDef);
00155
00156 static int      prlevel = 2;
00157 static ErExit  *erExit  = NULL;
00158 static void    *erExitArg  = NULL;
00159
00160 //************************************************************
00161 // Utility functions
00162 //============================================================
00163
00164 static void   grcc_fprintf(FILE* out, const char* fmt, ...);
00165 static void   erEnd(const char *msg);
00166 static Bool   nextPart(int nelem, int nclist, int *clist, int *nl, int *r);
00167 static void   prilist(int n, const int *a, const char *msg);
00168 static int   *newArray(int size, int val);
00169 static int   *deleteArray(int *a);
00170 static int  **newMat(int n0, int n1, int val);
00171 static int  **deleteMat(int **m, int n0);
00172 static void   prIntArray(int n, int *p, const char *msg);
00173 static void   prIntArrayErr(int n, int *p, const char *msg);
00174 static int   *intdup(int n, int *v);
00175 static int   *delintdup(int *v);
00176 static void   bsort(int n, int *ord, int *a);
00177 static void   sorti(int n, int *a);
00178 static int    sortb(int n, int *a);
00179 static int    toSList(int n, int *a);
00180 static int    intSetAdd(int n, int *a, int v, const int size);
00181 static int    intSListAdd(int n, int *a, int v, const int size);
00182 static int    cmpArray(int na, int *a, int nb, int *b);
00183 static Bool   leqArray(int n, int *a, int *b);
00184 static void   prMomStr(int mom, const char *ms, int mn);
00185 static void   listDiff(int na, int *a, int nb, int *b, int *np, int *p, int *nq, int *q, int *nr, int
      *r);
00186 static Bool   isSubSList(int na, int *a, int nb, int *b);
00187 static int    subtrSet(int na, int *a, int nb, int *b, int *c, int size);
00188 static int    nextPerm(int nelem, int nclass, int *cl, int *r, int *q, int *p, int count);
00189 static BigInt ipow(int n, int p);
00190 static BigInt factorial(int n);
00191
00192 // for debugging
00193 #ifdef CHECK
00194 static Bool   isIn(int n, int *a, int v);
00195 #endif
00196
00197 //============================================================
00198 // class Options
00199 //------------------------------------------------------------
00200 Options::Options(void)
00201 {
00202     if (nOptDef != GRCC_OPT_Size) {
00203         grcc_fprintf(GRCC_Stderr, "*** Options: inconsistent default values\n");
00204         grcc_fprintf(GRCC_Stderr, "nOptDef=%d, GRCC_OPT_Size=%d\n",
00205                 nOptDef, GRCC_OPT_Size);
00206         GRCC_ABORT();
00207     }
00208
00209     model  = NULL;
00210     proc   = NULL;
00211     sproc  = NULL;
00212
00213     outmg  = NULL;
00214     endmg  = NULL;
00215     outag  = NULL;
00216
00217     argmg  = NULL;
00218     argemg = NULL;
00219     argag  = NULL;
00220
00221     setDefaultValues();
00222
00223     // QGraf options
00224     if (nOptQGDef != GRCC_QGRAF_OPT_Size) {
00225         grcc_fprintf(GRCC_Stderr, "*** Options: inconsistent default values\n");
00226         grcc_fprintf(GRCC_Stderr, "nOptQGDef=%d, GRCC_QGRAF_OPT_Size=%d\n",
00227                 nOptQGDef, GRCC_QGRAF_OPT_Size);
00228         GRCC_ABORT();
00229     }
00230     nqgopt = 0;
00231     for (int j = 0; j < nOptQGDef; j++) {
00232         qgref[nqgopt].name  = optQGDef[j].name;
00233         qgref[nqgopt].index = j;
00234         qgref[nqgopt].sign  = +1;
00235         nqgopt++;
```

```
00236         if (strlen(optQGDef[j].cname) > 0) {
00237             qgref[nqgopt].name  = optQGDef[j].cname;
00238             qgref[nqgopt].index = j;
00239             qgref[nqgopt].sign  = -1;
00240             nqgopt++;
00241         }
00242     }
00243
00244     // default values
00245     for (int j = 0; j < GRCC_QGRAF_OPT_Size; j++) {
00246         qgopt[j] = 0;
00247     }
00248
00249     // for output
00250     out   = new Output(this);
00251
00252     // subrpocess
00253     sproc = NULL;
00254
00255     // measuring time
00256     time0  = 0.0;
00257     time1  = 0.0;
00258 }
00259
00260 //------------------------------------------------------------------
00261 Options::~Options(void)
00262 {
00263     outmg = NULL;
00264     outag = NULL;
00265
00266     argmg = NULL;
00267     argag = NULL;
00268
00269     delete out;
00270 }
00271
00272 //------------------------------------------------------------------
00273 void Options::setDefaultValues(void)
00274 {
00275     int j;
00276
00277     // default values
00278     for (j = 0; j < GRCC_OPT_Size; j++) {
00279         values[j] = optDef[j].defaultv;
00280     }
00281
00282     values[GRCC_OPT_Step] = GRCC_AGraph;
00283
00284     // print level
00285     prlevel = 1;
00286 }
00287
00288 //------------------------------------------------------------------
00289 void Options::setOldDefaultValues(void)
00290 {
00291     int j;
00292
00293     // default values
00294     for (j = 0; j < GRCC_OPT_Size; j++) {
00295         values[j] = optDef0[j].defaultv;
00296     }
00297
00298     values[GRCC_OPT_Step] = GRCC_AGraph;
00299
00300     // print level
00301     prlevel = 1;
00302 }
00303
00304 //------------------------------------------------------------------
00305 void Options::setOutMG(OutEGB *omg, void *pt)
00306 {
00307     outmg = omg;
00308     argmg = pt;
00309 }
00310
00311 //------------------------------------------------------------------
00312 void Options::setEndMG(OutEGB *emg, void *pt)
00313 {
00314     endmg  = emg;
00315     argemg = pt;
00316 }
00317
00318 //------------------------------------------------------------------
00319 void Options::setOutAG(OutEG *oag, void *pt)
00320 {
00321     outag = oag;
00322     argag = pt;
```

```
00323 }
00324
00325 //-------------------------------------------------------------
00326 void Options::setErExit(ErExit *ere, void *erearg)
00327 {
00328     erExit    = ere;
00329     erExitArg = erearg;
00330 }
00331
00332 //-------------------------------------------------------------
00333 void Options::printLevel(int l)
00334 {
00335     prlevel = l;
00336 }
00337
00338 //-------------------------------------------------------------
00339 void Options::setValue(int ind, int val)
00340 {
00341     if (0 <= ind && ind < GRCC_OPT_Size) {
00342         values[ind] = val;
00343     } else {
00344         if (prlevel > 0) {
00345             grcc_fprintf(GRCC_Stderr, "*** Options::setValue : invalid index=%d ",
00346                     ind);
00347             grcc_fprintf(GRCC_Stderr, "(val=%d)\n", val);
00348         }
00349         erEnd("Options::setValue : invalid index");
00350     }
00351 }
00352
00353 //-------------------------------------------------------------
00354 int Options::getValue(int ind)
00355 {
00356     if (0 <= ind && ind < GRCC_OPT_Size) {
00357         return values[ind];
00358     } else {
00359         if (prlevel > 0) {
00360             grcc_fprintf(GRCC_Stderr, "*** Options::getValue : invalid index=%d\n", ind);
00361         }
00362         erEnd("Options::getValue : invalid index");
00363     }
00364     return -1;
00365 }
00366
00367 //-------------------------------------------------------------
00368 void Options::setQGrafOpt(int *qg)
00369 {
00370     // reset options : default is to generate all connected graphs
00371
00372 #ifdef OLDDEFAULT
00373     values[GRCC_OPT_1PI]         = False;
00374     values[GRCC_OPT_NoSelfLoop]  = False;
00375     values[GRCC_OPT_NoTadpole]   = False;
00376     values[GRCC_OPT_No1PtBlock]  = False;
00377     values[GRCC_OPT_No2PtL1PI]   = False;
00378     values[GRCC_OPT_NoExtSelf]   = False;
00379     values[GRCC_OPT_NoAdj2PtV]   = False;
00380     values[GRCC_OPT_Block]       = False;
00381     values[GRCC_OPT_SymmInitial] = False;
00382     values[GRCC_OPT_SymmFinal]   = False;
00383 #endif
00384
00385     for (int j = 0; j < GRCC_QGRAF_OPT_Size; j++) {
00386         qgopt[j] = qg[j];
00387     }
00388
00389     // {"onepi", "onepr"},
00390     if (qgopt[GRCC_QGRAF_OPT_ONEPI] > 0) {
00391         values[GRCC_OPT_1PI] = True;
00392     } else if (qgopt[GRCC_QGRAF_OPT_ONEPI] < 0) {
00393         values[GRCC_OPT_1PI] = False;
00394     }
00395
00396     // {"onshell", "offshell"}
00397     if (qgopt[GRCC_QGRAF_OPT_ONSHELL] > 0) {
00398         values[GRCC_OPT_NoExtSelf] = 1;
00399     } else if (qgopt[GRCC_QGRAF_OPT_ONSHELL] < 0) {
00400         values[GRCC_OPT_NoExtSelf] = 0;
00401     }
00402
00403     // {"nosigma", "sigma"}
00404
00405     // {"nosnail",    "snail"},
00406     if (qgopt[GRCC_QGRAF_OPT_NOSNAIL] > 0) {
00407         values[GRCC_OPT_NoTadpole]  = True;
00408         values[GRCC_OPT_No1PtBlock] = True;
00409     } else if (qgopt[GRCC_QGRAF_OPT_NOSNAIL] < 0) {
```

```
00410         values[GRCC_OPT_NoTadpole]  = False;
00411         values[GRCC_OPT_No1PtBlock] = False;
00412     }
00413 
00414     // {"notadpole", "tadpole"}
00415     if (qgopt[GRCC_QGRAF_OPT_NOTADPOLE] > 0) {
00416         values[GRCC_OPT_NoTadpole] = True;
00417     } else if (qgopt[GRCC_QGRAF_OPT_NOTADPOLE] < 0) {
00418         values[GRCC_OPT_NoTadpole] = False;
00419     }
00420 
00421     // {"simple",     "notsimple"},
00422     if (qgopt[GRCC_QGRAF_OPT_SIMPLE] > 0) {
00423         values[GRCC_OPT_NoSelfLoop]  = True;
00424         values[GRCC_OPT_NoMultiEdge] = True;
00425     } else if (qgopt[GRCC_QGRAF_OPT_SIMPLE] < 0) {
00426         values[GRCC_OPT_NoSelfLoop]  = False;
00427         values[GRCC_OPT_NoMultiEdge] = False;
00428     }
00429 
00430     // {"bipart",     "nonbipart"},
00431     // {"cycli",      "cyclr"},
00432 
00433 #ifdef GRCC_QGRAF_OPT_TOPOL
00434     // {"topol",     ""},
00435     if (qgopt[GRCC_QGRAF_OPT_TOPOL] > 0) {
00436         values[GRCC_OPT_SymmInitial] = True;
00437         values[GRCC_OPT_SymmFinal]   = True;
00438     }
00439 #endif
00440 }
00441 
00442 //------------------------------------------------------------
00443 void Options::print(void)
00444 {
00445     int j;
00446 
00447     grcc_fprintf(GRCC_Stdout, "Options\n");
00448     grcc_fprintf(GRCC_Stdout, "+++ GRCC_OPT_Size=%d, print level=%d: ",
00449            GRCC_OPT_Size, prlevel);
00450     grcc_fprintf(GRCC_Stdout, "symbol = value (default)\n");
00451     for (j=0; j < GRCC_OPT_Size; j++) {
00452         grcc_fprintf(GRCC_Stdout, "   %4d GRCC_OPT_%-15s = %2d (%2d)\n",
00453                j, optDef[j].name, values[j], optDef[j].defaultv);
00454     }
00455     grcc_fprintf(GRCC_Stdout, "    outgrf=%s, outgrp=%s\n", out->outgrf, out->outgrp);
00456     grcc_fprintf(GRCC_Stdout, "    GRCC_QGRAF_OPT_Size=%d:\n", GRCC_QGRAF_OPT_Size);
00457     for (j=0; j < GRCC_QGRAF_OPT_Size; j++) {
00458         grcc_fprintf(GRCC_Stdout, "   %4d %-10s = %2d\n", j, optQGDef[j].name, qgopt[j]);
00459     }
00460 }
00461 
00462 //------------------------------------------------------------
00463 const OptDef *Options::getDef(void)
00464 {
00465     return optDef;
00466 }
00467 
00468 //------------------------------------------------------------
00469 const OptDef *Options::getOldDef(void)
00470 {
00471     return optDef0;
00472 }
00473 
00474 //------------------------------------------------------------
00475 const OptQGDef *Options::getQGDef(void)
00476 {
00477     return optQGDef;
00478 }
00479 
00480 //------------------------------------------------------------
00481 void Options::setOutputF(Bool outf, const char *fname)
00482 {
00483     values[GRCC_OPT_Outgrf] = outf;
00484     out->setOutgrf(fname);
00485 }
00486 
00487 //------------------------------------------------------------
00488 void Options::setOutputP(Bool outp, const char *fname)
00489 {
00490     values[GRCC_OPT_Outgrp] = outp;
00491     out->setOutgrp(fname);
00492 }
00493 
00494 //------------------------------------------------------------
00495 void Options::printModel(void)
00496 {
```

```
00497     if (prlevel > 0) {
00498         if (model != NULL) {
00499             model->prModel();
00500         } else {
00501             grcc_fprintf(GRCC_Stdout, "*** model is not defined\n");
00502         }
00503     }
00504 }
00505
00506 //---------------------------------------------------------------
00507 void Options::outModel(void)
00508 {
00509     if (out != NULL && model != NULL) {
00510         out->outModelF();
00511         out->outModelP();
00512     }
00513 }
00514
00515 //---------------------------------------------------------------
00516 void Options::begin(Model *mdl)
00517 {
00518     model = mdl;
00519     if (out != NULL) {
00520         if (values[GRCC_OPT_Outgrf]) {
00521             out->outBeginF(model, values[GRCC_OPT_Outgrf]);
00522             if (model != NULL) {
00523                 out->outModelF();
00524             }
00525         }
00526         if (values[GRCC_OPT_Outgrp]) {
00527             out->outBeginP(model, values[GRCC_OPT_Outgrp]);
00528             if (model != NULL) {
00529                 out->outModelP();
00530             }
00531         }
00532     }
00533 }
00534
00535 //---------------------------------------------------------------
00536 void Options::end(void)
00537 {
00538     if (prlevel > 1) {
00539         grcc_fprintf(GRCC_Stdout, "Optimization: ");
00540 #ifdef SIMPSEL
00541         grcc_fprintf(GRCC_Stdout, "SIMPSEL=1 ");
00542 #else
00543         grcc_fprintf(GRCC_Stdout, "SIMPSEL=0 ");
00544 #endif
00545 #ifdef MINMAXLEG
00546         grcc_fprintf(GRCC_Stdout, "MINMAXLEG=1 ");
00547 #else
00548         grcc_fprintf(GRCC_Stdout, "MINMAXLEG=0 ");
00549 #endif
00550 #ifdef OPTEXTONLY
00551         grcc_fprintf(GRCC_Stdout, "OPTEXTONLY=1 ");
00552 #else
00553         grcc_fprintf(GRCC_Stdout, "OPTEXTONLY=0 ");
00554 #endif
00555         grcc_fprintf(GRCC_Stdout, "\n");
00556     }
00557
00558     if (out != NULL && values[GRCC_OPT_Outgrf]) {
00559         out->outEndF();
00560     }
00561     if (out != NULL && values[GRCC_OPT_Outgrp]) {
00562         out->outEndP();
00563     }
00564     model = NULL;
00565 }
00566 //
00567 //---------------------------------------------------------------
00568 void Options::beginProc(Process *prc)
00569 {
00570     proc = prc;
00571     if (out != NULL && values[GRCC_OPT_Outgrf]) {
00572         out->outProcBeginF(prc);
00573     }
00574     if (out != NULL && values[GRCC_OPT_Outgrp]) {
00575         out->outProcBeginP(prc);
00576     }
00577     if (proc != NULL) {
00578         proc->mgrcount = 0;
00579         proc->agrcount = 0;
00580     }
00581 }
00582
00583 //---------------------------------------------------------------
```

```
00584 void Options::endProc(void)
00585 {
00586     int k;
00587 
00588     if (proc == NULL) {
00589         if (out != NULL && values[GRCC_OPT_Outgrf]) {
00590             out->outProcEndF();
00591         }
00592         if (out != NULL && values[GRCC_OPT_Outgrp]) {
00593             out->outProcEndP();
00594         }
00595         return;
00596     }
00597     if (prlevel > 0) {
00598         grcc_fprintf(GRCC_Stdout, "\n");
00599         grcc_fprintf(GRCC_Stdout, "+++ Proc %d: ext=%d, loop=%d, ",
00600                proc->id, proc->nExtern, proc->loop);
00601         if (model != NULL) {
00602             grcc_fprintf(GRCC_Stdout, "order=");
00603             prIntArray(model->ncouple, proc->clist, ": ");
00604             model->prParticleArray(proc->ninitl, proc->initlPart, "-->");
00605             model->prParticleArray(proc->nfinal, proc->finalPart, "");
00606         }
00607         grcc_fprintf(GRCC_Stdout, " (%8.2f sec)\n", proc->sec);
00608 
00609 
00610         grcc_fprintf(GRCC_Stdout, "    Proc    %d: Total M-Graphs=%ld, M-Graphs=",
00611                proc->id, proc->nMGraphs);
00612         proc->wMGraphs.print(" (Conn)\n");
00613 
00614         grcc_fprintf(GRCC_Stdout, "    Proc    %d: Total M-Graphs=%ld, M-Graphs=",
00615                proc->id, proc->nMOPI);
00616         proc->wMOPI.print(" (1PI)\n");
00617 
00618         grcc_fprintf(GRCC_Stdout, "    Proc    %d: Total A-Graphs=%ld, A-Graphs=",
00619                proc->id, proc->nAGraphs);
00620         proc->wAGraphs.print(" (Conn)\n");
00621 
00622         grcc_fprintf(GRCC_Stdout, "    Proc    %d: Total A-Graphs=%ld, A-Graphs=",
00623                proc->id, proc->nAOPI);
00624         proc->wAOPI.print(" (1PI)\n");
00625 
00626         grcc_fprintf(GRCC_Stdout, "#  { %d,{", proc->ninitl);
00627         for (k = 0; k < proc->ninitl; k++) {
00628             if (k != 0) {
00629                 grcc_fprintf(GRCC_Stdout, ", ");
00630             }
00631             if (proc->model != NULL) {
00632                 grcc_fprintf(GRCC_Stdout, "\"%s\"", proc->model->particleName(proc->initlPart[k]));
00633             } else {
00634                 grcc_fprintf(GRCC_Stdout, "%d", proc->initlPart[k]);
00635             }
00636         }
00637         grcc_fprintf(GRCC_Stdout, "}, %d,{", proc->nfinal);
00638         for (k = 0; k < proc->nfinal; k++) {
00639             if (k != 0) {
00640                 grcc_fprintf(GRCC_Stdout, ", ");
00641             }
00642             if (proc->model != NULL) {
00643                 grcc_fprintf(GRCC_Stdout, "\"%s\"", proc->model->particleName(proc->finalPart[k]));
00644             } else {
00645                 grcc_fprintf(GRCC_Stdout, "%d", proc->initlPart[k]);
00646             }
00647         }
00648         grcc_fprintf(GRCC_Stdout, "}, ");
00649         if (model != NULL) {
00650             grcc_fprintf(GRCC_Stdout, "{");
00651             for (k = 0; k < model->ncouple; k++) {
00652                 if (k != 0) {
00653                     grcc_fprintf(GRCC_Stdout, ", ");
00654                 }
00655                 grcc_fprintf(GRCC_Stdout, "%d", proc->clist[k]);
00656             }
00657             grcc_fprintf(GRCC_Stdout, "},");
00658         }
00659         grcc_fprintf(GRCC_Stdout, "},%6ldL,%6ldL,%3ldL, -1.0, %4.2f},\n",
00660                proc->nAOPI, proc->wAOPI.num, proc->wAOPI.den, proc->sec);
00661     }
00662 
00663     if (out != NULL && values[GRCC_OPT_Outgrf]) {
00664         out->outProcEndF();
00665     }
00666     if (out != NULL && values[GRCC_OPT_Outgrp]) {
00667         out->outProcEndP();
00668     }
00669     proc = NULL;
00670 }
```

```
00671
00672 //------------------------------------------------------------
00673 void Options::beginSubProc(SProcess *sprc)
00674 {
00675     sproc = sprc;
00676
00677     // 'beginProc' is called before
00678     if (out != NULL && values[GRCC_OPT_Outgrf]) {
00679         out->outSProcBeginF(sprc);
00680     }
00681     if (out != NULL && values[GRCC_OPT_Outgrp]) {
00682         out->outSProcBeginP(sprc);
00683     }
00684     if (proc != NULL) {
00685         return;
00686     }
00687     if (sprc != NULL) {
00688         sproc->mgrcount = 0;
00689         sproc->agrcount = 0;
00690     }
00691 }
00692
00693 //------------------------------------------------------------
00694 void Options::endSubProc(void)
00695 {
00696     MGraph *mgraph;
00697
00698     if (sproc == NULL) {
00699         if (out != NULL && values[GRCC_OPT_Outgrf]) {
00700             out->outProcEndF();
00701         }
00702         if (out != NULL && values[GRCC_OPT_Outgrp]) {
00703             out->outProcEndP();
00704         }
00705         return;
00706     }
00707     mgraph = sproc->mgraph;
00708     if (sproc != NULL) {
00709         sproc->resultMGraph(mgraph->cDiag, mgraph->wscon,
00710                             mgraph->c1PI, mgraph->wsopi);
00711     }
00712
00713     if (prlevel > 1) {
00714         grcc_fprintf(GRCC_Stdout, "\n");
00715         grcc_fprintf(GRCC_Stdout, "+++ Subproc %d: ext=%d, loop=%d, nodes=%d, edges=%d\n",
00716                sproc->id, sproc->nExtern, sproc->loop,
00717                sproc->nNodes, sproc->nEdges);
00718
00719         grcc_fprintf(GRCC_Stdout, "    Subproc %d: Total M-Graphs=%ld, M-Wsum=",
00720                sproc->id, sproc->nMGraphs);
00721         sproc->wMGraphs.print(" (Conn)\n");
00722
00723         grcc_fprintf(GRCC_Stdout, "    Subproc %d: Total M-Graphs=%ld, M-Wsum=",
00724                sproc->id, sproc->nMOPI);
00725         sproc->wMOPI.print(" (1PI)\n");
00726
00727         grcc_fprintf(GRCC_Stdout, "    Subproc %d: Total A-Graphs=%ld, A-Wsum=",
00728                sproc->id, sproc->nAGraphs);
00729         sproc->wAGraphs.print(" (Conn)\n");
00730
00731         grcc_fprintf(GRCC_Stdout, "    Subproc %d: Total A-Graphs=%ld, A-Wsum=",
00732                sproc->id, sproc->nAOPI);
00733         sproc->wAOPI.print(" (1PI)\n");
00734     }
00735     if (prlevel > 0) {
00736         grcc_fprintf(GRCC_Stdout, "\n");
00737         grcc_fprintf(GRCC_Stdout, "* Total %ld MGraphs; %ld 1PI", mgraph->cDiag, mgraph->c1PI);
00738         grcc_fprintf(GRCC_Stdout, " wscon = ");
00739         mgraph->wscon.print(" ( ");
00740         mgraph->wsopi.print(" 1PI)\n");
00741
00742 #ifdef MONITOR
00743         grcc_fprintf(GRCC_Stdout, "* refine:                    %ld\n", mgraph->nCallRefine);
00744         grcc_fprintf(GRCC_Stdout, "* discarded for refinement:  %ld\n", mgraph->discardRefine);
00745         grcc_fprintf(GRCC_Stdout, "* discarded for disconnected: %ld\n", mgraph->discardDisc);
00746         grcc_fprintf(GRCC_Stdout, "* discarded for duplication:  %ld\n", mgraph->discardIso);
00747 #endif
00748     }
00749
00750     if (proc != NULL) {
00751         return;
00752     }
00753     if (out != NULL && values[GRCC_OPT_Outgrf]) {
00754         out->outProcEndF();
00755     }
00756     if (out != NULL && values[GRCC_OPT_Outgrp]) {
00757         out->outProcEndP();
```

```
00758     }
00759 }
00760
00761 //-------------------------------------------------------------
00762 void Options::newMGraph(MGraph *mgr)
00763 {
00764     MGraph *mgraph = mgr;
00765
00766     if (proc != NULL) {
00767         proc->mgrcount++;
00768         mgr->egraph->mId = proc->mgrcount;
00769     } else if (sproc != NULL) {
00770         sproc->mgrcount++;
00771         mgr->egraph->mId = sproc->mgrcount;
00772     }
00773
00774     if (out != NULL
00775         && values[GRCC_OPT_Step] == GRCC_MGraph) {
00776         if (values[GRCC_OPT_Outgrf]) {
00777             out->outEGraphF(mgraph->egraph);
00778         }
00779         if (values[GRCC_OPT_Outgrp]) {
00780             out->outEGraphP(mgraph->egraph);
00781         }
00782     }
00783     if (sproc != NULL) {
00784         sproc->endMGraph(mgr);
00785     }
00786     if (endmg != NULL) {
00787         (*endmg)(mgr->egraph, argemg);
00788     }
00789 }
00790
00791 //-------------------------------------------------------------
00792 void Options::newAGraph(EGraph *egraph)
00793 {
00794     Fraction sf, zr;
00795
00796     if (proc != NULL) {
00797         proc->agrcount++;
00798         egraph->sId = proc->agrcount;
00799     } else if (sproc != NULL) {
00800         sproc->agrcount++;
00801         egraph->sId = sproc->agrcount;
00802     }
00803
00804     if (sproc != NULL) {
00805         zr = Fraction(0, 1);
00806         if ((egraph->nsym)*(egraph->esym) != 0) {
00807             sf = Fraction(egraph->extperm, egraph->nsym*egraph->esym);
00808         } else {
00809             sf = zr;
00810         }
00811         if (egraph->mgraph->opi) {
00812             sproc->resultAGraph(1, sf, 1, sf);
00813         } else {
00814             sproc->resultAGraph(1, sf, 0, zr);
00815         }
00816     }
00817
00818     if (out != NULL && values[GRCC_OPT_Outgrf]) {
00819         out->outEGraphF(egraph);
00820     }
00821     if (out != NULL && values[GRCC_OPT_Outgrp]) {
00822         out->outEGraphP(egraph);
00823     }
00824     if (outag != NULL) {
00825         (*outag)(egraph, argag);
00826     }
00827
00828     sproc->endAGraph(egraph);
00829 }
00830
00831 //=============================================================
00832 Output::Output(Options *optn)
00833 {
00834     opt     = optn;
00835     model   = NULL;
00836     proc    = NULL;
00837     sproc   = NULL;
00838     outgrfp = NULL;
00839     outgrpp = NULL;
00840     procId  = -1;
00841     outgrf  = NULL;
00842     outgrp  = NULL;
00843     outproc = False;
00844
```

```
00845 }
00846
00847 //-------------------------------------------------------------
00848 Output::~Output(void)
00849 {
00850     if (outgrfp != NULL) {
00851         if (prlevel > 0) {
00852             grcc_fprintf(GRCC_Stderr, "*** file has not been closed : \"%s\"\n",
00853                     outgrf);
00854         }
00855         fclose(outgrfp);
00856         outgrfp = NULL;
00857         erEnd("file has not been closed");
00858     }
00859     if (outgrf != NULL) {
00860         free(outgrf);
00861         outgrf = NULL;
00862     }
00863     if (outgrpp != NULL) {
00864         if (prlevel > 0) {
00865             grcc_fprintf(GRCC_Stderr, "*** file has not been closed : \"%s\"\n",
00866                     outgrp);
00867         }
00868         fclose(outgrpp);
00869         outgrpp = NULL;
00870         erEnd("file has not been closed");
00871     }
00872     if (outgrp != NULL) {
00873         free(outgrp);
00874         outgrp = NULL;
00875     }
00876 }
00877
00878 //-------------------------------------------------------------
00879 void Output::setOutgrf(const char *fname)
00880 {
00881     if (outgrf != NULL && strcmp(outgrf, fname) == 0) {
00882         return;
00883     }
00884
00885     if (outgrf != NULL) {
00886         // delete outgrf;
00887         free(outgrf);
00888     }
00889     if (fname == NULL || strlen(fname) < 1) {
00890         outgrf = NULL;
00891     } else {
00892         outgrf = strdup(fname);
00893     }
00894 }
00895
00896 //-------------------------------------------------------------
00897 void Output::setOutgrp(const char *fname)
00898 {
00899     if (outgrp != NULL && strcmp(outgrp, fname) == 0) {
00900         return;
00901     }
00902
00903     if (outgrp != NULL) {
00904         free(outgrp);
00905     }
00906     if (fname == NULL || strlen(fname) < 1) {
00907         outgrp = NULL;
00908     } else {
00909         outgrp = strdup(fname);
00910     }
00911 }
00912
00913 //-------------------------------------------------------------
00914 Bool Output::outBeginF(Model *mdl, Bool pr)
00915 {
00916     time_t tp;
00917     struct tm *tm;
00918     char   sdate[MAXSTRLEN];
00919
00920     model = mdl;
00921
00922     if (outgrf == NULL || strlen(outgrf) < 1) {
00923         outgrfp = NULL;
00924         return True;
00925     } else if (outgrfp != NULL) {
00926         if (prlevel > 0) {
00927             grcc_fprintf(GRCC_Stderr, "*** outBegin: \"%s\" is already opened\n",
00928                     outgrf);
00929         }
00930         erEnd("outBegin: file is already opened\n");
00931     }
```

```
00932
00933     if (!pr) {
00934         return True;
00935     }
00936     if ((outgrfp = fopen(outgrf, "w")) == NULL) {
00937         if (prlevel > 0) {
00938             grcc_fprintf(GRCC_Stderr, "*** outBegin: cannot open %s\n", outgrf);
00939         }
00940         return False;
00941     }
00942     tp = time(NULL);
00943     tm = localtime(&tp);
00944     strftime(sdate, MAXSTRLEN, "%Y/%m/%d %H:%M:%S", tm);
00945
00946     fprintf(outgrfp, "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
00947     fprintf(outgrfp, "%% Generated by grcc at \"%s\"\n", sdate);
00948     fprintf(outgrfp, "Version={2,2,0,0};\n");
00949     if (model != NULL) {
00950         fprintf(outgrfp, "Model=\"./%s.mdl\";\n", model->name);
00951     } else {
00952         fprintf(outgrfp, "Model=\"./Phi34.mdl\";\n");
00953     }
00954
00955
00956     return True;
00957 }
00958
00959 //------------------------------------------------------------
00960 Bool Output::outBeginP(Model *mdl, Bool pr)
00961 {
00962     time_t tp;
00963     struct tm *tm;
00964     char   sdate[MAXSTRLEN];
00965
00966     model = mdl;
00967
00968     if (outgrp == NULL || strlen(outgrp) < 1) {
00969         outgrpp = NULL;
00970         return True;
00971     } else if (outgrpp != NULL) {
00972         if (prlevel > 0) {
00973             grcc_fprintf(GRCC_Stderr, "*** outBegin: \"%s\" is already opened\n",
00974                     outgrp);
00975         }
00976         erEnd("outBegin: file is already opened\n");
00977     }
00978
00979     if (!pr) {
00980         return True;
00981     }
00982     if ((outgrpp = fopen(outgrp, "w")) == NULL) {
00983         if (prlevel > 0) {
00984             grcc_fprintf(GRCC_Stderr, "*** outBegin: cannot open %s\n", outgrp);
00985         }
00986         return False;
00987     }
00988     tp = time(NULL);
00989     tm = localtime(&tp);
00990     strftime(sdate, MAXSTRLEN, "%Y/%m/%d %H:%M:%S", tm);
00991
00992     fprintf(outgrpp, "################################\n");
00993     fprintf(outgrpp, "# Generated by 'grcc' at \"%s\"\n", sdate);
00994
00995     return True;
00996 }
00997
00998 //------------------------------------------------------------
00999 void Output::outEndF(void)
01000 {
01001     if (outgrfp == NULL) {
01002         return;
01003     }
01004     fprintf(outgrfp, "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
01005     fprintf(outgrfp, "End=1;\n");
01006     fprintf(outgrfp, "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
01007     fclose(outgrfp);
01008     outgrfp = NULL;
01009
01010     free(outgrf);
01011     outgrf = NULL;
01012 }
01013
01014 //------------------------------------------------------------
01015 void Output::outEndP(void)
01016 {
01017     if (outgrpp == NULL) {
01018         return;
```

```
01019     }
01020     fprintf(outgrpp, "################################\n");
01021     fprintf(outgrpp, "# End\n");
01022     fprintf(outgrpp, "################################\n");
01023     fclose(outgrpp);
01024     outgrpp = NULL;
01025
01026     free(outgrp);
01027     outgrp = NULL;
01028 }
01029
01030 //-------------------------------------------------------------
01031 void Output::outProcBeginF(Process *prc)
01032 {
01033     int k, ex;
01034
01035     if (prc == NULL) {
01036         return;
01037     }
01038
01039     proc    = prc;
01040     sproc   = NULL;
01041     procId  = proc->id;
01042
01043     if (outgrfp == NULL) {
01044         return;
01045     }
01046     if (outproc) {
01047         return;
01048     }
01049     outproc = True;
01050
01051     fprintf(outgrfp, "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
01052     fprintf(outgrfp, "Process=%d;\n", proc->id);
01053     fprintf(outgrfp, "External=%d;\n", proc->ninitl+proc->nfinal);
01054     for (k = 0, ex = 0; k < proc->ninitl; k++, ex++) {
01055         fprintf(outgrfp, "%4d= initial %s;\n", ex,
01056                 proc->model->particleName(proc->initlPart[k]));
01057     }
01058     for (k = 0; k < proc->nfinal; k++, ex++) {
01059         fprintf(outgrfp, "%4d= final   %s;\n", ex,
01060                 proc->model->particleName(proc->finalPart[k]));
01061     }
01062     fprintf(outgrfp, "Eend;\n");
01063     for (k = 0; k < proc->model->ncouple; k++) {
01064         fprintf(outgrfp, "%s=%d;  ", proc->model->cnlist[k], proc->clist[k]);
01065     }
01066     fprintf(outgrfp, "Loop=%d;\n", proc->loop);
01067     fprintf(outgrfp, "OPI=%s;\n", (opt->values[GRCC_OPT_1PI] > 0? "Yes" : "No"));
01068     fprintf(outgrfp, "Assign=%s;\n", (opt->values[GRCC_OPT_Step]==GRCC_AGraph ? "Yes" : "No"));
01069     fprintf(outgrfp, "%% Options : dummy values\n");
01070     fprintf(outgrfp, "Expand=Yes; ExpMin=0; ExpMax=-1;\n");
01071     fprintf(outgrfp, "Block=No; Tadpole=Yes; Extself=Yes;\n");
01072     fprintf(outgrfp, "Selfe=Yes; Countert=No; AnyCT=No;Undefp=No;\n");
01073 }
01074
01075 //-------------------------------------------------------------
01076 void Output::outProcBeginP(Process *prc)
01077 {
01078     if (prc == NULL) {
01079         return;
01080     }
01081
01082     proc    = prc;
01083     sproc   = NULL;
01084     procId  = proc->id;
01085
01086     if (outgrpp == NULL) {
01087         return;
01088     }
01089     if (outproc) {
01090         return;
01091     }
01092     outproc = True;
01093
01094     fprintf(outgrpp, "# Process=%d;\n", proc->id);
01095 }
01096
01097 //-------------------------------------------------------------
01098 void Output::outProcBegin0(int next, int couple, int loop)
01099 {
01100     int k, ex;
01101
01102     procId  = 0;
01103
01104     if (outgrfp == NULL) {
01105         return;
```

```
01106     }
01107     if (outproc) {
01108         return;
01109     }
01110     outproc = True;
01111 
01112     fprintf(outgrfp, "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
01113     fprintf(outgrfp, "Process=%d;\n", procId);
01114     fprintf(outgrfp, "External=%d;\n", next);
01115     int ninit = (next + 1) / 2;
01116     for (k = 0, ex = 0; k < ninit; k++, ex++) {
01117         fprintf(outgrfp, "%4d= initial undef;\n", ex);
01118     }
01119     for (k = 0; k < next - ninit; k++, ex++) {
01120         fprintf(outgrfp, "%4d= final   undef;\n", ex);
01121     }
01122     fprintf(outgrfp, "Eend;\n");
01123     fprintf(outgrfp, "GRCC_PHI=%d;  ", couple);
01124  // fprintf(outgrfp, "PHI=%d;  ", couple);
01125     fprintf(outgrfp, "Loop=%d;\n", loop);
01126     fprintf(outgrfp, "OPI=%s;\n", (opt->values[GRCC_OPT_1PI] > 0? "Yes" : "No"));
01127     fprintf(outgrfp, "Assign=No;\n");
01128     fprintf(outgrfp, "%% Options : dummy values\n");
01129     fprintf(outgrfp, "Expand=Yes; ExpMin=0; ExpMax=-1;\n");
01130     fprintf(outgrfp, "Block=No; Tadpole=Yes; Extself=Yes;\n");
01131     fprintf(outgrfp, "Selfe=Yes; Countert=No; AnyCT=No;Undefp=No;\n");
01132 }
01133 
01134 //--------------------------------------------------------------
01135 void Output::outSProcBeginF(SProcess *sprc)
01136 {
01137     int j, k, ex;
01138     PNodeClass *pnc;
01139 
01140     if (sprc == NULL) {
01141         return;
01142     }
01143 
01144     sproc  = sprc;
01145     procId = sproc->id;
01146     pnc    = sproc->pnclass;
01147 
01148     if (outgrfp == NULL) {
01149         return;
01150     }
01151     if (outproc) {
01152         return;
01153     }
01154     outproc = True;
01155     fprintf(outgrfp, "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
01156     fprintf(outgrfp, "Process=%d;\n", sproc->id);
01157     fprintf(outgrfp, "External=%d;\n", sproc->nExtern);
01158 
01159     ex = 0;
01160     for (j = 0; j < pnc->nclass; j++) {
01161         if (pnc->type[j] == GRCC_AT_Initial) {
01162             for (k = 0; k < pnc->count[j]; k++) {
01163                 fprintf(outgrfp, "%4d= initial %s;\n", ex,
01164                         model->particleName(pnc->particle[j]));
01165                 ex++;
01166             }
01167         } else if (pnc->type[j] == GRCC_AT_Final) {
01168             for (k = 0; k < pnc->count[j]; k++) {
01169                 fprintf(outgrfp, "%4d= final %s;\n", ex,
01170                         model->particleName(-pnc->particle[j]));
01171                 ex++;
01172             }
01173         }
01174     }
01175 
01176     fprintf(outgrfp, "Eend;\n");
01177     for (k = 0; k < sproc->model->ncouple; k++) {
01178         fprintf(outgrfp, "%s=%d;  ", sproc->model->cnlist[k], sproc->clist[k]);
01179     }
01180     fprintf(outgrfp, "Loop=%d;\n", sproc->loop);
01181     fprintf(outgrfp, "OPI=%s;\n", (opt->values[GRCC_OPT_1PI] > 0? "Yes" : "No"));
01182     fprintf(outgrfp, "Assign=%s;\n", (opt->values[GRCC_OPT_Step]==GRCC_AGraph ? "Yes" : "No"));
01183     fprintf(outgrfp, "%% Options : dummy values\n");
01184     fprintf(outgrfp, "Expand=Yes; ExpMin=0; ExpMax=-1;\n");
01185     fprintf(outgrfp, "Block=No; Tadpole=Yes; Extself=Yes;\n");
01186     fprintf(outgrfp, "Selfe=Yes; Countert=No; AnyCT=No;Undefp=No;\n");
01187 }
01188 
01189 //--------------------------------------------------------------
01190 void Output::outSProcBeginP(SProcess *sprc)
01191 {
01192     if (sprc == NULL) {
```

```
01193         return;
01194     }
01195
01196     sproc  = sprc;
01197     procId = sproc->id;
01198
01199     if (outgrpp == NULL) {
01200         return;
01201     }
01202     if (outproc) {
01203         return;
01204     }
01205     outproc = True;
01206     fprintf(outgrpp, "# SProcess=%d;\n", sproc->id);
01207 }
01208
01209 //-------------------------------------------------------------
01210 void Output::outProcEndF(void)
01211 {
01212     if (outgrfp == NULL) {
01213         return;
01214     }
01215     if (proc != NULL) {
01216         fprintf(outgrfp, "%%------------------------------\n");
01217         fprintf(outgrfp, "Pend=%d;\n", proc->id);
01218     } else if (sproc != NULL) {
01219         fprintf(outgrfp, "%%------------------------------\n");
01220         fprintf(outgrfp, "Pend=%d;\n", sproc->id);
01221     } else {
01222         fprintf(outgrfp, "%%------------------------------\n");
01223         fprintf(outgrfp, "Pend=1;\n");
01224     }
01225     fflush(outgrfp);
01226 }
01227
01228 //-------------------------------------------------------------
01229 void Output::outProcEndP(void)
01230 {
01231     if (outgrpp == NULL) {
01232         return;
01233     }
01234     fflush(outgrpp);
01235 }
01236
01237 //-------------------------------------------------------------
01238 void Output::outEGraphF(EGraph *egraph)
01239 {
01240     int nd, lg, ptcl, ed, intr, j, k, loop;
01241     Model *mdl = egraph->model;
01242     Bool popt;
01243     EFLine *fl;
01244
01245     if (outgrfp == NULL) {
01246         return;
01247     }
01248     fprintf(outgrfp, "%%------------------------------\n");
01249     if (egraph->assigned) {
01250         fprintf(outgrfp, "Graph=%ld;\n", egraph->sId);
01251         fprintf(outgrfp, "%% AGraph=%ld;\n", egraph->aId);
01252     } else {
01253         fprintf(outgrfp, "Graph=%ld;\n", egraph->mId);
01254     }
01255     fprintf(outgrfp, "Gtype=%ld;\n", egraph->mId);
01256     if (egraph->assigned) {
01257         fprintf(outgrfp, "Sfactor=%ld;\n",
01258                 egraph->nsym * egraph->esym * egraph->fsign);
01259     } else {
01260         fprintf(outgrfp, "Sfactor=%ld;\n", egraph->nsym * egraph->esym);
01261     }
01262     fprintf(outgrfp, "%% ExtPerm=%ld; NSym=%ld; ESym=%ld; NSym1=%ld;"
01263                      " Multp=%ld;\n",
01264             egraph->extperm, egraph->nsym, egraph->esym, egraph->nsym1,
01265             egraph->multp);
01266
01267     fprintf(outgrfp, "Vertex=%d;\n", egraph->nNodes - egraph->nExtern);
01268
01269     for (nd = 0; nd < egraph->nNodes; nd++) {
01270         fprintf(outgrfp, "%4d", nd);
01271         if (mdl != NULL && egraph->assigned && !egraph->isExternal(nd)) {
01272             intr = egraph->nodes[nd]->intrct;
01273             loop = mdl->interacts[intr]->loop;
01274             popt = False;
01275
01276             // not tree vertex
01277             if (loop > 0) {
01278                 fprintf(outgrfp, "[loop=%d", loop);
01279                 popt = True;
```

```
01280             }
01281
01282             // multiple coupling constants
01283             if (mdl->ncouple > 1) {
01284                 if (popt) {
01285                     fprintf(outgrfp, ";order={");
01286                 } else {
01287                     fprintf(outgrfp, "[order={");
01288                     popt = True;
01289                 }
01290                 for (j = 0; j < mdl->interacts[intr]->nclist; j++) {
01291                     if (j != 0) {
01292                         fprintf(outgrfp, ",");
01293                     }
01294                     fprintf(outgrfp, "%d", mdl->interacts[intr]->clist[j]);
01295                 }
01296                 fprintf(outgrfp, "}");
01297             }
01298             if (popt) {
01299                 fprintf(outgrfp, "]");
01300             }
01301         } else if (!egraph->isExternal(nd)) {
01302             loop = egraph->nodes[nd]->extloop;
01303             // not tree vertex
01304             if (loop > 0) {
01305                 fprintf(outgrfp, "[loop=%d]", loop);
01306             }
01307         }
01308         fprintf(outgrfp, "={");
01309
01310         // list of legs
01311         for (lg = 0; lg < egraph->nodes[nd]->deg; lg++) {
01312             if (lg != 0) {
01313                 fprintf(outgrfp, ", ");
01314             }
01315             ed = V2Iedge(egraph->nodes[nd]->edges[lg]);
01316          // fprintf(outgrfp, "%4d", egraph->nodes[nd]->edges[lg]);
01317             fprintf(outgrfp, "%4d", abs(egraph->nodes[nd]->edges[lg]));
01318             ptcl = egraph->edges[ed]->ptcl;
01319             if (mdl != NULL && ptcl != 0 && egraph->assigned) {
01320                 if (egraph->nodes[nd]->edges[lg] < 0) {
01321                     ptcl = mdl->normalParticle(-ptcl);
01322                 } else {
01323                     ptcl = mdl->normalParticle(ptcl);
01324                 }
01325                 fprintf(outgrfp, "[%s]", mdl->particleName(ptcl));
01326             } else {
01327                 fprintf(outgrfp, "[undef]");
01328             }
01329         }
01330         fprintf(outgrfp, "};\n");
01331     }
01332     // print Fermion line as comment line
01333     if (egraph->nFlines >= 0) {
01334         fprintf(outgrfp, "%%  FLines=%d; FSign=%d; sId=%ld;\n",
01335                 egraph->nFlines, egraph->fsign, egraph->sId);
01336     }
01337     for (j = 0; j < egraph->nFlines; j++) {
01338         fl = egraph->flines[j];
01339         fprintf(outgrfp, "%% %4d", j);
01340         if (fl->ftype == FL_Open) {
01341             fprintf(outgrfp, "[FOpen]=[");
01342         } else if (fl->ftype == FL_Closed) {
01343             fprintf(outgrfp, "[FLoop]=[");
01344         } else {
01345             fprintf(outgrfp, " ?%d", fl->ftype);
01346         }
01347         for (k = 0; k < fl->nlist; k++) {
01348             if (k != 0) {
01349                 fprintf(outgrfp, ", ");
01350             }
01351             fprintf(outgrfp, "%d", fl->elist[k]);
01352         }
01353         fprintf(outgrfp, "];\n");
01354     }
01355     fprintf(outgrfp, "Vend;\n");
01356     fprintf(outgrfp, "Gend;\n");
01357 }
01358
01359 //------------------------------------------------------------------
01360 #ifdef NEWOUTPY
01361 void Output::outEGraphP(EGraph *eg)
01362 {
01363     if (outgrpp == NULL) {
01364         return;
01365     }
01366
```

```
01367     if (eg->assigned) {
01368         grcc_fprintf(GRCC_Stdout, "outEGraphP:egraph:egraph[%ld]\n", eg->mId-1);
01369         eg->printPy(outgrpp, eg->mId);
01370         return;
01371     } else {
01372         grcc_fprintf(GRCC_Stdout, "outEGraphP:mgraph:egraph[%ld]\n", eg->mId-1);
01373         eg->mgraph->printPy(outgrpp, eg->mId);
01374         return;
01375     }
01376 }
01377 #else
01378 void Output::outEGraphP(EGraph *eg)
01379 {
01380     int      j, nd, lg, ed, k;
01381     ENode   *node;
01382     static char undef[] = "Undef";
01383     char    *s;
01384     EFLine  *fl;
01385 
01386     if (outgrpp == NULL) {
01387         return;
01388     }
01389 
01390     if (eg->assigned) {
01391         fprintf(outgrpp, "egraph[%ld] = {\n", eg->sId-1);
01392     } else {
01393         fprintf(outgrpp, "egraph[%ld] = {\n", eg->mId-1);
01394     }
01395 
01396     // process
01397     if (proc != NULL) {
01398         fprintf(outgrpp, "  \"Process\": {\n");
01399         proc->outProcP(outgrpp);
01400         fprintf(outgrpp, "  },\n");
01401     }
01402 
01403     // graph
01404     fprintf(outgrpp, "  \"GId\": [%ld, %ld, %ld],\n",
01405             eg->mId, eg->aId, eg->sId);
01406     fprintf(outgrpp, "  \"GParam\": {\"Assigned\":%d, "
01407             "\"NNodes\":%d, \"NEdges\":%d, \"NFLines\":%d},\n",
01408             eg->assigned, eg->nNodes, eg->nEdges, eg->nFlines);
01409     fprintf(outgrpp, "  \"Sym\": {\"FSign\":%d, \"NSym\":%ld, "
01410             "\"ESym\":%ld, \"NSym1\":%ld, \"Multp\":%ld},\n",
01411             eg->fsign, eg->nsym, eg->esym, eg->nsym1, eg->multp);
01412 
01413     // nodes
01414     fprintf(outgrpp, "  \"Nodes\": [\n");
01415     for (nd = 0; nd < eg->nNodes; nd++) {
01416         node = eg->nodes[nd];
01417         if (model==NULL) {
01418             s = undef;
01419         } else if(!eg->assigned) {
01420             if (eg->isExternal(nd)) {
01421                 ed = Abs(eg->nodes[nd]->edges[0])-1;
01422                 s = model->particleName(eg->edges[ed]->ptcl);
01423             } else {
01424                 s = undef;
01425             }
01426         } else if (eg->isExternal(nd)) {
01427             s = model->particleName(node->intrct);
01428         } else {
01429             s = model->interacts[node->intrct]->name;
01430         }
01431         fprintf(outgrpp, "    [%d, \"%s\", %d, [",
01432                 nd, s, node->extloop);
01433         for (lg = 0; lg < eg->nodes[nd]->deg; lg++) {
01434             if (lg != 0) {
01435                 fprintf(outgrpp, ", ");
01436             }
01437             fprintf(outgrpp, "%d", eg->nodes[nd]->edges[lg]);
01438         }
01439         fprintf(outgrpp, "]],\n");
01440     }
01441     fprintf(outgrpp, "  ],\n");
01442 
01443     // edges
01444     fprintf(outgrpp, "  \"Edges\": [\n");
01445     for (ed = 0; ed < eg->nEdges; ed++) {
01446         if (model != NULL) {
01447             s = model->particleName(eg->edges[ed]->ptcl);
01448             fprintf(outgrpp, "    [%d, \"%s\", [%d, %d]],\n",
01449                     ed+1, s, eg->edges[ed]->nodes[0],
01450                     eg->edges[ed]->nodes[1]);
01451         } else {
01452             fprintf(outgrpp, "    [%d, \"Undef\", [%d, %d]],\n",
01453                     ed+1, eg->edges[ed]->nodes[0],
```

```
01454                     eg->edges[ed]->nodes[1]);
01455         }
01456     }
01457
01458     fprintf(outgrpp, "  ],\n");
01459
01460     if (eg->nFlines >= 0) {
01461         // print Fermion line as comment line
01462         fprintf(outgrpp, "  \"FLines\": [\n");
01463         for (j = 0; j < eg->nFlines; j++) {
01464             fl = eg->flines[j];
01465             fprintf(outgrpp, "    [%d, ", j);
01466             if (fl->ftype == FL_Open) {
01467                 fprintf(outgrpp, "\"FOpen\", ");
01468             } else if (fl->ftype == FL_Closed) {
01469                 fprintf(outgrpp, "\"FLoop\", ");
01470             } else {
01471                 fprintf(outgrpp, "\"Undef%d\", ", fl->ftype);
01472             }
01473             fprintf(outgrpp, "[");
01474             for (k = 0; k < fl->nlist; k++) {
01475                 if (k != 0) {
01476                     fprintf(outgrpp, ", ");
01477                 }
01478                 fprintf(outgrpp, "%d", fl->elist[k]);
01479             }
01480             fprintf(outgrpp, "]],\n");
01481         }
01482         fprintf(outgrpp, "  ]\n");
01483     }
01484
01485     // end of a graph
01486     fprintf(outgrpp, "};\n");
01487 }
01488 #endif
01489
01490 //------------------------------------------------------------------
01491 void Output::outModelF(void)
01492 {
01493     char        *mdlfn;
01494     FILE        *mdlfp;
01495     Particle    *pt;
01496     Interaction *vt;
01497     int          j, k, l, ln;
01498     const char        *ptn;
01499
01500     if (model == NULL) {
01501         if (prlevel > 0) {
01502             grcc_fprintf(GRCC_Stderr, "*** Output::outModel : model is not defined\n");
01503         }
01504         return;
01505     }
01506     ln = strlen(model->name)+5;
01507     mdlfn = new char[ln];
01508     snprintf(mdlfn, ln, "%s.mdl", model->name);
01509
01510     if ((mdlfp = fopen(mdlfn, "w")) == NULL) {
01511         if (prlevel > 0) {
01512             grcc_fprintf(GRCC_Stderr, "*** Output::outModel : cannot open \"%s\"\n",
01513                     mdlfn);
01514         }
01515         return;
01516     }
01517
01518     for (l = 0; l < 60; l++) { putc('%', mdlfp); } putc('\n', mdlfp);
01519     fprintf(mdlfp, "%% File \"%s\"\n", mdlfn);
01520     fprintf(mdlfp, "%% Generated by graph generater\n");
01521     fprintf(mdlfp, " Version={2,2,0};\n");
01522
01523     // copupling constants
01524     fprintf(mdlfp, " Order={");
01525     for (j = 0; j < model->ncouple; j++) {
01526         if (j != 0) {
01527             fprintf(mdlfp, ", ");
01528         }
01529         fprintf(mdlfp, "%s", model->cnlist[j]);
01530     }
01531     fprintf(mdlfp, "};\n");
01532
01533     for (l = 0; l < 60; l++) { putc('%', mdlfp); } putc('\n', mdlfp);
01534     fprintf(mdlfp, "%% particles\n");
01535     for (l = 0; l < 60; l++) { putc('%', mdlfp); } putc('\n', mdlfp);
01536     for (j = 1; j < model->nParticles; j++) {
01537         pt = model->particles[j];
01538         fprintf(mdlfp, " Particle=%s; Antiparticle=%s;\n",
01539                 pt->name, pt->aname);
01540         ptn = pt->typeGName();
```

```
01541         fprintf(mdlfp, " PType=%s; Charge=0; Color=1; Mass=0; Width=0;\n", ptn);
01542         fprintf(mdlfp, " PCode=%d; Massless; \n", pt->pcode);
01543         fprintf(mdlfp, " Pend;\n");
01544         fprintf(mdlfp, "%%\n");
01545     }
01546     for (l = 0; l < 60; l++) { putc('%', mdlfp); } putc('\n', mdlfp);
01547     fprintf(mdlfp, "%% interactions\n");
01548     for (l = 0; l < 60; l++) { putc('%', mdlfp); } putc('\n', mdlfp);
01549     for (j = 0; j < model->nInteracts; j++) {
01550         vt = model->interacts[j];
01551         fprintf(mdlfp, " Vertex={");
01552         for (k = 0; k < vt->nplist; k++) {
01553             if (k != 0) {
01554                 fprintf(mdlfp, ", ");
01555             }
01556             fprintf(mdlfp, "%s", model->particleName(vt->plist[k]));
01557         }
01558         fprintf(mdlfp, "}; ");
01559         for (k = 0; k < model->ncouple; k++) {
01560             fprintf(mdlfp, "%s=%d; ", model->cnlist[k], vt->clist[k]);
01561         }
01562         fprintf(mdlfp, "FName=%s; Vend;\n", vt->name);
01563     }
01564     for (l = 0; l < 60; l++) { putc('%', mdlfp); } putc('\n', mdlfp);
01565     fprintf(mdlfp, " Mend;\n");
01566     for (l = 0; l < 60; l++) { putc('%', mdlfp); } putc('\n', mdlfp);
01567     fclose(mdlfp);
01568     delete[] mdlfn;
01569 }
01570
01571 //--------------------------------------------------------------
01572 void Output::outModelP(void)
01573 {
01574     char        *mdlfn;
01575     FILE        *mdlfp;
01576     Particle    *pt;
01577     Interaction *vt;
01578     int          j, k, l, ln;
01579     const char        *ptn;
01580
01581     if (model == NULL) {
01582         if (prlevel > 0) {
01583             grcc_fprintf(GRCC_Stderr, "*** Output::outModel : ");
01584             grcc_fprintf(GRCC_Stderr, "model is not defined\n");
01585         }
01586         return;
01587     }
01588     ln = strlen(model->name)+5;
01589     mdlfn = new char[ln];
01590     snprintf(mdlfn, ln, "%s.mdp", model->name);
01591
01592     if ((mdlfp = fopen(mdlfn, "w")) == NULL) {
01593         if (prlevel > 0) {
01594             grcc_fprintf(GRCC_Stderr, "*** Output::outModel : cannot open \"%s\"\n",
01595                     mdlfn);
01596         }
01597         return;
01598     }
01599
01600     for (l = 0; l < 60; l++) { putc('#', mdlfp); } putc('\n', mdlfp);
01601     fprintf(mdlfp, "# File \"%s\"\n", mdlfn);
01602     fprintf(mdlfp, "# Generated by graph generater grcc\n");
01603
01604     // copupling constants
01605     fprintf(mdlfp, "Model={\n");
01606     fprintf(mdlfp, "  \"Model\": [");
01607     fprintf(mdlfp, "\"%s\", %d, [",
01608             model->name, model->ncouple);
01609     for (j = 0; j < model->ncouple; j++) {
01610         if (j != 0) {
01611             fprintf(mdlfp, ", ");
01612         }
01613         fprintf(mdlfp, "\"%s\"", model->cnlist[j]);
01614     }
01615     fprintf(mdlfp, "], ");
01616     if (model->defpart == GRCC_DEFBYCODE) {
01617         fprintf(mdlfp, "\"ByCode\"");
01618     } else {
01619         fprintf(mdlfp, "\"ByName\"");
01620     }
01621     fprintf(mdlfp, "],\n");
01622
01623     fprintf(mdlfp, "  \"Particles\": [\n");
01624     for (j = 1; j < model->nParticles; j++) {
01625         pt = model->particles[j];
01626         ptn = pt->typeName();
01627         fprintf(mdlfp, "    [\"%s\", %d, \"%s\", %d, \"%s\", %d],\n",
```

```
01628                 pt->name, pt->pcode, pt->aname, pt->acode,
01629                 ptn, pt->extonly);
01630     }
01631     fprintf(mdlfp, "  ],\n");
01632     fprintf(mdlfp, "  \"Interactions\": [\n");
01633     for (j = 0; j < model->nInteracts; j++) {
01634         vt = model->interacts[j];
01635         fprintf(mdlfp, "    [\"%s\", %d, %d, [", vt->name, vt->id, vt->nplist);
01636         for (k = 0; k < vt->nplist; k++) {
01637             if (k != 0) {
01638                 fprintf(mdlfp, ", ");
01639             }
01640             if (model->defpart == GRCC_DEFBYCODE) {
01641                 fprintf(mdlfp, "%d", vt->plist[k]);
01642             } else {
01643                 fprintf(mdlfp, "\"%s\"", model->particleName(vt->plist[k]));
01644             }
01645         }
01646         fprintf(mdlfp, "], [");
01647         for (k = 0; k < model->ncouple; k++) {
01648             if (k != 0) {
01649                 fprintf(mdlfp, ",");
01650             }
01651             fprintf(mdlfp, "%d", vt->clist[k]);
01652         }
01653         fprintf(mdlfp, "]],\n");
01654     }
01655     fprintf(mdlfp, "  ],\n");
01656     fprintf(mdlfp, "};\n");
01657     fclose(mdlfp);
01658     delete[] mdlfn;
01659 }
01660
01661 //************************************************************
01662 // model.cc
01663
01664 //============================================================
01665
01666 static const char *ptypenames[]  = GRCC_PT_Table;
01667 static const char *pgtypenames[] = GRCC_PT_GTable;
01668
01669 //============================================================
01670 // class Particle
01671 //------------------------------------------------------------
01672 Particle::Particle(Model *modl, int pid, PInput *pinp)
01673 {
01674     static char buff[100];
01675     static int  nbuff=100;
01676     int j;
01677
01678     mdl     = modl;                  // the model
01679     id      = pid;                   // id of this particle
01680     if (pinp->name == NULL || strlen(pinp->name) < 1) {
01681         if (pinp->pcode != 0) {
01682             snprintf(buff, nbuff, "pa%d", pinp->pcode);
01683             name = strdup(buff);
01684         } else {
01685             snprintf(buff, nbuff, "pi%d", pid);
01686             name = strdup(buff);
01687         }
01688     } else {
01689         name = strdup(pinp->name);     // the name of the particle
01690     }
01691     if (pinp->aname == NULL) {
01692         if (pinp->acode != 0) {
01693             snprintf(buff, nbuff, "ap%d", Abs(pinp->acode));
01694             aname = strdup(buff);
01695         } else {
01696             snprintf(buff, nbuff, "ai%d", pid);
01697             aname = strdup(buff);
01698         }
01699     } else {
01700         aname   = strdup(pinp->aname); // the name of the anti-particle
01701     }
01702     pcode   = pinp->pcode;
01703     acode   = pinp->acode;
01704     if (Abs(mdl->defpart) == GRCC_DEFBYCODE) {
01705         neutral = (pcode == acode);
01706     } else if (Abs(mdl->defpart) == GRCC_DEFBYNAME) {
01707         neutral = ((strcmp(name, aname)==0) ? True : False);
01708     }
01709
01710     // convert ptype string to its code.
01711     if (Abs(mdl->defpart) == GRCC_DEFBYCODE) {
01712         if (pinp->ptypec < GRCC_PT_Undef || pinp->ptypec > GRCC_PT_Size) {
01713             if (prlevel > 0) {
01714                 grcc_fprintf(GRCC_Stderr, "*** particle type is not defined: %d\n",
```

```
01715                           pinp->ptypec);
01716               }
01717               erEnd("particle type is not defined (illegal code)");
01718           }
01719           ptype = pinp->ptypec;
01720       } else if (Abs(mdl->defpart) == GRCC_DEFBYNAME) {
01721           if (pinp->ptypen == NULL) {
01722               erEnd("particle type is not defined (NULL)");
01723           }
01724           ptype = -1;
01725           for (j = 0; j < GRCC_PT_Size; j++) {
01726               if (strcmp(pinp->ptypen, ptypenames[j]) == 0) {
01727                   ptype = j;
01728                   break;
01729               }
01730           }
01731           if (ptype < 0) {
01732               if (prlevel > 0) {
01733                   grcc_fprintf(GRCC_Stderr, "*** particle type \"%s\" is not defined\n",
01734                           pinp->ptypen);
01735               }
01736               erEnd("particle type is not defined (name)");
01737           }
01738       }
01739       if (ptype == GRCC_PT_Undef && ptype != 0) {
01740           erEnd("illegal ptype");
01741       }
01742
01743       extonly = pinp->extonly;
01744       cmindeg = 0;
01745       cmaxdeg = 0;
01746   }
01747
01748   //-------------------------------------------------------------
01749   Particle::~Particle(void)
01750   {
01751       free(aname);
01752       free(name);
01753   }
01754
01755   //-------------------------------------------------------------
01756   char *Particle::particleName(int p)
01757   {
01758       if (p < 0) {
01759           return aname;
01760       } else {
01761           return name;
01762       }
01763   }
01764
01765   //-------------------------------------------------------------
01766   int Particle::particleCode(int p)
01767   {
01768       if (p < 0) {
01769           return acode;
01770       } else {
01771           return pcode;
01772       }
01773   }
01774
01775   //-------------------------------------------------------------
01776   int Particle::isNeutral(void)
01777   {
01778       return neutral;
01779   }
01780
01781   //-------------------------------------------------------------
01782   const char *Particle::typeName(void)
01783   {
01784       return ptypenames[ptype];
01785   }
01786
01787   //-------------------------------------------------------------
01788   const char *Particle::typeGName(void)
01789   {
01790       return pgtypenames[ptype];
01791   }
01792
01793   //-------------------------------------------------------------
01794   char *Particle::aparticle(void)
01795   {
01796       static char prtcl[] = "Particle";
01797
01798       if (isNeutral()) {
01799           return prtcl;
01800       } else {
01801           return aname;
```

```
01802     }
01803 }
01804
01805 //------------------------------------------------------------------
01806 void Particle::prParticle(void)
01807 {
01808     if (isNeutral()) {
01809         grcc_fprintf(GRCC_Stdout, "pid=%d, name=\"%s\", real_field, ", id, name);
01810     } else {
01811         grcc_fprintf(GRCC_Stdout, "pid=%d, name=\"%s\", aname=\"%s\", ", id, name, aname);
01812     }
01813     grcc_fprintf(GRCC_Stdout, "ptype=%s, pcode=%d, acode=%d, extonly=%d, cdeg=(%d,%d)\n",
01814           ptypenames[ptype], pcode, acode, extonly, cmindeg, cmaxdeg);
01815 }
01816
01817 //==================================================================
01818 // class Interaction
01819 //------------------------------------------------------------------
01820 Interaction::Interaction(Model *modl, int iid, const char *nam, int icd, int *cpl, int nlgs, int
      *plst, int csm, int lp)
01821 {
01822     static char buff[100];
01823     static int  nbuff=100;
01824     static Bool prmsg = True;
01825     Particle   *p;
01826     int         j, ndir, sdir, jdir, nmaj, jmaj, ngho, sgho, jgho, ptcl;
01827     Bool        ok;
01828
01829     mdl    = modl;          // the model
01830     id     = iid;           // id of this interaction
01831     icode  = icd;           // id of this interaction
01832     csum   = csm;           // the total orders of coupling constants
01833     nlegs  = nlgs;          // the size of plist[]
01834     loop   = lp;            // the number of loops
01835
01836     if (nam == NULL || strlen(nam) < 1) {
01837         snprintf(buff, nbuff, "vt%d", icd);
01838         name = strdup(buff);
01839     } else {
01840         name = strdup(nam);  // the name of the interaction
01841     }
01842     nclist = mdl->ncouple;
01843     clist  = new int[nclist];
01844     icode  = icd;
01845     for (j = 0; j < mdl->ncouple; j++) {
01846         clist[j] = cpl[j];
01847     }
01848     plist  = new int[nlegs];
01849     nplist = nlegs;
01850     for (j = 0; j < nlegs; j++) {
01851         plist[j] = plst[j];
01852     }
01853     nslist = 0;
01854     slist  = NULL;
01855
01856     // check fermion number conservation
01857     ndir = 0;
01858     sdir = 0;
01859     jdir = 0;
01860     nmaj = 0;
01861     jmaj = 0;
01862     ngho = 0;
01863     sgho = 0;
01864     jgho = 0;
01865     ok = True;
01866     for (j = 0; j < nplist; j++) {
01867         if (plist[j] > 0) {
01868             p = mdl->particles[plist[j]];
01869             ptcl = 1;
01870         } else {
01871             p = mdl->particles[-plist[j]];
01872             ptcl = -1;
01873         }
01874
01875         if (p->ptype == GRCC_PT_Dirac) {
01876             ndir++;
01877             if (ptcl > 0) {
01878                 sdir++;
01879             } else {
01880                 sdir--;
01881             }
01882             if (Abs(sdir) == 1) {
01883                 jdir = j;
01884             } else if ((Abs(sdir) == 0 && j != jdir + 1) || Abs(sdir) > 1) {
01885                 grcc_fprintf(GRCC_Stderr, "*** Interaction: Dirac and anti-Dirac should "
01886                       "arranged in pairs in an interaction.\n");
01887                 ok = False;
```

```
01888             }
01889         } else if (p->ptype == GRCC_PT_Ghost) {
01890             ngho++;
01891             if (ptcl > 0) {
01892                 sgho++;
01893             } else {
01894                 sgho--;
01895             }
01896             if (Abs(sgho) == 1) {
01897                 jgho = j;
01898             } else if ((Abs(sgho) == 0 && j != jgho + 1) || Abs(sgho) > 1) {
01899                 grcc_fprintf(GRCC_Stderr, "*** Interaction: Ghost and anti-Ghost should "
01900                         "arranged in pairs in an interaction.\n");
01901                 ok = False;
01902             }
01903         } else if (p->ptype == GRCC_PT_Majorana) {
01904             nmaj++;
01905             if (nmaj % 2 == 1) {
01906                 jmaj = j;
01907             } else if (j != jmaj + 1) {
01908                 grcc_fprintf(GRCC_Stderr, "*** Interaction: two Majoranas should "
01909                         "arranged in pairs in an interaction.\n");
01910                 ok = False;
01911             }
01912         }
01913     }
01914     if (sdir != 0) {
01915         grcc_fprintf(GRCC_Stderr, "*** Interaction: Dirac number is not conserved\n");
01916         ok = False;
01917     }
01918     if (sgho != 0) {
01919         grcc_fprintf(GRCC_Stderr, "*** Interaction: Ghost number is not conserved\n");
01920         ok = False;
01921     }
01922     if (nmaj % 2 != 0) {
01923         grcc_fprintf(GRCC_Stderr, "*** Interaction: odd number of Majorana particles\n");
01924         ok = False;
01925     }
01926     if (ndir + nmaj + ngho > 2 && prmsg) {
01927         grcc_fprintf(GRCC_Stderr, "+++ Interaction: more than 2 "
01928                             "Dirac/Majorana/Ghost "
01929                             "particles are interacting.\n");
01930         grcc_fprintf(GRCC_Stderr, "    Interaction has %d Dirac, %d Ghost "
01931                             "and %d Majorana particles.\n",
01932                             ndir, nmaj, ngho);
01933         grcc_fprintf(GRCC_Stderr, "    Sign factors related to fermions may "
01934                             "be inconsistent with your calculation method.\n");
01935         prmsg = False;
01936     }
01937     if (!ok) {
01938         prInteraction();
01939         erEnd("illegal Dirac/Majorana/Ghost interaction");
01940     }
01941 }
01942
01943 //-------------------------------------------------------------
01944 Interaction::~Interaction(void)
01945 {
01946     if (slist != NULL) {
01947         delete[] slist;
01948     }
01949     if (plist != NULL) {
01950         delete[] plist;
01951     }
01952     if (clist != NULL) {
01953         delete[] clist;
01954     }
01955     free(name);
01956 }
01957
01958 //-------------------------------------------------------------
01959 void Interaction::prInteraction(void)
01960 {
01961     int j;
01962
01963     grcc_fprintf(GRCC_Stdout, "vid=%d, icode=%d, name=\"%s\", loop=%d, csum=%d, cpl=",
01964           id, icode, name, loop, csum);
01965     prIntArray(mdl->ncouple, clist, ", legs=[");
01966     for (j = 0; j < nplist; j++) {
01967         if (j != 0) {
01968             grcc_fprintf(GRCC_Stdout, ", ");
01969         }
01970         grcc_fprintf(GRCC_Stdout, "%s", mdl->particleName(plist[j]));
01971     }
01972     grcc_fprintf(GRCC_Stdout, "];\n");
01973
01974 }
```

```
01975
01976 //============================================================
01977 // class Model
01978 //------------------------------------------------------------
01979 Model::Model(MInput *minp)
01980 {
01981     static PInput pundef = {"undef", "undef", 0, 0, "undef", 0, 0};
01982     int j;
01983
01984     if (minp->name == NULL || strlen(minp->name) < 1) {
01985         erEnd("model should have a name");
01986     }
01987     name    = strdup(minp->name);
01988     ncouple = minp->ncouple;
01989     if (ncouple >= GRCC_MAXNCPLG) {
01990         erEnd("too many coupling constants in the model (GRCC_MAXNCPLG)");
01991     }
01992     // cnlist = minp->cnamlist;
01993     cnlist = new char*[ncouple];
01994     for (j = 0; j < ncouple; j++) {
01995         cnlist[j] = strdup(minp->cnamlist[j]);
01996     }
01997
01998     nParticles = 0;
01999     particles  = new Particle*[GRCC_MAXMPARTICLES];
02000     for (j = 0; j < GRCC_MAXMPARTICLES; j++) {
02001         particles[j] = NULL;
02002     }
02003     pdef = False;
02004
02005     nInteracts = 0;
02006     interacts  = new Interaction*[GRCC_MAXMINTERACT];
02007     for (j = 0; j < GRCC_MAXMINTERACT; j++) {
02008         interacts[j] = NULL;
02009     }
02010     vdef       = False;
02011
02012     if (particles == NULL || interacts == NULL) {
02013         erEnd("memory allocation failed");
02014     }
02015
02016 #ifdef UNIQINTR
02017     if (minp->defpart != GRCC_DEFBYNAME
02018         && minp->defpart != GRCC_DEFBYCODE) {
02019         erEnd("illegal value of defpart");
02020     }
02021 #else
02022     if (Abs(minp->defpart) != GRCC_DEFBYNAME
02023         && Abs(minp->defpart) != GRCC_DEFBYCODE) {
02024         erEnd("illegal value of defpart");
02025     }
02026 #endif
02027
02028     defpart = minp->defpart;
02029
02030     maxnlegs   = 0;
02031     maxcpl     = 0;
02032     maxloop    = 0;
02033     ncplgcp    = 0;
02034     cplgvl     = NULL;
02035
02036     addParticle(&pundef);
02037 }
02038
02039 //------------------------------------------------------------
02040 Model::~Model(void)
02041 {
02042     int j;
02043
02044     if (ncplgcp > 0) {
02045         for (j = ncplgcp-1; j >= 0; j--) {
02046             cplgvl[j] = delintdup(cplgvl[j]);
02047         }
02048         delete[] cplgvl;
02049         cplgnvl = delintdup(cplgnvl);
02050         cplglg  = delintdup(cplglg);
02051         cplgcp  = delintdup(cplgcp);
02052     }
02053
02054     for (j = GRCC_MAXMINTERACT-1; j >= 0; j--) {
02055         if (interacts[j] != NULL) {
02056             interacts[j]->slist = delintdup(interacts[j]->slist);
02057             delete interacts[j];
02058             interacts[j] = NULL;
02059         }
02060     }
02061     delete[] interacts;
```

```
02062     interacts = NULL;
02063
02064     for (j = GRCC_MAXMPARTICLES-1; j >= 0; j--) {
02065         if (particles[j] != NULL) {
02066             delete particles[j];
02067             particles[j] = NULL;
02068         }
02069     }
02070     delete[] particles;
02071     particles = NULL;
02072
02073     for (j=ncouple-1; j>=0; j--) {
02074         free(cnlist[j]);
02075     }
02076     delete[] cnlist;
02077
02078     free(name);
02079 }
02080
02081 //-------------------------------------------------------------
02082 void Model::prModel(void)
02083 {
02084     static char hdr[] = "#================================================\n";
02085     static char hd1[] = "#------------------------------------------------\n";
02086     int j;
02087
02088     grcc_fprintf(GRCC_Stdout, "%s", hdr);
02089     grcc_fprintf(GRCC_Stdout, "Model=\"%s\", ", name);
02090     grcc_fprintf(GRCC_Stdout, "coupling=[");
02091     for (j = 0; j < ncouple; j++) {
02092         if (j != 0) {
02093             grcc_fprintf(GRCC_Stdout, ", ");
02094         }
02095         grcc_fprintf(GRCC_Stdout, "\"%s\"", cnlist[j]);
02096     }
02097     grcc_fprintf(GRCC_Stdout, "];\n");
02098     grcc_fprintf(GRCC_Stdout, "%s", hd1);
02099     grcc_fprintf(GRCC_Stdout, "Particles=%d;\n", nParticles);
02100     for (j = 1; j < nParticles; j++) {
02101         particles[j]->prParticle();
02102     }
02103     grcc_fprintf(GRCC_Stdout, "EndParticle;\n");
02104     grcc_fprintf(GRCC_Stdout, "allPart (%d) = [", nallPart);
02105     for (j = 0; j < nallPart; j++) {
02106         if (j != 0) {
02107             grcc_fprintf(GRCC_Stdout, ", ");
02108         }
02109         grcc_fprintf(GRCC_Stdout, "%d", allPart[j]);
02110     }
02111     grcc_fprintf(GRCC_Stdout, "]\n");
02112
02113     grcc_fprintf(GRCC_Stdout, "%s", hd1);
02114     grcc_fprintf(GRCC_Stdout, "Interactions=%d;\n", nInteracts);
02115     for (j = 0; j < nInteracts; j++) {
02116         interacts[j]->prInteraction();
02117     }
02118     grcc_fprintf(GRCC_Stdout, "EndInteraction;\n");
02119     grcc_fprintf(GRCC_Stdout, "%s", hd1);
02120     grcc_fprintf(GRCC_Stdout, "InteractionTable;\n");
02121     grcc_fprintf(GRCC_Stdout, "#  %d class in (total coupling, degree)\n", ncplgcp);
02122     for (j = 0; j < ncplgcp; j++) {
02123         grcc_fprintf(GRCC_Stdout, "  class=%d: cp=%d, lg=%d, vl=", j, cplgcp[j], cplglg[j]);
02124         prIntArray(cplgnvl[j], cplgvl[j], "\n");
02125     }
02126     grcc_fprintf(GRCC_Stdout, "EndInteractionTable;\n");
02127     grcc_fprintf(GRCC_Stdout, "%s", hdr);
02128 }
02129
02130 //-------------------------------------------------------------
02131 void Model::addParticle(PInput *pinp)
02132 {
02133     int  nid, aid, pid, pcd, acd;
02134
02135     if (nParticles >= GRCC_MAXMPARTICLES) {
02136         erEnd("particle table overflow (GRCC_MAXMPARTICLES)");
02137     }
02138
02139     // uniqueness check
02140     if (Abs(defpart) == GRCC_DEFBYCODE) {
02141         pcd = findParticleCode(pinp->pcode);
02142         acd = findParticleCode(pinp->acode);
02143         if (pcd > 0 || acd > 0) {
02144             if (prlevel > 0) {
02145                 grcc_fprintf(GRCC_Stderr, "*** particle code [%d, %d] ",
02146                         pinp->pcode, pinp->acode);
02147                 grcc_fprintf(GRCC_Stderr, "is already used.\n");
02148             }
```

```
02149             erEnd("particle code is already defined");
02150         }
02151     } else if (Abs(defpart) == GRCC_DEFBYNAME) {
02152         if (pinp->name == NULL || pinp->aname == NULL ||
02153             strlen(pinp->name) < 1 || strlen(pinp->aname) < 1) {
02154             erEnd("no name of particle or anti-particle.");
02155         }
02156         nid = findParticleName(pinp->name);
02157         aid = findParticleName(pinp->aname);
02158         if (nid > 0 || aid > 0) {
02159             if (prlevel > 0) {
02160                 grcc_fprintf(GRCC_Stderr, "*** particle name [%s, %s] ",
02161                         pinp->name, pinp->aname);
02162                 grcc_fprintf(GRCC_Stderr, "is already used.\n");
02163             }
02164             erEnd("particle name is already defined.");
02165         }
02166     }
02167
02168     pid = nParticles++;
02169     particles[pid] = new Particle(this, pid, pinp);
02170 }
02171
02172 //-------------------------------------------------------------
02173 void Model::addParticleEnd(void)
02174 {
02175     int p, ap;
02176
02177     pdef = True;
02178
02179 #ifdef OPTEXTONLY
02180     nallPart = 0;
02181     for (p = nParticles-1; p >= 1; p--) {
02182         if (!particles[p]->extonly) {
02183             ap = antiParticle(p);
02184             if (ap != p) {
02185                 allPart[nallPart++] = ap;
02186             }
02187         }
02188     }
02189     for (p = 1; p < nParticles; p++) {
02190         if (!particles[p]->extonly) {
02191             allPart[nallPart++] = p;
02192         }
02193     }
02194     sorti(nallPart, allPart);
02195 #else
02196     nallPart = 0;
02197     for (p = nParticles-1; p >= 1; p--) {
02198         ap = antiParticle(p);
02199         if (ap != p) {
02200             allPart[nallPart++] = ap;
02201         }
02202     }
02203     for (p = 1; p < nParticles; p++) {
02204         allPart[nallPart++] = p;
02205     }
02206     sorti(nallPart, allPart);
02207 #endif
02208 }
02209
02210 //-------------------------------------------------------------
02211 void Model::addInteraction(IInput *iinp)
02212 {
02213     int vid, nlegs, j, k, c;
02214     int csum, lp2, lp;
02215     int plist[GRCC_MAXLEGS];
02216
02217     if (!pdef) {
02218         erEnd("call 'addParticleEnd' before 'addInteraction'");
02219     }
02220     if (nInteracts >= GRCC_MAXMINTERACT) {
02221         erEnd("too many interactions in the model (GRCC_MAXMINTERACT)");
02222     }
02223     nlegs = iinp->nplistn;
02224     if (nlegs >= GRCC_MAXLEGS) {
02225         erEnd("too many legs in an interaction (GRCC_MAXLEGS)");
02226     }
02227
02228     // check identifier
02229     if (defpart == GRCC_DEFBYCODE) {
02230         if (iinp->icode < 0) {
02231             if (prlevel > 0) {
02232                 grcc_fprintf(GRCC_Stderr, "*** vertex code %d should be positive\n",
02233                         iinp->icode);
02234             }
02235             erEnd("vertex code should be positive");
```

```
02236         } else {
02237             vid = findInteractionCode(iinp->icode);
02238             if (vid >= 0) {
02239                 if (prlevel > 0) {
02240                     grcc_fprintf(GRCC_Stderr, "*** vertex code %d is already used\n",
02241                             iinp->icode);
02242                 }
02243                 erEnd("vertex code is already used");
02244             }
02245         }
02246     } else if (defpart == GRCC_DEFBYNAME) {
02247         if (iinp->name == NULL || strlen(iinp->name) < 1) {
02248             erEnd("no name of vertex\n");
02249         }
02250         vid = findInteractionName(iinp->name);
02251         if (vid >= 0) {
02252             if (prlevel > 0) {
02253                 grcc_fprintf(GRCC_Stderr, "*** vertex name %s is already used\n",
02254                 iinp->name);
02255                 erEnd("vertex name is already used");
02256             }
02257         }
02258     } else {
02259 #ifdef UNIQINTR
02260         // defpart =< 0
02261         erEnd("illegal value of defpart");
02262 #else
02263         // don't care about duplicated name/code of interaction
02264         ;
02265 #endif
02266     }
02267     vid = nInteracts++;
02268
02269     for (j = 0; j < nlegs; j++) {
02270         if (Abs(defpart) == GRCC_DEFBYCODE) {
02271             plist[j] = findParticleCode(iinp->plistc[j]);
02272             if (plist[j] == 0) {
02273                 if (prlevel > 0) {
02274                     grcc_fprintf(GRCC_Stderr, "*** particle code %d ",
02275                             iinp->plistc[j]);
02276                     grcc_fprintf(GRCC_Stderr, "is not defined\n");
02277                 }
02278                 erEnd("particle is not defined (code)");
02279             }
02280         } else if (Abs(defpart) == GRCC_DEFBYNAME) {
02281             plist[j] = findParticleName(iinp->plistn[j]);
02282             if (plist[j] == 0) {
02283                 if (prlevel > 0) {
02284                     grcc_fprintf(GRCC_Stderr, "*** particle name %s ",
02285                             iinp->plistn[j]);
02286                     grcc_fprintf(GRCC_Stderr, "is not defined\n");
02287                 }
02288                 erEnd("particle is not defined (name)");
02289             }
02290         }
02291     }
02292
02293     csum = 0;
02294     for (j = 0; j < ncouple; j++) {
02295         c = iinp->cvallist[j];
02296         if (c < 0) {
02297             if (prlevel > 0) {
02298                 grcc_fprintf(GRCC_Stderr, "*** illegal value of c-constants \n");
02299                 for (k = 0; k < ncouple; k++) {
02300                       grcc_fprintf(GRCC_Stderr, "    %s = %d\n",
02301                               cnlist[k], iinp->cvallist[k]);
02302                 }
02303             }
02304             erEnd("illegal value of c-constants");
02305         }
02306         csum += c;
02307     }
02308     if (csum < 0) {
02309         if (prlevel > 0) {
02310             grcc_fprintf(GRCC_Stderr, "*** illegal total coupling-constants %d\n",
02311                     csum);
02312             for (k = 0; k < ncouple; k++) {
02313                 grcc_fprintf(GRCC_Stderr, "    %s = %d\n",
02314                         cnlist[k], iinp->cvallist[k]);
02315             }
02316         }
02317         erEnd("illegal value of c-constants");
02318     }
02319
02320     // assume : csum = nlegs - 2 + 2*loop
02321     lp2 = csum - nlegs + 2;
02322     if (lp2 % 2 != 0 || lp2 < 0) {
```

```
02323         if (prlevel > 0) {
02324             grcc_fprintf(GRCC_Stderr, "*** illegal coupling const : ");
02325             grcc_fprintf(GRCC_Stderr, "nlegs - 2 + 2*loop: 2*loop = %d\n", lp2);
02326         }
02327         erEnd("illegal value of c-constants");
02328     }
02329     lp = lp2/2;
02330
02331     maxnlegs = Max(maxnlegs, nlegs);
02332     maxloop  = Max(maxloop , lp);
02333     maxcpl   = Max(maxcpl  , csum);
02334
02335     interacts[vid] = new Interaction(this, vid, iinp->name, iinp->icode,
02336                               iinp->cvallist, nlegs, plist, csum, lp);
02337 }
02338
02339 //-------------------------------------------------------------
02340 void Model::addInteractionEnd(void)
02341 {
02342     int lst[GRCC_MAXMINTERACT];
02343     int tcp[GRCC_MAXMINTERACT], tlg[GRCC_MAXMINTERACT];
02344     int tnvl[GRCC_MAXMINTERACT], *tvl[GRCC_MAXMINTERACT];
02345     Interaction *vt;
02346     int cp, lg, j, k, p;
02347
02348     vdef = True;
02349     ncplgcp = 0;
02350     for (cp = 0; cp <= maxcpl; cp++) {
02351
02352         for (lg = 0; lg <= maxnlegs; lg++) {
02353             k = 0;
02354             for (j = 0; j < nInteracts; j++) {
02355                 vt = interacts[j];
02356                 if (vt->csum == cp && vt->nlegs == lg) {
02357                     lst[k++] = j;
02358                 }
02359             }
02360             if (k > 0) {
02361                 tcp[ncplgcp]  = cp;
02362                 tlg[ncplgcp]  = lg;
02363                 tnvl[ncplgcp] = k;
02364                 tvl[ncplgcp]  = intdup(k, lst);
02365                 ncplgcp++;
02366             }
02367         }
02368     }
02369     cplgcp  = intdup(ncplgcp, tcp);
02370     cplglg  = intdup(ncplgcp, tlg);
02371     cplgnvl = intdup(ncplgcp, tnvl);
02372     cplgvl  = new int*[ncplgcp];
02373     for (j = 0; j < ncplgcp; j++) {
02374         if (cplgnvl[j] > 0) {
02375             cplgvl[j] = tvl[j];
02376         } else {
02377             cplgvl[j] = NULL;
02378         }
02379     }
02380
02381     for (j = 0; j < nInteracts; j++) {
02382         vt = interacts[j];
02383         if (vt->slist != NULL) {
02384             erEnd("addInteractionEnd: vt->nslist != NULL");
02385         }
02386         vt->slist  = intdup(vt->nplist, vt->plist);
02387         vt->nslist = toSList(vt->nplist, vt->slist);
02388     }
02389
02390     for (p = 0; p < nParticles; p++) {
02391         particles[p]->cmindeg = 0;
02392         particles[p]->cmaxdeg = 0;
02393     }
02394     for (j = 0; j < nInteracts; j++) {
02395         vt = interacts[j];
02396         for (k = 0; k < vt->nplist; k++) {
02397             p = abs(vt->plist[k]);
02398             if (particles[p]->cmindeg == 0) {
02399                 particles[p]->cmindeg = vt->nlegs;
02400                 particles[p]->cmaxdeg = vt->nlegs;
02401             } else {
02402                 particles[p]->cmindeg = Min(particles[p]->cmindeg, vt->nlegs);
02403                 particles[p]->cmaxdeg = Max(particles[p]->cmaxdeg, vt->nlegs);
02404             }
02405         }
02406     }
02407 }
02408
02409 //-------------------------------------------------------------
```

```
02410 int Model::findParticleName(const char *name)
02411 {
02412     int j;
02413
02414     for (j = 0; j < nParticles; j++) {
02415         if (strcmp(name, particles[j]->name) == 0) {
02416             return j;
02417         }
02418     }
02419     for (j = 0; j < nParticles; j++) {
02420         if (strcmp(name, particles[j]->aname) == 0) {
02421             return -j;
02422         }
02423     }
02424     return 0;
02425 }
02426
02427 //-------------------------------------------------------------
02428 int Model::findParticleCode(int pcd)
02429 {
02430     int j;
02431
02432     for (j = 0; j < nParticles; j++) {
02433         if (pcd == particles[j]->pcode) {
02434             return j;
02435         } else if (pcd == particles[j]->acode) {
02436             return -j;
02437         }
02438     }
02439     return 0;
02440 }
02441
02442 //-------------------------------------------------------------
02443 char *Model::particleName(int p)
02444 {
02445     int q;
02446
02447     if (Abs(p) >= nParticles) {
02448         if (prlevel > 0) {
02449             grcc_fprintf(GRCC_Stderr, "\n*** Model::particleName: ");
02450             grcc_fprintf(GRCC_Stderr, "illegal particle id=%d\n", p);
02451         }
02452         erEnd("Model::particleName: illegal particle");
02453     }
02454     if (p < 0) {
02455         q = - p;
02456     } else {
02457         q = p;
02458     }
02459     return particles[q]->particleName(p);
02460 }
02461
02462 //-------------------------------------------------------------
02463 int Model::particleCode(int p)
02464 {
02465     int q;
02466
02467     if (Abs(p) >= nParticles) {
02468         grcc_fprintf(GRCC_Stderr, "\n*** Model::particleCode: illegal particle id=%d\n",
02469                 p);
02470         erEnd("Model::particleCode: illegal particle");
02471     }
02472     if (p < 0) {
02473         q = - p;
02474     } else {
02475         q = p;
02476     }
02477     return particles[q]->particleCode(p);
02478 }
02479
02480 //-------------------------------------------------------------
02481 void Model::prParticleArray(int n, int *a, const char *msg)
02482 {
02483     int j;
02484
02485     grcc_fprintf(GRCC_Stdout, "[");
02486     for (j = 0; j < n; j++) {
02487         if (j != 0) {
02488             grcc_fprintf(GRCC_Stdout, ", ");
02489         }
02490         grcc_fprintf(GRCC_Stdout, "%s", particleName(a[j]));
02491     }
02492     grcc_fprintf(GRCC_Stdout, "]%s", msg);
02493 }
02494
02495 //-------------------------------------------------------------
02496 int Model::findMClass(const int cpl, const int dgr)
```

```
02497 {
02498     int j;
02499
02500     for (j = 0; j < ncplgcp; j++) {
02501         if (cplgcp[j] == cpl && cplglg[j] == dgr) {
02502             return j;
02503         }
02504     }
02505     return -1;
02506 }
02507
02508 //-----------------------------------------------------------------
02509 int Model::normalParticle(int pt)
02510 {
02511     int ptc = Abs(pt);
02512     Particle *p;
02513
02514     if (ptc >= nParticles) {
02515         return 0;
02516     }
02517     p = particles[ptc];
02518     if (p->isNeutral()) {
02519         return ptc;
02520     } else {
02521         return pt;
02522     }
02523 }
02524
02525 //-----------------------------------------------------------------
02526 int Model::antiParticle(int pt)
02527 {
02528     int ptc = Abs(pt);
02529     Particle *p;
02530
02531     p = particles[ptc];
02532     if (p->isNeutral()) {
02533         return ptc;
02534     } else {
02535         return -pt;
02536     }
02537 }
02538
02539 //-----------------------------------------------------------------
02540 int *Model::allParticles(int *len)
02541 {
02542     *len = nallPart;
02543     return allPart;
02544 }
02545
02546 //-----------------------------------------------------------------
02547 int Model::findInteractionName(const char *name)
02548 {
02549     int j;
02550
02551     for (j = 0; j < nInteracts; j++) {
02552         if (strcmp(name, interacts[j]->name) == 0) {
02553             return j;
02554         }
02555     }
02556     return -1;
02557 }
02558
02559 //-----------------------------------------------------------------
02560 int Model::findInteractionCode(int icd)
02561 {
02562     int j;
02563
02564     for (j = 0; j < nInteracts; j++) {
02565         if (icd == interacts[j]->icode) {
02566             return j;
02567         }
02568     }
02569     return -1;
02570 }
02571
02572 //-----------------------------------------------------------------
02573 void Model::printMInput(MInput *min)
02574 {
02575     grcc_fprintf(GRCC_Stdout, "  \"Model\": [\"%s\", %d, [",
02576             min->name, min->ncouple);
02577     for (int j = 0; j < min->ncouple; j++) {
02578         if (j != 0) {
02579             grcc_fprintf(GRCC_Stdout, ", ");
02580         }
02581         grcc_fprintf(GRCC_Stdout, "\"%s\"", min->cnamlist[j]);
02582     }
02583     grcc_fprintf(GRCC_Stdout, "], ");
```

```
02584     if (min->defpart == GRCC_DEFBYCODE) {
02585         grcc_fprintf(GRCC_Stdout, "\"ByCode\"");
02586     } else {
02587         grcc_fprintf(GRCC_Stdout, "\"ByName\"");
02588     }
02589     grcc_fprintf(GRCC_Stdout, "],\n");
02590 }
02591
02592 //-------------------------------------------------------------
02593 void Model::printPInput(PInput *pin)
02594 {
02595     grcc_fprintf(GRCC_Stdout, "    [\"%s\"(%d), \"%s\"(%d), \"%s\"(%d), %d],\n",
02596            pin->name, pin->pcode, pin->aname, pin->acode,
02597            pin->ptypen, pin->ptypec, pin->extonly);
02598 }
02599
02600 //-------------------------------------------------------------
02601 void Model::printIInput(IInput *iin)
02602 {
02603     int j;
02604
02605     grcc_fprintf(GRCC_Stdout, "    [\"%s\"(%d), %d, [", iin->name, iin->icode, iin->nplistn);
02606     for (j = 0; j < iin->nplistn; j++) {
02607         if (j != 0) {
02608             grcc_fprintf(GRCC_Stdout, ", ");
02609         }
02610         grcc_fprintf(GRCC_Stdout, "\"%s\"(%d)", iin->plistn[j], iin->plistc[j]);
02611     }
02612     grcc_fprintf(GRCC_Stdout, "], [");
02613     for (j = 0; j < GRCC_MAXNCPLG; j++) {
02614         if (j != 0) {
02615             grcc_fprintf(GRCC_Stdout, ", ");
02616         }
02617         grcc_fprintf(GRCC_Stdout, "%d", iin->cvallist[j]);
02618     }
02619     grcc_fprintf(GRCC_Stdout, "],\n");
02620 }
02621
02622 //*************************************************************
02623 // proc.cc
02624
02625 //=============================================================
02626 // class PNodeClass
02627 //-------------------------------------------------------------
02628 PNodeClass::PNodeClass(SProcess *spc, int nnods, int nclss, NCInput *cls)
02629 {
02630     //  Create a class of nodes
02631     //    nnodes : # nodes
02632     //    nclss  : # classes
02633     //    cls    : table of class inputs
02634     //
02635     int j, k, nn, lp2;
02636     Bool ok = True;
02637
02638     sproc    = spc;
02639     nnodes   = nnods;
02640     nclass   = nclss;
02641     deg      = new int[nclass];
02642     type     = new int[nclass];
02643     particle = new int[nclass];
02644     count    = new int[nclass];
02645     couple   = new int[nclass];
02646     cmindeg  = new int[nclass];
02647     cmaxdeg  = new int[nclass];
02648     for (j = 0; j < nclass; j++) {
02649         deg[j]      = cls[j].cldeg;
02650         count[j]    = cls[j].clnum;
02651         type[j]     = cls[j].cltyp;
02652         cmindeg[j]  = cls[j].cmind;
02653         cmaxdeg[j]  = cls[j].cmaxd;
02654         particle[j] = cls[j].ptcl;
02655         couple[j]   = cls[j].cple;
02656         lp2 = (couple[j]-deg[j]+2);
02657         if (!isATExternal(type[j]) && (lp2 % 2 != 0 || lp2 < 0)) {
02658             if (prlevel > 0) {
02659                 grcc_fprintf(GRCC_Stderr, "*** PNodeClass: illegal loop: "
02660                        ": 2*loop=cpl[%d](%d)-deg[%d](%d)+2 =2*loop = %d\n",
02661                        j, couple[j], j, deg[j], lp2);
02662                 for (k = 0; k < nclss; k++) {
02663                     grcc_fprintf(GRCC_Stderr, "k=%d, deg=%d, typ=%d, ptcl=%d, ",
02664                            k, deg[k], type[k], particle[k]);
02665                     grcc_fprintf(GRCC_Stderr, "cpl=%d, cnt=%d\n",
02666                            couple[k], count[k]);
02667                 }
02668             }
02669             ok = False;
02670         }
```

```
02671     }
02672     if (!ok) {
02673         erEnd("PNodeClass: illegal loop");
02674     }
02675     cl2nd   = new int[nclass+1];
02676     nd2cl   = new int[nnodes];
02677     cl2mcl  = new int[nnodes];
02678 
02679     nn = 0;
02680     for (j = 0; j < nclass; j++) {
02681         cl2nd[j]  = nn;
02682         for (k = 0; k < count[j]; k++, nn++) {
02683             nd2cl[nn]  = j;
02684         }
02685         if (isATExternal(type[j])) {
02686             cl2mcl[j] = -1;
02687         } else {
02688             cl2mcl[j] = spc->model->findMClass(couple[j], deg[j]);
02689             if (cl2mcl[j] < 0) {
02690                 if (prlevel > 0) {
02691                     grcc_fprintf(GRCC_Stderr, "*** PNodeClass : no vertex : ");
02692                     grcc_fprintf(GRCC_Stderr, "coupling=%d, degree=%d\n",
02693                            couple[j], deg[j]);
02694                 }
02695                 erEnd("PNodeClass : no vertex");
02696             }
02697         }
02698     }
02699     cl2nd[nclass] = nnodes;
02700 }
02701 //-------------------------------------------------------------
02702 PNodeClass::PNodeClass(SProcess *spc, int nnods, int nclss, int *dgs, int *typ, int *ptcl, int *cpl,
      int *cnt, int *cmind, int *cmaxd)
02703 {
02704     //  Create a class of nodes
02705     //    nnodes : # nodes
02706     //    nclss  : # classes
02707     //    dgs  : degree of a node, which is common to the vertices in a class
02708     //    typ  : initial/final/vertex
02709     //    ptcl : particle code or list of possible interactions
02710     //    cpl  : values of coupling constants.
02711     //    cnt  : the number of nodes in this class
02712     //
02713     int j, k, nn, lp2;
02714     Bool ok = True;
02715 
02716     sproc    = spc;
02717     nnodes   = nnods;
02718     nclass   = nclss;
02719     deg      = new int[nclass];
02720     type     = new int[nclass];
02721     particle = new int[nclass];
02722     count    = new int[nclass];
02723     couple   = new int[nclass];
02724     cmindeg  = new int[nclass];
02725     cmaxdeg  = new int[nclass];
02726     for (j = 0; j < nclass; j++) {
02727         deg[j]      = dgs[j];
02728         type[j]     = typ[j];
02729         particle[j] = ptcl[j];
02730         count[j]    = cnt[j];
02731         couple[j]   = cpl[j];
02732         cmindeg[j]  = cmind[j];
02733         cmaxdeg[j]  = cmaxd[j];
02734         lp2 = (cpl[j]-dgs[j]+2);
02735         if (!isATExternal(type[j]) && (lp2 % 2 != 0 || lp2 < 0)) {
02736             if (prlevel > 0) {
02737                 grcc_fprintf(GRCC_Stderr, "*** PNodeClass: illegal loop: "
02738                        ": 2*loop=cpl[%d](%d)-deg[%d](%d)+2 =2*loop = %d\n",
02739                        j, cpl[j], j, dgs[j], lp2);
02740                 for (k = 0; k < nclss; k++) {
02741                     grcc_fprintf(GRCC_Stderr, "k=%d, dgs=%d, typ=%d, ptcl=%d, ",
02742                            k, dgs[k], typ[k], ptcl[k]);
02743                     grcc_fprintf(GRCC_Stderr, "cpl=%d, cnt=%d\n", cpl[k], cnt[k]);
02744                 }
02745             }
02746             ok = False;
02747         }
02748     }
02749     if (!ok) {
02750         erEnd("PNodeClass: illegal loop");
02751     }
02752     cl2nd   = new int[nclass+1];
02753     nd2cl   = new int[nnodes];
02754     cl2mcl  = new int[nnodes];
02755 
02756     nn = 0;
```

```
02757     for (j = 0; j < nclass; j++) {
02758         cl2nd[j]  = nn;
02759         for (k = 0; k < count[j]; k++, nn++) {
02760             nd2cl[nn]  = j;
02761         }
02762         if (isATExternal(type[j])) {
02763             cl2mcl[j] = -1;
02764         } else {
02765             cl2mcl[j] = spc->model->findMClass(couple[j], deg[j]);
02766             if (cl2mcl[j] < 0) {
02767                 if (prlevel > 0) {
02768                     grcc_fprintf(GRCC_Stderr, "*** PNodeClass : no vertex : ");
02769                     grcc_fprintf(GRCC_Stderr, "coupling=%d, degree=%d\n",
02770                             couple[j], deg[j]);
02771                 }
02772                 erEnd("PNodeClass : no vertex");
02773             }
02774         }
02775     }
02776     cl2nd[nclass] = nnodes;
02777 }
02778
02779 //-------------------------------------------------------------
02780 PNodeClass::~PNodeClass(void)
02781 {
02782     delete[] cl2mcl;
02783     delete[] nd2cl;
02784     delete[] cl2nd;
02785     delete[] cmaxdeg;
02786     delete[] cmindeg;
02787     delete[] couple;
02788     delete[] count;
02789     delete[] particle;
02790     delete[] type;
02791     delete[] deg;
02792 }
02793
02794 //-------------------------------------------------------------
02795 void PNodeClass::prPNodeClass(void)
02796 {
02797     int j;
02798
02799     grcc_fprintf(GRCC_Stdout, "+++ PNodeClass: nclass=%d, nnodes=%d\n", nclass, nnodes);
02800     for (j = 0; j < nclass; j++) {
02801         prElem(j);
02802     }
02803     grcc_fprintf(GRCC_Stdout, "\n");
02804 }
02805
02806 //-------------------------------------------------------------
02807 void PNodeClass::prElem(int j)
02808 {
02809     int tp;
02810
02811     grcc_fprintf(GRCC_Stdout, "%3d: %-7s(%2d), ", j, GRCC_AT_NdStr(type[j]), type[j]);
02812     grcc_fprintf(GRCC_Stdout, "deg=%d, count=%d, nodes[%d--%d], couple=%d, cmindeg=%d, cmaxdeg=%d",
02813           deg[j], count[j], cl2nd[j], cl2nd[j+1], couple[j], cmindeg[j], cmaxdeg[j]);
02814     tp = type[j];
02815     if (isATExternal(tp)) {
02816         if (sproc->model != NULL) {
02817             grcc_fprintf(GRCC_Stdout, ", ptcl=%s ", sproc->model->particleName(particle[j]));
02818         } else {
02819             grcc_fprintf(GRCC_Stdout, ", ptcl=%d ", particle[j]);
02820         }
02821     }
02822     grcc_fprintf(GRCC_Stdout, "\n");
02823
02824 }
02825
02826 //=============================================================
02827 // class SProcess
02828 //-------------------------------------------------------------
02829 SProcess::SProcess(Model *mdl, Process *prc, Options *opts, int sid, int *clst, int ncls, int *cdeg,
      int *ctyp, int *ptcl, int *cpl, int *cnum, int *cmind, int *cmaxd)
02830 {
02831     // Construct a Subprocess object
02832     //    mdl            : model
02833     //    prc            : process (may be NULL)
02834     //    opts           : options
02835     //    sid            : id of the sprocess
02836     //    ncls           : the number of classes
02837     //    cdeg[ncls]     : the table of degrees of nodes.
02838     //    ctyp[ncls]     : the table of nodes in the classes
02839     //    ptcl[ncls]     : particle(External)/interaction code(Internal)
02840     //    cpl[ncls]      : the table of total order of coupling constants.
02841     //    cnum[ncls]     : the table of nodes in the classes
02842     //    cmind[ncls]    : the table of min(deg of connectable vertex)
```

```
02843     //    cmaxd[ncls]    : the table of max(deg of connectable vertex)
02844
02845     int j, cp, lp2, ndeg, nvrt, tcpl0;
02846     bool ok;
02847
02848     if (ncls < 1) {
02849         grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: no class %d\n", ncls);
02850         erEnd("SProcess::SProcess: no Class");
02851     }
02852     id       = sid;
02853     model    = mdl;
02854     proc     = prc;
02855     opt      = opts;
02856     nclass   = ncls;
02857
02858     nNodes   = 0;
02859     nEdges   = 0;
02860     nExtern  = 0;
02861     tCouple  = 0;
02862     loop     = 0;
02863     mgraph   = NULL;
02864     egraph   = NULL;
02865     astack   = NULL;
02866
02867     mgrcount = 0;
02868     agrcount = 0;
02869     extperm  = 1;
02870
02871     // the results of the graph generation
02872     nMGraphs = 0;               // the number of generated M-graphs
02873     nMOPI    = 0;               // the number of 1PI M-graphs
02874     wMGraphs.setValue(0,1);   // the weighted sum of M-graphs
02875     wMOPI.setValue(0,1);      // the weighted sum of 1PI M-graphs
02876
02877     nAGraphs = 0;               // the number of generated A-graphs
02878     nAOPI    = 0;               // the number of 1PI A-graphs
02879     wAGraphs.setValue(0,1);   // the weighted sum of A-graphs
02880     wAOPI.setValue(0,1);      // the weighted sum of 1PI A-graphs
02881
02882     // check input
02883     nNodes  = 0;
02884     nExtern = 0;
02885
02886     ndeg     = 0;
02887     nvrt     = 0;
02888     ok = True;
02889
02890     tcpl0 = 0;
02891     for (j = 0; j < model->ncouple; j++) {
02892         clist[j] = clst[j];
02893         tcpl0 += clist[j];
02894     }
02895     for (j = 0; j < nclass; j++) {
02896         cp  = cpl[j];
02897         if (cp > 0) {
02898             lp2 = cpl[j] - cdeg[j] + 2;
02899             if (lp2 % 2 != 0 || lp2 < 0) {
02900                 if (prlevel > 0) {
02901                     grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: illegal loop:"
02902                            "class %d, cp=%d, deg=%d, lp2=%d\n",
02903                            j, cp, cdeg[j], lp2);
02904                 }
02905                 ok = False;
02906             }
02907         }
02908         tCouple +=      cp*cnum[j];
02909         ndeg    += cdeg[j]*cnum[j];
02910
02911         if (!isATExternal(ctyp[j])) {
02912             nvrt += cnum[j];
02913         } else if (isATExternal(ctyp[j])) {
02914             if (model->normalParticle(ptcl[j]) == 0) {
02915                 if (prlevel > 0) {
02916                     grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: ");
02917                     grcc_fprintf(GRCC_Stderr, "illegal particle code: ");
02918                     grcc_fprintf(GRCC_Stderr, "code: class %d, ptcl=%d\n", j, ptcl[j]);
02919                 }
02920                 ok = False;
02921             } else {
02922                 nExtern += cnum[j];
02923             }
02924             extperm *= factorial(cnum[j]);
02925
02926         } else {
02927             if (prlevel > 0) {
02928                 grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: illegal type: ");
02929                 grcc_fprintf(GRCC_Stderr, "class %d, type=%d\n", j, ctyp[j]);
```

```
02930             }
02931             ok = False;
02932         }
02933     }
02934     if (tcpl0 != tCouple) {
02935         if (prlevel > 0) {
02936             grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: ");
02937             grcc_fprintf(GRCC_Stderr, "illegal coupling constants:");
02938             grcc_fprintf(GRCC_Stderr, " %d != %d\n", tcpl0, tCouple);
02939         }
02940         ok = False;
02941     }
02942     if (ndeg == nExtern) {
02943         if (prlevel > 0) {
02944             grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: ");
02945             grcc_fprintf(GRCC_Stderr, "no vertices: ");
02946             grcc_fprintf(GRCC_Stderr, "nExtern=%d, ndeg=%d\n", nExtern, ndeg);
02947         }
02948         ok = False;
02949     }
02950     if (ndeg % 2 == 0) {
02951         nEdges = ndeg/2;
02952     } else {
02953         if (prlevel > 0) {
02954             grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: ");
02955             grcc_fprintf(GRCC_Stderr, "illegal total deg = %d (not even)\n", ndeg);
02956         }
02957         ok = False;
02958     }
02959     nNodes = nvrt + nExtern;
02960     if (nNodes >= GRCC_MAXNODES) {
02961         if (prlevel > 0) {
02962             grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: ");
02963             grcc_fprintf(GRCC_Stderr, "too many nodes = %d\n", nNodes);
02964             grcc_fprintf(GRCC_Stderr, "    nExtern=%d, nvert=%d (GRCC_MAXNODES)\n",
02965                     nExtern, nvert);
02966         }
02967         ok = False;
02968     }
02969     if (!ok) {
02970         prSProcess();
02971         erEnd("SProcess::SProcess: illegal input");
02972     }
02973     if (proc == NULL) {
02974         opt->beginProc(NULL);
02975     }
02976
02977     lp2 = tCouple - nExtern + 2;
02978     if (lp2 % 2 != 0 || lp2 < 0) {
02979         if (prlevel > 0) {
02980             grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: illegal loop: "
02981                    "tCouple=%d, nExtern=%d, 2*loop=%d\n",
02982                    tCouple, nExtern, lp2);
02983         }
02984         ok = False;
02985     }
02986     loop = lp2/2;
02987
02988     // stack for assignment
02989     if (opt->values[GRCC_OPT_Step] == GRCC_AGraph) {
02990         astack = new AStack(GRCC_MAXNSTACK, GRCC_MAXESTACK);
02991     }
02992
02993     // save to PNodeClass object
02994     pnclass = new PNodeClass(this, nNodes, nclass, cdeg, ctyp, ptcl, cpl, cnum, cmind, cmaxd);
02995 }
02996
02997 //------------------------------------------------------------------
02998 SProcess::SProcess(Model *mdl, Process *prc, Options *opts, int sid, int *clst, int ncls, NCInput
      *cls)
02999 {
03000     // Construct a Subprocess object
03001     //    mdl            : model
03002     //    prc            : process (may be NULL)
03003     //    opts           : options
03004     //    sid            : id of the sprocess
03005     //    ncls           : the number of classes
03006     //    cls            : input data of classes
03007
03008     int j, cp, lp2, ndeg, nvrt, tcpl0;
03009     bool ok;
03010
03011     if (ncls < 1) {
03012         grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: no class %d\n", ncls);
03013         erEnd("SProcess::SProcess: no Class");
03014     }
03015     id       = sid;
```

```
03016     model    = mdl;
03017     proc     = prc;
03018     opt      = opts;
03019     nclass   = ncls;
03020
03021     nNodes   = 0;
03022     nEdges   = 0;
03023     nExtern  = 0;
03024     tCouple  = 0;
03025     loop     = 0;
03026     mgraph   = NULL;
03027     egraph   = NULL;
03028     astack   = NULL;
03029
03030     mgrcount = 0;
03031     agrcount = 0;
03032     extperm  = 1;
03033
03034     // the results of the graph generation
03035     nMGraphs = 0;               // the number of generated M-graphs
03036     nMOPI    = 0;               // the number of 1PI M-graphs
03037     wMGraphs.setValue(0,1);   // the weighted sum of M-graphs
03038     wMOPI.setValue(0,1);      // the weighted sum of 1PI M-graphs
03039
03040     nAGraphs = 0;               // the number of generated A-graphs
03041     nAOPI    = 0;               // the number of 1PI A-graphs
03042     wAGraphs.setValue(0,1);   // the weighted sum of A-graphs
03043     wAOPI.setValue(0,1);      // the weighted sum of 1PI A-graphs
03044
03045     // check input
03046     nNodes  = 0;
03047     nExtern = 0;
03048
03049     ndeg     = 0;
03050     nvrt     = 0;
03051     ok = True;
03052
03053     tcpl0 = 0;
03054     for (j = 0; j < model->ncouple; j++) {
03055         clist[j] = clst[j];
03056         tcpl0 += clist[j];
03057     }
03058     for (j = 0; j < nclass; j++) {
03059         cp  = cls[j].cple;
03060         if (cp > 0) {
03061             lp2 = cls[j].cple - cls[j].cldeg + 2;
03062             if (lp2 % 2 != 0 || lp2 < 0) {
03063                 if (prlevel > 0) {
03064                     grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: illegal loop: "
03065                            "class %d, cp=%d, deg=%d, lp2=%d\n",
03066                            j, cp, cls[j].cldeg, lp2);
03067                 }
03068                 ok = False;
03069             }
03070         }
03071         tCouple += cp*cls[j].clnum;
03072         ndeg    += cls[j].cldeg*cls[j].clnum;
03073
03074         if (!isATExternal(cls[j].cltyp)) {
03075             nvrt += cls[j].clnum;
03076         } else if (isATExternal(cls[j].cltyp)) {
03077             if (model->normalParticle(cls[j].ptcl) == 0) {
03078                 if (prlevel > 0) {
03079                     grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: ");
03080                     grcc_fprintf(GRCC_Stderr, "illegal particle code: ");
03081                     grcc_fprintf(GRCC_Stderr, "code: class %d, ptcl=%d\n", j, cls[j].ptcl);
03082                 }
03083                 ok = False;
03084             } else {
03085                 nExtern += cls[j].clnum;
03086             }
03087             extperm *= factorial(cls[j].clnum);
03088
03089         } else {
03090             if (prlevel > 0) {
03091                 grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: illegal type: ");
03092                 grcc_fprintf(GRCC_Stderr, "class %d, type=%d\n", j, cls[j].cltyp);
03093             }
03094             ok = False;
03095         }
03096     }
03097     if (tcpl0 != tCouple) {
03098         if (prlevel > 0) {
03099             grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: ");
03100             grcc_fprintf(GRCC_Stderr, "illegal coupling constants:");
03101             grcc_fprintf(GRCC_Stderr, " %d != %d\n", tcpl0, tCouple);
03102         }
```

```
03103         ok = False;
03104     }
03105     if (ndeg == nExtern) {
03106         if (prlevel > 0) {
03107             grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: ");
03108             grcc_fprintf(GRCC_Stderr, "no vertices: ");
03109             grcc_fprintf(GRCC_Stderr, "nExtern=%d, ndeg=%d\n", nExtern, ndeg);
03110         }
03111         ok = False;
03112     }
03113     if (ndeg % 2 == 0) {
03114         nEdges = ndeg/2;
03115     } else {
03116         if (prlevel > 0) {
03117             grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: ");
03118             grcc_fprintf(GRCC_Stderr, "illegal total deg = %d (not even)\n", ndeg);
03119         }
03120         ok = False;
03121     }
03122     nNodes = nvrt + nExtern;
03123     if (nNodes >= GRCC_MAXNODES) {
03124         if (prlevel > 0) {
03125             grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: ");
03126             grcc_fprintf(GRCC_Stderr, "too many nodes = %d\n", nNodes);
03127             grcc_fprintf(GRCC_Stderr, "    nExtern=%d, nvert=%d (GRCC_MAXNODES)\n",
03128                     nExtern, nvert);
03129         }
03130         ok = False;
03131     }
03132     if (!ok) {
03133         erEnd("SProcess::SProcess: illegal input");
03134     }
03135     if (proc == NULL) {
03136         opt->beginProc(NULL);
03137     }
03138
03139     lp2 = tCouple - nExtern + 2;
03140     if (lp2 % 2 != 0 || lp2 < 0) {
03141         if (prlevel > 0) {
03142             grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: illegal loop: "
03143                    "tCouple=%d, nExtern=%d, 2*loop=%d\n",
03144                    tCouple, nExtern, lp2);
03145         }
03146         ok = False;
03147     }
03148     loop = lp2/2;
03149
03150     // stack for assignment
03151     if (opt->values[GRCC_OPT_Step] == GRCC_AGraph) {
03152         astack = new AStack(GRCC_MAXNSTACK, GRCC_MAXESTACK);
03153     }
03154
03155     // save to PNodeClass object
03156     pnclass = new PNodeClass(this, nNodes, nclass, cls);
03157 }
03158
03159 //------------------------------------------------------------
03160 SProcess::~SProcess(void)
03161 {
03162     delete pnclass;
03163     delete astack;
03164     delete mgraph;
03165     model = NULL;
03166 }
03167
03168 //------------------------------------------------------------
03169 void SProcess::prSProcess(void)
03170 {
03171     grcc_fprintf(GRCC_Stdout, "\n");
03172     grcc_fprintf(GRCC_Stdout, "+++ Subprocess %d, class=%d\n", id, nclass);
03173     if (pnclass != NULL) {
03174         pnclass->prPNodeClass();
03175     } else {
03176         grcc_fprintf(GRCC_Stdout, "  pnclass = NULL\n");
03177    }
03178 }
03179
03180 //------------------------------------------------------------
03181 BigInt SProcess::generate(void)
03182 {
03183     //  Entry point of Feynman graph generation
03184     int  cldeg[GRCC_MAXNODES], clnum[GRCC_MAXNODES], cltyp[GRCC_MAXNODES];
03185     int  cmind[GRCC_MAXNODES], cmaxd[GRCC_MAXNODES];
03186     int  ncl;
03187     BigInt ng;
03188
03189     ncl = toMNodeClass(cltyp, cldeg, clnum, cmind, cmaxd);
```

```
03190
03191     opt->beginSubProc(this);
03192
03193     mgraph = new MGraph(id, ncl, cldeg, clnum, cltyp, cmind, cmaxd, opt);
03194
03195     ng = mgraph->generate();
03196
03197     opt->endSubProc();
03198
03199     delete mgraph;
03200     mgraph = NULL;
03201
03202     return ng;
03203 }
03204
03205 //-------------------------------------------------------------
03206 int SProcess::toMNodeClass(int *cltyp, int *cldeg, int *clnum, int *cmind, int *cmaxd)
03207 {
03208     int j;
03209
03210     for (j = 0; j < nclass; j++) {
03211         if (isATExternal(pnclass->type[j])) {
03212             cltyp[j] = -1;
03213         } else {
03214             cltyp[j] = (pnclass->couple[j]-pnclass->deg[j]+2)/2;
03215         }
03216         cldeg[j] = pnclass->deg[j];
03217         clnum[j] = pnclass->count[j];
03218         cmind[j] = pnclass->cmindeg[j];
03219         cmaxd[j] = pnclass->cmaxdeg[j];
03220     }
03221     return nclass;
03222 }
03223
03224 //-------------------------------------------------------------
03225 void SProcess::assign(MGraph *mgr)
03226 {
03227     PNodeClass *pnc;
03228
03229     pnc = match(mgr);
03230
03231     agraph = new Assign(this, mgr, pnc);
03232
03233 #ifdef CHECK
03234     agraph->checkAG("SProcess::assign");
03235 #endif
03236     delete pnc;
03237     delete agraph;
03238     agraph = NULL;
03239 }
03240
03241 //-------------------------------------------------------------
03242 PNodeClass *SProcess::match(MGraph *mgr)
03243 {
03244     PNodeClass *pnc = NULL;
03245
03246     int typ[GRCC_MAXNODES], dgs[GRCC_MAXNODES];
03247     int cnt[GRCC_MAXNODES], ptcl[GRCC_MAXNODES];
03248     int cpl[GRCC_MAXNODES];
03249     int n2m[GRCC_MAXNODES];
03250     int cmind[GRCC_MAXNODES], cmaxd[GRCC_MAXNODES];
03251     int j, k, l, nd, mc, mcl, mclss;
03252
03253     if (mgr->nNodes != nNodes) {
03254         if (prlevel > 0) {
03255             grcc_fprintf(GRCC_Stderr, "*** SProcess::match: different nNodes=%d != %d\n",
03256                     nNodes, mgr->nNodes);
03257         }
03258         erEnd("SProcess::match: different nNodes");
03259     }
03260
03261     mcl = toMNodeClass(typ, dgs, cnt, cmind, cmaxd);
03262
03263     nd = 0;
03264     for (j = 0; j < mcl; j++) {
03265         for (k = 0; k < cnt[j]; k++, nd++) {
03266             n2m[nd] = j;
03267         }
03268     }
03269
03270     nd    = 0;
03271     mclss = mgr->curcl->nClasses;
03272     for (j = 0; j < mclss; j++) {
03273         for (k = 0; k < mgr->curcl->clist[j]; k++, nd++) {
03274             mc = mgr->nodes[nd]->clss;
03275             if (mc != n2m[nd]) {
03276                 if (prlevel > 0) {
```

```
03277                     grcc_fprintf(GRCC_Stderr, "*** SProcess::match: ");
03278                     grcc_fprintf(GRCC_Stderr, "inconsistent class\n");
03279                     for (l = 0; l < nNodes; l++) {
03280                         grcc_fprintf(GRCC_Stderr, "    %d: %d, %d\n",
03281                                 j, mgr->nodes[l]->clss, n2m[l]);
03282                     }
03283                 }
03284                 erEnd("SProcess::match: inconsistent class");
03285             }
03286         }
03287     }
03288     for (j = 0; j < mcl; j++) {
03289         dgs[j]    = pnclass->deg[j];
03290         typ[j]    = pnclass->type[j];
03291         ptcl[j]   = pnclass->particle[j];
03292         cnt[j]    = pnclass->count[j];
03293         cpl[j]    = pnclass->couple[j];
03294         cmind[j]  = pnclass->cmindeg[j];
03295         cmaxd[j]  = pnclass->cmaxdeg[j];
03296     }
03297     pnc = new PNodeClass(this, nNodes, mcl, dgs, typ, ptcl, cpl, cnt, cmind, cmaxd);
03298
03299     return pnc;
03300 }
03301
03302 //-------------------------------------------------------------
03303 void SProcess::resultMGraph(BigInt nmgraphs, Fraction mwsum, BigInt nmopi, Fraction mwopi)
03304 {
03305     nMGraphs += nmgraphs;
03306     nMOPI    += nmopi;
03307     wMGraphs.add(mwsum);
03308     wMOPI.add(mwopi);
03309 }
03310
03311 //-------------------------------------------------------------
03312 void SProcess::resultAGraph(BigInt nmgraphs, Fraction mwsum, BigInt nmopi, Fraction mwopi)
03313 {
03314     nAGraphs += nmgraphs;
03315     nAOPI    += nmopi;
03316     wAGraphs.add(mwsum);
03317     wAOPI.add(mwopi);
03318 }
03319
03320 //-------------------------------------------------------------
03321 void SProcess::endMGraph(MGraph *mgr)
03322 {
03323     egraph = mgr->egraph;
03324
03325     if (opt->values[GRCC_OPT_Step] != GRCC_MGraph) {
03326         assign(mgr);
03327     }
03328 }
03329
03330 //-------------------------------------------------------------
03331 void SProcess::endAGraph(EGraph *egr)
03332 {
03333     egraph = egr;
03334 }
03335
03336
03337 //=============================================================
03338 // class Process
03339 //-------------------------------------------------------------
03340 Process::Process(int pid, Model *modl, Options *optn, int nini, int *intlPrt, int nfin, int *finlPrt,
      int *coupling)
03341 {
03342     //  Define a process and construct a set of sprocesses
03343     //    model     : Model object
03344     //    optn      : Option object
03345     //    initlPart : list of particle-id of initlPart particles
03346     //    finalPart : list of particle-id of final particles
03347     //    coupling  : list of coupling constants
03348
03349     int j, lp2;
03350     Bool ok;
03351     char buff[MAXSTR];
03352
03353     id        = pid;
03354     model     = modl;
03355     opt       = optn;
03356     ninitl    = nini;
03357     nfinal    = nfin;
03358     initlPart = intdup(ninitl, intlPrt);
03359     finalPart = intdup(nfinal, finlPrt);
03360
03361     // count total number of coupling constants.
03362     ctotal = 0;
```

```
03363     for (j = 0; j < GRCC_MAXNCPLG; j++) {
03364         if (j < model->ncouple) {
03365             clist[j] = coupling[j];
03366             ctotal  += coupling[j];
03367         } else {
03368             clist[j] = 0;
03369         }
03370     }
03371     if (ctotal < 1) {
03372         erEnd("Process: coupling constant = 0");
03373     }
03374 
03375     nExtern   =  0;
03376     loop      = -1;
03377     maxnlegs  =  0;
03378     mgrcount = 0;
03379     agrcount = 0;
03380 
03381     // table of sprocesses
03382     nSubproc  = 0;
03383     sproc   = NULL;
03384 
03385     // the results of the graph generation
03386     nMGraphs = 0;                 // the number of generated M-graphs
03387     nMOPI    = 0;                 // the number of 1PI M-graphs
03388     wMGraphs.setValue(0,1);       // the weighted sum of M-graphs
03389     wMOPI.setValue(0,1);          // the weighted sum of 1PI M-graphs
03390 
03391     nAGraphs = 0;                 // the number of generated A-graphs
03392     nAOPI    = 0;                 // the number of 1PI A-graphs
03393     wAGraphs.setValue(0,1);       // the weighted sum of A-graphs
03394     wAOPI.setValue(0,1);          // the weighted sum of 1PI A-graphs
03395 
03396     ok = True;
03397 
03398     // numbers
03399     nExtern  = ninitl + nfinal;
03400     maxnlegs = Max(nExtern, model->maxnlegs);
03401 
03402     // count loops
03403     lp2 = ctotal - nExtern + 2;
03404     if (lp2 % 2 != 0) {
03405         if (prlevel > 0) {
03406             grcc_fprintf(GRCC_Stderr, "*** cannot generate : 2*loop is odd : "
03407                     "2*loop = %d, ctotal=%d, nExtern=%d\n",
03408                     lp2, ctotal, nExtern);
03409         }
03410         ok = False;
03411     } else if (lp2 < 0) {
03412         if (prlevel > 0) {
03413             grcc_fprintf(GRCC_Stderr, "*** cannot make a connected graph : "
03414                     "2*loop=%d, ctotal=%d, nExtern=%d\n",
03415                     lp2, ctotal, nExtern);
03416         }
03417         ok = False;
03418     }
03419 
03420     loop = lp2 / 2;
03421 
03422     if (!ok) {
03423         if (prlevel > 0) {
03424             grcc_fprintf(GRCC_Stderr, "*** Process: illegal input: lp2 = %d\n", lp2);
03425         }
03426         erEnd("Process: illegal input");
03427     }
03428 
03429     if (opt->out->outgrp != NULL) {
03430         snprintf(buff, MAXSTR, "out%d.prp", pid);
03431         prProcessP(buff);
03432     }
03433 
03434     // construct sprocesses
03435     mkSProcess();
03436 }
03437 
03438 //------------------------------------------------------------------
03439 Process::~Process(void)
03440 {
03441     initlPart = delintdup(initlPart);
03442     finalPart = delintdup(finalPart);
03443     for (int j = 0; j < nSubproc; j++) {
03444         delete sptbl[j];
03445     }
03446  // delete sproc;
03447 }
03448 
03449 //------------------------------------------------------------------
```

```
03450 void Process::prProcess(void)
03451 {
03452     grcc_fprintf(GRCC_Stdout, "+++ process options : OPI = %d, (Step) = %d, coupling=%d: ",
03453            opt->values[GRCC_OPT_1PI], opt->values[GRCC_OPT_Step], ctotal);
03454     prIntArray(model->ncouple, clist, "\n");
03455 }
03456
03457 //-------------------------------------------------------------
03458 void Process::prProcessP(const char *fname)
03459 {
03460     FILE *fp;
03461
03462     if ((fp = fopen(fname, "w")) == NULL) {
03463         grcc_fprintf(GRCC_Stderr, "*** cannot open \"%s\"\n", fname);
03464         return;
03465     }
03466     fprintf(fp, "Process = {\n");
03467     outProcP(fp);
03468     fprintf(fp, "};\n");
03469     fclose(fp);
03470 }
03471
03472 //-------------------------------------------------------------
03473 void Process::outProcP(FILE *fp)
03474 {
03475     int   j;
03476
03477     if (model == NULL) {
03478         return;
03479     }
03480     fprintf(fp, "    \"Model\": \"%s\",\n", model->name);
03481     fprintf(fp, "    \"Initial\": [");
03482     for (j = 0; j < ninitl; j++) {
03483         if (j != 0) {
03484             fprintf(fp, ", ");
03485         }
03486         fprintf(fp, "\"%s\"", model->particleName(initlPart[j]));
03487     }
03488     fprintf(fp, "],\n");
03489     fprintf(fp, "    \"Final\": [");
03490     for (j = 0; j < nfinal; j++) {
03491         if (j != 0) {
03492             fprintf(fp, ", ");
03493         }
03494         fprintf(fp, "\"%s\"", model->particleName(finalPart[j]));
03495     }
03496     fprintf(fp, "],\n");
03497     fprintf(fp, "    \"Couple\": [");
03498     for (j = 0; j < model->ncouple; j++) {
03499         if (j != 0) {
03500             fprintf(fp, ", ");
03501         }
03502         fprintf(fp, "%d", clist[j]);
03503     }
03504     fprintf(fp, "],\n");
03505     fprintf(fp, "    \"Options\": {\n");
03506     for (j = 0; j < GRCC_OPT_Size; j++) {
03507         if (j == GRCC_OPT_Outgrf || j == GRCC_OPT_Outgrp) {
03508             continue;
03509         }
03510         fprintf(fp, "      \"%s\":%d,\n", optDef[j].name, opt->values[j]);
03511     }
03512     fprintf(fp, "      \"%s\":", optDef[GRCC_OPT_Outgrf].name);
03513     if (opt->out->outgrf != NULL) {
03514         fprintf(fp, "\"%s\",\n", opt->out->outgrf);
03515     } else {
03516         fprintf(fp, "0,\n");
03517     }
03518     fprintf(fp, "      \"%s\":", optDef[GRCC_OPT_Outgrp].name);
03519     if (opt->out->outgrp != NULL) {
03520         fprintf(fp, "\"%s\",\n", opt->out->outgrp);
03521     } else {
03522         fprintf(fp, "0,\n");
03523     }
03524     fprintf(fp, "    }\n");
03525 }
03526
03527 //-------------------------------------------------------------
03528 void Process::mkSProcess(void)
03529 {
03530     //  Construct sprocesses
03531     //  Each sprocess is a set of nodes whose degree and order of
03532     //  coupling constants are determined.
03533     //  Only the total coupling constants are considered here even if
03534     //  two or more coupling constants are defined in the model
03535
03536     NCInput cls[GRCC_MAXMINTERACT];
```

```
03537     int r, j, k, lin2, nvtx, nleg, nc, n0;
03538     int nl[GRCC_MAXMINTERACT];
03539     double proct0 = 0, proct1 = 0;
03540
03541     if (model->ncplgcp < 1) {
03542         erEnd("function 'addInteractionEnd' has not been called");
03543     }
03544     // output file to start process
03545     opt->beginProc(this);
03546
03547     // starting time
03548     Second(&proct0);
03549
03550     // generate all possible number of (nl[i]),
03551     //   \sum_i nl[i]*cplgcp[i] = (total number of coupling constants)
03552
03553     r = -1;
03554     nSubproc = 0;
03555     ngraphs  = 0;
03556     nopi     = 0;
03557     wgraphs  = 0;
03558     wopi     = 0;
03559
03560     // for partitions of (leg, order)
03561
03562     while (nextPart(ctotal, model->ncplgcp, model->cplgcp, nl, &r)) {
03563         // count the total number of vertices and legs.
03564         nvtx = 0;
03565         nleg = 0;
03566         for (j = 0; j < model->ncplgcp; j++) {
03567             nvtx += nl[j];
03568             nleg += nl[j]*model->cplglg[j];
03569         }
03570
03571         // count loops
03572         lin2 = nExtern + nleg;
03573         if (lin2 % 2 != 0) {
03574             continue;
03575         }
03576         loop = lin2/2 - nExtern - nvtx + 1;
03577         if (loop < 0) {
03578             continue;
03579         }
03580
03581         nc = 0;
03582         if (opt->values[GRCC_OPT_SymmInitial]) {
03583             n0 = nc;
03584             for (j = 0; j < ninitl; j++) {
03585                 for (k = n0; k < nc; k++) {
03586                     if (cls[k].ptcl == initlPart[j]) {
03587                         cls[k].clnum++;
03588                         break;
03589                     }
03590                 }
03591                 if (k >= nc) {
03592                     cls[nc].ptcl  = initlPart[j];
03593                     cls[nc].clnum = 1;
03594                     cls[nc].cple  = 0;
03595                     cls[nc].clnum = 1;
03596                     cls[nc].cldeg = 1;
03597                     cls[nc].cltyp = GRCC_AT_Initial;
03598                     cls[nc].ptcl  = initlPart[j];
03599                     cls[nc].cmind
03600                         = model->particles[Abs(cls[nc].ptcl)]->cmindeg;
03601                     cls[nc].cmaxd
03602                         = model->particles[Abs(cls[nc].ptcl)]->cmaxdeg;
03603                     if (opt->values[GRCC_OPT_NoExtSelf] > 0 && nExtern > 2) {
03604                         cls[nc].cmind = Max(cls[nc].cmind, 3);
03605                     }
03606                     nc++;
03607                 }
03608             }
03609         } else {
03610             for (j = 0; j < ninitl; j++) {
03611                 cls[nc].ptcl  = initlPart[j];
03612                 cls[nc].clnum = 1;
03613                 cls[nc].cple  = 0;
03614                 cls[nc].cldeg = 1;
03615                 cls[nc].cltyp = GRCC_AT_Initial;
03616                 cls[nc].cmind = model->particles[Abs(cls[nc].ptcl)]->cmindeg;
03617                 cls[nc].cmaxd = model->particles[Abs(cls[nc].ptcl)]->cmaxdeg;
03618                 if (opt->values[GRCC_OPT_NoExtSelf] > 0 && nExtern > 2) {
03619                     cls[nc].cmind = Max(cls[nc].cmind, 3);
03620                 }
03621                 nc++;
03622             }
03623         }
```

```
03624         if (opt->values[GRCC_OPT_SymmFinal]) {
03625             n0 = nc;
03626             for (j = 0; j < nfinal; j++) {
03627                 for (k = n0; k < nc; k++) {
03628                     if (cls[k].ptcl == model->antiParticle(finalPart[j])) {
03629                         cls[k].clnum++;
03630                         break;
03631                     }
03632                 }
03633                 if (k >= nc) {
03634                     cls[nc].ptcl  = model->antiParticle(finalPart[j]);
03635                     cls[nc].clnum = 1;
03636                     cls[nc].cple  = 0;
03637                     cls[nc].cldeg = 1;
03638                     cls[nc].cltyp = GRCC_AT_Final;
03639                     cls[nc].cmind
03640                         = model->particles[Abs(cls[nc].ptcl)]->cmindeg;
03641                     cls[nc].cmaxd
03642                         = model->particles[Abs(cls[nc].ptcl)]->cmaxdeg;
03643                     if (opt->values[GRCC_OPT_NoExtSelf] > 0 && nExtern > 2) {
03644                         cls[nc].cmind = Max(cls[nc].cmind, 3);
03645                     }
03646                     nc++;
03647                 }
03648             }
03649         } else {
03650             for (j = 0; j < nfinal; j++) {
03651                 cls[nc].ptcl  = model->antiParticle(finalPart[j]);
03652                 cls[nc].cldeg = 1;
03653                 cls[nc].cple  = 0;
03654                 cls[nc].clnum = 1;
03655                 cls[nc].cltyp = GRCC_AT_Final;
03656                 cls[nc].cmind = model->particles[Abs(cls[nc].ptcl)]->cmindeg;
03657                 cls[nc].cmaxd = model->particles[Abs(cls[nc].ptcl)]->cmaxdeg;
03658                 if (opt->values[GRCC_OPT_NoExtSelf] > 0 && nExtern > 2) {
03659                     cls[nc].cmind = Max(cls[nc].cmind, 3);
03660                 }
03661                 nc++;
03662             }
03663         }
03664         for (j = 0; j < model->ncplgcp; j++) {
03665             if (nl[j] > 0) {
03666                 cls[nc].ptcl  = 0;
03667                 cls[nc].cple  = model->cplgcp[j];
03668                 cls[nc].cldeg = model->cplglg[j];
03669                 cls[nc].clnum = nl[j];
03670                 // number of loops
03671                 cls[nc].cltyp = (model->cplgcp[j] - model->cplglg[j] + 2)/2;
03672                 cls[nc].cmind = 0;
03673                 cls[nc].cmaxd = 0;
03674                 nc++;
03675             }
03676         }
03677         // create a sprocess ??? to be rewritten
03678         sproc = new SProcess(model, this, opt, nSubproc, clist, nc, cls);
03679
03680         // list of sprocesses
03681         if (nSubproc >= GRCC_MAXSUBPROCS) {
03682               erEnd("Subclass: too many sprocesses (GRCC_MAXSUBPROCS)");
03683         }
03684         sptbl[nSubproc++] = sproc;
03685
03686         // generate M-graphs
03687         sproc->generate();
03688
03689         // summation of the numbers and weighted numbers of graphs
03690         nMGraphs += sproc->nMGraphs;
03691         nMOPI    += sproc->nMOPI;
03692         wMGraphs.add(sproc->wMGraphs);
03693         wMOPI.add(sproc->wMOPI);
03694
03695         nAGraphs += sproc->nAGraphs;
03696         nAOPI    += sproc->nAOPI;
03697         wAGraphs.add(sproc->wAGraphs);
03698         wAOPI.add(sproc->wAOPI);
03699
03700         if (nAGraphs > 0) {
03701             ngraphs  = nAGraphs;
03702         } else {
03703             ngraphs  = nAGraphs;
03704         }
03705      // delete sproc;
03706      // sproc = NULL;
03707     }
03708
03709     // ending time
03710     Second(&proct1);
```

```
03711     sec = proc1-proc0;
03712
03713     // output file to finish process
03714     opt->endProc();
03715 }
03716
03717 //************************************************************
03718 // mgraph.cc
03719 //============================================================
03720 //------------------------------------------------------------
03721 MNodeClass::MNodeClass(int nnodes, int nclasses)
03722 {
03723     // Input
03724     //   nnodes   : possible maximal number of nodes
03725     //   nclasses : possible maximal number of classes
03726
03727     int j, k;
03728
03729     nNodes   = nnodes;
03730     nClasses = nclasses;
03731     maxdeg   = 0;
03732     for (j = 0; j < nNodes; j++) {
03733         for (k = 0; k < nNodes; k++) {
03734             clmat[j][k] = 0;
03735         }
03736         clist[j]   = 0;
03737         ndcl[j]    = 0;
03738         flist[j]   = 0;
03739         clord[j]   = 0;
03740         cmindeg[j] = 0;
03741         cmaxdeg[j] = 0;
03742     }
03743     flist[nNodes] = 0;
03744     flg0 = 0;
03745     flg1 = 0;
03746     flg2 = 0;
03747 }
03748
03749 //------------------------------------------------------------
03750 MNodeClass::~MNodeClass(void)
03751 {
03752 }
03753
03754 //============================================================
03755 // class MNodeClass
03756 //------------------------------------------------------------
03757 void MNodeClass::init(int *cl, int mxdeg, int **adjmat)
03758 {
03759     // initialize the object
03760     // Argument
03761     //   cl[j]        : list of number of nodes in the j-th class
03762     //   mxdeg        : maximum degree to nodes
03763     //   adjmat[j][k] : adjacency matrix
03764
03765     int j;
03766
03767     for (j = 0; j < nClasses; j++) {
03768         clist[j]  = cl[j];
03769         clord[j]  = j;
03770     }
03771     maxdeg   = mxdeg;
03772     mkNdCl();
03773     mkClMat(adjmat);
03774     mkFlist();
03775     flg0 = 0;
03776     flg1 = 0;
03777     flg2 = 0;
03778 }
03779
03780 //------------------------------------------------------------
03781 void MNodeClass::copy(MNodeClass* mnc)
03782 {
03783     // copy MNodeClass 'mnc' to this object
03784
03785     int j, k;
03786
03787     nClasses = mnc->nClasses;
03788     for (k = 0; k < nClasses; k++) {
03789         clist[k] = mnc->clist[k];
03790     }
03791     maxdeg   = mnc->maxdeg;
03792     for (j = 0; j < nNodes; j++) {
03793         ndcl[j]  = mnc->ndcl[j];
03794         clord[j] = mnc->clord[j];
03795         for (k = 0; k < nClasses; k++) {
03796             clmat[j][k] = mnc->clmat[j][k];
03797         }
```

```
03798     }
03799     for (k = 0; k < nClasses+1; k++) {
03800         flist[k] = mnc->flist[k];
03801     }
03802 }
03803
03804 //------------------------------------------------------------
03805 void MNodeClass::mkFlist(void)
03806 {
03807     //  Construct flist
03808     //    The set of nodes in class 'c' is [flist[c],...,flist[c+1]-1]
03809
03810     int  j, f;
03811
03812     f  = 0;
03813     for (j = 0; j < nClasses; j++) {
03814         flist[j] = f;
03815         f += clist[j];
03816     }
03817     flist[nClasses] = f;
03818 }
03819
03820 //------------------------------------------------------------
03821 void MNodeClass::mkNdCl(void)
03822 {
03823     //  Construct ndcl
03824     //    ndcl[nd] = (the class id in which node 'nd' belongs)
03825
03826     int  c, k;
03827     int  nd = 0;
03828
03829     for (c = 0; c < nClasses; c++) {
03830         for (k = 0; k < clist[c]; k++) {
03831             ndcl[nd++] = c;
03832         }
03833     }
03834 }
03835
03836 //------------------------------------------------------------
03837 int MNodeClass::clCmp(int nd0, int nd1, int cn)
03838 {
03839     //  Comparison of two nodes 'nd0' and 'nd1'
03840     //    in lexicographic ordering of [class, connection configuration]
03841
03842     int cmp;
03843     // Wether two nodes are in a same class or not.
03844
03845     cmp = ndcl[nd0] - ndcl[nd1];
03846     if (cmp != 0) {
03847         return cmp;
03848     }
03849
03850     // Sign '-' signifies the reverse ordering
03851     cmp = - cmpMNCArray(clmat[nd0], clmat[nd1], cn);
03852     if (cmp != 0) {
03853         return cmp;
03854     }
03855     return cmp;
03856 }
03857
03858 //------------------------------------------------------------
03859 void MNodeClass::mkClMat(int **adjmat)
03860 {
03861     //  Construct a matrix 'clmat[nd][tc]' which is the number
03862     //  of edges connecting 'nd' and all nodes in class 'tc'.
03863
03864     int  nd, td, tc;
03865
03866     for (nd = 0; nd < nNodes; nd++) {
03867         for (td = 0; td < nNodes; td++) {
03868             tc = ndcl[td];                      // another node
03869             if (nd == td) {
03870                 clmat[nd][tc] += CLWIGHTD(adjmat[nd][td]);
03871             } else {
03872                 clmat[nd][tc] += CLWIGHTO(adjmat[nd][td]);
03873             }
03874         }
03875     }
03876 }
03877
03878 //------------------------------------------------------------
03879 void MNodeClass::incMat(int nd, int td, int val)
03880 {
03881     //  Increase the number of edges by 'val' between 'nd' and 'td'.
03882
03883     int tdc = ndcl[td];             // class of 'td'
03884     clmat[nd][tdc] += val;          // modify matrix 'clmat'.
```

```
03885 }
03886
03887 //-------------------------------------------------------------
03888 void MNodeClass::printMat(void)
03889 {
03890     //  Print configuration matrix.
03891
03892     int j1, j2;
03893
03894     grcc_fprintf(GRCC_Stdout, "\n");
03895
03896     grcc_fprintf(GRCC_Stdout, "nClasses=%d", nClasses);
03897     grcc_fprintf(GRCC_Stdout, " clord="); prIntArray(nClasses, clord, "");
03898     grcc_fprintf(GRCC_Stdout, " flist="); prIntArray(nClasses, flist, "\n");
03899     grcc_fprintf(GRCC_Stdout, "flg = (%d, %d, %d)\n", flg0, flg1, flg2);
03900
03901     // the first line
03902     grcc_fprintf(GRCC_Stdout, "nd: cl:   ");
03903     for (j2 = 0; j2 < nClasses; j2++) {
03904         grcc_fprintf(GRCC_Stdout, "%2d ", j2);
03905     }
03906     grcc_fprintf(GRCC_Stdout, "\n");
03907
03908     // print raw
03909     for (j1 = 0; j1 < nNodes; j1++) {
03910         grcc_fprintf(GRCC_Stdout, "%2d; %2d: [", j1, ndcl[j1]);
03911         for (j2 = 0; j2 < nClasses; j2++) {
03912             grcc_fprintf(GRCC_Stdout, " %2d", clmat[j1][j2]);
03913         }
03914         grcc_fprintf(GRCC_Stdout, "]\n");
03915     }
03916 }
03917
03918 //-------------------------------------------------------------
03919 int MNodeClass::cmpMNCArray(int *a0, int *a1, int ma)
03920 {
03921     int j, k;
03922
03923     for (k = 0; k < ma; k++) {
03924         j = clord[k];
03925         if (a0[j] < a1[j]) {
03926             return -1;
03927         } else if (a0[j] > a1[j]) {
03928             return 1;
03929         }
03930     }
03931     return 0;
03932 }
03933
03934 //-------------------------------------------------------------
03935 void MNodeClass::reorder(MGraph *mg)
03936 {
03937     int flg[GRCC_MAXNODES];
03938     int co, cn;
03939     int f0, f1, f2;
03940
03941     f0 = 0;
03942     f1 = 0;
03943     f2 = 0;
03944     for (co = 0; co < nClasses; co++) {
03945         cn  = flist[co];
03946         if (mg->nodes[cn]->freelg < 1) {
03947             flg[co] = 0;
03948             f0++;
03949         } else if (mg->nodes[cn]->freelg < mg->nodes[cn]->deg) {
03950             flg[co] = mg->nodes[cn]->deg + maxdeg*clist[co];
03951             f1++;
03952         } else {
03953             flg[co] = maxdeg*(mg->nodes[cn]->deg + maxdeg*clist[co]);
03954             f2++;
03955         }
03956     }
03957     if (f0 < nClasses) {
03958         bsort(nClasses, clord, flg);
03959     }
03960     flg0 = f0;
03961     flg1 = f0 + f1;
03962     flg2 = f0 + f1 + f2;
03963 }
03964
03965 //=============================================================
03966 // class MNode : nodes in MGraph
03967 //-------------------------------------------------------------
03968 MNode::MNode(int vid, int vclss, NCInput *mgi)
03969 {
03970     // Arguments
03971     //   vid    : identifier of the vertex
```

```
03972     //   vdeg   : degree of the node
03973     //   vextlp : external node (-1) or looped vertex
03974     //   vclss  : class of the node
03975
03976     id      = vid;          // id of the node
03977     clss    = vclss;        // class to which the node belongs
03978     deg     = mgi->cldeg;   // degree of the node
03979     freelg  = mgi->cldeg;   // number of free legs
03980     extloop = mgi->cltyp;   // external node or not
03981     cmindeg = mgi->cmind;   // min(deg of connectable vertex)
03982     cmaxdeg = mgi->cmaxd;   // max(deg of connectable vertex)
03983 }
03984
03985 //-------------------------------------------------------------
03986 MNode::MNode(int vid, int vdeg, int vextlp, int vclss, int cmin, int cmax)
03987 {
03988     // Arguments
03989     //   vid    : identifier of the vertex
03990     //   vdeg   : degree of the node
03991     //   vextlp : external node (-1) or looped vertex
03992     //   vclss  : class of the node
03993
03994     id      = vid;       // id of the node
03995     deg     = vdeg;      // degree of the node
03996     freelg  = vdeg;      // number of free legs
03997     clss    = vclss;     // class to which the node belongs
03998     extloop = vextlp;    // external node or not
03999     cmindeg = cmin;      // min(deg of connectable vertex)
04000     cmaxdeg = cmax;      // max(deg of connectable vertex)
04001 }
04002
04003 //=============================================================
04004 //  class MGraph : scalar graph expressed by matrix form
04005 //-------------------------------------------------------------
04006 MGraph::MGraph(int pid, int ncl, int *cldeg, int *clnum, int *cltyp, int *cmind, int *cmaxd, Options
      *opts)
04007 {
04008     int nn, ne, j, k;
04009
04010     // initial conditions
04011     nClasses = ncl;
04012     clist    = new int[ncl];
04013     opt      = opts;
04014     mId      = -1;
04015
04016     mindeg   = -1;
04017     maxdeg   = -1;
04018     ne = 0;
04019     nn = 0;
04020     for (j = 0; j < nClasses; j++) {
04021         clist[j] = clnum[j];
04022         nn += clnum[j];
04023         ne += cldeg[j]*clnum[j];
04024         if (mindeg < 0) {
04025             mindeg = cldeg[j];
04026         } else {
04027             mindeg = Min(mindeg, cldeg[j]);
04028         }
04029         if (maxdeg < 0) {
04030             maxdeg = cldeg[j];
04031         } else {
04032             maxdeg = Max(maxdeg, cldeg[j]);
04033         }
04034     }
04035     if (ne % 2 != 0) {
04036         if (prlevel > 0) {
04037             grcc_fprintf(GRCC_Stderr, "Sum of degrees are not even\n");
04038             for (j = 0; j < nClasses; j++) {
04039                 grcc_fprintf(GRCC_Stderr, "class %2d: %2d %2d %2d\n",
04040                         j, cldeg[j], clnum[j], cltyp[j]);
04041             }
04042         }
04043         erEnd("illegal degrees of nodes");
04044     }
04045     pId    = pid;
04046     nNodes = nn;
04047     nodes  = new MNode*[nNodes];
04048     group  = new SGroup();
04049 #ifdef ORBITS
04050     orbits = new MOrbits();
04051 #endif
04052
04053     nEdges = ne / 2;
04054     nLoops = nEdges - nNodes + 1;
04055
04056     egraph = new EGraph(nNodes, nEdges, maxdeg);
04057     nn = 0;
```

```
04058     nExtern = 0;
04059     for (j = 0; j < nClasses; j++) {
04060         for (k = 0; k < clist[j]; k++, nn++) {
04061             nodes[nn] = new MNode(nn, cldeg[j], cltyp[j], j, cmind[j], cmaxd[j]);
04062             egraph->setExtLoop(nn, cltyp[j]);
04063             if (cltyp[j] < 0) {
04064                 nExtern++;
04065             }
04066         }
04067     }
04068     egraph->endSetExtLoop();
04069 
04070     init();
04071 }
04072 
04073 //------------------------------------------------------------------
04074 MGraph::MGraph(int pid, int ncl, NCInput *mgi, Options *opts)
04075 {
04076     int nn, ne, j, k;
04077 
04078     // initial conditions
04079     nClasses = ncl;
04080     clist    = new int[ncl];
04081     opt      = opts;
04082     mId      = -1;
04083 
04084     mindeg   = -1;
04085     maxdeg   = -1;
04086     ne = 0;
04087     nn = 0;
04088     for (j = 0; j < nClasses; j++) {
04089         clist[j] = mgi[j].clnum;
04090         nn += mgi[j].clnum;
04091         ne += mgi[j].cldeg*mgi[j].clnum;
04092         if (mindeg < 0) {
04093             mindeg = mgi[j].cldeg;
04094         } else {
04095             mindeg = Min(mindeg, mgi[j].cldeg);
04096         }
04097         if (maxdeg < 0) {
04098             maxdeg = mgi[j].cldeg;
04099         } else {
04100             maxdeg = Max(maxdeg, mgi[j].cldeg);
04101         }
04102     }
04103     if (ne % 2 != 0) {
04104         if (prlevel > 0) {
04105             grcc_fprintf(GRCC_Stderr, "Sum of degrees are not even\n");
04106             for (j = 0; j < nClasses; j++) {
04107                 grcc_fprintf(GRCC_Stderr, "class %2d: %2d %2d %2d\n",
04108                         j, mgi[j].cldeg, mgi[j].clnum, mgi[j].cltyp);
04109             }
04110         }
04111         erEnd("illegal degrees of nodes");
04112     }
04113     pId    = pid;
04114     nNodes = nn;
04115     if (nNodes < 0) {
04116         grcc_fprintf(GRCC_Stdout, "*** nNodes = %d\n", nNodes);
04117         erEnd("MGraph::MGrap : nNodes < 0");
04118     }
04119     nodes  = new MNode*[nNodes];
04120     group  = new SGroup();
04121 #ifdef ORBITS
04122     orbits = new MOrbits();
04123 #endif
04124 
04125     nEdges = ne / 2;
04126     nLoops = nEdges - nNodes + 1;
04127 
04128     egraph = new EGraph(nNodes, nEdges, maxdeg);
04129     nn = 0;
04130     nExtern = 0;
04131     for (j = 0; j < nClasses; j++) {
04132         for (k = 0; k < clist[j]; k++, nn++) {
04133             nodes[nn] = new MNode(nn, j, mgi+j);
04134             egraph->setExtLoop(nn, mgi[j].cltyp);
04135             if (mgi[j].cltyp < 0) {
04136                 nExtern++;
04137             }
04138         }
04139     }
04140     egraph->endSetExtLoop();
04141 
04142     init();
04143 }
04144 
```

```
04145 //-------------------------------------------------------------
04146 void MGraph::init(void)
04147 {
04148     // generated set of graphs
04149     cDiag          = 0;
04150     c1PI           = 0;
04151     cNoTadpole     = 0;
04152     cNoTadBlock    = 0;
04153     c1PINoTadBlock = 0;
04154
04155     // the current graph
04156     adjMat    = newMat(nNodes, nNodes, 0);
04157     nsym      = ToBigInt(0);
04158     esym      = ToBigInt(0);
04159     wscon     = Fraction(0, 1);
04160     wsopi     = Fraction(0, 1);
04161
04162     // current node classification
04163     curcl     = new MNodeClass(nNodes, nClasses);
04164
04165     // table of n-edge connected components
04166     mconn     = new MConn(nNodes, nEdges);
04167
04168     // measures of efficiency
04169     ngen      = 0;
04170     ngconn    = 0;
04171
04172 #ifdef MONITOR
04173     nCallRefine    = 0;
04174     discardRefine  = 0;
04175     discardDisc    = 0;
04176     discardIso     = 0;
04177 #endif
04178
04179     // work space for isIsomorphic
04180     modmat = newMat(nNodes, nNodes, 0);
04181
04182     // work space for bisearchME
04183     bidef = newArray(nNodes, 0);
04184     bilow = newArray(nNodes, 0);
04185     bicol = newArray(nNodes, 0);
04186     bicount = 0;
04187 }
04188
04189 //-------------------------------------------------------------
04190 MGraph::~MGraph(void)
04191 {
04192     int j;
04193
04194     // group->delGroup();
04195
04196     bicol = deleteArray(bicol);
04197     bilow = deleteArray(bilow);
04198     bidef = deleteArray(bidef);
04199
04200
04201     modmat = deleteMat(modmat, nNodes);
04202     delete mconn;
04203     delete curcl;
04204     adjMat = deleteMat(adjMat, nNodes);
04205 #ifdef ORBITS
04206     delete orbits;
04207 #endif
04208     delete egraph;
04209     delete group;
04210     for (j = 0; j < nNodes; j++) {
04211         delete nodes[j];
04212         nodes[j] = NULL;
04213     }
04214     delete[] nodes;
04215     delete[] clist;
04216
04217 }
04218
04219 //-------------------------------------------------------------
04220 void MGraph::printAdjMat(MNodeClass *cl)
04221 {
04222     int j1, j2;
04223
04224     grcc_fprintf(GRCC_Stdout, "      ");
04225     for (j2 = 0; j2 < nNodes; j2++) {
04226         grcc_fprintf(GRCC_Stdout, " %2d", j2);
04227     }
04228     grcc_fprintf(GRCC_Stdout, "\n");
04229     for (j1 = 0; j1 < nNodes; j1++) {
04230         grcc_fprintf(GRCC_Stdout, "%2d : [", j1);
04231         for (j2 = 0; j2 < nNodes; j2++) {
```

```
04232             grcc_fprintf(GRCC_Stdout, " %2d", adjMat[j1][j2]);
04233         }
04234         grcc_fprintf(GRCC_Stdout, "] %2d\n", cl->ndcl[j1]);
04235     }
04236 }
04237
04238 //-------------------------------------------------------------
04239 void MGraph::print(void)
04240 {
04241     int j;
04242
04243     grcc_fprintf(GRCC_Stdout, "MGraph: pId=%d, cDiag=%ld, c1PI=%ld\n", pId, cDiag, c1PI);
04244     grcc_fprintf(GRCC_Stdout, "     nNodes=%d, nEdges=%d, nExtern=%d, nLoops=%d "
04245            "mindeg=%d, maxdeg=%d, sym=(%ld, %ld)\n",
04246            nNodes, nEdges, nExtern, nLoops, mindeg, maxdeg, nsym, esym);
04247     grcc_fprintf(GRCC_Stdout, "  Nodes=%d\n", nNodes);
04248     for (j = 0; j < nNodes; j++) {
04249         grcc_fprintf(GRCC_Stdout, "    %2d: id=%d, deg=%d, clss=%d, extloop=%d, ",
04250                j, nodes[j]->id, nodes[j]->deg, nodes[j]->clss,
04251                nodes[j]->extloop);
04252         grcc_fprintf(GRCC_Stdout, "mind=%d, maxd=%d, freelg=%d\n",
04253                nodes[j]->cmindeg, nodes[j]->cmaxdeg, nodes[j]->freelg);
04254     }
04255
04256     curcl->printMat();
04257     grcc_fprintf(GRCC_Stdout, "\n");
04258
04259     printAdjMat(curcl);
04260 }
04261
04262 //-------------------------------------------------------------
04263 void MGraph::printPy(FILE *fp, long mId)
04264 {
04265     fprintf(fp, "#TGraph : (gseq, gid, nodes, amat)\n");
04266     fprintf(fp, "(%ld, (%d, %d),\n", mId, pId, -1);
04267     fprintf(fp, "  [ # nodes: (cid, deg0, loop, part)\n");
04268     for (int j = 0; j < nNodes; j++) {
04269         fprintf(fp, "    (%2d,%2d,%2d,%2d),   # %2d\n",
04270                 nodes[j]->clss, nodes[j]->deg, nodes[j]->extloop, 0, j);
04271     }
04272     fprintf(fp, "  ],\n");
04273     fprintf(fp, "  [\n");
04274     fprintf(fp, "  #");
04275     for (int j2 = 0; j2 < nNodes; j2++) {
04276         fprintf(fp, " %4d", j2);
04277     }
04278     fprintf(fp, "\n");
04279     for (int j1 = 0; j1 < nNodes; j1++) {
04280         fprintf(fp, "    [");
04281         for (int j2 = 0; j2 < nNodes; j2++) {
04282             fprintf(fp, " %2d, ", adjMat[j1][j2]);
04283         }
04284         fprintf(fp, "],   # %2d\n", j1);
04285     }
04286     fprintf(fp, "  ], # nodes\n");
04287     fprintf(fp, "  [ # group of 2 elements\n");
04288     fprintf(fp, "    [],\n");
04289     fprintf(fp, "  ], # group\n");
04290     fprintf(fp, ")\n");
04291 }
04292
04293 //-------------------------------------------------------------
04294 Bool MGraph::isConnected(void)
04295 {
04296     //  Check graph can be a connected one.
04297     //  If a connected component without free leg is not the whole graph then
04298     //  return False, otherwise return True.
04299
04300     int j, n, nv;
04301
04302     for (j = 0; j < nNodes; j++) {
04303         nodes[j]->visited = -1;
04304     }
04305     if (visit(0)) {
04306         return True;
04307     }
04308     nv = 0;
04309     for (n = 0; n < nNodes; n++) {
04310         if (nodes[n]->visited >= 0) {
04311             nv++;
04312         }
04313     }
04314     return (nv == nNodes);
04315 }
04316
04317 //-------------------------------------------------------------
04318 Bool MGraph::visit(int nd)
```

```
04319 {
04320     //  Visiting connected node used for 'isConnected'
04321     //  If child nodes has free legs, then this function returns True.
04322     //  otherwise it returns False.
04323
04324     int td;
04325
04326     // This node has free legs.
04327     if (nodes[nd]->freelg > 0) {
04328         return True;
04329     }
04330     nodes[nd]->visited = 0;
04331     for (td = 0; td < nNodes; td++) {
04332         if ((adjMat[nd][td] > 0) && (nodes[td]->visited < 0)) {
04333             if (visit(td)) {
04334                 return True;
04335             }
04336         }
04337     }
04338     // all the child nodes has no free legs.
04339     return False;
04340 }
04341
04342 //------------------------------------------------------------------
04343 Bool MGraph::isIsomorphic(MNodeClass *cl)
04344 {
04345     //  Check whether the current graph is the Representative
04346     //  of a isomorphic class.
04347     //    nsym = symmetry factor by the permutation of nodes.
04348     //    esym = symmetry factor by the permutation of edge.
04349     //  If this graph is not a representative, then returns False.
04350
04351     int j1, j2, cmp, nself;
04352     int *perm;
04353
04354     nsym = ToBigInt(0);
04355     esym = ToBigInt(1);
04356
04357     group->newGroup(nNodes, cl->nClasses, cl->clist);
04358
04359 #ifdef ORBITS
04360     orbits->initPerm(nNodes);
04361 #endif
04362
04363     while (True) {
04364         perm = group->genNext();
04365
04366         if (perm == NULL) {
04367             // calculate permutations of edges
04368             esym = ToBigInt(1);
04369             for (j1 = 0; j1 < nNodes; j1++) {
04370                 if (adjMat[j1][j1] > 0) {
04371                     nself = adjMat[j1][j1]/2;
04372                     esym *= factorial(nself);
04373                     esym *= ipow(2, nself);
04374                 }
04375                 for (j2 = j1+1; j2 < nNodes; j2++) {
04376                     if (adjMat[j1][j2] > 0) {
04377                         esym *= factorial(adjMat[j1][j2]);
04378                     }
04379                 }
04380             }
04381             return True;
04382         }
04383
04384         permMat(nNodes, perm, adjMat, modmat);
04385         cmp = compMat(nNodes, adjMat, modmat);
04386         if (cmp < 0) {
04387             return False;
04388         } else if (cmp == 0) {
04389             // if ngroup < 0, symmetry group is $S_{-group}$.
04390             group->addGroup(perm);
04391
04392             nsym = nsym + ToBigInt(1);
04393 #ifdef ORBITS
04394             orbits->fromPerm(perm);
04395 #endif
04396         }
04397     }
04398 }
04399
04400 //------------------------------------------------------------------
04401 void MGraph::permMat(int size, int *perm, int **mat0, int **mat1)
04402 {
04403     // apply permutation to matrix 'mat0' and obtain 'mat1'
04404
04405     int j1, j2;
```

```
04406
04407     for (j1 = 0; j1 < size; j1++) {
04408         for (j2 = 0; j2 < size; j2++) {
04409             mat1[j1][j2] = mat0[perm[j1]][perm[j2]];
04410         }
04411     }
04412 }
04413
04414 //-------------------------------------------------------------
04415 int MGraph::compMat(int size, int **mat0, int **mat1)
04416 {
04417     // comparison of matrix
04418
04419     int j1, j2, cmp;
04420
04421     for (j1 = 0; j1 < size; j1++) {
04422         for (j2 = 0; j2 < size; j2++) {
04423             cmp = mat0[j1][j2] - mat1[j1][j2];
04424             if (cmp != 0) {
04425                 return cmp;
04426             }
04427         }
04428     }
04429     return 0;
04430 }
04431
04432 //-------------------------------------------------------------
04433 MNodeClass *MGraph::refineClass(MNodeClass *cl)
04434 {
04435     //  Refine the classification
04436     //    cl : the current 'MNodeClass' object
04437     //    cn : the class number
04438     //  Returns (the new class number corresponds to 'cn') if OK, or
04439     //          'None' if ordering condition is not satisfied
04440
04441     MNodeClass *ccl = cl;
04442     MNodeClass *xcl = NULL;
04443     MNodeClass *ncl = NULL;
04444     int         ucl[GRCC_MAXNODES];
04445     int         ccn = cl->nClasses;
04446     int         nucl, nce;
04447     int         td, cmp;
04448     int         nccl;
04449
04450 #ifdef MONITOR
04451     nCallRefine++;
04452 #endif
04453     nucl = 0;
04454     while (ccl->nClasses != nucl) {    // repeat refinement.
04455         nce  = 0;
04456         nucl = 0;
04457         for (td = 1; td < nNodes; td++) {
04458           // 'td' is the next node and the current node is 'td-1'.
04459           // Count up the number of the elements in the current class
04460           // corresponding to the current node
04461           nce++;
04462           cmp = ccl->clCmp(td-1, td, ccn);
04463
04464           // the ordering condition is not satisfied.
04465           if (cmp > 0) {
04466             if (ccl != cl) {
04467                 delete ccl;
04468             }
04469             return NULL;
04470
04471           } else if (cmp < 0) {
04472               // 'td' is in the next class to the current node.
04473               ucl[nucl++] = nce;       // close the current class
04474
04475               // start new class
04476               nce = 0;
04477           }
04478           // nothing to do for the case of 'cmp == 0'.
04479
04480         }
04481
04482         // close array 'ucl'.
04483         ucl[nucl++] = nce + 1;
04484
04485 #ifdef CHECK
04486         // inconsistent
04487         if (nucl < ccl->nClasses) {
04488             erEnd("refineClasses : smaller number of classes");
04489         }
04490 #endif
04491
04492         // preparation of the next repetition.
```

```
04493         xcl = new MNodeClass(nNodes, nucl);
04494         xcl->init(ucl, maxdeg, adjMat);
04495         ccn = xcl->nClasses;
04496 
04497         nccl = ccl->nClasses;
04498 
04499         if (ccl != cl) {
04500             delete ccl;
04501         }
04502         ccl = xcl;
04503         if (ccl->nClasses == nccl) {
04504             ccl->reorder(this);
04505             return ccl;
04506         }
04507 
04508         nucl = 0;
04509     }
04510 
04511     return ncl;
04512 }
04513 
04514 //-------------------------------------------------------------------
04515 void MGraph::biconnME(void)
04516 {
04517     //  Count the number of 1PI components.
04518 
04519     int j, j1, root, vr, next, nart;
04520     MCOpi   mopi;
04521     MCBlock mblk;
04522     ULong   momset;
04523 
04524     // initialization
04525     bicount    = 0;
04526     mconn->init();
04527     mconn->initCEdges(this);
04528 
04529     for (j = 0; j < nNodes; j++) {
04530         bidef[j] = -1;
04531         bilow[j] = -1;
04532         bicol[j] =  0;
04533     }
04534     bipart = True;
04535 
04536     // find a root for the root of bisearch
04537     // root must be an external node
04538     // or an vertex when there are not external nodes.
04539     root = -1;
04540     vr   = -1;
04541     for (j = 0; j < nNodes; j++) {
04542         if (bidef[j] < 0) {
04543             if (isExternal(j)) {
04544                 root = j;
04545                 break;
04546             } else if (vr < 0) {
04547                 vr = j;
04548             }
04549         }
04550     }
04551     // case (2)
04552     if (root < 0) {
04553         root = vr;
04554     }
04555 
04556     if (root >= 0) {
04557         bisearchME(root, -1, 0, 1, &mopi, &mblk, &momset, &next, &nart);
04558 
04559         mconn->nlpopic = 0;
04560         mconn->nctopic = 0;
04561         for (j = 0; j < mconn->nopic; j++) {
04562             if (mconn->opics[j].loop > 0) {
04563                 mconn->nlpopic++;
04564             } else if (mconn->opics[j].ctloop > 0) {
04565                 mconn->nctopic++;
04566             }
04567         }
04568 
04569         mconn->ne0bridges  = 0;
04570         mconn->ne1bridges   = 0;
04571         for (j = 0; j < mconn->nbridges; j++) {
04572             if (mconn->bridges[j].next == 0) {
04573                 mconn->ne0bridges++;
04574             }
04575             if (mconn->bridges[j].next == 1) {
04576                 mconn->ne1bridges++;
04577             }
04578         }
04579 
```

```
04580         mconn->nselfloops = 0;
04581         for (j = 0; j < nNodes; j++) {
04582             mconn->nselfloops += adjMat[j][j]/2;
04583         }
04584 
04585         mconn->nmultiedges = 0;
04586         for (j = 0; j < nNodes; j++) {
04587             for (j1 = j+1; j1 < nNodes; j1++) {
04588                 if (adjMat[j][j1] > 1) {
04589                     mconn->nmultiedges++;
04590                 }
04591             }
04592         }
04593 
04594         mconn->na1blocks = 0;
04595         for (j = 0; j < mconn->nblocks; j++) {
04596             if (mconn->blocks[j].nartps == 1) {
04597                 mconn->na1blocks++;
04598             }
04599         }
04600         mconn->neblocks = mconn->nblocks + mconn->nctopic;
04601 
04602 #ifdef CHECK
04603         if (True) {
04604             if (isExternal(root)) {
04605                 momset |= MASK(root);
04606             }
04607             ULong m = 0;
04608             for (j = 0; j < nExtern; j++) {
04609                 m |= MASK(j);
04610             }
04611             bool ok = True;
04612             if (momset != m) {
04613                 grcc_fprintf(GRCC_Stdout, "*** momset = %ld != %ld\n", momset, m);
04614                 ok = False;
04615             }
04616             if (mconn->nbacked != nLoops) {
04617                 grcc_fprintf(GRCC_Stdout, "*** nbacked = %d != %c\n", mconn->nbacked, nLoops);
04618                 ok = False;
04619             }
04620             if (! ok) {
04621                 print();
04622                 egraph->print();
04623                 mconn->print();
04624                 erEnd("biconnME: illegal connection");
04625             }
04626         }
04627 #endif
04628 
04629     // no vertex.
04630     } else {
04631         // no vertices
04632         // Connected ==> only when ex=2, loop=0
04633         mconn->nopic      = 0;
04634         mconn->ne0bridges = 0;
04635         mconn->ne1bridges = 0;
04636     }
04637 }
04638 
04639 //-------------------------------------------------------------------
04640 void MGraph::bisearchME(int nd, int pd, int ned,  int col,
04641          MCOpi *mopi, MCBlock *mblk, ULong *momset, int *next, int *nart)
04642 {
04643     //  Search biconnected component
04644     //    visit : pd --> nd --> td
04645     //    ned : the number of edges between pd and nd.
04646     //  Output
04647     //
04648     //                 |
04649     //                 x----+
04650     //                 |    |
04651     //              ----x pd |
04652     //                 |    |
04653     //            x----x nd |  n art. pints.
04654     //            |   /|\   |
04655     //            |  / | \  |
04656     //   m art p. x-x  |  x-x
04657     //              n1  x  n3
04658     //                  n2
04659     //
04660     //   output of bisearchME(n1, nd, ...) ==> npart = m
04661     //   output of bisearchME(n2, nd, ...) ==> npart = 1
04662     //   output of bisearchME(n3, nd, ...) ==> npart = n
04663     //
04664     //   output of bisearchME(nd, pd, ...) ==> npart = n+1
04665     //       n1 => block of (m+1) articulation points
04666     //       n2 => block of    2  articulation points
```

```
04667     //       n3 => no block
04668     //
04669     Bool newv;
04670     int  td, td0, lp, conn, next1, nart1, nart2, nvisit, ndart;
04671     int  opit, opit0, blkt, l;
04672     MCOpi   mopi1;
04673     MCBlock mblk1;
04674     ULong   momset1, momset2, momset3;
04675
04676     mopi->init();
04677     mblk->init();
04678
04679     *momset = 0;     // # the set of momenta
04680     *next   = 0;     // # external below 'nd' inclusive
04681     *nart   = 0;     // # articulation points 'nd' inclusive
04682     nart1   = 0;     // # articulation points below 'nd'
04683     nart2   = 0;     // # 'nd' is articulation point then 1
04684     ndart   = 0;     // # the number of blocks attached to 'nd'
04685
04686     newv = (bidef[nd] < 0);
04687     if (! newv) {
04688         grcc_fprintf(GRCC_Stdout, "*** nd=%d, pd=%d, bidef[nd]=%d\n", nd, pd, bidef[nd]);
04689         erEnd("bisearchME: illegal connection");
04690     }
04691
04692     bidef[nd] = bicount;
04693     bilow[nd] = bicount;
04694     bicol[nd] = col;
04695     bicount++;
04696
04697     opit0 = mconn->opistkptr;
04698
04699     if (pd < 0) {
04700         mconn->pushNode(nd);
04701     }
04702
04703     // external node and not the root
04704     if (isExternal(nd)) {
04705         if (pd >= 0) {
04706             // not the root
04707             *momset      = MASK(nd);
04708             mopi->next   = 1;
04709             mopi->nlegs  = 1;
04710             mopi->mom0lg = 0;
04711             mblk->nartps = 1;
04712             *next        = 1;
04713             *nart        = 1;
04714             mconn->addCEdge(nd, pd, *momset);
04715             return;
04716         }
04717     }
04718
04719     nvisit = 0;
04720     td0    = -1;
04721     for (td = 0; td < nNodes; td++) {
04722         conn = adjMat[nd][td];
04723
04724         // td is not adjacent to nd
04725         if (conn < 1) {
04726             continue;
04727         }
04728         nvisit++;
04729         td0 = td;
04730
04731         momset1 = 0;
04732
04733         // external node
04734         if (isExternal(td) && bidef[td] < 0) {
04735             bicol[td] = - col;
04736             (*next)++;
04737             ndart++;
04738             mconn->addArtic(nd, 1);
04739             mblk->nartps++;
04740             nart2 = 1;
04741             momset1  = MASK(td);
04742             mopi->nlegs++;
04743             mopi->next++;
04744             mconn->addCEdge(td, nd, momset1);
04745             *momset |= momset1;
04746             if (newv) {
04747                 mconn->pushNode(td);
04748             }
04749
04750         // self-loop : pd --> nd --> nd
04751         } else if (td == nd) {
04752             lp  = conn/2;
04753             mopi->loop   += lp;
```

```
04754             mopi->nedges += lp;
04755             ndart += lp;
04756             bipart = False;
04757             mconn->addArtic(nd, lp);
04758             mconn->addBlockSelf(nd, lp);
04759             mblk->nartps++;
04760             nart2  = 1;
04761             for (l = 0; l < lp; l++) {
04762                 int m = nExtern + mconn->nbacked;
04763                 mconn->nbacked++;
04764                 momset1 = MASK(m);
04765                 mconn->addCEdge(td, nd, momset1);
04766             }
04767 
04768         // back to the parent : pd --> nd --> pd,  pd is a vertex
04769         } else if (td == pd) {
04770             if (ned > 1) {
04771                 bilow[nd] = Min(bilow[nd], bidef[pd]);
04772                 mopi->loop += ned - 1;
04773                 mblk->loop += ned - 1;
04774             } else {
04775                 // cancel this visit
04776                 nvisit--;
04777             }
04778 
04779         // td is already visited
04780         } else if (bidef[td] >= 0) {
04781             bipart &= (bicol[td] == - col);
04782             bilow[nd] = Min(bilow[nd], bidef[td]);
04783             if (bidef[nd] >= bidef[td]) {
04784                 // new back-edge is found
04785                 mopi->loop   += conn;
04786                 mopi->nedges += conn;
04787                 mblk->loop   += conn;
04788                 mconn->pushEdge(td, nd);
04789                 for (l = 0; l < conn; l++) {
04790                     int m = nExtern + mconn->nbacked;
04791                     mconn->nbacked++;
04792                     momset1 = MASK(m);
04793                     *momset |= momset1;
04794                     mconn->addCEdge(td, nd, momset1);
04795                 }
04796             } else {
04797                 // reverse direction of back-edge
04798                 int cn = 0;
04799 
04800                 for (int ed = 0; ed < mconn->sedges; ed++) {
04801                     if (mconn->cedges[ed].nodes[0] == nd &&
04802                         mconn->cedges[ed].nodes[1] == td &&
04803                         mconn->cedges[ed].momdir   > 0) {
04804                         cn++;
04805                         *momset &= ~ (mconn->cedges[ed].momset);
04806                     } else if (mconn->cedges[ed].nodes[1] == nd &&
04807                                mconn->cedges[ed].nodes[0] == td &&
04808                                mconn->cedges[ed].momdir   < 0) {
04809                         cn++;
04810                         *momset &= ~ (mconn->cedges[ed].momset);
04811                     }
04812                 }
04813                 if (cn != conn) {
04814                     grcc_fprintf(GRCC_Stdout, "*** revisit back-ed (%d --> %d) cn=%d != conn=%d\n",
04815                            nd, td, cn, conn);
04816                     grcc_fprintf(GRCC_Stdout, "    sedges = %d\n", mconn->sedges);
04817                     for (int ed = 0; ed < mconn->sedges; ed++) {
04818                         grcc_fprintf(GRCC_Stdout, "    %d : (%d --> %d) : [%2d*%2ld]\n", ed,
04819                             mconn->cedges[ed].nodes[0],
04820                             mconn->cedges[ed].nodes[1],
04821                             mconn->cedges[ed].momdir,
04822                             mconn->cedges[ed].momset);
04823                     }
04824                     print();
04825                     egraph->print();
04826                     mconn->print();
04827                     erEnd("bisearchME: illegal connection");
04828                 }
04829             }
04830 
04831         // new node
04832         } else if (bidef[td] < 0) {
04833 
04834             // stack pointer
04835             if (pd < 0 && isExternal(nd)) {
04836                 opit = opit0;
04837             } else {
04838                 opit = mconn->opistkptr;
04839             }
04840             blkt = mconn->blkstkptr;
```

```
04841             mblk1.init();
04842
04843             mconn->pushEdge(nd, td);
04844
04845             // visit child
04846             bisearchME(td, nd, adjMat[td][nd], - col,
04847                 &mopi1, &mblk1, &momset1, &next1, &nart1);
04848
04849             // momset
04850             momset2 = 0;
04851             for (l = 0; l < conn - 1; l++) {
04852                 int m = nExtern + mconn->nbacked;
04853                 mconn->nbacked++;
04854                 momset3 = MASK(m);
04855                 mconn->addCEdge(nd, td, momset3);
04856                 momset2 |= momset3;
04857             }
04858             mconn->addCEdge(td, nd, momset1 | momset2);
04859             *momset |= momset1;
04860             // articulation point of bridge
04861             if (bilow[td] > bidef[nd]) {
04862
04863                 // new OPI component (not including 'td')
04864                 int mom0lg = (momset1 == 0) ? 1 : 0;
04865                 mopi1.nlegs++;
04866                 mopi1.mom0lg += mom0lg;
04867
04868                 mconn->addOPIc(&mopi1, opit);
04869
04870                 mopi1.init();
04871                 mopi1.nlegs  = 1;
04872                 mopi1.mom0lg = mom0lg;
04873                 if (pd >= 0 || !isExternal(nd)) {
04874                     // opi
04875                     mconn->addBridge(nd, td, next1, nExtern);
04876
04877                     // block
04878                     ndart++;
04879                     mconn->addArtic(nd, 1);
04880                     // discard nparts from nodes below td
04881                     mblk1.nartps = 2;
04882                     mconn->addBlock(&mblk1, blkt);
04883                     mblk1.init();
04884                     mblk1.nartps = 1;
04885
04886                     nart1 = 0;
04887                     nart2 = 1;  // nd is an articulation point
04888                 }
04889             } else if (bilow[td] == bidef[nd]) {
04890                 // articulation point of a block
04891                 mopi1.nedges += conn;
04892                 if (pd >= 0 || !isExternal(nd)) {
04893                     ndart++;
04894                     mconn->addArtic(nd, 1);
04895                     mblk1.nartps++;
04896                     mconn->addBlock(&mblk1, blkt);
04897                     mblk1.init();
04898                     mblk1.nartps = 1;
04899                     nart1 = 0;
04900                     nart2 = 1;  // nd is an articulation point
04901                 }
04902
04903             } else {
04904                 // inside a block
04905                 mopi1.nedges += conn;
04906             }
04907
04908             bilow[nd] = Min(bilow[nd], bilow[td]);
04909             mopi->nlegs  += mopi1.nlegs;
04910             mopi->next   += mopi1.next;
04911             mopi->loop   += mopi1.loop;
04912             mopi->nedges += mopi1.nedges;
04913             mopi->ctloop += mopi1.ctloop;
04914             mopi->mom0lg += mopi1.mom0lg;
04915
04916             mblk->nartps += mblk1.nartps;
04917             mblk->loop   += mblk1.loop;
04918
04919             *next   += next1;
04920             *nart   += nart1;
04921         }  // of "if (bidef[td] < 0)"
04922     }  // end of for
04923
04924     // no connection except for 'pd'==>'nd'.  Then 'nd' is an art. point
04925     if (nvisit == 0) {
04926         nart2 = 1;
04927     }
```

```
04928     *nart += nart2;
04929
04930     // add # loop inside counter terms
04931     if (newv) {
04932         if (pd >= 0) {
04933             mconn->pushNode(nd);
04934         }
04935         mopi->ctloop += Max(0, nodes[nd]->extloop);
04936     }
04937
04938     // 'nd' is the root node
04939     if (pd < 0) {
04940         if (isExternal(nd)) {
04941             if (td0 >= 0) {
04942                 mconn->addArtic(td0, 1);
04943             }
04944             if (mconn->nopic > 0) {
04945                 (mconn->opics[mconn->nopic-1].next)++;
04946             }
04947         } else {
04948             mconn->addOPIc(mopi, opit0);
04949             if (ndart == 1) {
04950                 // 'nd' has only 1 child
04951                 // the root is not really an art. point
04952                 (*nart)--;
04953                 mconn->addArtic(nd, -1);
04954                 (mconn->blocks[mconn->nblocks-1].nartps)--;
04955             }
04956         }
04957     }
04958 }
04959
04960 //-------------------------------------------------------------------
04961 BigInt MGraph::generate(void)
04962 {
04963     //  Generate graphs
04964     //    the generation process starts from 'connectClass'.
04965
04966     MNodeClass *cl;
04967
04968     // Initial classification of nodes.
04969     cl = new MNodeClass(nNodes, nClasses);
04970     cl->init(clist, maxdeg, adjMat);
04971     cl->reorder(this);
04972     connectClass(cl);
04973
04974     delete cl;
04975
04976     return cDiag;
04977 }
04978
04979 //-------------------------------------------------------------------
04980 void MGraph::connectClass(MNodeClass *cl)
04981 {
04982     // Connect nodes in a class to others
04983
04984     int sc, sn;
04985     MNodeClass *xcl;
04986
04987     xcl = refineClass(cl);
04988
04989     if (xcl == NULL) {
04990 #ifdef MONITOR
04991         discardRefine++;
04992 #endif
04993     } else {
04994         if (xcl->flg0 >= xcl->nClasses) {
04995             newGraph(xcl);
04996         } else {
04997             sc = xcl->clord[xcl->flg0];
04998             sn = xcl->flist[sc];
04999             connectNode(xcl->flg0, sn, xcl);
05000         }
05001     }
05002     if (xcl != cl && xcl != NULL) {
05003         delete xcl;
05004     }
05005 }
05006
05007 //-------------------------------------------------------------
05008 void MGraph::connectNode(int so, int ss, MNodeClass *cl)
05009 {
05010     int sc, sn;
05011
05012     sc = cl->clord[so];
05013     if (ss >= cl->flist[sc+1]) {
05014         connectClass(cl);
```

```
05015         return;
05016     }
05017
05018     for (sn = ss; sn < cl->flist[sc+1]; sn++) {
05019         connectLeg(so, sn, so, sn, cl);
05020         return;
05021     }
05022 }
05023
05024 //------------------------------------------------------------
05025 void MGraph::connectLeg(int so, int sn, int to, int ts, MNodeClass *cl)
05026 {
05027     //  Add one connection between two legs.
05028     //    1. select another node to connect
05029     //    2. determine multiplicity of the connection
05030
05031     int sc, tc, tn, maxself, nc2, nc, maxcon, ts1, ncm, to1;
05032
05033     sc = cl->clord[so];
05034
05035 #ifdef CHECK
05036     if (sn >= cl->flist[sc+1]) {
05037         erEnd("connectLeg : illegal control");
05038         return;
05039     }
05040 #endif
05041
05042     // There remains no free legs in the node 'sn' : move to next node.
05043     if (nodes[sn]->freelg < 1) {
05044
05045         if (!isConnected()) {
05046 #ifdef MONITOR
05047             discardDisc++;
05048 #endif
05049         } else {
05050             // next node in the current class.
05051             connectNode(so, sn+1, cl);
05052         }
05053         return;
05054     }
05055
05056     // connect a free leg of the current node 'sn'.
05057     for (to1 = to; to1 < cl->nClasses; to1++) {
05058         tc = cl->clord[to1];
05059         if (to1 == to) {
05060             ts1 = ts;
05061         } else {
05062             ts1 = cl->flist[tc];
05063         }
05064 #ifdef MINMAXLEG
05065         if (ts1 >= nNodes) {
05066             continue;
05067         }
05068         if ((nodes[sn]->cmindeg > 0 && nodes[ts1]->deg < nodes[sn]->cmindeg)
05069           ||(nodes[sn]->cmaxdeg > 0 && nodes[ts1]->deg > nodes[sn]->cmaxdeg)
05070           | (nodes[ts1]->cmindeg > 0 && nodes[sn]->deg < nodes[ts1]->cmindeg)
05071           ||(nodes[ts1]->cmaxdeg > 0 && nodes[sn]->deg > nodes[ts1]->cmaxdeg)) {
05072             continue;
05073         }
05074 #endif
05075         for (tn = ts1; tn < cl->flist[tc+1]; tn++) {
05076             if (sc == tc && sn > tn) {
05077                 continue;
05078
05079             } else if (nodes[tn]->freelg < 1) {
05080                 continue;
05081
05082             // self-loop
05083             } else if (sn == tn) {
05084                 if (opt->values[GRCC_OPT_NoSelfLoop] > 0) {
05085                     continue;
05086                 }
05087                 if (nNodes > 1) {
05088                     // there are two or more nodes in the graph :
05089                     // avoid disconnected graph
05090                     maxself = Min((nodes[sn]->freelg)/2, (nodes[sn]->deg-1)/2);
05091                 } else {
05092                     // there is only one node the graph.
05093                     maxself = nodes[sn]->freelg/2;
05094                 }
05095
05096                 if ((nExtern > 1) && (opt->values[GRCC_OPT_1PI] > 0
05097                     || opt->values[GRCC_OPT_NoTadpole] > 0)
05098                     && nNodes > 1) {
05099                     // there are two or more nodes in the graph :
05100                     // avoid to generate tadpole
05101                     maxself = Min((nodes[sn]->deg-2)/2, maxself);
```

```
05102                 }
05103
05104                 // for the number of connection.
05105                 for (nc2 = maxself; nc2 > 0; nc2--) {
05106                     nc = 2*nc2;
05107                     ncm = CLWIGHTD(nc);
05108                     adjMat[sn][sn] = nc;
05109                     nodes[sn]->freelg -= nc;
05110                     cl->incMat(sn, tn, ncm);
05111
05112                     // next connection
05113                     connectLeg(so, sn, to1, tn+1, cl);
05114
05115                     // restore the configuration
05116                     cl->incMat(tn, sn, - ncm);
05117                     adjMat[sn][sn] = 0;
05118                     nodes[sn]->freelg += nc;
05119                 }
05120
05121             // connections between different nodes.
05122             } else {
05123                 // maximum possible connection number
05124                 maxcon = Min(nodes[sn]->freelg, nodes[tn]->freelg);
05125
05126                 // avoid disconnected graphs.
05127                 if (nNodes > 2 && nodes[sn]->deg == nodes[tn]->deg) {
05128                     maxcon = Min(maxcon, nodes[sn]->deg-1);
05129                 }
05130
05131                 if (opt->values[GRCC_OPT_NoMultiEdge] > 0) {
05132                     maxcon = Min(maxcon, 1);
05133                 }
05134
05135 #ifdef CHECK
05136                 if ((adjMat[sn][tn] != 0) ||
05137                     (adjMat[sn][tn] != adjMat[tn][sn])) {
05138                     grcc_fprintf(GRCC_Stdout, "*** inconsistent connection: sn=%d, tn=%d",
05139                            sn, tn);
05140                     printAdjMat(cl);
05141                     erEnd("inconsistent connection ");
05142                 }
05143 #endif
05144
05145                 // vary number of connections
05146                 for (nc = maxcon; nc > 0; nc--) {
05147                     ncm = CLWIGHTO(nc);
05148                     adjMat[sn][tn] = nc;
05149                     adjMat[tn][sn] = nc;
05150                     nodes[sn]->freelg -= nc;
05151                     nodes[tn]->freelg -= nc;
05152                     cl->incMat(sn, tn, ncm);
05153                     cl->incMat(tn, sn, ncm);
05154
05155                     // next connection
05156                     connectLeg(so, sn, to1, tn+1, cl);
05157
05158                     // restore configuration
05159                     cl->incMat(sn, tn, - ncm);
05160                     cl->incMat(tn, sn, - ncm);
05161                     adjMat[sn][tn] = 0;
05162                     adjMat[tn][sn] = 0;
05163                     nodes[sn]->freelg += nc;
05164                     nodes[tn]->freelg += nc;
05165                 }
05166             }
05167         }
05168     }
05169 }
05170
05171 //-------------------------------------------------------------
05172 Bool MGraph::isOptM(void)
05173 {
05174     Bool ok = True;
05175
05176     opi       = (mconn->nopic == 1);
05177     opiloop   = (mconn->nlpopic <= 1);
05178     tadpole   = (mconn->ne0bridges >= ((nExtern == 1) ? 0 : 1));
05179     selfloop  = (mconn->nselfloops > 0);
05180     multiedge = (mconn->nmultiedges > 0);
05181     tadblock  = (mconn->na1blocks > 0);
05182     block     = (mconn->neblocks <= 1);
05183     extself   = (mconn->ne1bridges > 0);
05184
05185     if (opt->values[GRCC_OPT_1PI] > 0) {
05186         ok = ok && opi;
05187     } else if (opt->values[GRCC_OPT_1PI] < 0) {
05188         ok = ok && !opi;
```

```
05189     }
05190     if (opt->values[GRCC_OPT_NoExtSelf] > 0) {
05191         ok = ok && !extself;
05192     } else if (opt->values[GRCC_OPT_NoExtSelf] < 0) {
05193         ok = ok && extself;
05194     }
05195     if (opt->values[GRCC_OPT_NoTadpole] > 0) {
05196 #ifdef OLDOPT
05197         ok = ok && !tadpole;
05198 #else
05199         if (nExtern > 1) {
05200             ok = ok && !tadpole;
05201         }
05202 #endif
05203     } else if (opt->values[GRCC_OPT_NoTadpole] < 0) {
05204 #ifdef OLDOPT
05205         ok = ok && !tadpole;
05206 #else
05207         if (nExtern > 1) {
05208             ok = ok && tadpole;
05209         }
05210 #endif
05211     }
05212     if (opt->values[GRCC_OPT_NoSelfLoop] > 0) {
05213     } else if (opt->values[GRCC_OPT_NoSelfLoop] < 0) {
05214         ok = ok && selfloop;
05215     }
05216     if (opt->values[GRCC_OPT_NoMultiEdge] > 0) {
05217     } else if (opt->values[GRCC_OPT_NoMultiEdge] < 0) {
05218         ok = ok && multiedge;
05219     }
05220     if (opt->values[GRCC_OPT_No1PtBlock] > 0) {
05221         if (nExtern > 1) {
05222             ok = ok && !tadblock;
05223         }
05224     } else if (opt->values[GRCC_OPT_No1PtBlock] < 0) {
05225         if (nExtern > 1) {
05226             ok = ok && tadblock;
05227         }
05228     }
05229     if (opt->values[GRCC_OPT_Block] > 0) {
05230         ok = ok && block;
05231     } else if (opt->values[GRCC_OPT_Block] < 0) {
05232         ok = ok && !block;
05233     }
05234     return ok;
05235 }
05236
05237 //------------------------------------------------------------
05238 void MGraph::newGraph(MNodeClass *cl)
05239 {
05240     //  A new candidate diagram is obtained.
05241     //  It may be a disconnected graph
05242     //
05243     //  Things to be done in the steps followed by this function.
05244     //  1. convert data format which treat the edges as objects.
05245     //  2. definition of loop momenta to the edges.
05246     //  3. analysis of loop structure in a graph.
05247     //  4. assignment of particles to edges and vertices to nodes
05248     //  5. recalculate symmetry factor
05249
05250     int connected;
05251     MNodeClass *xcl;
05252     Bool ok;
05253
05254     ngen++;
05255
05256     // refine class and check ordering condition
05257     xcl = refineClass(cl);
05258     if (xcl == NULL) {
05259 #ifdef MONITOR
05260         discardRefine++;
05261 #endif
05262
05263     // check whether this is a connected graph or not.
05264     } else {
05265         connected = isConnected();
05266         if (!connected) {
05267 #ifdef MONITOR
05268             discardDisc++;
05269 #endif
05270
05271         // connected graph : check isomorphism of the graph
05272         } else {
05273             ngconn++;
05274             if (!isIsomorphic(xcl)) {
05275 #ifdef MONITOR
```

```
05276                 discardIso++;
05277 #endif
05278
05279             // We got a new connected and a unique representation of a class.
05280             } else {
05281                 biconnME();
05282                 if (isOptM()) {
05283                     egraph->fromMGraph(this);
05284                     if (egraph->isOptE()) {
05285                         curcl->copy(xcl);
05286 #ifdef ORBITS
05287                         orbits->toOrbits();
05288 #endif
05289                         ok = True;
05290                         if (opt->outmg != NULL) {
05291                             if (opt->proc != NULL) {
05292                                 egraph->mId = opt->proc->mgrcount + 1;
05293                                 egraph->sId = opt->proc->agrcount + 1;
05294                             } else if (opt->sproc != NULL) {
05295                                 egraph->mId = opt->sproc->mgrcount + 1;
05296                                 egraph->sId = opt->sproc->agrcount + 1;
05297                             } else {
05298                                 egraph->mId = cDiag;
05299                             }
05300                             ok = (*(opt->outmg))(egraph, opt->argmg);
05301                         }
05302
05303                         if (ok) {
05304                             // # generated graphs
05305                             cDiag++;
05306                             wscon.add(1, nsym*esym);
05307
05308                             // # generated 1PI graphs
05309                             if (opi) {
05310                                 wsopi.add(1, nsym*esym);
05311                                 c1PI++;
05312                             }
05313
05314                             // # no tadpoles
05315                             if (!tadpole) {
05316                                 cNoTadpole++;
05317                             }
05318                             // # no tadblocks
05319                             if (!tadblock) {
05320                                 cNoTadBlock++;
05321                                 if (opi) {
05322                                     c1PINoTadBlock++;
05323                                 }
05324                             }
05325
05326 #ifdef MONITOR
05327                             grcc_fprintf(GRCC_Stdout, "\n");
05328                             grcc_fprintf(GRCC_Stdout, "Graph : %ld (%ld)", cDiag, ngen);
05329                             grcc_fprintf(GRCC_Stdout, " sym. factor = (%ld*%ld)\n", nsym, esym);
05330                             printAdjMat(cl);
05331                             // cl->printMat();
05332 #  ifdef ORBITS
05333                             orbits->print();
05334 #  endif
05335 #endif
05336
05337                             // go to next step
05338                             opt->newMGraph(this);
05339                         } else {
05340                         }
05341                     } else {
05342                     }
05343                 } else {
05344                     ;
05345                 }
05346             }
05347         }
05348     }
05349     if (xcl != cl && xcl != NULL) {
05350       delete xcl;
05351     }
05352 }
05353
05354 //============================================================
05355 // class MOrbits: orbits of nodes
05356 //------------------------------------------------------------
05357 MOrbits::MOrbits(void)
05358 {
05359     nOrbits = 0;
05360     nNodes  = 0;
05361 }
05362
05363 //------------------------------------------------------------
```

```
05363 MOrbits::~MOrbits(void)
05364 {
05365 }
05366
05367 //------------------------------------------------------------
05368 void MOrbits::print(void)
05369 {
05370     int c;
05371
05372     grcc_fprintf(GRCC_Stdout, "Orbits : nOrbits=%d: nd2or=", nOrbits);
05373     prIntArray(nNodes, nd2or, "\n");
05374     for (c = 0; c < nOrbits; c++) {
05375         grcc_fprintf(GRCC_Stdout, "    %2d: (%2d -- %2d) :", c, flist[c], flist[c+1]-1);
05376         prIntArray(flist[c+1]-flist[c], or2nd+flist[c], "\n");
05377     }
05378 }
05379
05380 //------------------------------------------------------------
05381 void MOrbits::initPerm(int nnodes)
05382 {
05383     int j;
05384
05385     nNodes = nnodes;
05386     for (j = 0; j < nNodes; j++) {
05387         nd2or[j] = j;
05388     }
05389 }
05390
05391 //------------------------------------------------------------
05392 void MOrbits::fromPerm(int *perm)
05393 {
05394     // update the orbits of nodes by the permutation
05395
05396     int j, j1, k, l;
05397
05398     for (j = 0; j < nNodes; j++) {
05399         if (nd2or[j] != j) {
05400             // not the first element of an old orbit
05401             continue;
05402         }
05403         // merge orbits to orbit 'j'
05404         k = j;
05405         for (j1 = 0; j1 < nNodes && (k = perm[k]) != j; j1++) {
05406             if (nd2or[k] == j) {
05407                 // 'k' is already in 'j'
05408                 ;
05409             } else {
05410                 // old orbit 'k' had several elements: merge to orbit 'j'
05411                 for (l = 0; l < nNodes; l++) {
05412                     if (nd2or[l] == k) {
05413                         nd2or[l] = j;
05414                     }
05415                 }
05416             }
05417         }
05418 #ifdef CHECK
05419         if (j1 >= nNodes) {
05420             grcc_fprintf(GRCC_Stdout, "*** fromPerm: illegal control: j=%d, k=%d\n", j, k);
05421             grcc_fprintf(GRCC_Stdout, "perm=");
05422             prIntArray(nNodes, perm,  " nd2or=");
05423             prIntArray(nNodes, nd2or, "\n");
05424             erEnd("fromPerm: illegal control");
05425         }
05426 #endif
05427     }
05428 }
05429
05430 //------------------------------------------------------------
05431 void MOrbits::toOrbits(void)
05432 {
05433     // fill elements from 'nd2or'
05434     int nd, nel, k, lor;
05435
05436     lor = -1;
05437     nel = 0;
05438     nOrbits = 0;
05439     for (nd = 0; nd < nNodes; nd++) {
05440         if (nd2or[nd] > lor) {
05441             // lowest element of a new orbit
05442             lor = nd2or[nd];
05443             flist[nOrbits++] = nel;
05444             for (k = nd; k < nNodes; k++) {
05445                 if (nd2or[k] == lor) {
05446                     or2nd[nel++] = k;
05447                 }
05448             }
05449         }
```

```
05450     }
05451     flist[nOrbits] = nNodes;
05452 }
05453
05454 //************************************************************
05455 // n-edge connected components
05456 //============================================================
05457 // class MCEdge
05458 //------------------------------------------------------------
05459 MCEdge::MCEdge(void)
05460 {
05461     ;
05462 }
05463
05464 //------------------------------------------------------------
05465 MCEdge::~MCEdge(void)
05466 {
05467     ;
05468 }
05469
05470 //============================================================
05471 // class MCOpi: one n-edge connected component
05472 //------------------------------------------------------------
05473 MCOpi::MCOpi(void)
05474 {
05475     init();
05476 }
05477
05478 //------------------------------------------------------------
05479 MCOpi::~MCOpi(void)
05480 {
05481     nodes = NULL;
05482 }
05483
05484 //------------------------------------------------------------
05485 void MCOpi::init(void)
05486 {
05487     nodes  = NULL;
05488     nnodes = 0;
05489     nlegs  = 0;
05490     next   = 0;
05491     nedges = 0;
05492     loop   = 0;
05493     ctloop = 0;
05494     mom0lg = 0;
05495 }
05496
05497 //============================================================
05498 // class MCBridge
05499 //------------------------------------------------------------
05500 MCBridge::MCBridge(void)
05501 {
05502 }
05503
05504 //------------------------------------------------------------
05505 MCBridge::~MCBridge(void)
05506 {
05507 }
05508
05509 //============================================================
05510 // class MCBlock
05511 //------------------------------------------------------------
05512 MCBlock::MCBlock(void)
05513 {
05514     init();
05515 }
05516
05517 //------------------------------------------------------------
05518 MCBlock::~MCBlock(void)
05519 {
05520 }
05521
05522 //------------------------------------------------------------
05523 void MCBlock::init(void)
05524 {
05525     edges   = NULL;
05526     nmedges = 0;
05527     nartps  = 0;
05528     loop    = 0;
05529 }
05530
05531 //============================================================
05532 // class MConn: table of n-edge connected components
05533 //------------------------------------------------------------
05534 MConn::MConn(int nnod, int nedg)
05535 {
05536     // nnod : the number of nodes
```

```
05537     // nedg : the number of edges
05538
05539     snodes = nnod;
05540     sedges = nedg;
05541
05542     cedges   = new MCEdge[sedges];
05543     opics    = new MCOpi[snodes];
05544     bridges  = new MCBridge[sedges];
05545     blocks   = new MCBlock[sedges];
05546     articuls = new int[snodes];
05547
05548     // workspace
05549     opisp    = new int[snodes];
05550     opistk   = new int[snodes];
05551     blksp    = new Edge2n[sedges];
05552     blkstk   = new Edge2n[sedges];
05553     init();
05554 }
05555
05556 //------------------------------------------------------------
05557 MConn::~MConn(void)
05558 {
05559     delete[] blkstk;
05560     delete[] blksp;
05561     delete[] opistk;
05562     delete[] opisp;
05563
05564     delete[] articuls;
05565     delete[] blocks;
05566     delete[] bridges;
05567     delete[] opics;
05568     delete[] cedges;
05569 }
05570
05571 //------------------------------------------------------------
05572 void MConn::init(void)
05573 {
05574     int j;
05575
05576     nopic       = 0;
05577     nlpopic     = 0;
05578     nbacked     = 0;
05579     nctopic     = 0;
05580     nbridges    = 0;
05581     ne0bridges  = 0;
05582     ne1bridges  = 0;
05583     nselfloops  = 0;
05584     nmultiedges = 0;
05585     nbacked     = 0;
05586
05587     nblocks     = 0;
05588     na1blocks   = 0;
05589     narticuls   = 0;
05590     neblocks    = 0;
05591
05592     opistkptr   = 0;
05593     nopisp      = 0;
05594     blkstkptr   = 0;
05595     nblksp      = 0;
05596
05597     for (j = 0; j < snodes; j++) {
05598         articuls[j] = 0;
05599     }
05600 }
05601
05602 //------------------------------------------------------------
05603 void MConn::pushNode(int nd)
05604 {
05605     if (opistkptr >= snodes) {
05606         erEnd("MConn::pushNode: opistkptr >= snodes");
05607     }
05608     opistk[opistkptr++] = nd;
05609 }
05610
05611 //------------------------------------------------------------
05612 void MConn::pushEdge(int n0, int n1)
05613 {
05614     if (blkstkptr >= sedges) {
05615         erEnd("MConn::pushEdge: blkstkptr >= sedges");
05616     }
05617     if (n0 <= n1) {
05618         blkstk[blkstkptr][0] = n0;
05619         blkstk[blkstkptr][1] = n1;
05620     } else {
05621         blkstk[blkstkptr][0] = n1;
05622         blkstk[blkstkptr][1] = n0;
05623     }
```

```
05624     blkstkptr++;
05625 }
05626
05627
05628 //-------------------------------------------------------------
05629 void MConn::initCEdges(MGraph *mg)
05630 {
05631     int ed, n0, n1, e;
05632
05633     ed = 0;
05634     for (n0 = 0; n0 < mg->nNodes; n0++) {
05635         for (e = 0; e < mg->adjMat[n0][n0]/2; e++, ed++) {
05636             cedges[ed].nodes[0] = n0;
05637             cedges[ed].nodes[1] = n0;
05638             cedges[ed].momdir   = 0;
05639         }
05640         for (n1 = n0+1; n1 < mg->nNodes; n1++) {
05641             for (e = 0; e < mg->adjMat[n0][n1]; e++, ed++) {
05642                 cedges[ed].nodes[0] = n0;
05643                 cedges[ed].nodes[1] = n1;
05644                 cedges[ed].momdir   = 0;
05645             }
05646         }
05647     }
05648     if (ed != sedges) {
05649         grcc_fprintf(GRCC_Stdout, "*** ed=%d != sedges=%d\n", ed, sedges);
05650         erEnd("MConn::initCEdge: table overflow");
05651     }
05652 }
05653
05654 //-------------------------------------------------------------
05655 void MConn::addCEdge(int n0, int n1, ULong momset)
05656 {
05657     int m0, m1, dir, ed;
05658
05659     if (n0 <= n1) {
05660         m0  = n0;
05661         m1  = n1;
05662         dir = 1;
05663     } else {
05664         m0  = n1;
05665         m1  = n0;
05666         dir = -1;
05667     }
05668     for (ed = 0; ed < sedges; ed++) {
05669         if (cedges[ed].nodes[0] == m0 && cedges[ed].nodes[1] == m1 &&
05670                 cedges[ed].momdir   == 0) {
05671             cedges[ed].momdir = dir;
05672             cedges[ed].momset = momset;
05673             return;
05674         }
05675     }
05676 }
05677
05678 //-------------------------------------------------------------
05679 void MConn::addOPIc(MCOpi *mopi, int stp)
05680 {
05681     int j, nn;
05682
05683     if (nopic >= snodes) {
05684         erEnd("MConn::addOPIc: table overflow");
05685     }
05686
05687     nn = opistkptr - stp;
05688     for (j = 0; j < nn; j++) {
05689         opisp[nopisp+j] = opistk[stp+j];
05690     }
05691     opics[nopic].nnodes = nn;
05692     opics[nopic].nlegs  = mopi->nlegs;
05693     opics[nopic].nedges = mopi->nedges;
05694     opics[nopic].next   = mopi->next;
05695     opics[nopic].loop   = mopi->loop;
05696     opics[nopic].ctloop = mopi->ctloop;
05697     opics[nopic].mom0lg = mopi->mom0lg;
05698     opics[nopic].nodes  = opisp  + nopisp;
05699     nopisp   += nn;
05700     opistkptr = stp;
05701
05702     nopic++;
05703 }
05704
05705 //-------------------------------------------------------------
05706 void MConn::addBridge(int n0, int n1, int nex, int nextot)
05707 {
05708     if (nbridges >= sedges) {
05709         erEnd("MConn::addBridge: table overflow");
05710     }
```

```
05711     if (n0 <= n1) {
05712         bridges[nbridges].nodes[0] = n0;
05713         bridges[nbridges].nodes[1] = n1;
05714     } else {
05715         bridges[nbridges].nodes[0] = n1;
05716         bridges[nbridges].nodes[1] = n0;
05717     }
05718     bridges[nbridges].next = Min(nex, nextot-nex);
05719     nbridges++;
05720 }
05721
05722 //-------------------------------------------------------------
05723 void MConn::addArtic(int nd, int mul)
05724 {
05725     articuls[nd] += mul;
05726     if (articuls[nd] == mul) {
05727         narticuls++;
05728     } else if (articuls[nd] == 0) {
05729         narticuls--;
05730     }
05731 }
05732
05733 //-------------------------------------------------------------
05734 void MConn::addBlock(MCBlock *mblk, int stp)
05735 {
05736     int j, nn;
05737
05738     if (nopic >= snodes) {
05739         erEnd("MConn::addOPIc: table overflow");
05740     }
05741
05742     nn = blkstkptr - stp;
05743     for (j = 0; j < nn; j++) {
05744         blksp[nblksp+j][0] = blkstk[stp+j][0];
05745         blksp[nblksp+j][1] = blkstk[stp+j][1];
05746     }
05747     blocks[nblocks].edges   = blksp + nblksp;
05748     blocks[nblocks].nmedges = nn;
05749     blocks[nblocks].nartps  = mblk->nartps;
05750     blocks[nblocks].loop    = mblk->loop;
05751     nblksp    += nn;
05752     blkstkptr = stp;
05753
05754     nblocks++;
05755 }
05756
05757 //-------------------------------------------------------------
05758 void MConn::addBlockSelf(int nd, int mult)
05759 {
05760     MCBlock mblk;
05761     int j, stp;
05762
05763     mblk.edges   = NULL;
05764     mblk.nmedges = 0;
05765     mblk.nartps  = 1;
05766     mblk.loop    = 1;
05767
05768     for (j = 0; j < mult; j++) {
05769         stp = blkstkptr;
05770         pushEdge(nd, nd);
05771         addBlock(&mblk, stp);
05772     }
05773 }
05774
05775 //-------------------------------------------------------------
05776 void MConn::print(void)
05777 {
05778     int j, k;
05779
05780     grcc_fprintf(GRCC_Stdout, "+++ MConn object: snodes=%d, sedges=%d\n", snodes, sedges);
05781     grcc_fprintf(GRCC_Stdout, "    nopic=%d, nlpopic=%d, nbacked=%d, nctopic=%d, "
05782            "nbridges=%d, ne0bridges=%d, ne1bridges=%d\n",
05783            nopic, nlpopic, nbacked, nctopic,
05784            nbridges, ne0bridges, ne1bridges);
05785     grcc_fprintf(GRCC_Stdout, "    nblocks=%d, neblocks=%d, na1blocks=%d, narticuls=%d,
      nselfloop=%d\n",
05786            nblocks, neblocks, na1blocks, narticuls, nselfloops);
05787     grcc_fprintf(GRCC_Stdout, "    nmultiedges=%d\n", nmultiedges);
05788
05789     grcc_fprintf(GRCC_Stdout, "  cEdges (%d)\n", sedges);
05790     if (sedges > 0) {
05791         for (j = 0; j < sedges; j++) {
05792             grcc_fprintf(GRCC_Stdout, "    %2d: (%d,%d)[%2d*%2ld] \n", j,
05793                    cedges[j].nodes[0], cedges[j].nodes[1],
05794                    cedges[j].momdir, cedges[j].momset);
05795         }
05796     }
```

```
05797     grcc_fprintf(GRCC_Stdout, "\n");
05798
05799     grcc_fprintf(GRCC_Stdout, "  1PI components (%d)\n", nopic);
05800     for (j = 0; j < nopic; j++) {
05801         grcc_fprintf(GRCC_Stdout, "    %d: nleg=%d, nnodes=%d, nedge=%d, ",
05802                 j, opics[j].nlegs, opics[j].nnodes, opics[j].nedges);
05803         grcc_fprintf(GRCC_Stdout, "next=%d, loop=%d, ctlp=%d: m0lg=%d [",
05804                 opics[j].next, opics[j].loop, opics[j].ctloop,
05805                 opics[j].mom0lg);
05806         for (k = 0; k < opics[j].nnodes; k++) {
05807             grcc_fprintf(GRCC_Stdout, " %d", opics[j].nodes[k]);
05808         }
05809         grcc_fprintf(GRCC_Stdout, "]\n");
05810     }
05811
05812     grcc_fprintf(GRCC_Stdout, "  bridges (%d)\n", nbridges);
05813     if (nbridges > 0) {
05814         grcc_fprintf(GRCC_Stdout, "    ");
05815         for (j = 0; j < nbridges; j++) {
05816             grcc_fprintf(GRCC_Stdout, "(%d,%d)[mom=%d] ",
05817                    bridges[j].nodes[0], bridges[j].nodes[1], bridges[j].next);
05818         }
05819         grcc_fprintf(GRCC_Stdout, "\n");
05820     }
05821
05822     grcc_fprintf(GRCC_Stdout, "  blocks (%d)\n", nblocks);
05823     for (j = 0; j < nblocks; j++) {
05824         grcc_fprintf(GRCC_Stdout, "    %d: nmedges=%d, nartps=%d, loop=%d: [",
05825                j, blocks[j].nmedges, blocks[j].nartps, blocks[j].loop);
05826         for (k = 0; k < blocks[j].nmedges; k++) {
05827             grcc_fprintf(GRCC_Stdout, " (%d,%d)", blocks[j].edges[k][0], blocks[j].edges[k][1]);
05828         }
05829         grcc_fprintf(GRCC_Stdout, "]\n");
05830     }
05831
05832     grcc_fprintf(GRCC_Stdout, "  articulation points (%d)\n", narticuls);
05833     if (narticuls > 0) {
05834         grcc_fprintf(GRCC_Stdout, "    ");
05835         for (j = 0; j < snodes; j++) {
05836             if (articuls[j] != 0) {
05837 #if 0
05838                 grcc_fprintf(GRCC_Stdout, "%d(%d) ", j, articuls[j]);
05839 #else
05840                 grcc_fprintf(GRCC_Stdout, "%d ", j);
05841 #endif
05842             }
05843         }
05844         grcc_fprintf(GRCC_Stdout, "\n");
05845     }
05846 }
05847
05848 //-------------------------------------------------------------
05849 void MConn::prEdges(void)
05850 {
05851     int j;
05852
05853     grcc_fprintf(GRCC_Stdout, "  cEdges (%d)", sedges);
05854     if (sedges > 0) {
05855         for (j = 0; j < sedges; j++) {
05856             if (j % 5 == 0) {
05857                 grcc_fprintf(GRCC_Stdout, "\n    ");
05858             }
05859             grcc_fprintf(GRCC_Stdout, "(%d,%d)[%2d*%3ld] ",
05860                    cedges[j].nodes[0], cedges[j].nodes[1],
05861                    cedges[j].momdir, cedges[j].momset);
05862         }
05863         grcc_fprintf(GRCC_Stdout, "\n");
05864     }
05865 }
05866
05867 //*************************************************************
05868 // sgroup.cc
05869 //=============================================================
05870 // compile options
05871 //-------------------------------------------------------------
05872 // table of group elements
05873 //-------------------------------------------------------------
05874 SGroup::SGroup(void)
05875 {
05876     // should be called before graph generation
05877     nnodes = -1;
05878     nelem  = 0;
05879     elem   = NULL;
05880 }
05881
05882 //-------------------------------------------------------------
05883 SGroup::~SGroup(void)
```

```
05884 {
05885     int j;
05886 
05887     if (elem != NULL) {
05888         for (j = GRCC_MAXGROUP-1; j >= 0; j--) {
05889             if (elem[j] != NULL) {
05890                 delete[] elem[j];
05891                 elem[j] = NULL;
05892             }
05893         }
05894         delete[] elem;
05895         elem = NULL;
05896     }
05897 }
05898 
05899 //-------------------------------------------------------------
05900 void SGroup::print(void)
05901 {
05902     int j;
05903 
05904     grcc_fprintf(GRCC_Stdout, "SGroup : nnodes = %d, nelem=%ld, cgen=%d, csav=%d\n",
05905             nnodes, nelem, cgen, csav);
05906     grcc_fprintf(GRCC_Stdout, "  eclass =");
05907     prIntArray(neclass, eclass, "\n");
05908     for (j = 0; j < nelem; j++) {
05909         grcc_fprintf(GRCC_Stdout, "    %4d: ", j);
05910         prIntArray(nnodes, elem[j], "\n");
05911     }
05912 }
05913 
05914 //-------------------------------------------------------------
05915 void SGroup::newGroup(int nnd, int nclss, int *clss)
05916 {
05917     int j;
05918 
05919     clearGroup();
05920     nnodes  = nnd;
05921     neclass = nclss;
05922 
05923     if (neclass > GRCC_MAXNODES) {
05924         erEnd("newGroup : too many classes (GRCC_MAXNODES)");
05925     }
05926     for (j = 0; j < neclass; j++) {
05927         eclass[j] = clss[j];
05928     }
05929 
05930     nelem = 0;
05931     if (elem == NULL) {
05932         elem  = new int*[GRCC_MAXGROUP];
05933         for (j = 0; j < GRCC_MAXGROUP; j++) {
05934             elem[j] = NULL;
05935         }
05936     }
05937 }
05938 
05939 //-------------------------------------------------------------
05940 void SGroup::clearGroup(void)
05941 {
05942     nelem  = 0;
05943     csav   = -1;
05944     cgen   = -1;
05945 }
05946 
05947 //-------------------------------------------------------------
05948 void SGroup::delGroup(void)
05949 {
05950     int j;
05951 
05952     for (j = 0; j < GRCC_MAXGROUP; j++) {
05953         if (elem[j] != NULL) {
05954             delete[] elem[j];
05955             elem[j] = NULL;
05956         }
05957     }
05958     delete[] elem;
05959     elem = NULL;
05960     clearGroup();
05961 }
05962 
05963 //-------------------------------------------------------------
05964 int *SGroup::genNext(void)
05965 {
05966     cgen = nextPerm(nnodes, neclass, eclass, pgr, pgq, permg, cgen);
05967     if (cgen < 0) {
05968         return NULL;
05969     }
05970     return permg;
```

```
05971 }
05972
05973 //-------------------------------------------------------------
05974 void SGroup::addGroup(int *p)
05975 {
05976     int j;
05977
05978     if (nelem >= GRCC_MAXGROUP) {
05979         erEnd("addGroup : too many elements (GRCC_MAXGROUP)");
05980     }
05981     if (elem[nelem] == NULL) {
05982         elem[nelem] = new int[GRCC_MAXNODES];
05983     }
05984     for (j = 0; j < nnodes; j++) {
05985         elem[nelem][j] = p[j];
05986     }
05987     nelem++;
05988 }
05989
05990 //-------------------------------------------------------------
05991 BigInt SGroup::nElem(void)
05992 {
05993     return nelem;
05994 }
05995
05996 //-------------------------------------------------------------
05997 int *SGroup::nextElem(void)
05998 {
05999     if (nelem < 0) {
06000         cgen = nextPerm(nelem, neclass, eclass,
06001                     psr, psq, perms, cgen);
06002         return perms;
06003     }
06004     if (cgen < 0) {
06005         cgen = 0;
06006     }
06007     if (cgen >= nelem) {
06008         return NULL;
06009     } else {
06010         return elem[cgen++];
06011     }
06012 }
06013
06014 //*************************************************************
06015 // egraph.cc
06016 // Generate scalar connected Feynman graphs.
06017 //=============================================================
06018 static const char *GRCC_ND_names[] = GRCC_ND_NAMES;
06019 static const char *GRCC_ED_names[] = GRCC_ED_NAMES;
06020
06021 //=============================================================
06022 // class ENode
06023 //-------------------------------------------------------------
06024 ENode::ENode(void)
06025 {
06026     id      = -1;
06027     egraph  = NULL;
06028     edges   = NULL;
06029     maxdeg  = 0;
06030     deg     = 0;
06031
06032     klow    = NULL;
06033     extloop = 0;
06034     ndtype  = GRCC_ND_Undef;
06035
06036     intrct  = -1;
06037 }
06038
06039 //-------------------------------------------------------------
06040 ENode::ENode(EGraph *egrph, int loops, int sdeg)
06041
06042 {
06043     id      = -1;
06044     egraph  = egrph;
06045     edges   = NULL;
06046     maxdeg  = sdeg;
06047     deg     = 0;
06048
06049     klow    = NULL;
06050     extloop = 0;
06051     ndtype  = GRCC_ND_Undef;
06052
06053     intrct  = -1;
06054
06055     edges = new int[sdeg];
06056     klow  = new int[loops+1];
06057 }
```

```
06058
06059 //------------------------------------------------------------
06060 ENode::~ENode(void)
06061 {
06062     if (klow != NULL) {
06063         delete[] klow;
06064         delete[] edges;
06065         klow  = NULL;
06066         edges = NULL;;
06067     }
06068 }
06069
06070 //------------------------------------------------------------
06071 void ENode::copy(ENode *en)
06072 {
06073     int d;
06074
06075     deg = en->deg;
06076     extloop = en->extloop;
06077     ndtype  = en->ndtype;
06078     for (d = 0; d < deg; d++) {
06079         edges[d] = en->edges[d];
06080     }
06081 }
06082
06083 //------------------------------------------------------------
06084 void ENode::initAss(EGraph *egrph, int nid, int sdg)
06085 {
06086     id      = nid;
06087     egraph  = egrph;
06088     maxdeg  = sdg;
06089 }
06090
06091 //------------------------------------------------------------
06092 void ENode::setId(EGraph *egrph, const int nid)
06093 {
06094     id      = nid;
06095     egraph  = egrph;
06096 }
06097
06098 //------------------------------------------------------------
06099 void ENode::setExtern(int typ, int pt)
06100 {
06101     intrct  = pt;
06102     if (typ == GRCC_AT_Initial || typ == GRCC_AT_Final) {
06103         extloop = typ;
06104     } else {
06105         grcc_fprintf(GRCC_Stderr, "** ENode::setExternal:: illegal typ=%d\n",
06106                 typ);
06107         erEnd("ENode::setExternal:: illegal typ");
06108     }
06109 }
06110
06111 //------------------------------------------------------------
06112 void ENode::setType(int typ)
06113 {
06114 #ifdef CHECK
06115     if (ndtype != GRCC_ND_Undef && ndtype != typ) {
06116         grcc_fprintf(GRCC_Stderr, "*** ndtype is already defined : old=%d, new = %d\n",
06117                 ndtype, typ);
06118         erEnd("ndtype is already defined");
06119     }
06120 #endif
06121     ndtype = typ;
06122 }
06123
06124 //------------------------------------------------------------
06125 void ENode::print(void)
06126 {
06127     int j;
06128
06129     grcc_fprintf(GRCC_Stdout, "Enode %d deg=%d, extl=%2d, intr=%d ",
06130             id, deg, extloop, intrct);
06131     if (egraph->bicount > 0) {
06132         grcc_fprintf(GRCC_Stdout, "(%-8s) ", GRCC_ND_names[ndtype]);
06133     }
06134     grcc_fprintf(GRCC_Stdout, "edge=[");
06135     for (j = 0; j < deg; j++) {
06136         grcc_fprintf(GRCC_Stdout, " %2d", edges[j]);
06137     }
06138     grcc_fprintf(GRCC_Stdout, "]\n");
06139 }
06140
06141 //============================================================
06142 // class EEdge
06143 //------------------------------------------------------------
06144 EEdge::EEdge(void)
```

```
06145 {
06146     id       = -1;
06147     egraph   = NULL;
06148     nodes[0] = -1;
06149     nodes[1] = -1;
06150     // nlegs[0] = -1;
06151     // nlegs[1] = -1;
06152     ptcl     = GRCC_PT_Undef;
06153     deleted  = False;
06154 
06155     edtype   = GRCC_ED_Undef;
06156     cut      = False;
06157 
06158     emom = NULL;
06159     lmom = NULL;
06160 
06161     momset = 0;
06162     momdir = 0;
06163 }
06164 
06165 //------------------------------------------------------------------
06166 EEdge::EEdge(EGraph *egrph, int nedges, int nloops)
06167 {
06168     id       = -1;
06169     egraph   = egrph;
06170     nodes[0] = -1;
06171     nodes[1] = -1;
06172     nlegs[0] = -1;
06173     nlegs[1] = -1;
06174     ptcl     = GRCC_PT_Undef;
06175     deleted  = False;
06176 
06177     edtype   = GRCC_ED_Undef;
06178     cut      = False;
06179 
06180     emom = new int[nedges];
06181     lmom = new int[nloops];
06182 }
06183 
06184 //------------------------------------------------------------------
06185 EEdge::~EEdge(void)
06186 {
06187     if (lmom != NULL) {
06188         delete[] lmom;
06189     }
06190     if (emom != NULL) {
06191         delete[] emom;
06192     }
06193 }
06194 
06195 //------------------------------------------------------------------
06196 void EEdge::copy(EEdge *ee)
06197 {
06198     nodes[0] = ee->nodes[0];
06199     nodes[1] = ee->nodes[1];
06200     id       = ee->id;
06201     ext      = ee->ext;
06202     dir      = ee->dir;
06203     edtype   = ee->edtype;
06204     cut      = ee->cut;
06205 }
06206 
06207 //------------------------------------------------------------------
06208 void EEdge::setId(EGraph *egrph, const int eid)
06209 {
06210     id      = eid;
06211     egraph  = egrph;
06212 }
06213 
06214 //------------------------------------------------------------------
06215 void EEdge::setType(int typ)
06216 {
06217 #ifdef CHECK
06218     if (edtype != GRCC_ED_Undef) {
06219         erEnd("edtype is already defined");
06220     }
06221 #endif
06222     edtype = typ;
06223 }
06224 
06225 //------------------------------------------------------------------
06226 void EEdge::setEMom(int nedges, int *em, int dir)
06227 {
06228     int e;
06229 
06230     for (e = 0; e < nedges; e++) {
06231         if (em[e] != 0) {
```

```
06232             emom[e] = dir;
06233         }
06234     }
06235 }
06236
06237 //------------------------------------------------------------
06238 void EEdge::setLMom(int k, int dir)
06239 {
06240     lmom[k] = dir;
06241 }
06242
06243 //------------------------------------------------------------
06244 void EEdge::print(void)
06245 {
06246     int zero, n, k;
06247
06248     grcc_fprintf(GRCC_Stdout, "Edge %2d ext=%d ptcl=%2d ", id, ext, ptcl);
06249     if (egraph == NULL || !egraph->assigned) {
06250         grcc_fprintf(GRCC_Stdout, "[%d, %d]", nodes[0], nodes[1]);
06251     } else {
06252         grcc_fprintf(GRCC_Stdout, "[(%d,%d), (%d,%d))", nodes[0], nlegs[0], nodes[1], nlegs[1]);
06253     }
06254     if (momdir != 0) {
06255         grcc_fprintf(GRCC_Stdout, "[%2d*%2lu]", momdir, momset);
06256     }
06257
06258     if (egraph != NULL && egraph->bicount > 0) {
06259         if (cut) {
06260             grcc_fprintf(GRCC_Stdout, " %-9s ", "Cut");
06261         } else {
06262             grcc_fprintf(GRCC_Stdout, " %-9s ", GRCC_ED_names[edtype]);
06263         }
06264         grcc_fprintf(GRCC_Stdout, "c%2d: ", opicomp);
06265         grcc_fprintf(GRCC_Stdout, "%s%d =", (ext)?"Q":"p", id);
06266         zero = True;
06267         for (n = 0; n < egraph->nEdges; n++) {
06268             if (emom[n] != 0) {
06269                 prMomStr(emom[n], "Q", n);
06270                 zero = False;
06271             }
06272         }
06273         for (k = 0; k < egraph->nLoops; k++) {
06274             if (lmom[k] != 0) {
06275                 prMomStr(lmom[k], "k", k);
06276                 zero = False;
06277             }
06278         }
06279         if (zero) {
06280             grcc_fprintf(GRCC_Stdout, " 0");
06281         }
06282     } else {
06283         if (egraph == NULL) {
06284             grcc_fprintf(GRCC_Stdout, " egraph=NULL");
06285         } else {
06286             grcc_fprintf(GRCC_Stdout, " egraph->bicount=%d", egraph->bicount);
06287         }
06288     }
06289     grcc_fprintf(GRCC_Stdout, "\n");
06290 }
06291
06292 //============================================================
06293 // class EFLine
06294 //------------------------------------------------------------
06295 EFLine::EFLine(void)
06296 {
06297     ftype = FL_Open;
06298     fkind = 0;
06299     nlist = 0;
06300 }
06301
06302 //------------------------------------------------------------
06303 void EFLine::print(const char *msg)
06304 {
06305     if (ftype == FL_Open) {
06306         grcc_fprintf(GRCC_Stdout, " Open");
06307     } else if (ftype == FL_Closed) {
06308         grcc_fprintf(GRCC_Stdout, " Loop");
06309     } else {
06310         grcc_fprintf(GRCC_Stdout, " ?%d", ftype);
06311     }
06312     if (fkind == GRCC_PT_Dirac) {
06313         grcc_fprintf(GRCC_Stdout, " Dirac");
06314     } else if (fkind == GRCC_PT_Majorana) {
06315         grcc_fprintf(GRCC_Stdout, " Major");
06316     } else if (fkind == GRCC_PT_Ghost) {
06317         grcc_fprintf(GRCC_Stdout, " Ghost");
06318     } else {
```

```
06319         grcc_fprintf(GRCC_Stdout, " ?%d", fkind);
06320     }
06321     grcc_fprintf(GRCC_Stdout, " len=%d ", nlist);
06322     prIntArray(nlist, elist, msg);
06323 }
06324
06325 //=============================================================
06326 // class EGraph
06327 //-------------------------------------------------------------
06328 EGraph::EGraph(int nnodes, int nedges, int mxdeg)
06329 {
06330     // Arguments
06331     //   nnodes : the number of nodes
06332     //   nedges : the number of edges
06333     //   mxdeg  : the maximum number of degrees of nodes
06334
06335     int j;
06336
06337     opt     = NULL;
06338     mgraph  = NULL;
06339     econn   = NULL;
06340
06341     sNodes  = nnodes;
06342     sEdges  = nedges;
06343     sMaxdeg = mxdeg;
06344     sLoops  = sEdges - sNodes + 1;   // assume connected graph
06345
06346     pId     = -1;
06347     mId     = -1;
06348     aId     = -1;
06349     sId     =  0;
06350     gSubId  = -1;
06351     assigned = False;
06352
06353     nNodes  = sNodes;
06354     nEdges  = sEdges;
06355     nLoops  = 0;
06356     nExtern = 0;
06357
06358     nsym    = 1;
06359     esym    = 1;
06360     extperm = 1;
06361     nsym1   = 1;
06362     multp   = 1;
06363     totalc  = 0;
06364
06365     nodes = new ENode*[sNodes];
06366     for (j = 0; j < sNodes; j++) {
06367         nodes[j] = new ENode(this, sNodes, sMaxdeg);
06368     }
06369     edges = new EEdge*[sEdges];
06370     for (j = 0; j < sEdges; j++) {
06371         edges[j] = new EEdge(this, sEdges, sLoops);
06372     }
06373
06374     bidef   = new int[sNodes];
06375     bilow   = new int[sNodes];
06376     bicount = -1;
06377
06378     extMom  = new int[sEdges+1];
06379
06380     fsign   = 1;
06381     nFlines = -1;
06382     for (j = 0; j < GRCC_MAXFLINES; j++) {
06383         flines[j] = NULL;
06384     }
06385 }
06386
06387 //-------------------------------------------------------------
06388 EGraph::~EGraph(void)
06389 {
06390     int j;
06391
06392     for (j = GRCC_MAXFLINES-1; j >= 0; j--) {
06393         if (flines[j] != NULL) {
06394             delete flines[j];
06395             flines[j] = NULL;
06396         }
06397     }
06398
06399     delete[] extMom;
06400
06401     delete[] bilow;
06402     delete[] bidef;
06403
06404     for (j = 0; j < sEdges; j++) {
06405         delete edges[j];
```

```
06406         edges[j] = NULL;
06407     }
06408     delete[] edges;
06409
06410     for (j = 0; j < sNodes; j++) {
06411         delete nodes[j];
06412         nodes[j] = NULL;
06413     }
06414     delete[] nodes;
06415
06416 }
06417
06418 //-------------------------------------------------------------
06419 void EGraph::copy(EGraph *eg)
06420 {
06421     int nd, ed;
06422
06423 #ifdef CHECK
06424     if (sNodes < eg->nNodes || sEdges < eg->nEdges ||
06425         sMaxdeg < eg->sMaxdeg || sLoops < eg->nLoops) {
06426         erEnd("EGraph::copy: sizes are too small");
06427     }
06428 #endif
06429     model   = eg->model;
06430     proc    = eg->proc;
06431     sproc   = eg->sproc;
06432     mgraph  = eg->mgraph;
06433     opt     = mgraph->opt;
06434     econn   = mgraph->mconn;
06435
06436     pId     = eg->pId;
06437     mId     = eg->mId;
06438     gSubId  = eg->gSubId;
06439     nNodes  = eg->nNodes;
06440     nEdges  = eg->nEdges;
06441     nExtern = eg->nExtern;
06442     nLoops  = eg->nLoops;
06443
06444     for (nd = 0; nd < nNodes; nd++) {
06445         nodes[nd]->copy(eg->nodes[nd]);
06446     }
06447     for (ed = 0; ed < nEdges; ed++) {
06448         edges[ed]->copy(eg->edges[ed]);
06449     }
06450
06451     nsym    = eg->nsym;
06452     esym    = eg->esym;
06453     extperm = eg->extperm;
06454     nsym1   = eg->nsym1;
06455     multp   = eg->multp;
06456
06457     bicount = -1;
06458 }
06459
06460 //-------------------------------------------------------------
06461 void EGraph::setExtLoop(int nd, int val)
06462 {
06463     // set the node 'nd' being an external node (-1) or a looped vertex
06464     nodes[nd]->extloop = val;
06465 }
06466
06467 //-------------------------------------------------------------
06468 void EGraph::endSetExtLoop(void)
06469 {
06470     // end of calling 'setExtLoop'.
06471
06472     int n;
06473
06474     nExtern = 0;
06475     for (n = 0; n < nNodes; n++) {
06476         if (isExternal(n)) {
06477             nExtern++;
06478         }
06479     }
06480 }
06481
06482 //-------------------------------------------------------------
06483 void EGraph::fromDGraph(DGraph *dg)
06484 {
06485     int deg[GRCC_MAXNODES];
06486     int maxdeg, nextern, nconn, tc;
06487     int n0, n1, e;
06488
06489     if (dg->nnodes > GRCC_MAXNODES) {
06490         grcc_fprintf(GRCC_Stderr, "*** too many nodes (GRCC_MAXNODES)\n");
06491         GRCC_ABORT();
06492     }
```

```
06493     if (dg->nedges > GRCC_MAXEDGES) {
06494         grcc_fprintf(GRCC_Stderr, "*** too many edges (GRCC_MAXEDGES)\n");
06495         GRCC_ABORT();
06496     }
06497 
06498     for (e = 0; e < dg->nedges; e++) {
06499         if (dg->edges[e][0] >= dg->nnodes || dg->edges[e][0] >= dg->nnodes) {
06500             grcc_fprintf(GRCC_Stderr, "*** undefined node:");
06501             grcc_fprintf(GRCC_Stderr, "edge[%d] = {%d, %d}\n",
06502                     e, dg->edges[e][0], dg->edges[e][0]);
06503             GRCC_ABORT();
06504         }
06505     }
06506 
06507     for (n0 = 0; n0 < dg->nnodes; n0++) {
06508         deg[n0] = 0;
06509     }
06510     for (e = 0; e < dg->nedges; e++) {
06511         deg[dg->edges[e][0]]++;
06512         deg[dg->edges[e][1]]++;
06513     }
06514     maxdeg  = 0;
06515     nextern = 0;
06516     for (n0 = 0; n0 < dg->nnodes; n0++) {
06517         maxdeg = Max(maxdeg, deg[n0]);
06518         if (deg[n0] == 1) {
06519             nextern++;
06520         } if (deg[n0] < 0) {
06521             grcc_fprintf(GRCC_Stderr, "+++ node %d is isolated\n", n0);
06522         }
06523     }
06524 
06525     mgraph = NULL;
06526     model  = NULL;
06527     proc   = NULL;
06528     sproc  = NULL;
06529     opt    = NULL;
06530 
06531     nNodes  = dg->nnodes;
06532     nEdges  = dg->nedges;
06533     nLoops  = 0;
06534     nExtern = nextern;
06535 
06536     pId     = 0;
06537     mId     = 0;
06538     aId     = 0;
06539     gSubId  = 0;
06540     nsym    = 1;
06541     nsym1   = 1;
06542     esym    = 1;
06543     extperm = 1;
06544     multp   = 1;
06545     //maxdeg  = maxdeg;
06546 
06547     for (n0 = 0; n0 < dg->nnodes; n0++) {
06548         nodes[n0]->deg = 0;
06549         nodes[n0]->extloop = dg->nodes[n0].extloop;
06550     }
06551     for (e = 0; e < dg->nedges; e++) {
06552         n0 = dg->edges[e][0];
06553         n1 = dg->edges[e][1];
06554         edges[e]->nodes[0] = n0;
06555         edges[e]->nodes[1] = n1;
06556         edges[e]->ext = (isExternal(n0) || isExternal(n1));
06557         edges[e]->momdir = 0;
06558 
06559         nodes[n0]->edges[nodes[n0]->deg++] = I2Vedge(e, -1);
06560         nodes[n1]->edges[nodes[n1]->deg++] = I2Vedge(e, +1);
06561     }
06562     for (n0 = 0; n0 < dg->nnodes; n0++) {
06563         nodes[n0]->setId(this, n0);
06564         nodes[n0]->intrct = dg->nodes[n0].intrct;
06565     }
06566 
06567     for (e = 0; e < nEdges; e++) {
06568         edges[e]->setId(this, e);
06569     }
06570     tc = 0;
06571     for (n0 = 0; n0 < nNodes; n0++) {
06572         if (isExternal(n0)) {
06573             setExtern(n0, 1, GRCC_AT_Initial);
06574         } else {
06575             tc += 2*nodes[n0]->extloop + nodes[n0]->deg - 2;
06576         }
06577     }
06578     totalc = tc;
06579 
```

```
06580     // get the number of connected components
06581     nconn = connComp();
06582     nLoops = nEdges - nNodes + nconn;
06583 
06584     bicount = -1;
06585 }
06586 
06587 
06588 //------------------------------------------------------------
06589 void EGraph::fromMGraph(MGraph *mg)
06590 {
06591     // construct EGraph from adjacency matrix.
06592     // This function should be called after
06593     //   EGraph(), setExtLoop() and endSetExtLoop().
06594 
06595     int n0, n1, m0, m1, ed, e;
06596     int j, ni, nf;
06597     PNodeClass *pnc;
06598     int k;
06599 
06600 #ifdef CHECK
06601     if (sNodes < mg->nNodes || sEdges < mg->nEdges || sMaxdeg < mg->maxdeg) {
06602         erEnd("EGraph::fromMGraph: sizes are too small");
06603     }
06604     if (sLoops < mg->nLoops) {
06605         grcc_fprintf(GRCC_Stdout, "too small sLoops : %d < %d\n", sLoops, mg->nLoops);
06606         erEnd("too small sLoops");
06607     }
06608 #endif
06609     mgraph  = mg;
06610     opt     = mgraph->opt;
06611     econn   = mgraph->mconn;
06612     sproc   = opt->sproc;
06613     if (sproc == NULL) {
06614         proc  = NULL;
06615         model = NULL;
06616     } else {
06617         proc  = sproc->proc;
06618         model = sproc->model;
06619     }
06620 
06621     nNodes  = mg->nNodes;
06622     nEdges  = mg->nEdges;
06623     nLoops  = mg->nLoops;
06624     nExtern = mg->nExtern;
06625     pId     = mg->pId;
06626     mId     = mg->cDiag;
06627     aId     = -1;
06628     if (opt->proc != NULL) {
06629         mId = mg->opt->proc->mgrcount;
06630         sId = mg->opt->proc->agrcount;
06631     } else if (mg->opt->sproc != NULL) {
06632         mId = mg->opt->sproc->mgrcount;
06633         sId = mg->opt->sproc->agrcount;
06634     } else {
06635         mId = mg->cDiag;
06636         sId = mg->cDiag;
06637     }
06638     gSubId  = 0;
06639     nsym    = mg->nsym;
06640     esym    = mg->esym;
06641     extperm = 1;
06642     nsyml   = nsym;
06643     multp   = 1;
06644     maxdeg  = mg->maxdeg;
06645 
06646     totalc  = 0;
06647     if (proc != NULL) {
06648         for (j = 0; j < proc->model->ncouple; j++) {
06649             totalc += proc->clist[j];
06650         }
06651     }
06652     for (ed = 0; ed < nEdges; ed++) {
06653         edges[ed]->edtype = GRCC_ED_Undef;
06654     }
06655 
06656     ed = 0;
06657     for (n0 = 0; n0 < nNodes; n0++) {
06658         nodes[n0]->deg = 0;
06659     }
06660     for (n0 = 0; n0 < nNodes; n0++) {
06661         for (e = 0; e < mg->adjMat[n0][n0]/2; e++, ed++) {
06662             edges[ed]->nodes[0] = n0;
06663             edges[ed]->nodes[1] = n0;
06664             nodes[n0]->edges[nodes[n0]->deg++] = I2Vedge(ed, -1);
06665             nodes[n0]->edges[nodes[n0]->deg++] = I2Vedge(ed, +1);
06666             edges[ed]->ext = isExternal(n0);
```

```
06667         }
06668         for (n1 = n0+1; n1 < nNodes; n1++) {
06669             for (e = 0; e < mg->adjMat[n0][n1]; e++, ed++) {
06670                 edges[ed]->nodes[0] = n0;
06671                 edges[ed]->nodes[1] = n1;
06672                 nodes[n0]->edges[nodes[n0]->deg++] = I2Vedge(ed, -1);
06673                 nodes[n1]->edges[nodes[n1]->deg++] = I2Vedge(ed, +1);
06674                 edges[ed]->ext = (isExternal(n0) || isExternal(n1));
06675             }
06676         }
06677     }
06678 #ifdef CHECK
06679     if (ed != nEdges) {
06680         grcc_fprintf(GRCC_Stdout, "*** EGraph::init: ed=%d != nEdges=%d\n",
06681                ed, nEdges+1);
06682         erEnd("EGraph::init: illegal connection");
06683     }
06684 #endif
06685
06686     for (j = 0; j < nNodes; j++) {
06687         nodes[j]->setId(this, j);
06688         nodes[j]->intrct = mg->nodes[j]->extloop;
06689     }
06690
06691     for (j = 0; j < nEdges; j++) {
06692         edges[j]->setId(this, j);
06693     }
06694
06695     ni = 0;
06696     nf = 0;
06697     if (proc != NULL) {
06698         ni = proc->ninitl;
06699         for (j = 0; j < ni; j++) {
06700             setExtern(j, proc->initlPart[j], GRCC_AT_Initial);
06701         }
06702
06703         nf = proc->nfinal;
06704         for (j = 0; j <  nf; j++) {
06705             setExtern(j+ni, proc->finalPart[j], GRCC_AT_Final);
06706         }
06707         nExtern = ni + nf;
06708     } else if (sproc != NULL) {
06709         pnc = sproc->pnclass;
06710         for (j = 0; j < sproc->pnclass->nclass; j++) {
06711             if (pnc->type[j] == GRCC_AT_Initial
06712                 || pnc->type[j] == GRCC_AT_External) {
06713                 for (k = pnc->cl2nd[j]; k < pnc->cl2nd[j+1]; k++) {
06714                     setExtern(j, k, GRCC_AT_Initial);
06715                     ni++;
06716                 }
06717             } else if (sproc->pnclass->type[j] == GRCC_AT_Final) {
06718                 for (k = pnc->cl2nd[j]; k < pnc->cl2nd[j+1]; k++) {
06719                     setExtern(j, k, GRCC_AT_Final);
06720                     nf++;
06721                 }
06722             }
06723
06724         }
06725         nExtern = ni + nf;
06726     }
06727     totalc = 0;
06728     for (j = 0; j < nNodes; j++) {
06729         if (!isExternal(j)) {
06730             totalc += 2*nodes[j]->extloop + nodes[j]->deg - 2;
06731         }
06732     }
06733
06734     for (ed = 0; ed < nEdges; ed++) {
06735         edges[ed]->momdir = 0;
06736     }
06737     for (e = 0; e < econn->sedges; e++) {
06738         m0 = econn->cedges[e].nodes[0];
06739         m1 = econn->cedges[e].nodes[1];
06740
06741         bool found = False;
06742         for (ed = 0; ed < nEdges; ed++) {
06743             if (edges[ed]->momdir != 0) {
06744                 continue;
06745             }
06746             n0 = edges[ed]->nodes[0];
06747             n1 = edges[ed]->nodes[1];
06748             if (m0 == n0 && m1 == n1) {
06749                 edges[ed]->momdir = econn->cedges[e].momdir;
06750                 edges[ed]->momset = econn->cedges[e].momset;
06751                 found = True;
06752                 break;
06753             } else if (m0 == n1 && m1 == n0) {
```

```
06754                 edges[ed]->momdir = - econn->cedges[e].momdir;
06755                 edges[ed]->momset =   econn->cedges[e].momset;
06756                 found = True;
06757                 break;
06758             }
06759         }
06760 
06761         if (! found) {
06762             grcc_fprintf(GRCC_Stdout, "*** EGraph::fromMGraph:edge (%d->%d) is not found\n",
06763                    m0, m1);
06764         }
06765     }
06766 
06767     bicount = -1;
06768 }
06769 
06770 //-------------------------------------------------------------
06771 ENode *EGraph::setExtern(int n0, int pt, int ndtyp)
06772 {
06773     ENode *nd;
06774     int npt, ept, eg;
06775 
06776     nd = nodes[n0];
06777     if (nd->deg != 1) {
06778         if (prlevel > 0) {
06779             grcc_fprintf(GRCC_Stderr, "*** illegal external particle %d, %d, %d\n",
06780                    n0,pt,ndtyp);
06781         }
06782         erEnd("illegal external particle");
06783     }
06784 
06785     // particle comes into the node
06786     if (ndtyp == GRCC_AT_Initial) {
06787         npt = pt;
06788     } else if (ndtyp == GRCC_AT_Final) {
06789         npt = -pt;
06790     } else {
06791         npt = 0;
06792         if (prlevel > 0) {
06793             grcc_fprintf(GRCC_Stderr, "*** illegal type of an external particle : "
06794                    "node = %d, particle = %d, type = %d", n0, pt, ndtyp);
06795         }
06796         erEnd("illegal type of an external particle");
06797     }
06798 
06799     // edge attached to the external node
06800     eg = n0;
06801 
06802     // particle flows on e from leg=0 to 1.
06803     if (model != NULL){
06804         npt = model->normalParticle(npt);
06805         ept = model->normalParticle(npt);
06806     } else {
06807         npt = 0;
06808         ept = 0;
06809     }
06810 
06811     nd->setExtern(ndtyp, npt);
06812     edges[eg]->ptcl = ept;
06813 
06814     return nd;
06815 }
06816 
06817 //-------------------------------------------------------------
06818 void EGraph::print(void)
06819 {
06820     // print the EGraph
06821 
06822     int  nd, ed, nlp;
06823 
06824     nlp = nEdges - nNodes + 1;
06825     grcc_fprintf(GRCC_Stdout, "\nEGraph\n");
06826     grcc_fprintf(GRCC_Stdout, "    pId=%d, gSubId=%ld, mId=%ld, aId=%ld, sId=%ld\n",
06827            pId, gSubId, mId, aId, sId);
06828     grcc_fprintf(GRCC_Stdout, "    sNodes=%d, sEdges=%d, sMaxdeg=%d, sLoops=%d\n",
06829            sNodes, sEdges, sMaxdeg, sLoops);
06830     grcc_fprintf(GRCC_Stdout, "    nNodes=%d, nEdges=%d, nExtern=%d, nLoops=%d, totalc=%d\n",
06831            nNodes, nEdges, nExtern, nlp, totalc);
06832     grcc_fprintf(GRCC_Stdout, "    ");
06833     if (model == NULL) {
06834         grcc_fprintf(GRCC_Stdout, "model=NULL,");
06835     } else {
06836         grcc_fprintf(GRCC_Stdout, "model=\"%s\",", model->name);
06837     }
06838     if (proc == NULL) {
06839         grcc_fprintf(GRCC_Stdout, "proc=NULL,");
06840     } else {
```

```
06841         grcc_fprintf(GRCC_Stdout, "proc=%d,", proc->id);
06842     }
06843     if (sproc == NULL) {
06844         grcc_fprintf(GRCC_Stdout, "sproc=NULL,");
06845     } else {
06846         grcc_fprintf(GRCC_Stdout, "sproc=%d,", sproc->id);
06847     }
06848     grcc_fprintf(GRCC_Stdout, "\n");
06849     grcc_fprintf(GRCC_Stdout, "    assigned=%d, sym = (%ld * %ld) ",
06850             assigned, nsym, esym);
06851     grcc_fprintf(GRCC_Stdout, "extperm=%ld, nsym1=%ld, multp=%ld\n",
06852             extperm, nsym1, multp);
06853 
06854     grcc_fprintf(GRCC_Stdout, "  Nodes\n");
06855     for (nd = 0; nd < nNodes; nd++) {
06856         if (isExternal(nd)) {
06857             grcc_fprintf(GRCC_Stdout, "    %2d Extern ", nd);
06858         } else {
06859             grcc_fprintf(GRCC_Stdout, "    %2d Vertex ", nd);
06860         }
06861         nodes[nd]->print();
06862     }
06863     grcc_fprintf(GRCC_Stdout, "  Edges\n");
06864     for (ed = 0; ed < nEdges; ed++) {
06865         if (edges[ed]->ext) {
06866             grcc_fprintf(GRCC_Stdout, "    %2d Extern ", ed);
06867         } else {
06868             grcc_fprintf(GRCC_Stdout, "    %2d Intern ", ed);
06869         }
06870         edges[ed]->print();
06871     }
06872     if (bicount > 0) {
06873         grcc_fprintf(GRCC_Stdout, "  Biconn: nopicomp=%d, nopi2p=%d, opi2plp=%d, nadj2ptv=%d\n",
06874                 nopicomp, nopi2p, opi2plp, nadj2ptv);
06875     }
06876     grcc_fprintf(GRCC_Stdout, "\n");
06877 
06878     if (nFlines > 0) {
06879         prFLines();
06880     }
06881 }
06882 
06883 //-------------------------------------------------------------
06884 void EGraph::printPy(FILE *fp, long mId)
06885 {
06886     if (!assigned) {
06887         mgraph->printPy(fp, mId);
06888         return;
06889     }
06890 
06891     fprintf(fp, "#AGraph : (gseq, {spid, gid, ...}, nodes, node-group)\n");
06892     fprintf(fp, "(%ld,\n", mId);
06893 
06894     //  {'proc':0, 'subproc':0, 'tgraph':1, ... }
06895     fprintf(fp, "  {'proc':%d, 'subproc':%d, ", proc->id, sproc->id);
06896     fprintf(fp, "'tgraph':%ld, ", mId);
06897     fprintf(fp, "'OPI':%s, ", BOOLSTR(opt->values[GRCC_OPT_1PI]));
06898     fprintf(fp, "'agraph':%ld, \n", aId);
06899     fprintf(fp, "   'nnodes':%d, 'nnsym':%ld, 'nesym':%ld, },\n",
06900             nNodes, nsym, esym);
06901     //  #nodes
06902     fprintf(fp, "  [ #nodes\n");
06903     for (int nd = 0; nd < nNodes; nd++) {
06904         ENode *end = nodes[nd];
06905 
06906         // #node  0: external=phi
06907         // #node  4: phi3=0
06908         fprintf(fp, "    [ #node  %d: ", nd);
06909         fprintf(fp, "\n");
06910 
06911         // {'deg':1, 'cid':-3, ..., }
06912         fprintf(fp, "      {'deg':%d, 'cid':%d, 'extern':%s, ",
06913                 end->deg, 0, BOOLSTR(isExternal(nd)));
06914         fprintf(fp, "'intr':%d, 'loop':%d, },\n",
06915                 end->intrct, end->extloop);
06916 
06917         // legs : (edge, leg, particle)
06918         // [ (0, 1, 0), ... ],
06919         fprintf(fp, "      [ ");
06920         for (int lg = 0; lg < end->deg; lg++) {
06921             int ed = V2Iedge(end->edges[lg]);
06922             int ptcl = edges[ed]->ptcl;
06923             int edlg = (edges[ed]->nodes[0] == nd) ? 1 : 0;
06924             int nnd  = edges[ed]->nodes[edlg];
06925             int nlg  = edges[ed]->nlegs[edlg];
06926             fprintf(fp, "(%d, %d, %d), ", nnd, ptcl, nlg);
06927         }
```

```
06928         fprintf(fp, "],\n");
06929         fprintf(fp, "    ],\n");   // end #node
06930     }
06931     fprintf(fp, "  ], #node\n");
06932     fprintf(fp, "  [ # group of 1 elements\n");
06933     fprintf(fp, "    [],    #\n");
06934     fprintf(fp, "  ], # group\n");
06935     fprintf(fp, ")\n");
06936 }
06937
06938 //------------------------------------------------------------
06939 void EGraph::prFLines(void)
06940 {
06941     int j;
06942
06943     grcc_fprintf(GRCC_Stdout, "  Fermion lines %d, sign=%d (mId=%ld, aId=%ld)\n",
06944             nFlines, fsign, mId, aId);
06945     for (j = 0; j < nFlines; j++) {
06946         grcc_fprintf(GRCC_Stdout, "%4d ", j);
06947         flines[j]->print("\n");
06948     }
06949 }
06950
06951 //------------------------------------------------------------
06952 int EGraph::dirEdge(int n, int e)
06953 {
06954     if (nodes[n]->edges[e] > 0) {
06955         return 1;
06956     } else {
06957         return -1;
06958     }
06959 }
06960
06961 //------------------------------------------------------------
06962 int EGraph::cmpMom(int *lm0, int *em0, int *lm1, int *em1)
06963 {
06964     int lp, ed, cmp, sgn = 0;
06965
06966     for (lp = 0; lp < nLoops; lp++) {
06967         if (lm0[lp] != 0) {
06968             if (lm1[lp] == 0) {
06969                 return 1;
06970             } else if (sgn == 0) {
06971                 sgn = lm0[lp]*lm1[lp];   // cancel first non-zero elements
06972 #ifdef CHECK
06973                 if (Abs(sgn) != 1) {
06974                     erEnd("cmpMom : illegal element of lmom");
06975                 }
06976 #endif
06977             } else {
06978                 cmp = lm0[lp] - sgn*lm1[lp];
06979                 if (cmp != 0) {
06980                     return cmp;
06981                 }
06982             }
06983         } else if (lm1[lp] != 0) {   // lm0[lp] == 0
06984             return -1;
06985         }
06986     }
06987     for (ed = 0; ed < nEdges; ed++) {
06988         if (em0[ed] != 0) {
06989             if (em1[ed] == 0) {
06990                 return 1;
06991             } else if (sgn == 0) {
06992                 sgn = em0[ed]*em1[ed];   // cancel first non-zero elements
06993 #ifdef CHECK
06994                 if (Abs(sgn) != 1) {
06995                     erEnd("cmpMom : illegal element of emom");
06996                 }
06997 #endif
06998             } else {
06999                 cmp = em0[ed] - sgn*em1[ed];
07000                 if (cmp != 0) {
07001                     return cmp;
07002                 }
07003             }
07004         } else if (em1[ed] != 0) {   // em0[ed] == 0
07005             return -1;
07006         }
07007     }
07008     return 0;
07009 }
07010
07011 //------------------------------------------------------------
07012 int EGraph::groupLMom(int *grp, int *ed2gr)
07013 {
07014     //  grp[g] : the number of elements in the group g (in [0,...,(ngrp-1)])
```

```
07015     //  ed2gr[ed] : group number of edge ed.
07016
07017     int ed0, ed1, cmp, ngrp = -1;
07018
07019     for (ed0 = 0; ed0 < nEdges; ed0++) {
07020         ed2gr[ed0] = -1;
07021         grp[ed0]   = 0;
07022     }
07023
07024     for (ed0 = 0; ed0 < nEdges; ed0++) {
07025         if (ed2gr[ed0] < 0) {
07026             ngrp++;
07027             ed2gr[ed0] = ngrp;
07028             grp[ngrp]++;
07029             for (ed1 = ed0+1; ed1 < nEdges; ed1++) {
07030                 cmp = cmpMom(edges[ed0]->lmom, edges[ed0]->emom,
07031                              edges[ed1]->lmom, edges[ed1]->emom);
07032                 if (cmp == 0) {
07033                     ed2gr[ed1] = ngrp;
07034                     grp[ngrp]++;
07035                 }
07036             }
07037         }
07038     }
07039     ngrp++;
07040
07041     return ngrp;
07042 }
07043
07044 //------------------------------------------------------------
07045 Bool EGraph::optQGrafM(Options *opt)
07046 {
07047     int *qgopt = opt->qgopt;
07048     int nopis[GRCC_MAXEDGES];
07049     ULong mext = 0;
07050     mext = (~ mext) « nExtern;
07051
07052 #ifdef PRINT
07053     grcc_fprintf(GRCC_Stdout, "optQGrafM:");
07054     print();
07055 #endif
07056
07057 #ifdef PRINT
07058     grcc_fprintf(GRCC_Stdout, "optQGrafM: %8ld\n", mId);
07059     econn->print();
07060 #endif
07061
07062     // count the number of self-energy 1PI components.
07063     int maxlegs = 0;
07064     for (int j = 0; j < econn->nopic; j++) {
07065         maxlegs = Max(maxlegs, econn->opics[j].nlegs);
07066     }
07067     if (maxlegs >= GRCC_MAXEDGES) {
07068         grcc_fprintf(GRCC_Stdout, "*** table overflow\n");
07069         GRCC_ABORT();
07070     }
07071     for (int k = 0; k < GRCC_MAXEDGES; k++) {
07072         nopis[k] = 0;
07073     }
07074     for (int j = 0; j < econn->nopic; j++) {
07075         nopis[econn->opics[j].nlegs]++;
07076     }
07077
07078     //
07079     if (qgopt[GRCC_QGRAF_OPT_ONEPI] > 0) {
07080         if (econn->nopic != 1) {
07081             return False;
07082         }
07083     } else if (qgopt[GRCC_QGRAF_OPT_ONEPI] < 0) {
07084         if (econn->nopic < 2) {
07085             return False;
07086         }
07087     }
07088
07089     if (qgopt[GRCC_QGRAF_OPT_ONSHELL] != 0) {
07090         if (nExtern == 1) {
07091             if (qgopt[GRCC_QGRAF_OPT_ONSHELL] > 0) {
07092                 if (econn->nopic != 1) {
07093                     return False;
07094                 }
07095             } else if (qgopt[GRCC_QGRAF_OPT_ONSHELL] < 0) {
07096                 if (econn->nopic == 1) {
07097                     return False;
07098                 }
07099             }
07100         } else {
07101             if (qgopt[GRCC_QGRAF_OPT_ONSHELL] > 0) {
```

```
07102                 if (econn->ne1bridges > 0) {
07103                     return False;
07104                 }
07105             } else if (qgopt[GRCC_QGRAF_OPT_ONSHELL] < 0) {
07106                 if (econn->ne1bridges <= 0) {
07107                     return False;
07108                 }
07109             }
07110         }
07111     }
07112
07113     if (qgopt[GRCC_QGRAF_OPT_NOSNAIL] != 0) {
07114         if (qgopt[GRCC_QGRAF_OPT_NOSNAIL] > 0) {
07115             if (nExtern == 1) {
07116                 if (econn->nblocks != 1) {
07117                     return False;
07118                 }
07119             } else {
07120                 if (mgraph->mconn->ne0bridges >= 1 ||
07121                     mgraph->mconn->na1blocks  >= 1) {
07122                     return False;
07123                 }
07124             }
07125         } else if (qgopt[GRCC_QGRAF_OPT_NOSNAIL] < 0) {
07126             if (nExtern == 1) {
07127                 if (econn->nblocks == 1) {
07128                     return False;
07129                  }
07130             } else {
07131                 if (mgraph->mconn->ne0bridges < 1 &&
07132                     mgraph->mconn->na1blocks  < 1) {
07133                     return False;
07134                 }
07135             }
07136         }
07137     }
07138
07139     if (qgopt[GRCC_QGRAF_OPT_NOTADPOLE] != 0) {
07140         if (nExtern == 1) {
07141             if (qgopt[GRCC_QGRAF_OPT_NOTADPOLE] > 0) {
07142                 if (econn->nopic != 1) {
07143                     return False;
07144                 }
07145             } else if (qgopt[GRCC_QGRAF_OPT_NOTADPOLE] < 0) {
07146                 if (econn->nopic == 1) {
07147                     return False;
07148                 }
07149             }
07150         } else {
07151             if (qgopt[GRCC_QGRAF_OPT_NOTADPOLE] > 0) {
07152                 if (mgraph->mconn->ne0bridges != 0) {
07153                     return False;
07154                 }
07155             } else if (qgopt[GRCC_QGRAF_OPT_NOTADPOLE] < 0) {
07156                 if (mgraph->mconn->ne0bridges == 0) {
07157                     return False;
07158                 }
07159             }
07160         }
07161     }
07162
07163     if (qgopt[GRCC_QGRAF_OPT_NOSIGMA] != 0) {
07164         bool ok = True;
07165         if (nExtern != 2) {
07166             if (nopis[2] > 1) {
07167                 ok = False;
07168             }
07169         }
07170         for (int j = 0; j < econn->nopic; j++) {
07171             if (econn->opics[j].nlegs >= 2 &&
07172                 econn->opics[j].nlegs == econn->opics[j].mom0lg) {
07173                 ok = False;
07174             }
07175         }
07176
07177         // loop momenta
07178         for (int j = 0; j < econn->sedges - 1; j++) {
07179             ULong momj = econn->cedges[j].momset;
07180             if (momj == 0) {
07181                 continue;
07182             }
07183             int extj = 0;
07184             if (econn->cedges[j].nodes[0] < nExtern) {
07185                 extj = 1;
07186             } else if (econn->cedges[j].nodes[1] < nExtern) {
07187                 extj = 1;
07188             }
```

```
07189             for (int k = j+1; k < econn->sedges; k++) {
07190                 ULong momk = econn->cedges[k].momset;
07191                 if (momk == 0) {
07192                      continue;
07193                 }
07194                 int extk = 0;
07195                 if (econn->cedges[k].nodes[0] < nExtern) {
07196                     extk = 1;
07197                 } else if (econn->cedges[k].nodes[1] < nExtern) {
07198                     extk = 1;
07199                 }
07200                 if ( momj == momk) {
07201                     if (nExtern == 2) {
07202                         if (extj + extk != 2) {
07203                             ok = False;
07204                         }
07205                     } else {
07206                         ok = False;
07207                     }
07208                 }
07209             }
07210         }
07211         if (qgopt[GRCC_QGRAF_OPT_NOSIGMA] > 0) {
07212             if (! ok) {
07213                 return False;
07214             }
07215         } else if (qgopt[GRCC_QGRAF_OPT_NOSIGMA] < 0) {
07216             if (ok) {
07217                 return False;
07218             }
07219         }
07220     }
07221
07222     if (qgopt[GRCC_QGRAF_OPT_SIMPLE] > 0) {
07223         if (mgraph->selfloop || mgraph->multiedge) {
07224             return False;
07225         }
07226     } else if (qgopt[GRCC_QGRAF_OPT_SIMPLE] < 0) {
07227         if (!mgraph->selfloop && !mgraph->multiedge) {
07228             return False;
07229         }
07230     }
07231
07232     if (qgopt[GRCC_QGRAF_OPT_BIPART] > 0) {
07233         if (! mgraph->bipart) {
07234             return False;
07235         }
07236     } else if (qgopt[GRCC_QGRAF_OPT_BIPART] < 0) {
07237         if (mgraph->bipart) {
07238             return False;
07239         }
07240     }
07241     // GRCC_QGRAF_OPT_CYCLI
07242     if (qgopt[GRCC_QGRAF_OPT_CYCLI] != 0) {
07243         int nb = 0;
07244         for (int k = 0; k < econn->nblocks; k++) {
07245             if (econn->blocks[k].loop > 0) {
07246                 nb++;
07247             }
07248         }
07249         if (qgopt[GRCC_QGRAF_OPT_CYCLI] > 0) {
07250             if (nb > 1) {
07251                 return False;
07252             }
07253         } else if (qgopt[GRCC_QGRAF_OPT_CYCLI] < 0) {
07254             if (nb <= 1) {
07255                 return False;
07256             }
07257         }
07258     }
07259
07260     return True;
07261 }
07262
07263 //------------------------------------------------------------------
07264 Bool EGraph::optQGrafA(Options *opt)
07265 {
07266 #ifdef PRINT
07267     grcc_fprintf(GRCC_Stdout, "optQGrafA: %8ld\n", mId);
07268     econn->print();
07269 #endif
07270     Bool retval = True;
07271     if (opt->qgopt[GRCC_QGRAF_OPT_FLOOP] != 0) {
07272         for (int fl=0; fl < nFlines; fl++) {
07273             if (flines[fl]->ftype == FL_Closed) {
07274                 if (flines[fl]->nlist % 2 != 0) {
07275                     retval = False;
```

```
07276                     break;
07277                 }
07278             }
07279         }
07280         // `notfloop_' is the dual of `floop_', so flip the decision:
07281         if (opt->qgopt[GRCC_QGRAF_OPT_FLOOP] == -1) {
07282             retval = (retval == True ? False : True);
07283         }
07284     }
07285     return retval;
07286 }
07287
07288 //------------------------------------------------------------
07289 Bool EGraph::isOptE(void)
07290 {
07291     EGraph edupv = EGraph(sNodes, sEdges, sMaxdeg);
07292     EGraph *edup = &edupv;
07293     int    grp[GRCC_MAXEDGES], ed2gr[GRCC_MAXEDGES];
07294     int    g, ed, ngrp;
07295     Bool   ok;
07296     int    minopi2p;
07297
07298
07299  // if (opt->values[GRCC_OPT_No2PtL1PI] == 0
07300  //     && opt->values[GRCC_OPT_NoAdj2PtV] == 0) {
07301  //     return True;
07302  // }
07303
07304     for (ed = 0; ed < nEdges; ed++) {
07305         edges[ed]->cut = False;
07306     }
07307
07308     biconnE();
07309
07310     // QGraf options
07311     if (! optQGrafM(opt)) {
07312         return False;
07313     }
07314
07315     if (opt->values[GRCC_OPT_NoAdj2PtV] > 0) {
07316         if (nadj2ptv > 0) {
07317             return False;
07318         }
07319     } else if (opt->values[GRCC_OPT_NoAdj2PtV] < 0) {
07320         if (nadj2ptv < 1) {
07321             return False;
07322         }
07323     }
07324     if (opt->values[GRCC_OPT_No2PtL1PI] == 0) {
07325         return True;
07326     }
07327
07328     if (nExtern == 2 && nopicomp == 1) {
07329         minopi2p = 2;
07330     } else {
07331         minopi2p = 1;
07332     }
07333
07334     if (opi2plp > 0 && nopi2p >= minopi2p) {
07335         if (opt->values[GRCC_OPT_No2PtL1PI] > 0) {
07336             return False;
07337         } else if (opt->values[GRCC_OPT_No2PtL1PI] < 0) {
07338             return True;
07339         }
07340     }
07341
07342     edup->copy(this);
07343     ngrp = groupLMom(grp, ed2gr);
07344
07345     for (g = 0; g < ngrp; g++) {
07346         ok = True;
07347         if (grp[g] < 2) {
07348             continue;
07349         }
07350         for (ed = 0; ed < nEdges; ed++) {
07351             if (ed2gr[ed] == g) {
07352                 if (edges[ed]->ext) {
07353                     ok = False;
07354                     break;
07355                 }
07356                 edup->edges[ed]->cut = True;
07357             } else {
07358                 edup->edges[ed]->cut = False;
07359             }
07360         }
07361         if (ok) {
07362             edup->gSubId++;
```

```
07363             edup->biconnE();
07364
07365             if (edup->nExtern == 2 && edup->nopicomp == 1) {
07366                 minopi2p = 2;
07367             } else {
07368                 minopi2p = 1;
07369             }
07370
07371             if (edup->opi2plp > 0 && edup->nopi2p >= minopi2p) {
07372                 if (opt->values[GRCC_OPT_No2PtL1PI] > 0) {
07373                     return False;
07374                 } else if (opt->values[GRCC_OPT_No2PtL1PI] < 0) {
07375                     return True;
07376                 }
07377             }
07378         }
07379     }
07380
07381     if (opt->values[GRCC_OPT_No2PtL1PI] > 0) {
07382         return True;
07383     } else if (opt->values[GRCC_OPT_No2PtL1PI] < 0) {
07384         return False;
07385     } else {
07386         erEnd("isOptE: illegal control");
07387         return False;
07388     }
07389 }
07390
07391 //-------------------------------------------------------------
07392 int EGraph::findRoot(void)
07393 {
07394     int root, vr, er, e, nd0, nd1;
07395
07396     root = -1;
07397     vr   = -1;
07398     er   = -1;
07399     for (e = 0; e < nEdges; e++) {
07400         if (edges[e]->cut || edges[e]->edtype == GRCC_ED_Deleted) {
07401             continue;
07402         }
07403         if (edges[e]->visited) {
07404             continue;
07405         }
07406         if (root < 0) {
07407             if (edges[e]->ext) {
07408                 nd0 = edges[e]->nodes[0];
07409                 nd1 = edges[e]->nodes[1];
07410                 if (isExternal(nd0) && !isExternal(nd1)) {
07411                     root = nd1;
07412                     break;
07413                 } else if (!isExternal(nd0) && isExternal(nd1)) {
07414                     root = nd0;
07415                     break;
07416                 }
07417             }
07418         }
07419         if (er < 0) {
07420             er = edges[e]->nodes[0];
07421         }
07422         if (vr < 0) {
07423             vr = edges[e]->nodes[0];
07424         }
07425     }
07426     // if root >= 0 : vertex adjacent to an external node
07427     if (root < 0) {
07428         // if vr >= 0 : a vertex
07429         if (vr >= 0) {
07430             root = vr;
07431
07432         // no vertex, only external nodes;
07433         } else {
07434             root = er;
07435         }
07436     }
07437     return root;
07438 }
07439
07440 //-------------------------------------------------------------
07441 int EGraph::connComp(void)
07442 {
07443     //  Count the number of connected component
07444     //  If a connected component without free leg is not the whole graph then
07445     //  return False, otherwise return True.
07446
07447     int j, ncc, nelem;
07448
07449     for (j = 0; j < nNodes; j++) {
```

```
07450         nodes[j]->visited = -1;
07451     }
07452     ncc   = 0;    // # connected components
07453     nelem = 0;    // # visited nodes
07454     while (nelem < nNodes) {
07455         for (j = 0; j < nNodes; j++) {
07456             if (nodes[j]->visited < 0) {
07457                 break;
07458             }
07459         }
07460         if (j >= nNodes) {
07461             break;
07462         }
07463         nelem += connVisit(j, ncc);
07464         ncc++;
07465     }
07466     if (nelem != nNodes) {
07467         erEnd("EGraph::connComp: illegal control");
07468     }
07469     return ncc;
07470 }
07471
07472 //-------------------------------------------------------------
07473 int EGraph::connVisit(int nd, int ncc)
07474 {
07475     //  Visiting connected node used for 'connComp'
07476
07477     int e, en, nn, j, nelem;
07478
07479     nelem = 1;
07480     nodes[nd]->visited = ncc;
07481     for (e = 0; e < nodes[nd]->deg; e++) {
07482         en = V2Iedge(nodes[nd]->edges[e]);
07483         for (j = 0; j < 2; j++) {
07484             nn = edges[en]->nodes[j];
07485             if (nodes[nn]->visited < 0) {
07486                 nelem += connVisit(nn, ncc);
07487             }
07488         }
07489     }
07490     return nelem;
07491 }
07492
07493 //-------------------------------------------------------------
07494 void EGraph::biconnE(void)
07495 {
07496     int e, ie, root;
07497     int extlst[GRCC_MAXEDGES], intlst[GRCC_MAXEDGES];
07498     int opiext, opiloop;
07499
07500     nadj2ptv = 0;
07501     for (e = 0; e < nEdges; e++) {
07502         if (nodes[edges[e]->nodes[0]]->deg == 2 &&
07503             nodes[edges[e]->nodes[1]]->deg == 2 &&
07504             edges[e]->nodes[0] != edges[e]->nodes[1]) {
07505             nadj2ptv++;
07506         }
07507     }
07508
07509     biinitE();      // initialization
07510     bconn = 0;      // the number of connected components
07511
07512     opiext  = 0;
07513     opiloop = 0;
07514
07515     for (e = 0; e < nEdges; e++) {
07516         // root of a spanning tree to be searched.
07517         root = findRoot();
07518         if (root < 0) {
07519             // no root : end of search
07520             break;
07521         }
07522
07523         // recursive search
07524         bisearchE(root, extlst, intlst, &opiext, &opiloop);
07525
07526         // opi2plp = Max(opi2plp, opiloop);      // ???
07527
07528         // adjust the #external for the root
07529         if (isExternal(root)) {
07530             opiext++;
07531         }
07532
07533         // 2 point function
07534         if (opiext == 2) {
07535             opi2plp = Max(opi2plp, opiloop);
07536             nopi2p++;
```

```
07537         }
07538         for (ie = 0; ie < nEdges; ie++) {
07539             if (intlst[ie]) {
07540                 edges[ie]->opicomp = nopicomp;
07541             }
07542         }
07543         nopicomp++;
07544 
07545         bconn++;    // the number of connected component
07546     }
07547 
07548     // momentum conservation of external particles
07549     extMomConsv();
07550 
07551 #ifdef CHECK
07552     chkMomConsv();
07553 #endif
07554     return;
07555 }
07556 
07557 //------------------------------------------------------------
07558 void EGraph::biinitE(void)
07559 {
07560     int n, j, e;
07561 
07562     bicount = 0;                  // counter of visiting node
07563     loopm   = 0;                  // # of independent loop momentum
07564 
07565     nopicomp = 0;
07566     opi2plp  = 0;
07567     nopi2p   = 0;
07568 
07569     for (n = 0; n < nNodes; n++) {
07570         bidef[n] = -1;
07571         bilow[n] = -1;
07572         nodes[n]->ndtype  = GRCC_ND_Undef;
07573         for (j = 0; j < nLoops; j++) {
07574             nodes[n]->klow[j] = -1;
07575         }
07576     }
07577 
07578     for (e = 0; e < nEdges; e++) {
07579         // edges[e]->cut is defined before
07580         edges[e]->visited = False;
07581         edges[e]->conid   = -1;                  // connected component
07582         edges[e]->edtype  = GRCC_ED_Undef;
07583         edges[e]->opicomp = -1;
07584         for (j = 0; j < nEdges; j++) {
07585             edges[e]->emom[j] = 0;            // coefficients of ext. mom.
07586         }
07587         for (j = 0; j < nLoops; j++) {
07588             edges[e]->lmom[j] = 0;            // coefficients of loop mom.
07589         }
07590     }
07591 }
07592 
07593 //------------------------------------------------------------
07594 void EGraph::bisearchE(int nd, int *extlst, int *intlst, int *opiext, int *opiloop)
07595 {
07596     // Search in the spanning tree below 'nd'.
07597     // Arguments:
07598     //   nd      : input  : node to be visited
07599     //   extlst  : output : set of external lines in the current 1PI comp.
07600     //   intlst  : output : set of internal lines in the current 1PI comp.
07601     //   opiext  : output : no. external lines of the current 1PI component
07602     //   opiloop : output : no. loops of the current 1PI component
07603     //
07604     // EGraph variables
07605     //   bicount   : counter of visiting node
07606     //   loopm     : no. of independent loop momentum
07607     //   nopicomp  : no. of OPI components
07608     //   opi2plp   : maximum number loops among 2-point looped OPI components
07609     //   nopi2p    : no. of 2-point OPI components
07610     //   edges[ie]->opicomp : OPI component no. of the edge
07611     //
07612 
07613     int k, e, ie, ed, td, dir, j;
07614     int opiext1, opiloop1;
07615     int extlst1[GRCC_MAXEDGES];   // set of external nodes below
07616     int intlst1[GRCC_MAXEDGES];   // set of vertices in the 1PI component
07617 
07618     (*opiext)  = 0;
07619     (*opiloop) = 0;
07620 
07621     // visit node 'nd'
07622     bidef[nd] = bicount;
07623     bilow[nd] = bicount;
```

```
07624     for (k = 0; k < nLoops; k++) {
07625         nodes[nd]->klow[k] = bicount;
07626     }
07627     bicount++;
07628     for (j = 0; j < nEdges; j++) {
07629         extlst[j] = False;
07630         intlst[j] = False;
07631     }
07632
07633     // go to children : nd --> ed --> td
07634     for (e = 0; e < nodes[nd]->deg; e++) {
07635         ed = V2Iedge(nodes[nd]->edges[e]);
07636
07637         // check this edge is already visited
07638         if (edges[ed]->edtype == GRCC_ED_Deleted) {
07639             // permanently deleted edge
07640             continue;
07641         } else if (edges[ed]->visited) {
07642             // already visited (backword visit)
07643             continue;
07644         } else if (edges[ed]->cut) {
07645             // cut edge : treat as nd is an external
07646             (*opiext)++;
07647             nodes[nd]->setType(GRCC_ND_CPoint);
07648             continue;
07649         } else if (!isExternal(nd) && edges[ed]->ext) {
07650             // external
07651             edges[ed]->setType(GRCC_ED_Extern);
07652             edges[ed]->visited = True;
07653             edges[ed]->emom[ed] = 1;
07654             extlst[ed] = True;
07655             (*opiext)++;
07656             nodes[nd]->setType(GRCC_ND_CPoint);
07657             continue;
07658         }
07659
07660         // mark visited
07661         edges[ed]->visited = True;
07662
07663         // momentum is assigned on the backward move. (nd) 1 --> 0 (td)
07664         td  = edges[ed]->nodes[0];
07665         dir = 1;
07666         if (td == nd) {
07667             td = edges[ed]->nodes[1];
07668             dir = -1;
07669         }
07670
07671         // biconnected component
07672         if (bidef[td] >= 0) {
07673             // a back edge is found. Already visited. Don't go further
07674             bilow[nd] = Min(bilow[nd], bilow[td]);
07675             edges[ed]->setType(GRCC_ED_Back);
07676             intlst[ed] = True;
07677             (*opiloop)++;
07678
07679             // create new loop momentum
07680             k = loopm++;
07681             nodes[nd]->klow[k] = Min(nodes[nd]->klow[k], nodes[td]->klow[k]);
07682             edges[ed]->setLMom(k, dir);
07683
07684             // self-loop
07685             if (td == nd) {
07686                 nodes[nd]->setType(GRCC_ND_CPoint);
07687             }
07688
07689         } else {
07690             // not a back edge
07691             // go down further
07692             bisearchE(td, extlst1, intlst1, &opiext1, &opiloop1);
07693
07694             // the set of external particles
07695             for (j = 0; j < nEdges; j++) {
07696                 extlst[j] = extlst[j] || extlst1[j];
07697             }
07698
07699             // loop momenta
07700             for (k = 0; k < nLoops; k++) {
07701                 if (nodes[td]->klow[k] <= nodes[nd]->klow[k]) {
07702                     // inside loop 'k'
07703                     edges[ed]->setLMom(k, dir);
07704                 }
07705                 nodes[nd]->klow[k] = Min(nodes[nd]->klow[k], nodes[td]->klow[k]);
07706             }
07707
07708             // cut point ?
07709             if (bilow[td] >= bidef[nd]) {
07710                 // a cut point is found
```

```
07711                 if (nodes[nd]->ndtype == GRCC_ND_Undef) {
07712                     nodes[nd]->setType(GRCC_ND_CPoint);
07713 #ifdef CHECK
07714                 } else if (nodes[nd]->ndtype != GRCC_ND_CPoint) {
07715                     if (prlevel > 0) {
07716                         grcc_fprintf(GRCC_Stderr, "bisearch: node %d is a cut point ",
07717                                 nd);
07718                         grcc_fprintf(GRCC_Stderr, "(not undef %d)\n",
07719                                 nodes[nd]->ndtype);
07720                     }
07721 #endif
07722                 }
07723             }  // cut point
07724
07725             // bridge ?
07726             if (bilow[td] > bidef[nd]) {
07727                 // a bridge is found
07728                 if (edges[ed]->edtype == GRCC_ED_Undef) {
07729                     edges[ed]->setType(GRCC_ED_Bridge);
07730                     nodes[td]->setType(GRCC_ND_CPoint);
07731 #ifdef CHECK
07732                 } else if (edges[ed]->edtype != GRCC_ED_Bridge) {
07733                     if (prlevel > 0) {
07734                         grcc_fprintf(GRCC_Stderr, "bisearch: edges %d is a bridge ", ed);
07735                         grcc_fprintf(GRCC_Stderr, "(not undef %d)\n", edges[ed]->edtype);
07736                     }
07737 #endif
07738                 }
07739
07740                 if (!edges[ed]->ext) {
07741                     // internal bridge
07742                     opiext1++;            // from bridge
07743                     for (ie = 0; ie < nEdges; ie++) {
07744                         if (intlst1[ie]) {
07745                             // OPI component No.
07746                             edges[ie]->opicomp = nopicomp;
07747                             intlst1[ie] = False;
07748                         }
07749                     }
07750                     // 2pt OPI component
07751                     if (opiext1 == 2) {
07752                         opi2plp = Max(opi2plp, opiloop1);
07753
07754                         // the number of 2pt looped OPI components
07755                         nopi2p++;
07756                     }
07757                     // the number of OPI components
07758                     nopicomp++;
07759                 }
07760                 opiext1  = 1;    // from bridge
07761                 opiloop1 = 0;
07762
07763             } else {
07764                 // not a bridge
07765
07766                 bilow[nd] = Min(bilow[nd], bilow[td]);
07767
07768                 if (edges[ed]->edtype == GRCC_ED_Undef) {
07769                     edges[ed]->setType(GRCC_ED_Inloop);
07770 #ifdef CHECK
07771                 } else if (edges[ed]->edtype != GRCC_ED_Inloop) {
07772                     if (prlevel > 0) {
07773                         grcc_fprintf(GRCC_Stderr, "bisearch: ");
07774                         grcc_fprintf(GRCC_Stderr, "edges %d is not undef (%d)\n",
07775                                 ed, edges[ed]->edtype);
07776                     }
07777 #endif
07778                 }
07779                 intlst1[ed] = True;
07780             }
07781
07782             // merge to the current variables
07783             for (ie = 0; ie < nEdges; ie++) {
07784                 intlst[ie] = intlst[ie] || intlst1[ie];
07785             }
07786             (*opiext)  += opiext1;
07787             (*opiloop) += opiloop1;
07788
07789             // linear combination of external momenta
07790             edges[ed]->setEMom(nEdges, extlst1, dir);
07791
07792         }  // end of a child
07793     }  // for (ed)
07794
07795     return;
07796 }
07797
```

```
07798 //-------------------------------------------------------------
07799 void EGraph::extMomConsv(void)
07800 {
07801     int e, ex, lex, rs;
07802 
07803     lex = -1;
07804     for (e = 0; e < nEdges; e++) {
07805         if (edges[e]->ext) {
07806             lex = e;
07807             extMom[e] = edges[e]->emom[e];
07808         } else {
07809             extMom[e] = 0;
07810         }
07811     }
07812     // eliminate momentum of the last external particle
07813     if (lex < 0) {
07814         return;
07815     }
07816     for (e = 0; e < nEdges; e++) {
07817         rs = edges[e]->emom[lex];
07818         if (rs != 0) {
07819             for (ex = 0; ex < nEdges; ex++) {
07820                 edges[e]->emom[ex] -= rs*extMom[ex];
07821             }
07822         }
07823     }
07824 }
07825 
07826 //-------------------------------------------------------------
07827 void EGraph::chkMomConsv(void)
07828 {
07829     // check momentum conservation
07830 
07831     int  esum[GRCC_MAXEDGES];
07832     int  lsum[GRCC_MAXNODES];
07833     Bool ok, okn;
07834     int  n, ex, lk, ej, e, dir;
07835 
07836     if (bicount < 1) {
07837         return;
07838     }
07839     ok = True;
07840     for (n = 0; n < nNodes; n++) {
07841         if (isExternal(n)) {
07842             continue;
07843         }
07844         for (ex = 0; ex < nEdges; ex++) {
07845             esum[ex] = 0;
07846         }
07847         for (lk = 0; lk < nLoops; lk++) {
07848             lsum[lk] = 0;
07849         }
07850         for (ej = 0; ej < nodes[n]->deg; ej++) {
07851             e = V2Iedge(nodes[n]->edges[ej]);
07852             if (edges[e]->nodes[0] == edges[e]->nodes[0]) {
07853                 continue;
07854             }
07855             dir = dirEdge(n, ej);
07856             for (ex = 0; ex < nEdges; ex++) {
07857                 if (edges[e]->ext) {
07858                     esum[ex] += dir*edges[e]->emom[ex];
07859                 }
07860             }
07861             for (lk = 0; lk < nLoops; lk++) {
07862                 lsum[lk] += dir*edges[e]->lmom[lk];
07863             }
07864         }
07865 
07866         okn = True;
07867         for (ex = 0; ex < nEdges; ex++) {
07868             if (esum[ex] != 0) {
07869                 okn = False;
07870                 grcc_fprintf(GRCC_Stderr, "chkMomConsv:n=%d, esum[%d]=%d\n",
07871                         n, ex, esum[ex]);
07872             }
07873         }
07874 
07875         for (lk = 0; lk < nLoops; lk++) {
07876             if (lsum[lk] != 0) {
07877                 okn = False;
07878                 grcc_fprintf(GRCC_Stderr, "chkMomConsv:n=%d, lsum[%d]=%d\n",
07879                         n, lk, lsum[lk]);
07880 
07881             }
07882         }
07883 
07884         if (!okn) {
```

```
07885             ok = False;
07886             grcc_fprintf(GRCC_Stderr, "*** Violation of momentum conservation ");
07887             grcc_fprintf(GRCC_Stderr, "at node =%d\n",n);
07888         }
07889     }
07890
07891     if (!ok) {
07892         print();
07893         erEnd("inconsistent momentum");
07894     }
07895 }
07896
07897 //------------------------------------------------------------
07898 int  EGraph::legParticle(int ed, int el)
07899 {
07900     // outgoing particle from (edge, leg) = (ed, el)
07901
07902     return model->normalParticle((2*el-1)*edges[ed]->ptcl);
07903 }
07904
07905 //------------------------------------------------------------
07906 int  EGraph::isFermion(int ed)
07907 {
07908     // return if particle on the edge ed follows Fermi statistics or not.
07909
07910     int ptcl, ptype;
07911
07912     ptcl  = edges[ed]->ptcl;
07913     ptype = model->particles[Abs(ptcl)]->ptype;
07914
07915     return (ptype == GRCC_PT_Dirac || ptype == GRCC_PT_Majorana
07916             || ptype == GRCC_PT_Ghost);
07917 }
07918
07919 //------------------------------------------------------------
07920 int  EGraph::fltrace(int fk, int nd0, int *fl)
07921 {
07922     // fk  : kind of fermion
07923     //       = (GRCC_PT_Dirac, GRCC_PT_Majorana or GRCC_PT_Ghost)
07924     // nd0 : the last node visited
07925     // fl  : list of signed edge on the fermion line.
07926     //       fl[j] :
07927     //       (V2Iedge(fl[j]), V2Ileg(fl[j])) is the next node
07928     //       fl[0] should be already defined
07929
07930     int nfl, k, i, nd, nl, e, ed, el, fgcnt, fkind, lk;
07931
07932     nfl = 1;
07933     // maximal possible length of a fline is nEdges
07934     for (k = 0; k < nEdges; k++) {
07935         if (k >= nfl) {
07936             grcc_fprintf(GRCC_Stdout, "*** fltrace:illegal contorl: k=%d, nEdges=%d\n", k, nEdges);
07937             break;
07938         }
07939
07940         // get next node : nd0 ---- nd (nl)
07941         e  = fl[k];
07942         ed = V2Iedge(e);
07943         el = V2Ileg(e);
07944 #ifdef CHECK
07945         if (ed > nEdges) {
07946             grcc_fprintf(GRCC_Stderr, "*** fltrace: ed=%d > nEdges=%d, fl=",
07947                     ed, nEdges);
07948             prIntArray(nNodes, fl, "\n");
07949             erEnd("fltrace: illegal fl");
07950         }
07951 #endif
07952         nd = edges[ed]->nodes[el];
07953         nl = edges[ed]->nlegs[el];
07954
07955         // end of the fline
07956         if (isExternal(nd) || nd == nd0) {
07957             fgcnt = 1;
07958             break;
07959         }
07960
07961         // find leg of nd for going next
07962         fgcnt = 0;
07963         ed = 0;
07964         lk = 0;
07965         for (i = 0; i < nodes[nd]->deg; i++) {
07966             e  = nodes[nd]->edges[i];
07967             ed = V2Iedge(e);
07968             fkind = model->particles[Abs(edges[ed]->ptcl)]->ptype;
07969             // lk =  1 : (# of fkind particle) is even
07970             // lk = -1 : (# of fkind particle) is odd
07971             if (fkind == fk) {
```

```
07972                 if (lk == 1) {
07973                     lk = -1;
07974                 } else {
07975                     lk = 1;
07976                 }
07977             } else {
07978                 lk = 0;
07979             }
07980             if (!edges[ed]->visited) {
07981                 if (!isFermion(ed)) {
07982                     edges[ed]->visited = True;
07983                 } else {
07984                     if (fkind == fk) {
07985                         // i should be the neighbor of nl
07986                         // (nl, i) or (i, nl) should be a pair of fkind
07987                         // (nl, i) : lk = -1
07988                         // (i, nl) : lk =  1
07989                         if ((lk == 1 && nl == i+1) || (lk == -1 && nl == i-1)) {
07990                             edges[ed]->visited = True;
07991                             fgcnt++;
07992                             if (fgcnt == 1) {
07993                                 // -e = (signed edge points the next node)
07994                                 fl[nfl++] = - e;
07995                                 break;
07996                             }
07997                         }
07998                     }
07999                 }
08000             } // not visited
08001         }  // end of for k
08002         if (fgcnt == 0) {
08003             grcc_fprintf(GRCC_Stdout, "*** fline: Fermion number is not conserved\n");
08004             grcc_fprintf(GRCC_Stdout, "    nd=%d, e=%d, ed=%d, fgcnt=%d\n",
08005                     nd, e, ed, fgcnt);
08006             // erEnd("fline: Fermion number is not conserved");
08007         } else if (fgcnt > 1) {
08008             grcc_fprintf(GRCC_Stdout, "+++ fline: more than two fermions: check fsign\n");
08009             grcc_fprintf(GRCC_Stdout, "    nd=%d, e=%d, ed=%d, fgcnt=%d\n",
08010                     nd, e, ed, fgcnt);
08011         }
08012     }
08013     if (k >= nEdges || k >= nfl) {
08014         grcc_fprintf(GRCC_Stdout, "*** fline: illegal control\n");
08015         grcc_fprintf(GRCC_Stdout, "    nEdges=%d, nfl=%d, k=%d, ", nEdges, nfl, k);
08016         prIntArray(nfl, fl, "\n");
08017         erEnd("fline: illegal control");
08018     }
08019     return nfl;
08020 }
08021
08022 //-------------------------------------------------------------
08023 void EGraph::getFLines(void)
08024 {
08025     int fl[GRCC_MAXNODES];
08026     int nextn, exto[GRCC_MAXNODES];
08027     int e, ed, nd, floop, nswap, nfl, fkind, el, ptcl;
08028
08029     nFlines = 0;
08030     for (ed = 0; ed < nEdges; ed++) {
08031         edges[ed]->visited = False;
08032     }
08033
08034     nextn = 0;
08035     for (ed = 0; ed < nEdges; ed++) {
08036         if (edges[ed]->visited) {
08037             continue;
08038         }
08039         if (!edges[ed]->ext) {
08040             continue;
08041         }
08042         el = 0;
08043         nd = edges[ed]->nodes[el];
08044         if (!isExternal(nd)) {
08045             el = 1;
08046             nd = edges[ed]->nodes[el];
08047             if (!isExternal(nd)) {
08048                 erEnd("*** illegal control");
08049             }
08050         }
08051         if (!isFermion(ed)) {
08052             continue;
08053         }
08054         ptcl = legParticle(ed, 1-el);
08055         if (ptcl < 0) {
08056             continue;
08057         }
08058
```

```
08059         //     el   ed   el1
08060         //      x----<----x       e = (signed edge of (ed, el1))
08061         //     nd   ptcl(>= 0)
08062         //  external
08063
08064         fl[0] = - I2Vedge(ed, el);
08065         nfl   = 1;
08066         fkind = model->particles[Abs(edges[ed]->ptcl)]->ptype;
08067         edges[ed]->visited = True;
08068
08069         nfl = fltrace(fkind, nd, fl);
08070         addFLine(FL_Open, fkind, nfl, fl);
08071         e  = fl[nfl-1];
08072         exto[nextn++] = edges[V2Iedge(e)]->nodes[V2Ileg(e)];
08073     }
08074
08075     // closed Fermion line
08076     floop = 0;
08077     for (ed = 0; ed < nEdges; ed++) {
08078         if (edges[ed]->visited) {
08079             continue;
08080         }
08081         if (!isFermion(ed)) {
08082             continue;
08083         }
08084         el = 0;
08085         ptcl = legParticle(ed, 1-el);
08086         if (ptcl < 0) {
08087             el = 1;
08088         }
08089         nd = edges[ed]->nodes[el];
08090
08091         //     el    ed  el1
08092         //      x----<----x       e = (signed edge of (ed, el1))
08093         //     nd   ptcl
08094
08095         fl[0] = - I2Vedge(ed, el);
08096         nfl   = 1;
08097         edges[ed]->visited = True;
08098
08099         fkind = model->particles[Abs(edges[ed]->ptcl)]->ptype;
08100         nfl = fltrace(fkind, nd, fl);
08101         addFLine(FL_Closed, fkind, nfl, fl);
08102         floop++;
08103     }
08104     if (nextn > 0) {
08105         nswap = sortb(nextn, exto);
08106     } else {
08107         nswap = 0;
08108     }
08109     if ((floop+nswap) % 2 == 0) {
08110         fsign = 1;
08111     } else {
08112         fsign = -1;
08113     }
08114 }
08115
08116 //------------------------------------------------------------
08117 void EGraph::addFLine(const FLType ft, int fk, int nfl, int *fl)
08118 {
08119     int j;
08120
08121     if (nFlines >= GRCC_MAXFLINES) {
08122         erEnd("too many Fermion lines (GRCC_MAXEDGES)");
08123     }
08124     if (flines[nFlines] == NULL) {
08125         flines[nFlines] = new EFLine();
08126     }
08127     flines[nFlines]->ftype = ft;
08128     flines[nFlines]->fkind = fk;
08129     flines[nFlines]->nlist = nfl;
08130     for (j = 0; j < nfl; j++) {
08131         flines[nFlines]->elist[j] = fl[j];
08132     }
08133     nFlines++;
08134 }
08135
08136
08137 //************************************************************
08138 // assign.cc
08139 //============================================================
08140 // Assign particles to the edges and interactions to nodes.
08141 // Completed graph data is saved in the form of EGraph.
08142
08143 // method : selection of assignable node
08144
08145 //============================================================
```

```
08146 // class NCand
08147 NCand::NCand(const NCandSt sta, const int dega, const int nilst, int *ilst)
08148 {
08149     // candidate list of interactions attached to a vertex.
08150 
08151     int j;
08152 
08153     deg    = dega;
08154     st     = sta;
08155     nilist = nilst;
08156     for (j = 0; j < nilist; j++) {
08157         ilist[j] = ilst[j];
08158     }
08159 
08160 #ifdef CHECK
08161     if (st == AS_Assigned) {
08162         if (nilist != 1) {
08163             erEnd("illegal assignment");
08164         }
08165     } else if (st == AS_AssExt) {
08166         if (nilist != 1) {
08167             erEnd("illegal assignment");
08168         }
08169     }
08170 #endif
08171 }
08172 
08173 //-------------------------------------------------------------------
08174 NCand::~NCand(void)
08175 {
08176 }
08177 
08178 //-------------------------------------------------------------------
08179 void NCand::prNCand(const char* msg)
08180 {
08181     grcc_fprintf(GRCC_Stdout, "%d %d ",  st, deg);
08182     prIntArray(nilist, ilist, msg);
08183 }
08184 
08185 //===================================================================
08186 // class ECand
08187 ECand::ECand(int dt, int nplst, int *plst)
08188 {
08189     int j;
08190 
08191     det    = dt;
08192     nplist = nplst;
08193     for (j = 0; j < nplist; j++) {
08194         plist[j] = plst[j];
08195     }
08196 
08197     if (det && nplist != 1) {
08198         if (prlevel > 0) {
08199             grcc_fprintf(GRCC_Stderr, "*** ECand : len(plist) != 1 : det=%d ", det);
08200             prIntArrayErr(nplist, plist, "\n");
08201         }
08202         erEnd("ECand : len(plist) != 1");
08203     }
08204 }
08205 
08206 //-------------------------------------------------------------------
08207 ECand::~ECand(void)
08208 {
08209 }
08210 
08211 //-------------------------------------------------------------------
08212 void ECand::prECand(const char *msg)
08213 {
08214     prIntArray(nplist, plist, "");
08215     grcc_fprintf(GRCC_Stdout, " (det=%d)%s", det, msg);
08216 }
08217 
08218 //===================================================================
08219 // class ANode
08220 ANode::ANode(int dg)
08221 {
08222     int j;
08223 
08224     deg    = dg;
08225     nlegs  = 0;
08226     anodes = new int[deg];
08227     aedges = new int[deg];
08228     aelegs = new int[deg];
08229     cand   = NULL;
08230     for (j = 0; j < deg; j++) {
08231         anodes[j] = -1;
08232         aedges[j] = -1;
```

```
08233         aelegs[j] = -1;
08234     }
08235 }
08236
08237 //------------------------------------------------------------
08238 ANode::~ANode(void)
08239 {
08240     if (cand != NULL) {
08241         delete cand;
08242         cand = NULL;
08243     }
08244     if (aelegs != NULL) {
08245         delete[] aelegs;
08246         delete[] aedges;
08247         delete[] anodes;
08248         aelegs = NULL;
08249         aedges = NULL;
08250         anodes = NULL;
08251     }
08252 }
08253
08254 //------------------------------------------------------------
08255 int ANode::newleg(void)
08256 {
08257     // add a slot for a leg
08258
08259     int lg = nlegs;
08260
08261     nlegs++;
08262 #ifdef CHECK
08263     if (nlegs > deg) {
08264         grcc_fprintf(GRCC_Stderr, "*** ANode::newleg : nlegs = %d > deg = %d\n",
08265                 nlegs, deg);
08266         erEnd("ANode::newleg : nlegs > deg");
08267     }
08268 #endif
08269
08270     return lg;
08271 }
08272
08273 //============================================================
08274 // class AEdge
08275 AEdge::AEdge(int n0, int l0, int n1, int l1)
08276 {
08277     nodes[0]  = n0;       // node of 0 side of the edge
08278     nodes[1]  = n1;       // node of 1 side of the edge
08279     nlegs[0]  = l0;       // leg ot the node of 0 side of the edge
08280     nlegs[1]  = l1;       // leg ot the node of 1 side of the edge
08281     ptcl   = GRCC_PT_Undef;    // particle defined in the model
08282     cand   = NULL;        // candidate
08283 }
08284
08285 //------------------------------------------------------------
08286 AEdge::~AEdge(void)
08287 {
08288     if (cand != NULL) {
08289         delete cand;
08290     }
08291 }
08292
08293 //============================================================
08294 // class Assign
08295 Assign::Assign(SProcess *sprc, MGraph *mgr, PNodeClass *pnc)
08296 {
08297     Bool ok;
08298     int  j;
08299
08300     if (sprc == NULL) {
08301         erEnd("Assign: sproc == NULL");
08302     }
08303
08304     // pointers to related objects
08305     sproc        = sprc;
08306     model        = sproc->model;
08307     opt          = sproc->opt;
08308     mgraph       = mgr;
08309     pnclass      = pnc;
08310
08311     egraph = mgraph->egraph;
08312     if (egraph == NULL) {
08313         erEnd("Assign: egraph == NULL");
08314     }
08315
08316     astack       = sproc->astack;
08317     if (astack == NULL) {
08318         erEnd("Assign: astack == NULL");
08319     }
```

```
08320
08321     nNodes       = mgraph->nNodes;
08322     nEdges       = mgraph->nEdges;
08323     nExtern      = sproc->nExtern;
08324     nETotal      = 0;               // total number of edges
08325
08326     // counters of graphs
08327     nAGraphs     = 0;
08328     wAGraphs     = Fraction(0,1);
08329     nAOPI        = 0;
08330     wAOPI        = Fraction(0,1);
08331
08332     nodes        = new ANode*[nNodes];
08333     edges        = new AEdge*[nEdges];
08334
08335     for (j = 0; j < nNodes; j++) {
08336         nodes[j] = NULL;
08337     }
08338     for (j = 0; j < nEdges; j++) {
08339         edges[j] = NULL;
08340     }
08341     for (j = 0; j < model->ncouple; j++) {
08342         cplleft[j] = sproc->clist[j];
08343     }
08344
08345     // initialize astack
08346     astack->setAGraph(this);
08347     astack->checkPoint(checkpoint0);
08348
08349     // import from mgraph
08350     ok = fromMGraph();
08351 #ifdef CHECK
08352     checkAG("Assign::Assign");
08353 #endif
08354
08355     if (ok) {
08356         // start assignment
08357         assignAllVertices();
08358     } else {
08359         // cannot assign
08360     }
08361
08362     // restore from the stack
08363 #ifdef CHECK
08364     astack->restoreMsg(checkpoint0, "Assign");
08365 #else
08366     astack->restore(checkpoint0);
08367 #endif
08368 }
08369
08370 //============================================================
08371 Assign::~Assign(void)
08372 {
08373     int j;
08374
08375     for (j = 0; j < nEdges; j++) {
08376         delete edges[j];
08377         edges[j] = NULL;
08378     }
08379     delete[] edges;
08380     edges = NULL;
08381
08382     for (j = 0; j < nNodes; j++) {
08383         delete nodes[j];
08384         nodes[j] = NULL;
08385     }
08386     delete[] nodes;
08387     nodes = NULL;
08388 }
08389
08390 //------------------------------------------------------------
08391 void Assign::prCand(const char *msg)
08392 {
08393     //  Print candidate table
08394     int    n, e, ne;
08395     AEdge *ed;
08396
08397     grcc_fprintf(GRCC_Stdout, "\n");
08398     grcc_fprintf(GRCC_Stdout, "+++ Candidate list: %s\n", msg);
08399     grcc_fprintf(GRCC_Stdout, "  Nodes %d\n", nNodes);
08400     for (n = 0; n < nNodes; n++) {
08401        grcc_fprintf(GRCC_Stdout, "%d: edges=", n);
08402        prIntArray(nodes[n]->deg, nodes[n]->aedges, ": cand=");
08403        if (nodes[n]->cand == NULL) {
08404            grcc_fprintf(GRCC_Stdout, "NULL\n");
08405        } else {
08406            nodes[n]->cand->prNCand("\n");
```

```
08407         }
08408     }
08409     ne = Min(nEdges, nETotal);
08410     grcc_fprintf(GRCC_Stdout, "  Edges %d\n", ne);
08411     for (e = 0; e < ne; e++) {
08412        ed = edges[e];
08413        if (ed == NULL) {
08414            grcc_fprintf(GRCC_Stdout, "NULL_Edge\n");
08415        } else {
08416            grcc_fprintf(GRCC_Stdout, "%d: %d->%d: cand=", e, ed->nodes[0], ed->nodes[1]);
08417            if (edges[e]->cand == NULL) {
08418                grcc_fprintf(GRCC_Stdout, "NULL\n");
08419            } else {
08420                edges[e]->cand->prECand("\n");
08421            }
08422        }
08423     }
08424 }
08425
08426 //=============================================================
08427 void Assign::checkAG(const char *msg)
08428 {
08429     int n, lg;
08430     Bool ok = True;
08431
08432     for (n = 0; n < nNodes; n++) {
08433         for (lg = 0; lg < nodes[n]->deg; lg++) {
08434             if (nodes[n]->aedges[lg] < 0) {
08435                 grcc_fprintf(GRCC_Stdout, "*** checkAG:%s: n=%d, lg=%d, aedges=%d\n",
08436                         msg, n, lg, nodes[n]->aedges[lg]);
08437                 ok = False;
08438             }
08439         }
08440     }
08441     if (!ok) {
08442         prCand("checkAG");
08443         erEnd("checkAG: failed");
08444     }
08445 }
08446
08447 //=============================================================
08448 // control of the process of particle/interaction assignment
08449
08450 //-------------------------------------------------------------
08451 Bool Assign::assignAllVertices(void)
08452 {
08453     //  Entry point of the assignment procedure
08454     Bool   ok;
08455
08456 #ifdef CHECK
08457     checkCand("assignAllVertices:1");
08458 #endif
08459
08460     // start main part
08461 #ifdef SIMPSEL
08462     ok = selectVertexSimp(-1);
08463 #else
08464     ok = selectVertex();
08465 #endif
08466
08467 #ifdef CHECK
08468     checkCand("assignAllVertices:2");
08469 #endif
08470
08471     return ok;
08472 }
08473
08474 //-------------------------------------------------------------
08475 Bool Assign::selectVertexSimp(int lastv)
08476 {
08477     //  Select a vertex for assignment of particles to legs
08478     //
08479     //  Argument
08480     //    lastv : previously assigned vertex; Nothing when lastv<0.
08481
08482     Bool ok;
08483     int v;
08484
08485 #ifdef CHECK
08486         checkCand("selectVertexSimp");
08487 #endif
08488
08489     // find a vertex to which interaction is assigned
08490     v = selUnAssVertexSimp(lastv);
08491
08492     // no more vertex
08493     if (v < 0) {
```

```
08494
08495         ok = allAssigned();
08496         if (ok) {
08497             if (!egraph->optQGrafA(opt)) {
08498                 return False;
08499             }
08500          // egraph->biconnE();      // necessary ???
08501             opt->newAGraph(egraph);
08502         }
08503
08504         return ok;
08505     }
08506
08507     // select a leg and assign a particle to it
08508     ok = selectLeg(v, -1);
08509
08510     return ok;
08511 }
08512
08513 //-------------------------------------------------------------
08514 Bool Assign::selectVertex(void)
08515 {
08516     //  Select a vertex for assignment of particles to legs
08517     //
08518     //  Argument
08519     //    lastv : previously assigned vertex; Nothing when lastv<0.
08520
08521     Bool ok;
08522     int v;
08523
08524 #ifdef CHECK
08525         checkCand("selectVertex");
08526 #endif
08527
08528     // find a vertex to which interaction is assigned
08529     v = selUnAssVertex();
08530
08531     // no more vertex
08532     if (v < 0) {
08533
08534         ok = allAssigned();
08535         if (ok) {
08536             if (!egraph->optQGrafA(opt)) {
08537                 return False;
08538             }
08539             egraph->biconnE();
08540             opt->newAGraph(egraph);
08541         }
08542
08543         return ok;
08544     }
08545
08546     // select a leg and assign a particle to it
08547     ok = selectLeg(v, -1);
08548
08549     return ok;
08550 }
08551
08552 //-------------------------------------------------------------
08553 Bool Assign::selectLeg(int v, int lastlg)
08554 {
08555     //  Select a leg of vertex 'v' for assignment of particle,
08556     //  where the last assigned leg was 'lastlg'.
08557
08558     int      pt, ln, j;
08559     Bool     ok;
08560     CheckPt  sav, sav0;
08561     int      nplist, plist[GRCC_MAXMPARTICLES2];
08562
08563 #ifdef CHECK
08564     checkNode(v, "selectLeg:0");
08565     checkCand("selectLeg");
08566 #endif
08567
08568     // find a leg to be assigned
08569     ln = selUnAssLeg(v, lastlg);
08570
08571     // no more assignable leg for the current node
08572     if (ln < 0) {
08573         // move to next vertex
08574         ok = assignVertex(v);
08575
08576         return ok;
08577     }
08578
08579     // make the list of possible particles, incoming to the vertex
08580     astack->checkPoint(sav0);
```

```
08581     nplist = candPart(v, ln, plist, GRCC_MAXMPARTICLES2);
08582
08583     // no candidate
08584     if (nplist < 1) {
08585         return False;
08586     }
08587
08588     // assign each of possible particle to the leg 'lg'.
08589     astack->checkPoint(sav);
08590     for (j = 0; j < nplist; j++) {
08591         pt = plist[j];
08592
08593         // try to assign 'pt' to (v, ln)
08594         if(assignPLeg(v, ln, pt)) {
08595             // move to next leg
08596             selectLeg(v, ln);
08597         }
08598
08599 #ifdef CHECK
08600         astack->restoreMsg(sav, "selectLeg2");
08601 #else
08602         astack->restore(sav);
08603 #endif
08604     }
08605
08606 #ifdef CHECK
08607     astack->restoreMsg(sav0, "selectLeg2");
08608 #else
08609     astack->restore(sav0);
08610 #endif
08611     return True;
08612 }
08613
08614 //-------------------------------------------------------------
08615 void Assign::saveCouple(int *sav)
08616 {
08617     int j;
08618
08619     if (model->ncouple > 1) {
08620         for (j = 0; j < model->ncouple; j++) {
08621             sav[j] = cplleft[j];
08622         }
08623     }
08624 }
08625
08626 //-------------------------------------------------------------
08627 void Assign::restoreCouple(int *sav)
08628 {
08629     int j;
08630
08631     if (model->ncouple > 1) {
08632         for (j = 0; j < model->ncouple; j++) {
08633             cplleft[j] = sav[j];
08634         }
08635     }
08636 }
08637
08638 //-------------------------------------------------------------
08639 Bool Assign::subCouple(int *cpl)
08640 {
08641     int j;
08642
08643     if (model->ncouple > 1) {
08644         for (j = 0; j < model->ncouple; j++) {
08645             cplleft[j] -= cpl[j];
08646             if (cplleft[j] < 0) {
08647                 return False;
08648             }
08649         }
08650     }
08651     return True;
08652 }
08653
08654 //-------------------------------------------------------------
08655 Bool Assign::assignVertex(int v)
08656 {
08657     //  Assign interactions to vertex 'v'
08658
08659     Bool      done, ok;
08660     CheckPt   sav, sav1;
08661     NCand    *nc;
08662     int       ia, n, j, cl;
08663     NCandSt   vst;
08664     Bool      ok1;
08665     int       savc0[GRCC_MAXNCPLG], savc1[GRCC_MAXNCPLG];
08666
08667 #ifdef CHECK
```

```
08668     checkCand("assignVertex");
08669 #endif
08670
08671     // save the configuration
08672     astack->checkPoint(sav);
08673     saveCouple(savc0);
08674
08675     // assign to all vertices with only one candidate
08676     done = False;
08677     for (n = 0; n < nNodes; n++) {
08678
08679         // check if vertex
08680         cl = pnclass->nd2cl[n];
08681         if (!isATExternal(pnclass->type[cl])) {
08682
08683             nc = nodes[n]->cand;
08684             if (nc->st == AS_UnAssLegs && nc->nilist == 1) {
08685
08686                 // bind interaction to the vertex
08687                 ia = nc->ilist[0];
08688                 vst = assignIVertex(n, ia);
08689                 if (vst == AS_Impossible) {
08690                     // discard the current configuration
08691 #ifdef CHECK
08692                     astack->restoreMsg(sav, "assignVertex");
08693 #else
08694                     astack->restore(sav);
08695 #endif
08696                     restoreCouple(savc0);
08697                     return False;
08698
08699                 } else if (vst == AS_Assigned) {
08700                     if (!subCouple(model->interacts[ia]->clist)) {
08701 #ifdef CHECK
08702                         astack->restoreMsg(sav, "assignVertex");
08703 #else
08704                         astack->restore(sav);
08705 #endif
08706                         restoreCouple(savc0);
08707                         return False;
08708                     }
08709                 }
08710                 if (n == v) {
08711                     done = (vst == AS_Assigned);
08712                 }
08713             }
08714         }
08715     }
08716
08717     // uniquely determined
08718     ok = True;
08719     if (done) {
08720         // move to the next vertex
08721 #ifdef SIMPSEL
08722         ok = selectVertexSimp(v);
08723 #else
08724         ok = selectVertex();
08725 #endif
08726
08727 #ifdef CHECK
08728         astack->restoreMsg(sav, "assignVertex");
08729 #else
08730         astack->restore(sav);
08731 #endif
08732         restoreCouple(savc0);
08733
08734         return ok;
08735     }
08736
08737     // there are still several possibilities.
08738
08739     astack->checkPoint(sav1);
08740     saveCouple(savc1);
08741     for (j = 0; j < nodes[v]->cand->nilist; j++) {
08742         ia = nodes[v]->cand->ilist[j];
08743
08744         // bind interaction to the vertex
08745         ok1 = True;
08746         vst = assignIVertex(v, ia);
08747         if (vst == AS_Assigned) {
08748             ok1 = subCouple(model->interacts[ia]->clist);
08749         }
08750         if (ok1 && (vst == AS_Assigned || vst == AS_Assigned0)) {
08751             // move to the next vertex
08752 #ifdef SIMPSEL
08753             ok = selectVertexSimp(v);
08754 #else
```

```
08755             ok = selectVertex();
08756 #endif
08757         }
08758
08759 #ifdef CHECK
08760         astack->restoreMsg(sav1, "assignVertex");
08761 #else
08762         astack->restore(sav1);
08763 #endif
08764         restoreCouple(savc1);
08765     }
08766
08767 #ifdef CHECK
08768     astack->restoreMsg(sav, "assignVertex");
08769 #else
08770     astack->restore(sav);
08771 #endif
08772     restoreCouple(savc0);
08773
08774     return ok;
08775 }
08776
08777 //------------------------------------------------------------
08778 Bool Assign::allAssigned(void)
08779 {
08780     // Assignment of particles and interactions is finished.
08781     // Check isomorphism etc. and count symmetry factor
08782     // The sign from Fermi statistics is calculated inf fillEGraph
08783
08784     BigInt  nsym, esym, nsym1;
08785     MNodeClass *cl;
08786     Bool ok = True;
08787 #ifdef OPTEXTONLY
08788 #else
08789     Bool ext;
08790     int  e, p;
08791 #endif
08792
08793 #ifdef CHECK
08794     checkCand("allAssigned");
08795 #endif
08796
08797     // check order of coupling constants
08798 #ifdef CHECK
08799     if (!checkOrderCpl()) {
08800         erEnd("allAssigned: checkOrderCpl = False");
08801     }
08802 #endif
08803
08804     // check duplication by violating ordering condition
08805     if (!isOrdLegs()) {
08806         return False;
08807     }
08808
08809     // classification of nodes
08810     cl = mgraph->curcl;
08811
08812     // check isomorphism and calculate sfactor
08813     nsym = 0;
08814     esym = 0;
08815
08816     ok = isIsomorphic(cl, &nsym, &esym, &nsym1);
08817     if (!ok || nsym < 1 || esym < 1) {
08818         return False;
08819     }
08820
08821     //--------------------------
08822     // Now we got a new agraph.
08823
08824     // Update counters
08825     nAGraphs++;
08826     wAGraphs.add(1,nsym*esym);
08827     if (mgraph->opi) {
08828         nAOPI++;
08829         wAOPI.add(1,nsym*esym);
08830     }
08831
08832 #ifdef CHECK
08833     checkAG("allAssigned");
08834 #endif
08835     // fill information to resulting Egraph
08836     fillEGraph(nAGraphs, nsym, esym, nsym1);
08837
08838 #ifdef OPTEXTONLY
08839 #else
08840     // check extonly particles
08841     for (e = 0; e < nEdges; e++) {
```

```
08842         p   = Abs(egraph->edges[e]->ptcl);
08843         ext = egraph->edges[e]->ext;
08844         if (!ext) {
08845             if (model->particles[p]->extonly) {
08846                 return False;
08847             }
08848         }
08849     }
08850 #endif
08851
08852     return True;
08853 }
08854
08855 //============================================================
08856 // Interface to MGraph and EGraph
08857 //------------------------------------------------------------
08858 Bool Assign::fromMGraph(void)
08859 {
08860     // Import from MGraph
08861
08862     int     npall, *pall;
08863     int     np[1];
08864     Bool    ok;
08865     int     j, k, n, nn, n1, deg, nc, ptcl, cl, typ, mcl, cl1, typ1;
08866     int     lcn, ia;
08867     int     nilist, ilist[GRCC_MAXMINTERACT];
08868     NCandSt st;
08869
08870     // create ANodes
08871     for (j = 0; j < nNodes; j++) {
08872         nodes[j] = new ANode(mgraph->nodes[j]->deg);
08873     }
08874
08875     // initialization of edge table
08876     nETotal = 0;
08877
08878     // List of all particles
08879     pall = model->allParticles(&npall);
08880
08881     // add nodes
08882     for (n = 0; n < mgraph->nNodes; n++) {
08883         cl   = pnclass->nd2cl[n];
08884         typ  = pnclass->type[cl];
08885
08886         // external particle
08887         if (isATExternal(typ)) {
08888
08889 #ifdef CHECK
08890             if (mgraph->nodes[n]->deg != 1) {
08891                 grcc_fprintf(GRCC_Stdout, "*** assign:fromMGraph : "
08892                        "external but deg[%d] = %d != 1, type=%d\n",
08893                        n, mgraph->nodes[n]->deg, typ);
08894                 mgraph->print();
08895                 erEnd("assign:fromMGraph : illegal external particle");
08896             }
08897 #endif
08898             np[0] = pnclass->particle[cl];
08899             nodes[n]->cand = new NCand(AS_AssExt, 1, 1, np);
08900             for (n1 = 0; n1 < mgraph->nNodes; n1++) {
08901                 nc = mgraph->adjMat[n][n1];
08902                 for (k = 0; k < nc; k++) {
08903                     addEdge(n, n1, 1, np);
08904                 }
08905             }
08906
08907         // vertex
08908         } else {
08909
08910             // candidates of vertices
08911             deg    = mgraph->nodes[n]->deg;
08912             st     = AS_UnAssLegs;
08913             cl     = sproc->pnclass->nd2cl[n];     // class in the process
08914             mcl    = sproc->pnclass->cl2mcl[cl];   // class in the model
08915
08916             if (nodes[n]->cand != NULL) {
08917                 delete nodes[n]->cand;
08918             }
08919             lcn = model->ncouple;
08920
08921             // multiple coupling constants are defined in the model.
08922             if (lcn > 1) {
08923                 nilist = 0;
08924                 for (j = 0; j < model->cplgnvl[mcl]; j++) {
08925                     ia = model->cplgvl[mcl][j];
08926
08927                     // select by coupling constants
08928                     if (leqArray(lcn, model->interacts[ia]->clist, cplleft)) {
```

```
08929                         ilist[nilist++] = ia;
08930                     }
08931                 }
08932                 nodes[n]->cand = new NCand(st, deg, nilist, ilist);
08933
08934             // there is only one coupling constant
08935             } else {
08936                 nodes[n]->cand =
08937                     new NCand(st, deg, model->cplgnvl[mcl], model->cplgvl[mcl]);
08938             }
08939
08940             // vertices
08941             for (n1 = n; n1 < mgraph->nNodes; n1++) {
08942                 cl1   = pnclass->nd2cl[n1];
08943                 typ1  = pnclass->type[cl1];
08944                 if (!isATExternal(typ1)) {
08945                     nc = mgraph->adjMat[n][n1];
08946                     if (n == n1) {
08947                         nc = nc/2;
08948                     }
08949                     for (k = 0; k < nc; k++) {
08950                         addEdge(n, n1, npall, pall);
08951                     }
08952                 }
08953             }
08954         }
08955     }
08956 #ifdef CHECK
08957     if (nETotal != nEdges) {
08958         grcc_fprintf(GRCC_Stdout, "*** Assign::fromMGraph nETotal=%d != nEdges=%d\n",
08959                nETotal, nEdges);
08960         erEnd("Assign::fromMGraph nETotal= != nEdges");
08961     }
08962 #endif
08963
08964     // assign particle of an edge next to an external particle
08965     for (n = 0; n < mgraph->nNodes; n++) {
08966         cl  = pnclass->nd2cl[n];
08967         typ = pnclass->type[cl];
08968         if (isATExternal(typ)) {
08969
08970             cl   = pnclass->nd2cl[n];
08971             ptcl = pnclass->particle[cl];
08972
08973             // particle ptcl     flows in to   the node and
08974             // particle (- ptcl) flows in from the edge.
08975
08976 #ifdef CHECK
08977             if (ptcl == 0) {
08978                 erEnd("fromMGraph: ptcl=0");
08979             }
08980 #endif
08981             ok = assignPLeg(n, 0, -ptcl);
08982             if (!ok) {
08983                 // impossible config
08984                 return False;
08985             }
08986             nn = nodes[n]->anodes[0];
08987             ok = updateCandNode(nn);
08988             if (!ok) {
08989                 // impossible config
08990                 return False;
08991             }
08992         }
08993     }
08994
08995 #ifdef CHECK
08996     checkCand("fromMGraph");
08997     checkAG("fromMGraph");
08998 #endif
08999
09000     return True;
09001 }
09002
09003 //-------------------------------------------------------------------
09004 void Assign::addEdge(int n0, int n1, int nplist, int *plist)
09005 {
09006     //  add an edge and set connection information
09007     //  n0 -- (new edge) -- n1, with candidates 'plist'
09008
09009     int lg0, lg1, eid;
09010     AEdge *aed;
09011
09012 #ifdef CHECK
09013     if (n0 >= nNodes || n1 >= nNodes) {
09014         grcc_fprintf(GRCC_Stdout, "*** Assign::addEdge : undefined nodes %d: [%d, %d]",
09015               nETotal, n0, n1);
```

```
09016         erEnd("Assign::addEdge : undefined nodes");
09017     }
09018 #endif
09019
09020     // legs to be connected
09021     lg0 = nodes[n0]->newleg();
09022     lg1 = nodes[n1]->newleg();
09023
09024     // create edge
09025 #ifdef CHECK
09026     if (nETotal >= nEdges) {
09027         erEnd("too many edges");
09028     }
09029 #endif
09030
09031     aed = new AEdge(n0, lg0, n1, lg1);
09032     aed->cand = new ECand(False, nplist, plist);
09033
09034 #ifdef CHECK
09035     if (edges[nETotal] != NULL) {
09036         erEnd("edges[nETotal] != NULL");
09037     }
09038 #endif
09039
09040     // registor to the edge table
09041     eid = nETotal;
09042     edges[nETotal++] = aed;
09043
09044     // connect edge and nodes
09045     connect(n0, lg0, eid, 0, n1, lg1);
09046 }
09047
09048 //-------------------------------------------------------------------
09049 void Assign::connect(int n0, int l0, int eg, int el, int n1, int l1)
09050 {
09051     //  Connect (n0, l0) -- (eg, el) -- (n1)
09052
09053     ANode *nd0, *nd1;
09054     AEdge *ed;
09055     int    eo;
09056
09057     nd0 = nodes[n0];
09058     nd1 = nodes[n1];
09059     ed  = edges[eg];
09060     eo  = 1 - el;
09061
09062     nd0->anodes[l0] = n1;
09063     nd0->aedges[l0] = eg;
09064     nd0->aelegs[l0] = el;
09065
09066     nd1->anodes[l1] = n0;
09067     nd1->aedges[l1] = eg;
09068     nd1->aelegs[l1] = eo;
09069
09070     ed->nodes[el] = n0;
09071     ed->nlegs[el] = l0;
09072
09073     ed->nodes[eo] = n1;
09074     ed->nlegs[eo] = l1;
09075 }
09076
09077 //-------------------------------------------------------------------
09078 Bool Assign::fillEGraph(int aid, BigInt nsym, BigInt esym, BigInt nsym1)
09079 {
09080     //  Set variables of egraph with the result of particle assignment
09081     //  Adjust the ordering of legs of vertices in a consistent way
09082     //  with the interaction defined in the model.
09083
09084     ANode *an;
09085     ENode *en;
09086     int    work[3][GRCC_MAXLEGS];
09087     int    n, lr, e;
09088     int   *elist;
09089     int    lg, ed, eg;
09090 #ifdef CHECK
09091     int    cl, ok = True;
09092 #endif
09093
09094     egraph->assigned = True;
09095
09096     egraph->aId   = aid;
09097     egraph->nsym  = nsym;
09098     egraph->esym  = esym;
09099     egraph->bicount = -1;
09100     if (sproc != NULL) {
09101         egraph->extperm = sproc->extperm;
09102     } else {
```

```
09103         egraph->extperm = 1;
09104     }
09105     egraph->nsym1 = nsym1;
09106     egraph->multp = (egraph->extperm * egraph->nsym1) / egraph->nsym;
09107
09108 #ifdef CHECK
09109     checkAG("fillEGraph:0");
09110 #endif
09111     // external node
09112     for (n = 0; n < nNodes; n++) {
09113         // vertices
09114 #ifdef CHECK
09115         cl  = pnclass->nd2cl[n];
09116         if (isATExternal(pnclass->type[cl])) {
09117             ;
09118         } else if (nodes[n]->cand->st != AS_Assigned) {
09119             grcc_fprintf(GRCC_Stdout, "*** fillEGraph : node %d is not assigned", n);
09120             prCand("fillEGraph: node");
09121             erEnd("fillEGraph : node is not assigned");
09122         }
09123 #endif
09124
09125         an = nodes[n];
09126         en = egraph->nodes[n];
09127
09128         en->initAss(egraph, n, an->deg);
09129
09130         en->intrct = an->cand->ilist[0];
09131         elist = reordLeg(n, work[0], work[1], work[2]);
09132         for (lr = 0; lr < nodes[n]->deg; lr++) {
09133             if (elist == NULL) {
09134                 lg = lr;
09135             } else {
09136                 lg = elist[lr];
09137             }
09138 #ifdef CHECK
09139             if (lg < 0 || lg >= nodes[n]->deg) {
09140                 grcc_fprintf(GRCC_Stdout, "*** fillEGraph: n=%d, lr=%d: 0 <= lg=%d < %d\n",
09141                         n, lr, lg, nodes[n]->deg);
09142                 erEnd("fillEGraph: illegal reordering");
09143             }
09144 #endif
09145             ed = an->aedges[lg];
09146             eg = an->aelegs[lg];
09147
09148             en->edges[lr] = I2Vedge(ed, 2*eg-1);
09149
09150             egraph->edges[ed]->nodes[eg] = n;
09151             egraph->edges[ed]->nlegs[eg] = lr;
09152         }
09153     }
09154     // edges
09155     for (e = 0; e < nEdges; e++) {
09156 #ifdef CHECK
09157         if (edges[e]->cand->nplist != 1) {
09158             grcc_fprintf(GRCC_Stdout, "*** fillEGraph : edge %d is not assigned", e);
09159             prCand("fillEGraph: edge");
09160             erEnd("fillEGraph : edge is not assigned");
09161         }
09162 #endif
09163         egraph->edges[e]->ptcl = edges[e]->cand->plist[0];
09164     }
09165 #ifdef EDGEPORDER
09166     for (e = 0; e < nEdges; e++) {
09167         if (egraph->edges[e]->ptcl < 0) {
09168             egraph->edges[e]->ptcl = - edges[e]->cand->plist[0];
09169             n  = egraph->edges[e]->nodes[0];
09170             lr = egraph->edges[e]->nlegs[0];
09171             egraph->edges[e]->nodes[0] = egraph->edges[e]->nodes[1];
09172             egraph->edges[e]->nlegs[0] = egraph->edges[e]->nlegs[1];
09173             egraph->edges[e]->nodes[1] = n;
09174             egraph->edges[e]->nlegs[1] = lr;
09175
09176             egraph->nodes[n]->edges[lr] = - egraph->nodes[n]->edges[lr];
09177             n  = egraph->edges[e]->nodes[0];
09178             lr = egraph->edges[e]->nlegs[0];
09179             egraph->nodes[n]->edges[lr] = - egraph->nodes[n]->edges[lr];
09180         }
09181     }
09182 #endif
09183 #ifdef CHECK
09184     for (ed = 0; ed < nEdges; ed++) {
09185         n  = egraph->edges[ed]->nodes[0];
09186         lr = egraph->edges[ed]->nlegs[0];
09187         if (egraph->nodes[n]->edges[lr] != -ed-1) {
09188             grcc_fprintf(GRCC_Stderr, "+++ node[%d][%d]=%d != - (edge[%d][0] + 1) = %d\n",
09189                     n, lr, egraph->nodes[n]->edges[lr], e, -ed-1);
```

```
09190             ok = False;
09191         }
09192         n = egraph->edges[ed]->nodes[1];
09193         lr = egraph->edges[ed]->nlegs[1];
09194         if (egraph->nodes[n]->edges[lr] != ed+1) {
09195             grcc_fprintf(GRCC_Stderr, "+++ node[%d][%d]=%d != + (edge[%d][0] + 1) = %d\n",
09196                     n, lr, egraph->nodes[n]->edges[lr], e, ed+1);
09197             ok = False;
09198         }
09199     }
09200     if (!ok) {
09201         egraph->print();
09202         erEnd("fillEGraph: illegal connection");
09203     }
09204 #endif
09205 
09206 
09207     // analyse fermion line and determine Fermi statistical sign factor
09208     egraph->getFLines();
09209 
09210 #ifdef CHECK
09211     checkAG("fillEGraph:0");
09212 #endif
09213     return True;
09214 
09215 }
09216 
09217 //-------------------------------------------------------------
09218 int *Assign::reordLeg(int n, int *reord, int *plist, int *used)
09219 {
09220     // Reorder legs of node 'n' in accordance with the definition
09221     // of the interaction in the model
09222     //
09223     // plist[record[j]] = model->interacts[ia]->plist[j]
09224 
09225     int lg, ia, lr, deg, pt, ed;
09226     int *ilegs;
09227 #ifdef CHECK
09228     Bool found;
09229 #endif
09230 
09231     // external node
09232     if (nodes[n]->cand->st == AS_AssExt) {
09233         reord[0] = 0;
09234         return 0;
09235     }
09236 
09237     // degree of the node
09238     deg = nodes[n]->deg;
09239 
09240     if (deg <= 1) {
09241         reord[0] = 0;
09242         return 0;
09243     }
09244 
09245     // list of particles at the legs of the node 'n'
09246     for (lg = 0; lg < deg; lg++) {
09247         ed = nodes[n]->aedges[lg];
09248         pt = edges[ed]->cand->plist[0];
09249         plist[lg] = legEdgeParticle(n, lg, pt);
09250         used[lg]  = 0;
09251     }
09252 
09253     // list of legs in the interaction
09254     ia    = nodes[n]->cand->ilist[0];
09255     ilegs = model->interacts[ia]->plist;
09256 
09257     // reorder
09258 #ifdef CHECK
09259     found = False;
09260 #endif
09261     for (lr = 0; lr < deg; lr++) {
09262         for (lg = 0; lg < deg; lg++) {
09263             if (used[lg] == 0 && plist[lg] == ilegs[lr]) {
09264                 reord[lr] = lg;
09265                 used[lg]  = 1;
09266 #ifdef CHECK
09267                 found = True;
09268 #endif
09269                 break;
09270             }
09271         }
09272 
09273 #ifdef CHECK
09274         if (!found) {
09275             grcc_fprintf(GRCC_Stdout, "*** reordLeg: illegal list of particles:"
09276                     "interaction %d ", ia);
```

```
09277             prIntArray(deg, ilegs, "; ");
09278             grcc_fprintf(GRCC_Stdout, "vertex %d ", n);
09279             prIntArray(deg, plist, "\n");
09280             prCand("reordLeg");
09281             erEnd("reordLeg: illegal list of particles");
09282         }
09283 #endif
09284     }
09285
09286     return reord;
09287 }
09288
09289 //============================================================
09290 // Adjust the direction of a particle on an edge and on a leg of
09291 // node.
09292 //------------------------------------------------------------
09293 int Assign::getLegParticle(int n,  int ln)
09294 {
09295     //  Get particle code of (n, ln) when particle 'pt' runs
09296     //  in the direction of the edge at (n, ln).
09297     //
09298     //  Get particle code in the direction the edge at (n, ln)
09299     //  when particle 'pt' is at leg (n, ln).
09300     //
09301     //  These two cases are realized by the same function
09302
09303     ANode *nd;
09304     int    elg, ed, pt;
09305
09306     nd  = nodes[n];
09307     elg = nd->aelegs[ln];
09308     ed  = nd->aedges[ln];
09309     pt  = edges[ed]->cand->plist[0];
09310
09311     // normalization of sign for neutral particle
09312     if (elg == 0) {
09313         return model->normalParticle(-pt);
09314     } else {
09315         return model->normalParticle(pt);
09316     }
09317 }
09318
09319 //------------------------------------------------------------
09320 int Assign::legEdgeParticle(int n, int ln, int pt)
09321 {
09322     //  Get particle code of (n, ln) when particle 'pt' runs
09323     //  in the direction of the edge at (n, ln).
09324     //
09325     //  Get particle code in the direction the edge at (n, nl)
09326     //  when particle 'pt' is at leg (n, ln).
09327     //
09328     //  These two cases are realized by the same function
09329
09330     // normalization of sign for neutral particle
09331
09332     if (nodes[n]->aelegs[ln] == 0) {
09333         return model->normalParticle(-pt);
09334     } else {
09335         return model->normalParticle(pt);
09336     }
09337 }
09338
09339 //------------------------------------------------------------
09340 int Assign::legPart(int v, int lg, int nplist, int *plist, int *rlist, const int size)
09341 {
09342     // Convert list of candidate particles in 'plist' at (v, lg) ==> edge
09343     // or edge ==> (v, lg)
09344
09345     int    j, nrlist;
09346
09347     nrlist = 0;
09348     for (j = 0; j < nplist; j++) {
09349         nrlist = intSetAdd(nrlist, rlist,
09350                            legEdgeParticle(v, lg, plist[j]), size);
09351     }
09352     return nrlist;
09353 }
09354
09355 //------------------------------------------------------------
09356 int Assign::candPart(int v, int ln, int *plist, const int size)
09357 {
09358     //  update candidate particles incoming to (v, ln)
09359
09360     int      en;
09361     int      ntplist;
09362     ECand   *ec;
09363
```

```
09364     en = nodes[v]->aedges[ln];
09365
09366 #ifdef CHECK
09367     if (edges[en]->cand->det) {
09368         grcc_fprintf(GRCC_Stdout, "*** candPart : particle of leg (%d, %d) "
09369                "is assigned to %d\n",
09370                v, ln, edges[en]->cand->plist[0]);
09371         checkCand("candPart");
09372         mgraph->printAdjMat(mgraph->curcl);
09373         prCand("candPart");
09374         erEnd("candPart : particle of leg is assigned");
09375     }
09376 #endif
09377
09378     ec = edges[en]->cand;
09379
09380     ntplist = legPart(v, ln, ec->nplist, ec->plist, plist, size);
09381
09382     return  ntplist;
09383 }
09384
09385 //============================================================
09386 // tools for assignment
09387
09388 //-------------------------------------------------------------------
09389 int Assign::selUnAssVertexSimp(int lastv)
09390 {
09391     //  Select a vertex for assignment to its leg.
09392     //  We take one with minimum number of candidates
09393     //
09394     //  There may be better method.
09395
09396     int v0, v;
09397
09398     // select vertex one by one in the sequential order of node number
09399     v0 = Max(lastv+1, nExtern);
09400     for (v = v0; v < nNodes; v++) {
09401         if (nodes[v]->cand->st == AS_UnAssLegs) {
09402             return v;
09403         }
09404     }
09405     return -1;
09406 }
09407
09408 //-------------------------------------------------------------------
09409 int Assign::selUnAssVertex(void)
09410 {
09411     //  Select a vertex for assignment to its leg.
09412     //  We take one with minimum number of candidates
09413     //
09414     //  There may be better method.
09415
09416     int v0, v, nl;
09417
09418     // select vertex one by one in the sequential order of node number
09419     v0 = -1;
09420     nl = 0;
09421     for (v = 0; v < nNodes; v++) {
09422         if (nodes[v]->cand->st == AS_UnAssLegs) {
09423             if (nodes[v]->cand->nilist == 1) {
09424                 return v;
09425             } else if (nodes[v]->cand->nilist > 1) {
09426                 // select node with fewer candidates
09427                 if (v0 < 0 || nodes[v]->cand->nilist < nl) {
09428                     v0 = v;
09429                     nl = nodes[v]->cand->nilist;
09430                 }
09431             }
09432         }
09433     }
09434     return v0;
09435 }
09436
09437 //-------------------------------------------------------------------
09438 int Assign::selUnAssLeg(int v, int lastlg)
09439 {
09440     // Select a leg of vertex 'v' for the assignment.
09441     // The lastly selected leg was 'lastlg'.
09442
09443     int lg0, lg, e;
09444 #ifdef CHECK
09445     int n0, n1;
09446 #endif
09447
09448     // lg0 = Max(lastlg, lastlg + 1);
09449     lg0 = lastlg + 1;
09450
```

```
09451     for (lg = lg0; lg < nodes[v]->deg; lg++) {
09452
09453 #ifdef CHECK
09454         if (lastlg >= 0) {
09455             n0 = nodes[v]->anodes[lastlg];
09456         } else {
09457             n0 = -1;
09458         }
09459         n1 = nodes[v]->anodes[lg];
09460         if (n0 > n1) {
09461             grcc_fprintf(GRCC_Stdout, "*** selUnAssLeg: n0=%d > n1=%d\n", n0, n1);
09462             grcc_fprintf(GRCC_Stdout, "*** illegal connection\n");
09463             erEnd("selUnAssLeg: n0 > n1");
09464         }
09465 #endif
09466
09467         e  = nodes[v]->aedges[lg];
09468         if (!edges[e]->cand->det) {
09469             return lg;
09470         }
09471
09472     }
09473     return -1;
09474 }
09475
09476 //-----------------------------------------------------------------
09477 NCandSt Assign::assignIVertex(int v, int ia)
09478 {
09479     //  Try to assign interaction 'ia' to 'v'
09480
09481     int     lplist[GRCC_MAXLEGS], iplist[GRCC_MAXLEGS];
09482     int     lg, e, pt, deg;
09483
09484
09485     if (nodes[v]->cand->st == AS_Assigned || nodes[v]->cand->st == AS_AssExt) {
09486         return AS_Assigned0;
09487     }
09488
09489     deg = nodes[v]->deg;
09490     if (deg != model->interacts[ia]->nlegs) {
09491         return AS_Impossible;
09492     }
09493
09494     // list of particles at the legs of the vertex
09495     for (lg = 0; lg < deg; lg++) {
09496         e = nodes[v]->aedges[lg];
09497         if (edges[e]->cand->nplist != 1) {
09498             return AS_UnAssLegs;
09499         } else {
09500             pt = edges[e]->cand->plist[0];
09501             lplist[lg] = legEdgeParticle(v, lg, pt);
09502         }
09503         iplist[lg] = model->interacts[ia]->plist[lg];
09504     }
09505
09506     sorti(deg, lplist);
09507     sorti(deg, iplist);
09508
09509     // from interaction
09510     if (cmpArray(deg, lplist, deg, iplist) == 0) {
09511         // orders of coupling constants should be checked ???
09512         astack->pushNode(v);
09513         nodes[v]->cand->st = AS_Assigned;
09514         nodes[v]->cand->nilist = 1;
09515         nodes[v]->cand->ilist[0] = ia;
09516         return AS_Assigned;
09517     }
09518
09519     return AS_Impossible;
09520 }
09521
09522 //-----------------------------------------------------------------
09523 Bool Assign::assignPLeg(int n, int ln, int pt)
09524 {
09525     //  A particle 'pt' is assigned to leg 'ln' of node 'n'
09526
09527     ANode *nd;
09528     int e, ept;
09529     Bool ok;
09530
09531 #ifdef CHECK
09532     checkNode(n, "assignPLeg0");
09533     checkCand("assignPLeg");
09534 #endif
09535
09536     // nodes and the edge
09537     nd  = nodes[n];
```

```
09538     e   = nd->aedges[ln];
09539
09540     // already determined
09541     if (edges[e]->cand->det) {
09542         return True;
09543     }
09544
09545     if (!isOrdPLeg(n, ln, pt)) {
09546         return False;
09547     }
09548
09549     // particle on the edge
09550     ept = legEdgeParticle(n, ln, pt);
09551
09552 #ifdef CHECK
09553     if (!isIn(edges[e]->cand->nplist, edges[e]->cand->plist, ept)) {
09554         grcc_fprintf(GRCC_Stdout, "*** assignPLeg: particle %d is not in the cand. of e=%d",
09555                 ept, e);
09556         edges[e]->cand->prECand("\n");
09557         prCand("assignPLeg");
09558         erEnd("assignPLeg: particle is not in the cand.");
09559     }
09560 #endif
09561
09562     // save the current configuration
09563     astack->pushEdge(e);
09564
09565     // determine the particle on the edge
09566     edges[e]->cand->nplist   = 1;
09567     edges[e]->cand->plist[0] = ept;
09568
09569     ok = detEdge(e);
09570
09571     return ok;
09572 }
09573
09574 //------------------------------------------------------------------
09575 Bool Assign::detEdge(int e)
09576 {
09577     Bool ok0, ok1;
09578
09579     if (edges[e]->cand->nplist != 1) {
09580         return False;
09581     }
09582     edges[e]->cand->det = True;
09583
09584     // update candidate list
09585     ok0 = updateCandNode(edges[e]->nodes[0]);
09586     if (ok0) {
09587         ok1 = updateCandNode(edges[e]->nodes[1]);
09588     } else {
09589         ok1 = False;
09590     }
09591
09592     return (ok0 && ok1);
09593 }
09594
09595 //------------------------------------------------------------------
09596 Bool Assign::isOrdPLeg(int n, int ln, int pt)
09597 {
09598     int e0, el, ln0, ep0, pt0, nn, n0, n1, ln1, pt1, e1, ep1;
09599
09600     if (nodes[n]->deg < 2) {
09601         return True;
09602     }
09603     nn = nodes[n]->anodes[ln];
09604     if (n <= nn) {
09605         n0  = n;
09606         n1  = nn;
09607         ln0 = ln;
09608         pt0 = pt;
09609     } else {
09610         n0 = nn;
09611         n1 = n;
09612         e0  = nodes[n]->aedges[ln];
09613         el  = nodes[n]->aelegs[ln];
09614         ln0 = edges[e0]->nlegs[1-el];
09615         ep0 = legEdgeParticle(n,  ln,  pt);
09616         pt0 = legEdgeParticle(n0, ln0, ep0);
09617     }
09618
09619     for (ln1 = 0; ln1 < nodes[n0]->deg; ln1++) {
09620         if (ln1 == ln0 || n1 != nodes[n0]->anodes[ln1]) {
09621             continue;
09622         }
09623         e1  = nodes[n0]->aedges[ln1];
09624         if (!edges[e1]->cand->det) {
```

```
09625             continue;
09626         }
09627         ep1 = edges[e1]->cand->plist[0];
09628         pt1 = legEdgeParticle(n0, ln1, ep1);
09629         if ((ln0 < ln1 && pt0 > pt1) ||
09630             (ln0 > ln1 && pt0 < pt1)) {
09631             return False;
09632         }
09633     }
09634     return True;
09635 }
09636
09637 //============================================================
09638 // Operations of candidate
09639 //------------------------------------------------------------
09640 Bool Assign::candPartClassify(int v, int *npdass, int *pdass, int *npuass, int *puass, const int size)
09641 {
09642     // Construct the classified lists of particles possible to assign to
09643     // the legs of node v
09644     //
09645     // pdass = (duplicated set of particles where edges has a unique
09646     // candidate)
09647     // puass = (duplicated set of other candidate particles)
09648
09649     int lg, e, npl, ass, jd, ju, j, pt, pts;
09650
09651     jd = 0;
09652     ju = 0;
09653     for (lg = 0; lg < nodes[v]->deg; lg++) {
09654         e   = nodes[v]->aedges[lg];
09655         npl = edges[e]->cand->nplist;
09656         if (npl < 1) {
09657             return False;
09658         }
09659         ass = (npl == 1);
09660         for (j = 0; j < edges[e]->cand->nplist; j++) {
09661             pt  = edges[e]->cand->plist[j];
09662             pts = legEdgeParticle(v, lg, pt);
09663             if (ass) {
09664                 jd = intSListAdd(jd, pdass, pts, size);
09665             } else {
09666                 ju = intSListAdd(ju, puass, pts, size);
09667             }
09668         }
09669     }
09670     *npdass = jd;
09671     *npuass = ju;
09672
09673     return True;
09674 }
09675
09676 //------------------------------------------------------------
09677 Bool Assign::updateCandNode(int v)
09678 {
09679     //  Update the configuration
09680     //  - nodes[v]->cand->ilist
09681     //  - edges[e]->cand->plist for the adjacent edge
09682     //  - nodes[n]->cand->unass for the adjacent node 'n' of 'e'
09683
09684     int   npdass,  pdass[GRCC_MAXPSLIST];
09685     int   npuass,  puass[GRCC_MAXPSLIST];
09686     int    nsub0,   sub0[GRCC_MAXPSLIST];
09687     int  niplist, iplist[GRCC_MAXPSLIST];
09688     int      nd0,     d0[GRCC_MAXMPARTICLES2];
09689     int      nd1,     d1[GRCC_MAXMPARTICLES2];
09690     int      ns0,     s0[GRCC_MAXMPARTICLES2];
09691     int  neplist, eplist[GRCC_MAXMPARTICLES2];
09692     int  nitlist, itlist[GRCC_MAXMINTERACT];
09693     int e, it, j, k, i, lg;
09694
09695     // external node
09696     if (nodes[v]->cand->st == AS_AssExt) {
09697 #ifdef CHECK
09698         e = nodes[v]->aedges[0];
09699         if (e < 0 || edges[e]->cand->nplist != 1) {
09700             grcc_fprintf(GRCC_Stdout, "*** illegal external node: v=%d e=%d :", v, e);
09701             prCand("updateCandNode");
09702             erEnd("illegal external node");
09703         }
09704 #endif
09705
09706         return True;
09707     }
09708
09709     // impossible configuration.
09710     if (nodes[v]->cand->nilist < 1) {
09711         return False;
```

```
09712     }
09713
09714     // list of possible particles on the edges of the vertex.
09715     // pdass = (set of assigned particles)
09716     // puass = (list of particles of edges with two of more candidates)
09717
09718     niplist = 0;
09719     nitlist = 0;
09720     if (!candPartClassify(v, &npdass, pdass, &npuass, puass, GRCC_MAXPSLIST)) {
09721         return False;
09722     }
09723
09724     if (npdass < 1) {
09725         return False;
09726     }
09727
09728     // from interaction : slist
09729     // Conditions :
09730     // 1. pdass \subset slist
09731     // 2. slist-pdass \subset puass
09732     // from interaction : slist
09733     //  conditions : pdass \subset slist \subset (pdass + puass)
09734     //  sub0 = slist  -   pdass,
09735     nitlist = 0;
09736     for (j = 0; j < nodes[v]->cand->nilist; j++) {
09737         it = nodes[v]->cand->ilist[j];
09738         // conditions:
09739         // 1. pdass \subset slist <==> pdass-slist = \emptyset
09740         // 2. slist-pdass \subset puass <==> (slist-pdass)-puass = \emptyset
09741         if (isSubSList(npdass, pdass,
09742                        model->interacts[it]->nslist,
09743                        model->interacts[it]->slist) ) {
09744             nsub0 = subtrSet(model->interacts[it]->nslist,
09745                              model->interacts[it]->slist,
09746                              npdass, pdass, sub0, GRCC_MAXPSLIST);
09747             if (nsub0 < 1) {
09748                 // slist == pdass : no additional candidate particle
09749                 nitlist = intSetAdd(nitlist, itlist, it, GRCC_MAXMINTERACT);
09750
09751             } else {
09752                 if (isSubSList(nsub0, sub0, npuass, puass)) {
09753                     nitlist = intSetAdd(nitlist, itlist, it, GRCC_MAXMINTERACT);
09754                     for (k = 0; k < nsub0; k++) {
09755                         niplist = intSetAdd(niplist,iplist,sub0[k],GRCC_MAXPSLIST);
09756                     }
09757                 }
09758             }
09759         }
09760     }
09761
09762     if (nitlist < 1) {
09763         return False;
09764     }
09765
09766     if (nitlist != nodes[v]->cand->nilist) {
09767         astack->pushNode(v);
09768         nodes[v]->cand->nilist = nitlist;
09769         for (i = 0; i < nitlist; i++) {
09770             nodes[v]->cand->ilist[i] = itlist[i];
09771         }
09772 #ifdef CHECK
09773     } else if (nitlist > nodes[v]->cand->nilist) {
09774         erEnd("larger ncand");
09775 #endif
09776     }
09777
09778     // edge
09779     for (lg = 0; lg < nodes[v]->deg; lg++) {
09780         e = nodes[v]->aedges[lg];
09781         if (edges[e]->cand->nplist > 1) {
09782             // elist = {candidate particles in the direction of the edge}
09783             // s0 = plist \cap elist
09784             neplist = legPart(v, lg, niplist, iplist, eplist, GRCC_MAXMPARTICLES2);
09785             listDiff(edges[e]->cand->nplist, edges[e]->cand->plist,
09786                      neplist, eplist,
09787                      &nd0, d0, &ns0, s0, &nd1, d1);
09788
09789             if (ns0 < 1) {
09790                 return False;
09791             }
09792             if (ns0 < edges[e]->cand->nplist) {
09793                 astack->pushEdge(e);
09794                 edges[e]->cand->nplist = ns0;
09795                 for (i = 0; i < ns0; i++) {
09796                     edges[e]->cand->plist[i] = s0[i];
09797                 }
09798             }
```

```
09799         }
09800     }
09801     for (lg = 0; lg < nodes[v]->deg; lg++) {
09802         e = nodes[v]->aedges[lg];
09803         if (edges[e]->cand->nplist == 1 && !edges[e]->cand->det) {
09804             if(!detEdge(e)) {
09805                 return False;
09806             }
09807         }
09808     }
09809
09810     return True;
09811 }
09812
09813 //============================================================
09814 // Check order of coupling constants, duplication and isomorphism
09815 //------------------------------------------------------------
09816 Bool Assign::checkOrderCpl(void)
09817 {
09818     //  Check order of coupling constants
09819
09820     int ord[GRCC_MAXNCPLG];
09821     int lcn, n, ia, j, cl;
09822
09823     // list of coupling constants
09824     lcn = model->ncouple;
09825
09826     if (lcn == 1) {
09827         return True;
09828     }
09829
09830     // sum up for each coupling constants
09831     for (j = 0; j < GRCC_MAXNCPLG; j++) {
09832         ord[j] = 0;
09833     }
09834     for (n = 0; n < nNodes; n++) {
09835         cl = pnclass->nd2cl[n];
09836         if (!isATExternal(pnclass->type[cl])) {
09837             ia = nodes[n]->cand->ilist[0];
09838             for (j = 0; j < lcn; j++) {
09839                 ord[j] += model->interacts[ia]->clist[j];
09840             }
09841         }
09842     }
09843
09844     // comparison
09845     for (j = 0; j < lcn; j++) {
09846         if (sproc->clist[j] != ord[j]) {
09847             return False;
09848         }
09849     }
09850     return True;
09851 }
09852
09853 //------------------------------------------------------------
09854 Bool Assign::isOrdLegs(void)
09855 {
09856     //  Whether graph is duplicated because of the violation
09857     //  of ordering in the case of multiple connections
09858     //    v0 <-- v --> v1
09859
09860     int v, l0, v0, l1, v1, e0, e1, q0, q1, p0, p1, cl;
09861
09862     for (v = 0; v < nNodes; v++) {
09863         cl = pnclass->nd2cl[v];
09864         if (!isATExternal(pnclass->type[cl])) {
09865             for (l0 = 0; l0 < nodes[v]->deg; l0++) {
09866                 v0 = nodes[v]->anodes[l0];
09867                 for (l1 = l0+1; l1 < nodes[v]->deg; l1++) {
09868                     v1 = nodes[v]->anodes[l1];
09869
09870                     if (v0 == v1 && v <= v0) {
09871                         // multiple connections (v, l0) and (v, l1)
09872                         e0 = nodes[v]->aedges[l0];
09873                         e1 = nodes[v]->aedges[l1];
09874                         q0 = edges[e0]->cand->plist[0];
09875                         q1 = edges[e1]->cand->plist[0];
09876                         p0 = legEdgeParticle(v, l0, q0);
09877                         p1 = legEdgeParticle(v, l1, q1);
09878                         if (p0 > p1) {
09879                             // violation of ordering
09880                             return False;
09881                         }
09882                     }
09883                 }
09884             }
09885         }
```

```
09886     }
09887
09888     return True;
09889 }
09890
09891 //------------------------------------------------------------
09892 Bool Assign::isIsomorphic(MNodeClass *cl, BigInt *nsym, BigInt *esym, BigInt *nsym1)
09893 {
09894     //  Check whether the current graph is the representive of
09895     //  a isomorphic class.
09896     //  Returns (nsym, esym)
09897     //    nsym = symmetry factor by the permutation of nodes.
09898     //    esym = symmetry factor by the permutation of edge.
09899     //  If this graph is not a representative, then returns (0,0).
09900
09901     int j, cmp, n, cln;
09902     BigInt ngelem;
09903     int *p;
09904     Bool invext;
09905
09906     ngelem = mgraph->group->nElem();
09907 #ifdef CHECK
09908     if (mgraph->nsym > 1 && ngelem <= 1) {
09909         grcc_fprintf(GRCC_Stdout, "*** isIsomorphic: illegal group: "
09910                "ngelem=%ld, mgraph->sym=(%ld, %ld)\n",
09911                ngelem, mgraph->nsym, mgraph->esym);
09912         erEnd("Assign::isIsomorphic: illegal group");
09913     }
09914 #endif
09915     if (ngelem == 0) {
09916         *nsym  = 1;
09917         *nsym1 = 1;
09918     } else {
09919         *nsym  = 0;
09920         *nsym1 = 0;
09921         for (j = 0; j < ngelem; j++) {
09922             // check the graph is the representative and count 'nsym'.
09923
09924             //p = mgraph->group->nextElem();
09925             p = mgraph->group->elem[j];
09926
09927             cmp = cmpPermGraph(p, cl);
09928
09929             if (cmp < 0) {        // duplicated graph
09930                 return False;
09931             } else if(cmp == 0) { // not duplicated
09932                 (*nsym)++;
09933
09934                 if (opt->values[GRCC_OPT_SymmInitial] || opt->values[GRCC_OPT_SymmFinal]) {
09935                     invext = True;
09936                     for (n = 0; n < nNodes; n++) {
09937                         cln = pnclass->nd2cl[n];
09938                         if (!isATExternal(pnclass->type[cln])) {
09939                             continue;
09940                         }
09941                         if (p[n] != n) {
09942                             invext = False;
09943                         }
09944                     }
09945                     if (invext) {
09946                         (*nsym1)++;
09947                     }
09948                 } else {
09949                     (*nsym1)++;
09950                 }
09951             }
09952         }
09953     }
09954     if (*nsym1 == 0) {
09955         *nsym1 = *nsym;
09956     }
09957
09958     // calculate permutations of edges
09959     *esym = edgeSym();
09960     if (*esym < 1) {
09961         return False;
09962     }
09963
09964     return True;
09965 }
09966
09967 //------------------------------------------------------------
09968 int Assign::cmpPermGraph(int *p, MNodeClass *cl)
09969 {
09970     //  compare the graph with one permutated by 'p'.
09971
09972     int n, cmp, n1, n2, p1, p2, j;
```

```
09973     ANode *nd1, *np1;
09974     // ANode *nd2, *np2;
09975     int    njn, jn[GRCC_MAXLEGS];
09976     int    njp, jp[GRCC_MAXLEGS];
09977
09978 #ifdef CHECK
09979     if (p == NULL) {
09980         erEnd("Assign::cmpPermGraph: p==NULL");
09981     }
09982 #endif
09983     for (n = 0; n < nNodes; n++) {
09984         if (!isATExternal(pnclass->type[pnclass->nd2cl[n]])) {
09985             cmp = cmpNodes(n, p[n], cl);
09986             if (cmp != 0) {
09987                 return cmp;
09988             }
09989         }
09990     }
09991
09992     njn = 0;
09993     njp = 0;
09994     for (n1 = 0; n1 < nNodes; n1++) {
09995         p1  = p[n1];
09996         nd1 = nodes[n1];
09997         np1 = nodes[p1];
09998         for (n2 = n1; n2 < nNodes; n2++) {
09999             if (mgraph->adjMat[n1][n2] == 0) {
10000                 continue;
10001             }
10002
10003             p2  = p[n2];
10004             if (p1 == n1 && p2 == n2) {
10005                 continue;
10006             }
10007             cmp = mgraph->adjMat[n1][n2] - mgraph->adjMat[p1][p2];
10008             if (cmp != 0) {
10009                 return cmp;
10010             }
10011
10012             njn = 0;
10013             njp = 0;
10014             for (j = 0; j < nd1->deg; j++) {
10015                 if (nd1->anodes[j] == n2) {
10016                     jn[njn++] = getLegParticle(n1, j);
10017                 }
10018             }
10019
10020             for (j = 0; j < np1->deg; j++) {
10021                 if (np1->anodes[j] == p2) {
10022                     jp[njp++] = getLegParticle(p1, j);
10023                 }
10024             }
10025
10026             cmp = njn - njp;
10027             if (cmp != 0) {
10028                 return cmp;
10029             }
10030
10031             // ignore ordering for multiple connections
10032             if (mgraph->adjMat[n1][n2] >= 2) {
10033                 sorti(njn, jn);
10034                 sorti(njn, jp);
10035             }
10036
10037             for (j = 0; j < njn; j++) {
10038                 cmp = jn[j] - jp[j];
10039                 if (cmp != 0) {
10040                     return cmp;
10041                 }
10042             }
10043         }
10044     }
10045
10046     return 0;
10047 }
10048
10049 //-------------------------------------------------------------
10050 int Assign::cmpNodes(int nd0, int nd1, MNodeClass *cn)
10051 {
10052     //  Comparison of two nodes 'nd0' and 'nd1'
10053     //    Ordering is lexicographical (class, connection configuration)
10054     //
10055
10056     int cmp;
10057
10058     // Wether two nodes are in a same class or not.
10059     cmp = cn->ndcl[nd0] - cn->ndcl[nd1];
```

```
10060     if (cmp != 0) {
10061         return cmp;
10062     }
10063
10064     // interaction
10065     cmp = nodes[nd0]->cand->ilist[0] - nodes[nd1]->cand->ilist[0];
10066     return cmp;
10067 }
10068
10069 //-------------------------------------------------------------
10070 BigInt Assign::edgeSym(void)
10071 {
10072     //  calculate permutations of edges
10073
10074     int lg[GRCC_MAXLEGS], lt[GRCC_MAXLEGS], lc;
10075     ANode *nd1;
10076     int n1, n2, j, k, mult;
10077     BigInt  esym;
10078
10079     esym = 1;
10080     for (n1 = 0; n1 < nNodes; n1++) {
10081         nd1 = nodes[n1];
10082         for (n2 = n1; n2 < nNodes; n2++) {
10083             if (mgraph->adjMat[n1][n2] > 1) {
10084                 lc = 0;
10085                 for (j = 0; j < nd1->deg; j++) {
10086                     if (nd1->anodes[j] == n2) {
10087                         lg[lc] = 1;
10088                         lt[lc] = getLegParticle(n1, j);
10089                         lc++;
10090                     }
10091                 }
10092                 for (j = 0; j < lc-1; j++) {
10093                     if (lg[j] > 0) {
10094                         for (k = lc-1; k > j; k--) {
10095                             if (lt[j] == lt[k]) {
10096                                 lg[j] += lg[k];
10097                                 lg[k]  = 0;
10098                             }
10099                         }
10100                     }
10101                 }
10102
10103                 // calculate symmetry factor
10104                 for (j = 0; j < lc; j++) {
10105                     if (lg[j] < 1) {
10106                         continue;
10107                     }
10108                     mult = lg[j];
10109                     if (mult > 1) {
10110                         if (n1 == n2) {
10111                             // self-loop
10112                             esym *= ipow(2,mult/2)*factorial(mult/2);
10113                         } else {
10114                             esym *= factorial(mult);
10115                         }
10116                     }
10117                 }
10118             }
10119         }
10120     }
10121
10122     return esym;
10123 }
10124
10125 #ifdef CHECK
10126 //=============================================================
10127 // check
10128 //-------------------------------------------------------------
10129 Bool Assign::checkCand(const char *msg)
10130 {
10131     //  Check the consistency of Candidate data
10132
10133     Bool  ok;
10134     ANode *na;
10135     NCand *nc;
10136     ECand *ec;
10137     int   n, lg, e;
10138
10139     // check 'nodes->cand'
10140     ok = True;
10141     for (n = 0; n < nNodes; n++) {
10142         na = nodes[n];
10143         nc = na->cand;
10144
10145         // check unassigned vertex
10146         if (nc->st == AS_UnAssLegs) {
```

```
10147
10148         // check assigned vertex
10149         } else if (nc->st == AS_Assigned) {
10150             if (nc->nilist < 1) {
10151                 grcc_fprintf(GRCC_Stdout, "*** checkCand:7:%s:status (%d) of node %d says"
10152                        " interaction is assigned to %d but ilist=",
10153                        msg, nc->st, n, nc->st);
10154                 prIntArray(nc->nilist, nc->ilist, "\n");
10155                 ok = False;
10156             }
10157
10158             for (lg = 0; lg < nc->deg; lg++) {
10159                 e  = na->aedges[lg];
10160                 ec = edges[e]->cand;
10161                 if (ec->nplist != 1) {
10162                     grcc_fprintf(GRCC_Stdout, "*** checkCand:8:%s:status (%d) of node %d says"
10163                           " interaction is assigned "
10164                           " but unassigned edge %d is found\n",
10165                           msg, nc->st, n, e);
10166                     ok = False;
10167                 }
10168             }
10169
10170         // external particle
10171         } else if (nc->st == AS_AssExt) {
10172             // pt  = nc->ilist[0];
10173             // *** pte = legEdgeParticle(n, 0, - pt);
10174         } else {
10175             grcc_fprintf(GRCC_Stdout, "*** checkCand:10:%s:illegal status of node %d : %d",
10176                    msg, n, nc->st);
10177             ok = False;
10178         }
10179     }
10180
10181     // check edges
10182
10183     for (e = 0; e < nEdges; e++) {
10184         ec = edges[e]->cand;
10185         if (ec != NULL && ec->nplist < 1) {
10186             grcc_fprintf(GRCC_Stdout, "*** checkCand:12:%s:illegal edge %d\n", msg, e);
10187             ok = False;
10188         }
10189     }
10190
10191     if (!ok) {
10192         grcc_fprintf(GRCC_Stdout, "*** checkCand:15:%s:illegal configuration\n", msg);
10193         prCand("checkCand");
10194         grcc_fprintf(GRCC_Stdout, "*** checkCand:16:illegal configuration\n");
10195         erEnd("checkCand:16:illegal configuration");
10196     }
10197     return ok;
10198 }
10199
10200 //-------------------------------------------------------------
10201 void Assign::checkNode(int n, const char *msg)
10202 {
10203     int j, it;
10204
10205     for (j = 0; j < nodes[n]->cand->nilist; j++) {
10206         it = nodes[n]->cand->ilist[j];
10207         if (Abs(it) >= GRCC_MAXMINTERACT) {
10208             grcc_fprintf(GRCC_Stdout, "*** %s: n=%d, j=%d, it=%d\n", msg, n, j, it);
10209             nodes[n]->cand->prNCand(msg);
10210             grcc_fprintf(GRCC_Stdout, "\n");
10211             erEnd("checkNode:illegal it");
10212         }
10213     }
10214 }
10215 #endif // CHECK
10216
10217 //*************************************************************
10218 // astack.cc
10219 //=============================================================
10220 // Assign particles to the edges and interactions to nodes.
10221 // Completed graph data is saved in the form of EGraph.
10222
10223 //=============================================================
10224 // stack operations
10225 //-------------------------------------------------------------
10226 void NStack::print(const char *msg)
10227 {
10228     grcc_fprintf(GRCC_Stdout, "  node=%d, deg=%d, st=%d, ilist=", noden, deg, st);
10229     prilist(nilist, ilist, msg);
10230 }
10231
10232 //-------------------------------------------------------------
10233 void EStack::print(const char *msg)
```

```
10234 {
10235     grcc_fprintf(GRCC_Stdout, "  edge=%d, det=%d, plist=", edgen, det);
10236     prilist(nplist, plist, msg);
10237 }
10238 
10239 //-----------------------------------------------------------------
10240 AStack::AStack(int nsize, int esize)
10241 {
10242     int j;
10243 
10244     agraph  = NULL;
10245 
10246     nSize   = nsize;
10247     eSize   = esize;
10248     if (nSize > 0) {
10249         nStack = new NStack*[nSize];
10250     } else {
10251         nStack = NULL;
10252     }
10253     if (eSize > 0) {
10254         eStack = new EStack*[eSize];
10255     } else {
10256         eStack = NULL;
10257     }
10258 
10259     nStackP = 0;
10260     eStackP = 0;
10261 
10262     for (j = 0; j < nSize; j++) {
10263         nStack[j] = new NStack();
10264     }
10265 
10266     for (j = 0; j < eSize; j++) {
10267         eStack[j] = new EStack();
10268     }
10269 }
10270 
10271 //-----------------------------------------------------------------
10272 AStack::~AStack(void)
10273 {
10274     int j;
10275 
10276     if (eStack != NULL) {
10277         for (j = eSize-1; j >= 0; j--) {
10278             if (eStack[j] != NULL) {
10279                 delete eStack[j];
10280                 eStack[j] = NULL;
10281             }
10282         }
10283         delete[] eStack;
10284         eStack = NULL;
10285     }
10286 
10287     if (nStack != NULL) {
10288         for (j = nSize-1; j >= 0; j--) {
10289             if (nStack[j] != NULL) {
10290                 delete nStack[j];
10291                 nStack[j] = NULL;
10292             }
10293         }
10294         delete[] nStack;
10295         nStack = NULL;
10296     }
10297 }
10298 
10299 //-----------------------------------------------------------------
10300 void AStack::setAGraph(Assign *ag)
10301 {
10302     agraph = ag;
10303 }
10304 
10305 //-----------------------------------------------------------------
10306 void AStack::pushNode(int n)
10307 {
10308     NCand  *nc;
10309     NStack *ns;
10310     int j;
10311 
10312 #ifdef CHECK
10313     if (agraph == NULL) {
10314         erEnd("pushNode: agraph is not defined");
10315     }
10316 #endif
10317     if (nStackP >= GRCC_MAXNSTACK) {
10318         erEnd("N-stack overflow (GRCC_MAXNSTACK)");
10319     }
10320     nc = agraph->nodes[n]->cand;
```

```
10321     if (nc->nilist >= GRCC_MAXMINTERACT) {
10322         erEnd("N-stack: too long list (GRCC_MAXMINTERACT)");
10323     }
10324     ns = nStack[nStackP];
10325     ns->noden  = n;
10326     ns->deg    = nc->deg;
10327     ns->st     = nc->st;
10328     ns->nilist = nc->nilist;
10329     for (j = 0; j < nc->nilist; j++) {
10330         ns->ilist[j] = nc->ilist[j];
10331     }
10332     nStackP++;
10333 }
10334
10335 //-------------------------------------------------------------
10336 void AStack::pushEdge(int e)
10337 {
10338     ECand  *ec;
10339     EStack *es;
10340     int j;
10341
10342 #ifdef CHECK
10343     if (agraph == NULL) {
10344         erEnd("pushEdge: agraph is not defined");
10345     }
10346 #endif
10347     if (eStackP >= GRCC_MAXESTACK) {
10348         erEnd("E-stack overflow (GRCC_MAXESTACK)");
10349     }
10350     ec = agraph->edges[e]->cand;
10351     if (ec->nplist >= GRCC_MAXMPARTICLES2) {
10352         erEnd("E-stack: Too long list (GRCC_MAXMPARTICLES)");
10353     }
10354     es = eStack[eStackP];
10355     es->edgen  = e;
10356     es->det    = ec->det;
10357     es->nplist = ec->nplist;
10358     for (j = 0; j < ec->nplist; j++) {
10359         es->plist[j] = ec->plist[j];
10360     }
10361     eStackP++;
10362 }
10363
10364 //-------------------------------------------------------------
10365 void AStack::checkPoint(CheckPt sav)
10366 {
10367     sav[0] = nStackP;
10368     sav[1] = eStackP;
10369 }
10370
10371 //-------------------------------------------------------------
10372 void AStack::restoreNode(int spr)
10373 {
10374     NStack *stc;
10375     int     sp, n, j;
10376
10377     for (sp = nStackP-1; sp >= spr; sp--) {
10378         stc = nStack[sp];
10379         n = stc->noden;
10380         agraph->nodes[n]->cand->deg     = stc->deg;
10381         agraph->nodes[n]->cand->st      = stc->st;
10382         agraph->nodes[n]->cand->nilist  = stc->nilist;
10383         for (j = 0; j < stc->nilist; j++) {
10384             agraph->nodes[stc->noden]->cand->ilist[j]  = stc->ilist[j];
10385         }
10386     }
10387     nStackP = spr;
10388 }
10389
10390 //-------------------------------------------------------------
10391 void AStack::restoreEdge(int spr)
10392 {
10393     EStack *stc;
10394     int     sp, e, j;
10395
10396     for (sp = eStackP-1; sp >= spr; sp--) {
10397         stc = eStack[sp];
10398         e   = eStack[sp]->edgen;
10399         agraph->edges[e]->cand->det = stc->det;
10400         agraph->edges[e]->cand->nplist = stc->nplist;
10401         for (j = 0; j < stc->nplist; j++) {
10402             agraph->edges[e]->cand->plist[j] = stc->plist[j];
10403         }
10404     }
10405     eStackP = spr;
10406 }
10407
```

```
10408 //-------------------------------------------------------------
10409 void AStack::restore(CheckPt sav)
10410 {
10411     restoreNode(sav[0]);
10412     restoreEdge(sav[1]);
10413 }
10414
10415 #ifdef CHECK
10416 //-------------------------------------------------------------
10417 void AStack::restoreMsg(CheckPt sav, const char *msg)
10418 {
10419     if (agraph == NULL) {
10420         erEnd("restore: agraph is not defined");
10421     }
10422
10423     restore(sav);
10424
10425     if (!agraph->checkCand("restore")) {
10426         grcc_fprintf(GRCC_Stdout, "restore is called from %s\n", msg);
10427     }
10428 }
10429 #endif
10430
10431 //-------------------------------------------------------------
10432 void AStack::prStack(void)
10433 {
10434     int j;
10435
10436     grcc_fprintf(GRCC_Stdout, "+++ prStack : (%d, %d)", nStackP, eStackP);
10437     for (j = 0; j < nStackP; j++) {
10438         grcc_fprintf(GRCC_Stdout, "N:%4d ", j);
10439         nStack[j]->print("\n");
10440     }
10441     for (j = 0; j < eStackP; j++) {
10442         grcc_fprintf(GRCC_Stdout, "E:%4d ", j);
10443         eStack[j]->print("\n");
10444     }
10445 }
10446
10447 //*************************************************************
10448 // class Fraction
10449 //=============================================================
10450 Fraction::Fraction(BigInt n, BigInt d)
10451 {
10452     BigInt g;
10453
10454     g     = gcd(n, d);
10455     num   = n/g;
10456     den   = d/g;
10457     ratio = Real(n)/Real(d);
10458 }
10459
10460 //-------------------------------------------------------------
10461 void Fraction::print(const char *msg)
10462 {
10463     double err = Abs(Real(num)/Real(den) - ratio);
10464
10465     if (err > GRCC_FRACERROR) {
10466         grcc_fprintf(GRCC_Stdout, "%ld/%ld(%g)(overflow)%s", num, den, ratio, msg);
10467     } else {
10468         grcc_fprintf(GRCC_Stdout, "%ld/%ld(%g)%s", num, den, ratio, msg);
10469     }
10470 }
10471
10472 //-------------------------------------------------------------
10473 void Fraction::setValue(BigInt n, BigInt d)
10474 {
10475     num = n;
10476     den = d;
10477     ratio = Real(n)/Real(d);
10478 }
10479
10480 //-------------------------------------------------------------
10481 void  Fraction::setValue(Fraction &f)
10482 {
10483     num   = f.num;
10484     den   = f.den;
10485     ratio = f.ratio;
10486 }
10487
10488 //-------------------------------------------------------------
10489 BigInt Fraction::gcd(BigInt n0, BigInt n1)
10490 {
10491     BigInt nn[2], r;
10492     int    nc;
10493
10494     nn[0] = Abs(n0);
```

```
10495     nn[1] = Abs(n1);
10496     nc    = 0;
10497
10498     while (1) {
10499         r = nn[nc] % nn[1-nc];
10500         if (r == 0) {
10501             return nn[1-nc];
10502         }
10503         nn[nc] = r;
10504         nc = 1 - nc;
10505     }
10506 }
10507
10508 //--------------------------------------------------------------
10509 void Fraction::normal(void)
10510 {
10511     BigInt g;
10512
10513     if (den == 0) {
10514         erEnd("Fraction: den==0");
10515     }
10516     g   = gcd(num, den);
10517     num /= g;
10518     den /= g;
10519     if (den < 0) {
10520         num = - num;
10521         den = - den;
10522     }
10523 }
10524
10525 //--------------------------------------------------------------
10526 void Fraction::add(BigInt n, BigInt d)
10527 {
10528     BigInt g, d1;
10529
10530     if (d < 0) {
10531         n = -n;
10532         d = -d;
10533     }
10534     if (den < 0) {
10535         num = - num;
10536         den = - den;
10537     }
10538     g    = gcd(den, d);
10539     d1   = d/g;
10540     num  = num * d1 + n * (den/g);
10541     den  = d1*den;
10542     g    = gcd(num, den);
10543     num /= g;
10544     den /= g;
10545     ratio += Real(n)/Real(d);
10546 }
10547
10548 //--------------------------------------------------------------
10549 void Fraction::add(Fraction f)
10550 {
10551     double r = ratio + f.ratio;
10552
10553     add(f.num, f.den);
10554     ratio = r;
10555 }
10556
10557 //--------------------------------------------------------------
10558 void Fraction::sub(Fraction f)
10559 {
10560     double r = ratio - f.ratio;
10561
10562     add(-f.num, f.den);
10563     ratio = r;
10564 }
10565
10566 //--------------------------------------------------------------
10567 Bool Fraction::isEq(Fraction f)
10568 {
10569     return (num == f.num && den == f.den);
10570 }
10571
10572 //**************************************************************
10573 // common.cc
10574 //==============================================================
10575
10576 // Wrapper function for printing messages. This allows the use
10577 // of FORM MesPrint when compiled as part of FORM, and the
10578 // usual fprintf to GRCC_Stdout or GRCC_Stderr otherwise.
10579 static void grcc_fprintf(FILE* out, const char* fmt, ...)
10580 {
10581     va_list args;
```

```
10582     va_start(args, fmt);
10583
10584 #ifndef NOFORM
10585     DUMMYUSE(out);
10586     // the second call of vsnprintf requires a copy of args
10587     va_list args_copy;
10588     va_copy(args_copy, args);
10589     // determine the required buffer size for the formatted string:
10590     const int len = vsnprintf(NULL, 0, fmt, args);
10591     char *buffer = new char[len+1];
10592     vsnprintf(buffer, len+1, fmt, args_copy);
10593     MLOCK(ErrorMessageLock);
10594     MesPrint("%s%", buffer);
10595     MUNLOCK(ErrorMessageLock);
10596     delete[] buffer;
10597     va_end(args_copy);
10598 #else
10599     vfprintf(out, fmt, args);
10600 #endif
10601
10602     va_end(args);
10603     return;
10604 }
10605
10606 static void erEnd(const char *msg)
10607 {
10608     if (erExit != NULL) {
10609         (*erExit)(msg, erExitArg);
10610     }
10611     grcc_fprintf(GRCC_Stderr, "*** Error : %s\n\n", msg);
10612     GRCC_ABORT();
10613 }
10614
10615 #define MAXPART 100
10616 //-------------------------------------------------------------
10617 static Bool nextPart(int nelem, int nclist, int *clist, int *nl, int *r)
10618 {
10619     // Generate configuration nl[] sequentially such that
10620     //    sum_j^{nclist} nl[j]*clist[j] = nelem
10621     //    0 <= nl[j]    (j = 0, ..., nclis-1)
10622     // For control
10623     //    for the first call : *r < 0
10624     //    otherwise          : *r >= 0,  nl is the last configuration
10625     // Returns
10626     //    True  : succeeded
10627     //    False : no more configuration
10628
10629     int rem, pn, j, c;
10630
10631     if (*r < 0) {
10632         *r = 0;
10633         rem = nelem;
10634         for (j = 0; j < nclist; j++) {
10635             nl[j] = rem/clist[j];
10636             rem -= nl[j]*clist[j];
10637         }
10638         if (rem == 0) {
10639             return True;
10640         }
10641     } else {
10642         rem = 0;
10643     }
10644     for (c = 0; c < MAXPART; c++) {
10645         rem += nl[nclist-1]*clist[nclist-1];
10646         for (pn = nclist-2; pn >= 0 && nl[pn] == 0; pn--) {
10647             ;
10648         }
10649         if (pn < 0) {
10650             return False;
10651         }
10652         rem += clist[pn];
10653         nl[pn]--;
10654         for (j = pn+1; j < nclist; j++) {
10655             nl[j] = rem/clist[j];
10656             rem -= nl[j]*clist[j];
10657         }
10658         if (rem == 0) {
10659             return True;
10660         }
10661     }
10662     erEnd("*** nextPart : illegal control : too many repetition");
10663     return False;
10664 }
10665
10666 //-------------------------------------------------------------
10667 static int   *intdup(int n, int *a)
10668 {
```

```
10669     int *r, j;
10670
10671     r = new int[n];
10672     for (j = 0; j < n; j++) {
10673         r[j] = a[j];
10674     }
10675     return r;
10676 }
10677
10678 //---------------------------------------------------------------
10679 static int   *delintdup(int *a)
10680 {
10681     if (a != NULL) {
10682         delete[] a;
10683     }
10684     return NULL;
10685 }
10686
10687 //---------------------------------------------------------------
10688 static void   prilist(int n, const int *a, const char *msg)
10689 {
10690     grcc_fprintf(GRCC_Stdout, "[");
10691     for (int j = 0; j < n; j++) {
10692         if (j!=0) grcc_fprintf(GRCC_Stdout, ", ");
10693         grcc_fprintf(GRCC_Stdout, "%d", a[j]);
10694     }
10695     grcc_fprintf(GRCC_Stdout, "]%s", msg);
10696 }
10697
10698 //---------------------------------------------------------------
10699 static int nextPerm(int nelem, int nclass, int *cl, int *r, int *q, int *p, int count)
10700 {
10701     // Sequential generation of all permutations.
10702
10703     int  j, k, n, e, t;
10704     Bool b;
10705
10706     for (j = 0; j < nelem; j++) {
10707         p[j] = j;
10708     }
10709     if (count < 1) {
10710         for (j = 0; j < nelem; j++) {
10711             q[j] = 0;
10712             r[j] = 0;
10713         }
10714         j = 0;
10715         for (k = 0; k < nclass; k++) {
10716             n = cl[k];
10717             for (e = 0; e < n; e++) {
10718                 r[j] = n - e - 1;
10719                 j++;
10720             }
10721         }
10722 #ifdef CHECK
10723         if (j != nelem) {
10724             erEnd("inconsistent # elements");
10725         }
10726 #endif
10727         return 1;
10728     }
10729     b = False;
10730     for (j = nelem-1; j >= 0; j--) {
10731         if (q[j] < r[j]) {
10732             for (k = j+1; k < nelem; k++) {
10733                 q[k] = 0;
10734             }
10735             q[j]++;
10736             b = True;
10737             break;
10738         }
10739     }
10740     if (!b) {
10741         return (-count);
10742     }
10743
10744     for (j = 0; j < nelem; j++) {
10745         k = j + q[j];
10746         t = p[j];
10747         p[j] = p[k];
10748         p[k] = t;
10749     }
10750     return count + 1;
10751 }
10752
10753 //---------------------------------------------------------------
10754 static BigInt factorial(int n)
10755 {
```

```
10756     // returns 1 for n < 1
10757
10758     int r, j;
10759
10760     r = 1;
10761     for (j = 2; j <= n; j++) {
10762         r *= j;
10763     }
10764     return r;
10765 }
10766
10767 //-------------------------------------------------------------
10768 static BigInt ipow(int n, int p)
10769 {
10770     int r, j;
10771
10772     r = 1;
10773     for (j = 0; j < p; j++) {
10774         r *= n;
10775     }
10776     return r;
10777 }
10778
10779 //-------------------------------------------------------------
10780 static int   *newArray(int size, int val)
10781 {
10782     // memory allocation of an array
10783
10784     int *a, j;
10785
10786     a = new int[size];
10787     for (j = 0; j < size; j++) {
10788         a[j] = val;
10789     }
10790     return a;
10791 }
10792
10793 //-------------------------------------------------------------
10794 static int   *deleteArray(int *a)
10795 {
10796     // memory allocation of an array
10797
10798     delete[] a;
10799     return NULL;
10800 }
10801
10802 //-------------------------------------------------------------
10803 static int  **newMat(int n0, int n1, int val)
10804 {
10805     int **m, j, k;
10806
10807     m = new int*[n0];
10808     for (j = 0; j < n0; j++) {
10809         m[j] = new int[n1];
10810         for (k = 0; k < n1; k++) {
10811             m[j][k] = val;
10812         }
10813     }
10814     return m;
10815 }
10816
10817 //-------------------------------------------------------------
10818 static int **deleteMat(int **m, int n0)
10819 {
10820     int j;
10821
10822     for (j = n0-1; j >= 0; j--) {
10823         delete[] m[j];
10824     }
10825     delete[] m;
10826
10827     return NULL;
10828 }
10829
10830 //-------------------------------------------------------------
10831 static void bsort(int n, int *ord, int *a)
10832 {
10833     int i, j, t;
10834
10835     for (j = n-1; j > 0; j--) {
10836         for (i = 0; i < j; i++) {
10837             if(a[ord[i+1]] < a[ord[i]]) {
10838                 t = ord[i];
10839                 ord[i] = ord[i+1];
10840                 ord[i+1] = t;
10841             }
10842         }
```

```
10843     }
10844 }
10845
10846 //-------------------------------------------------------------
10847 static void prMomStr(int mom, const char *ms, int mn)
10848 {
10849     if (mom == 0) {
10850         return;
10851     } else if (mom == 1) {
10852         grcc_fprintf(GRCC_Stdout, " + %s%d", ms, mn);
10853     } else if (mom > 0) {
10854         grcc_fprintf(GRCC_Stdout, " + %d*%s%d", mom, ms, mn);
10855     } else if (mom == -1) {
10856         grcc_fprintf(GRCC_Stdout, " - %s%d", ms, mn);
10857     } else {
10858         grcc_fprintf(GRCC_Stdout, " - %d*%s%d", -mom, ms, mn);
10859     }
10860 }
10861
10862 //-------------------------------------------------------------
10863 static void prIntArray(int n, int *p, const char *msg)
10864 {
10865
10866     int j;
10867
10868     grcc_fprintf(GRCC_Stdout, "[");
10869     for (j = 0; j < n; j++) {
10870         if (j!=0) grcc_fprintf(GRCC_Stdout, ", ");
10871         grcc_fprintf(GRCC_Stdout, "%2d", p[j]);
10872     }
10873     grcc_fprintf(GRCC_Stdout, "]%s", msg);
10874 }
10875
10876 //-------------------------------------------------------------
10877 static void prIntArrayErr(int n, int *p, const char *msg)
10878 {
10879
10880     int j;
10881
10882     grcc_fprintf(GRCC_Stderr, "[");
10883     for (j = 0; j < n; j++) {
10884         if (j!=0) grcc_fprintf(GRCC_Stderr, ", ");
10885         grcc_fprintf(GRCC_Stderr, "%2d", p[j]);
10886     }
10887     grcc_fprintf(GRCC_Stderr, "]%s", msg);
10888 }
10889
10890 //-------------------------------------------------------------
10891 static void sortrec(int l, int r, int *a)
10892 {
10893     // quick sort : taken from N. Wirth
10894     int i,j, x, w;
10895
10896     i = l;
10897     j = r;
10898     x = a[(l+r)/2];
10899     do {
10900         while(a[i] < x) i++;
10901         while(x < a[j]) j--;
10902         if (i <= j) {
10903             w = a[i];  a[i] = a[j];  a[j] = w;
10904             i++;  j--;
10905         }
10906     } while (i < j);
10907     if (l < j) sortrec(l, j, a);
10908     if (i < r) sortrec(i, r, a);
10909 }
10910
10911 //-------------------------------------------------------------
10912 static void sorti(int size, int *a)
10913 {
10914     sortrec(0, size-1, a);
10915 }
10916
10917 //-------------------------------------------------------------
10918 static int sortb(int n, int *a)
10919 {
10920     int i, j, t, nswap;
10921
10922     nswap = 0;
10923     for (j = n-1; j > 0; j--) {
10924         for (i = 0; i < j; i++) {
10925             if(a[i+1] < a[i]) {
10926                 t = a[i];
10927                 a[i] = a[i+1];
10928                 a[i+1] = t;
10929                 nswap++;
```

```
10930             }
10931         }
10932     }
10933
10934     // the sign of the permutation is (-1)^{nswap}
10935     return nswap;
10936 }
10937
10938 #ifdef CHECK
10939 //------------------------------------------------------------------
10940 static Bool isIn(int n, int *a, int v)
10941 {
10942     int j;
10943
10944     for (j = 0; j < n; j++) {
10945         if (a[j] == v) {
10946             return True;
10947         }
10948     }
10949     return False;
10950 }
10951 #endif
10952
10953 //------------------------------------------------------------------
10954 static int intSetAdd(int n, int *a, int v, const int size)
10955 {
10956     // Add 'v' into 'a'.
10957     // 'n' is the current # of element.
10958     // 'a' is assumed sorted without duplicated elements.
10959     // Size of 'a' should be large enough.
10960
10961     int j, k;
10962
10963     if (n >= size) {
10964         grcc_fprintf(GRCC_Stderr, "*** intSetAdd : array out of range (>%d)\n", size);
10965         erEnd("intSetAdd : array out of range (GRCC_MAXPSLIST)");
10966     }
10967     for (j = 0; j < n; j++) {
10968         if (a[j] > v) {
10969             break;
10970         } else if (a[j] == v) {
10971             return n;
10972         }
10973     }
10974
10975     // 'j' can be equal to 'n'
10976     for (k = n-1; k >= j; k--) {
10977         a[k+1] = a[k];
10978     }
10979     a[j] = v;
10980     return n+1;
10981 }
10982
10983 //------------------------------------------------------------------
10984 static int intSListAdd(int n, int *a, int v, const int size)
10985 {
10986     // Add 'v' into 'a'.
10987     // 'n' is the current # of element.
10988     // 'a' is assumed sorted without duplicated elements.
10989     // Size of 'a' should be large enough.
10990
10991     int j, k;
10992
10993     if (n >= size) {
10994         grcc_fprintf(GRCC_Stderr, "*** intSListAdd : array out of range (>%d)\n", size);
10995         erEnd("intSListAdd : array out of range");
10996     }
10997     for (j = 0; j < n; j++) {
10998         if (a[j] >= v) {
10999             break;
11000         }
11001     }
11002
11003     // 'j' can be equal to 'n'
11004     for (k = n-1; k >= j; k--) {
11005         a[k+1] = a[k];
11006     }
11007     a[j] = v;
11008     return n+1;
11009 }
11010
11011 //------------------------------------------------------------------
11012 static int    cmpArray(int na, int *a, int nb, int *b)
11013 {
11014     int j, cmp;
11015
11016     cmp = na - nb;
```

```
11017     if (cmp != 0) {
11018         return cmp;
11019     }
11020     for (j = 0; j < na; j++) {
11021         cmp = a[j] - b[j];
11022         if (cmp != 0) {
11023             return cmp;
11024         }
11025     }
11026     return 0;
11027 }
11028
11029 //-------------------------------------------------------------
11030 static Bool   leqArray(int n, int *a, int *b)
11031 {
11032     int j;
11033
11034     for (j = 0; j < n; j++) {
11035         if (a[j] > b[j]) {
11036             return False;
11037         }
11038     }
11039     return True;
11040 }
11041
11042 //-------------------------------------------------------------
11043 // convert list to sorted list
11044 static int   toSList(int n, int *a)
11045 {
11046     sorti(n, a);
11047     return n;
11048 }
11049
11050 //-------------------------------------------------------------
11051 // as set operation assuming a and b are sorted with duplication
11052 //     p := a \ b;  q := a \cap b;  r := b \ a;
11053 // p, q, r are sorted without duplication
11054 static void   listDiff(int na, int *a, int nb, int *b, int *np, int *p, int *nq, int *q, int *nr, int
      *r)
11055 {
11056     int ja, jb;
11057
11058     *np = *nq = *nr = 0;
11059     ja = jb = 0;
11060     while (ja < na && jb < nb) {
11061         while (ja < na && a[ja] < b[jb]) {
11062             p[(*np)++] = a[ja++];
11063         }
11064         while (jb < nb && b[jb] < a[ja]) {
11065             r[(*nr)++] = b[jb++];
11066         }
11067         if (ja < na && jb < nb && a[ja] == b[jb]) {
11068             q[(*nq)++] = a[ja++];
11069             jb++;
11070         }
11071     }
11072     for (; ja < na; ja++) {
11073         p[(*np)++] = a[ja];
11074     }
11075     for (; jb < nb; jb++) {
11076         r[(*nr)++] = b[jb];
11077     }
11078 }
11079
11080 //-------------------------------------------------------------
11081 static int isSubSList(int na, int *a, int nb, int *b)
11082 {
11083     int ja, jb;
11084
11085     ja = jb = 0;
11086     while (ja < na && jb < nb) {
11087         for ( ; jb < nb && b[jb] < a[ja]; jb++) {
11088             ;
11089         }
11090         if (jb >= nb || b[jb] > a[ja]) {
11091             return False;
11092         }
11093         for ( ; jb < nb && ja < na && b[jb] == a[ja]; jb++, ja++) {
11094             ;
11095         }
11096     }
11097     return (ja >= na);
11098 }
11099
11100 //-------------------------------------------------------------
11101 static int subtrSet(int na, int *a, int nb, int *b, int *c, int size)
11102 {
```

```
11103     // set subtraction 'c' := 'a' - 'b'.
11104     // 'a' and 'b' are sorted lists (not always unique)
11105     // 'c' is the set (sorted and unique elements)
11106
11107     int ja, jb, jc;
11108
11109     jc = 0;
11110     ja = jb = 0;
11111     while (ja < na && jb < nb) {
11112         while (ja < na && a[ja] < b[jb]) {
11113             if (jc == 0 || c[jc-1] != a[ja]) {
11114                 if (jc >= size) {
11115                     erEnd("subsutSet: too small size of array (GRCC_MAXPSLIST)");
11116                 }
11117                 c[jc++] = a[ja];
11118             }
11119             ja++;
11120         }
11121         while (jb < nb && b[jb] < a[ja]) {
11122             ;
11123         }
11124         if (ja < na && jb < nb && a[ja] == b[jb]) {
11125             ja++;
11126             jb++;
11127         }
11128     }
11129     for (; ja < na; ja++) {
11130         if (jc == 0 || c[jc-1] != a[ja]) {
11131             c[jc++] = a[ja];
11132         }
11133     }
11134     return jc;
11135 }
11136
11137 // } ) ]
```

4.60 grcc.h

```
00001 #pragma once
00002
00003 /* #[ License : */
00004 /*
00005  *   Copyright (C) 2023-2026 T. Kaneko
00006  *   When using this file you are requested to refer to the publication
00007  *   Comput.Phys.Commun. 92 (1995) 127-152
00008  *
00009  *   This file is part of FORM.
00010  *
00011  *   FORM is free software: you can redistribute it and/or modify it under the
00012  *   terms of the GNU General Public License as published by the Free Software
00013  *   Foundation, either version 3 of the License, or (at your option) any later
00014  *   version.
00015  *
00016  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00017  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00018  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00019  *   details.
00020  *
00021  *   You should have received a copy of the GNU General Public License along
00022  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00023  */
00024 /* #] License : */
00025
00026 #include <stdio.h>
00027 #include <stdlib.h>
00028 #include <string.h>
00029 #include <time.h>
00030
00031 #define CHECK
00032 //============================================================
00033 extern "C" {
00034 #include "grccparam.h"
00035 }
00036
00037 //============================================================
00038 // Name space of this library
00039 // See 'grccparam.h' for #define
00040 #ifdef GRCC_NAMESPACE
00041 namespace Grcc {
00042 #endif
00043
00044 //============================================================
00045 // Common types
```

```
00046 //
00047 #define True   1
00048 #define False  0
00049 typedef int Bool;
00050
00051 //============================================================
00052 // Big integer (may be replaced by suitable one)
00053
00054 #define BigInt       long
00055 #define ToBigInt(x)  ((BigInt) (x))
00056
00057 //============================================================
00058 // Macro functions
00059 #define Real(x)      ((double) (x))
00060 #define Abs(x)       (((x) >= 0)? (x): (-x))
00061 #define Sign(x)      (((x) >= 0)? (1): (-1))
00062 #define Max(x, y)    ((x) > (y) ? (x) : (y))
00063 #define Min(x, y)    ((x) < (y) ? (x) : (y))
00064 #define Second(t) ((double)(*(t)=(double)(clock()/((double)CLOCKS_PER_SEC))))
00065 #define isATExternal(x) ((x)==GRCC_AT_Initial||(x)==GRCC_AT_Final||(x)==GRCC_AT_External)
00066
00067 //============================================================
00068 // types and classes
00069 typedef int  Edge2n[2];   // pair of nodes expressing an edge
00070 class Assign;
00071 class AStack;
00072 class EEdge;
00073 class EGraph;
00074 class ENode;
00075 class Interaction;
00076 class MGraph;
00077 class MNodeClass;
00078 class MCEdge;
00079 class MCOpi;
00080 class MCBridge;
00081 class MCBlock;
00082 class MConn;
00083 class Model;
00084 class MOrbits;
00085 class Options;
00086 class Output;
00087 class Particle;
00088 class PNodeClass;
00089 class Process;
00090 class SGroup;
00091 class SProcess;
00092 class DCGraph;
00093
00094 //============================================================
00095 // type of output functions
00096 typedef Bool OutEGB(EGraph *, void *);
00097 typedef void OutEG(EGraph *, void *);
00098 typedef void ErExit(const char *msg, void *);
00099
00100 //============================================================
00101 class Options {
00102   public:
00103
00104     Model      *model;
00105     Process    *proc;
00106     SProcess   *sproc;
00107     Output     *out;         // for output
00108     OutEGB     *outmg;       // call back function of mgraph
00109     OutEGB     *endmg;       // call back function of end of mgraph
00110     OutEG      *outag;       // call back function of agraph
00111     void       *argmg;       // additional argument to outmg
00112     void       *argemg;      // additional argument to endmg
00113     void       *argag;       // additional argument to outag
00114
00115     //switches for graph generation
00116     OptQGRef qgref[2*GRCC_QGRAF_OPT_Size]; // array of reference to QG-options
00117     int      values[GRCC_OPT_Size];        // array of options
00118     int      qgopt[GRCC_QGRAF_OPT_Size];   // array of QGRAF options
00119
00120     int      nqgopt;                       // effective length of qgref
00121
00122     int          DUMMYPADDING;
00123
00124     // measuring time
00125     double time0;
00126     double time1;
00127
00128     //------------------
00129     Options(void);
00130     ~Options(void);
00131     void setDefaultValues(void);
00132     void setOldDefaultValues(void);
```

```
00133
00134     void setValue(int ind, int val);
00135     int  getValue(int ind);
00136
00137     void setQGrafOpt(int *qgopt);
00138
00139     void print(void);
00140
00141     void setOutputF(Bool outgrf, const char *fname);
00142     void setOutputP(Bool outgrp, const char *fname);
00143     void printLevel(int l);
00144
00145     void printModel(void);
00146     void setOutMG(OutEGB *omg, void *pt);
00147     void setOutAG(OutEG  *oag, void *pt);
00148     void setEndMG(OutEGB *omg, void *pt);
00149     void setErExit(ErExit  *ere, void *pt);
00150     const OptDef *getDef(void);
00151     const OptDef *getOldDef(void);
00152     const OptQGDef *getQGDef(void);
00153
00154     void begin(Model *mdl);
00155     void end(void);
00156     void beginProc(Process *prc);
00157     void endProc(void);
00158     void beginSubProc(SProcess *sprc);
00159     void endSubProc(void);
00160     void newMGraph(MGraph *mgr);
00161     void newAGraph(EGraph *egr);
00162     void outModel(void);
00163 };
00164
00165 //============================================================
00166 class Output {
00167   public:
00168
00169     Options    *opt;
00170     Model      *model;
00171     Process    *proc;
00172     SProcess   *sproc;
00173     char       *outgrf;
00174     FILE       *outgrfp;
00175     char       *outgrp;
00176     FILE       *outgrpp;
00177     int         procId;
00178     Bool        outproc;
00179
00180     Output(Options *optn);
00181     ~Output(void);
00182
00183     void setOutgrf(const char *fname);
00184     void setOutgrp(const char *fname);
00185     Bool outBeginF(Model *mdl, Bool pr);
00186     Bool outBeginP(Model *mdl, Bool pr);
00187     void outEndF(void);
00188     void outEndP(void);
00189     void outProcBeginF(Process *prc);
00190     void outProcBeginP(Process *prc);
00191     void outProcBegin0(int next, int couple, int loop);
00192     void outSProcBeginF(SProcess *sprc);
00193     void outSProcBeginP(SProcess *sprc);
00194     void outProcEndF(void);
00195     void outProcEndP(void);
00196     void outEGraphF(EGraph *egraph);
00197     void outEGraphP(EGraph *egraph);
00198     void outModelF(void);
00199     void outModelP(void);
00200
00201 };
00202
00203 //============================================================
00204 class Fraction {
00205   public:
00206     BigInt num, den;
00207     double ratio;
00208
00209     Fraction() { num = 0;  den = 1; };
00210     Fraction(BigInt n, BigInt d);
00211
00212     void   print(const char *msg);
00213     void   setValue(BigInt n, BigInt d);
00214     void   setValue(Fraction &f);
00215     void   add(BigInt n, BigInt d);
00216     void   add(Fraction f);
00217     void   sub(Fraction f);
00218     BigInt gcd(BigInt n0, BigInt n1);
00219     void   normal(void);
```

```
00220     Bool   isEq(Fraction f);
00221 };
00222
00223 //************************************************************
00224 // Model
00225 //============================================================
00226 //------------------------------------------------------------
00227 class Particle {
00228   public:
00229     Model *mdl;             // the model
00230     char  *name;            // the name of the particle
00231     char  *aname;           // the name of the anti-particle
00232     int    id;              // id of this particle
00233     int    ptype;           // the type of partice : GRCC_PT_Scalar, etc.
00234     int    neutral;         // True if particle==anti-particle
00235     int    pcode;           // Particle code
00236     int    acode;           // Anti-particle code
00237     int    cmindeg;         // min(leg of connectable interactions)
00238     int    cmaxdeg;         // max(leg of connectable interactions)
00239
00240     int    extonly;         // can appear only as external particle
00241
00242     //-----------------------------
00243     Particle(Model *modl, int pid, PInput *pinp);
00244     ~Particle(void);
00245
00246     char *particleName(int p);
00247     int   particleCode(int p);
00248     char *interactionName(int p);
00249     char *aparticle(void);
00250     void  prParticle(void);
00251     int   isNeutral(void);
00252     const char *typeName(void);
00253     const char *typeGName(void);
00254
00255 };
00256
00257 //------------------------------------------------------------
00258 class Interaction {
00259   public:
00260     Model *mdl;           // the model
00261     char  *name;          // the name of the interaction
00262     int   *plist;        // the list of particle codes
00263     int   *clist;        // the orders of coupling constants
00264     int   *slist;        // the sorted list of particle codes
00265
00266     int    id;            // id of this interaction
00267     int    csum;          // the total orders of coupling constants
00268     int    nlegs;         // the number of legs
00269     int    loop;          // the number of loops
00270     int    icode;         // user defined interaction code
00271
00272     int    nplist;        // the list of particle codes
00273     int    nclist;        // the orders of coupling constants
00274     int    nslist;        // the sorted list of particle codes
00275
00276     //-----------------------------
00277     Interaction(Model *modl, int iid, const char* nam, int icode, int *cpl, int nlgs, int *plst, int
    csm, int lp);
00278     ~Interaction(void);
00279
00280     void prInteraction(void);
00281 };
00282
00283 //------------------------------------------------------------
00284 class Model {
00285   public:
00286     char         *name;        // the name of this model
00287     char        **cnlist;       // the list of coupling constant names
00288     int           ncouple;     // the number of coupling consants.
00289
00290     int           nParticles;  // the number of particles
00291     Particle    **particles;   // the list of particles
00292     int           pdef;        // the def. of partcls is ended
00293
00294     // list of particles and anti-p. without Undef.
00295     int           nallPart;
00296     int           allPart[GRCC_MAXMPARTICLES2];
00297
00298     // list of particles and anti-p. without ones of extloop=True
00299     int           nintPart;
00300     int           intPart[GRCC_MAXMPARTICLES2];
00301
00302     int           nInteracts;  // the number of interactions.
00303     Interaction **interacts;   // the list of interactions
00304     int           vdef;        // the definition of interaction is ended
00305
```

```
00306     int           maxnlegs;    // maximum number of legs of an interaction
00307     int           maxcpl;      // maximum number of coupling const.
00308     int           maxloop;     // maximum number of loops inside an intr.
00309
00310     // classification of interactions
00311     int          *cplgcp;      // coupling constants
00312     int          *cplglg;      // degree
00313     int          *cplgnvl;     // number of vertices
00314     int         **cplgvl;      // list of vertices
00315     int           ncplgcp;     // the number of classes
00316
00317     int           defpart;     // GRCC_DEFBYNAME or GRCC_DEFBYCODE
00318
00319     // methods
00320     Model(MInput *minp);
00321     ~Model(void);
00322
00323     void  prModel(void);
00324     void  addParticle(PInput *pinp);
00325     void  addParticleEnd(void);
00326     void  addInteraction(IInput *iinp);
00327     void  addInteractionEnd(void);
00328     int   findParticleName(const char *name);
00329     int   findParticleCode(int pcd);
00330     int   findInteractionName(const char *name);
00331     int   findInteractionCode(int icd);
00332     char *particleName(int p);
00333     int   particleCode(int p);
00334     int   normalParticle(int pt);
00335     int   antiParticle(int pt);
00336     int  *allParticles(int *len);
00337     int   findMClass(const int cpl, const int dgr);
00338     void  prParticleArray(int n, int *a, const char *msg);
00339
00340     static void printMInput(MInput *min);
00341     static void printPInput(PInput *pin);
00342     static void printIInput(IInput *iin);
00343 };
00344
00345 //************************************************************
00346 // Process
00347 //============================================================
00348 class PNodeClass {
00349   public:
00350     SProcess *sproc;
00351     int      *deg;       // The degree of each node
00352     int      *type;      // The type of the class : GRCC_AT_xxx
00353     int      *particle;  // The particle code of external node.
00354     int      *couple;    // The orders of coupling constans
00355     int      *cmindeg;   // min(deg of connectable vertex)
00356     int      *cmaxdeg;   // max(deg of connectable vertex)
00357     int      *count;     // The number of nodes in the class
00358     int      *cl2nd;     // cl2nd[class] <= nd < cl2nd[class+1] = set of nodes
00359     int      *nd2cl;     // nd2cl[node]  = class
00360     int      *cl2mcl;    // cl2mcl[class] = class defined in the model.
00361     int       nnodes;
00362     int       nclass;
00363
00364     PNodeClass(SProcess *sprc, int nnods, int nclss, NCInput *cls);
00365     PNodeClass(SProcess *sprc, int nnods, int nclss, int *dgs, int *typ, int *ptcl, int *cpl, int
      *cnt, int *cmind, int *cmaxd);
00366     ~PNodeClass(void);
00367
00368     void prPNodeClass(void);
00369     void prElem(int e);
00370 };
00371
00372 //============================================================
00373 class SProcess {
00374   //  In sprocess the number of nodes and edges are fixed.
00375   //  A node is considered as
00376   //    (degree, total order of coupling constants),
00377   //  which pair of data defines a class of nodes.
00378
00379   public:
00380     Model       *model;       // model
00381     Process     *proc;        // mother process
00382     Options     *opt;         // options
00383     PNodeClass  *pnclass;     // list of classes (cpl and deg is determined)
00384     AStack      *astack;
00385
00386     MGraph      *mgraph;
00387     EGraph      *egraph;
00388     Assign      *agraph;
00389
00390     int         *cl2nd;       // (code of nodes in the class[c])
00391                               //  = [cl2nd[c], ..., cl2nd[c+1]-1]
```

```
00392     int         *nd2cl;       // node nd is in the class pnclass[nd2cl[nd]]
00393     int          nclass;      // the number of classes
00394
00395     int          ninitl;      // the number of initial particles
00396     int          nfinal;      // the number of final particles
00397     int          nvert;       // the number of vertices
00398
00399     int          clist[GRCC_MAXNCPLG];  // coupling constans of the process
00400     int          id;          // sprocess id
00401     int          loop;        // the number of loops
00402     int          nNodes;      // the number of nodes
00403     int          nEdges;      // the number of edges
00404     int          nExtern;     // the number of external particles
00405     int          ncouple;     // the number of coupling constants
00406     int          tCouple;     // the total coupling constants
00407
00408     int          DUMMYPADDING;
00409
00410     BigInt       mgrcount;    // count generated mgraph
00411     BigInt       agrcount;    // count generated agraph
00412     BigInt       extperm;    // count generated agraph
00413
00414     // the results of the graph generation
00415     BigInt   nMGraphs;        // the number of generated M-graphs
00416     BigInt   nMOPI;           // the number of 1PI M-graphs
00417     Fraction wMGraphs;        // the weighted sum of M-graphs
00418     Fraction wMOPI;           // the weighted sum of 1PI M-graphs
00419
00420     BigInt   nAGraphs;        // the number of generated A-graphs
00421     BigInt   nAOPI;           // the number of 1PI A-graphs
00422     Fraction wAGraphs;        // the weighted sum of A-graphs
00423     Fraction wAOPI;           // the weighted sum of 1PI A-graphs
00424
00425     // methods
00426     SProcess(Model *mdl, Process *prc, Options *opts, int sid, int *clst, int ncls, NCInput *cls);
00427     SProcess(Model *mdl, Process *prc, Options *opts, int sid, int *clst, int ncls, int *cdeg, int
      *ctyp, int *ptcl, int *cpl, int *cnum, int *cmind, int *cmaxd);
00428
00429     ~SProcess(void);
00430
00431     void prSProcess(void);
00432     BigInt generate(void);
00433     void assign(MGraph *mgr);
00434
00435     int toMNodeClass(int *ctyp, int *cldeg, int *clnum, int *cmind, int *cmaxd);
00436     PNodeClass *match(MGraph *mgr);
00437
00438     void endMGraph(MGraph *mgr);
00439     void endAGraph(EGraph *egr);
00440
00441     void resultMGraph(BigInt nmgraphs, Fraction mwsum, BigInt nmopi, Fraction mwopi);
00442     void resultAGraph(BigInt nagraphs, Fraction awsum, BigInt naopi, Fraction awopi);
00443
00444 };
00445
00446 //============================================================
00447 class Process {
00448   public:
00449     Model       *model;            // model used for this process
00450     Options     *opt;              // options
00451     BigInt       mgrcount;         // count generated mgraph
00452     BigInt       agrcount;         // count generated agraph
00453     int         *initlPart;        // list of ids of initial particles
00454     int         *finalPart;        // list of ids of final particles
00455
00456     int          id;               // process id
00457     int          ninitl;           // the number of initial particles
00458     int          nfinal;           // the number of final particles
00459     int          ctotal;           // the total number of coupling constants
00460     int          nExtern;          // the number of external particles
00461     int          loop;             // the number of external particles
00462     int          maxnlegs;         // the maximum possible degree of node
00463     int          clist[GRCC_MAXNCPLG];  // coupling constans of the process
00464
00465     // table of sprocesses
00466     int        nSubproc;           // the number of sprocesses
00467     SProcess  *sptbl[GRCC_MAXSUBPROCS]; // the table of sprocesses
00468     SProcess  *sproc;              // the current sprocess
00469
00470     // stack for the assignment;
00471     AStack      *astack;
00472
00473     // the number of graphs
00474     BigInt       ngraphs;
00475     BigInt       nopi;
00476     BigInt       wgraphs;
00477     BigInt       wopi;
```

```
00478
00479     // the results of the graph generation
00480     BigInt       nMGraphs;             // the number of generated M-graphs
00481     BigInt       nMOPI;                // the number of 1PI M-graphs
00482     Fraction     wMGraphs;             // the weighted sum of M-graphs
00483     Fraction     wMOPI;                // the weighted sum of 1PI M-graphs
00484
00485     BigInt       nAGraphs;             // the number of generated A-graphs
00486     BigInt       nAOPI;                // the number of 1PI A-graphs
00487     Fraction     wAGraphs;             // the weighted sum of A-graphs
00488     Fraction     wAOPI;                // the weighted sum of 1PI A-graphs
00489
00490     double       sec;
00491
00492     Process(int pid, Model *model, Options *opt, int nin, int *initlPart, int nfin, int *finalPart,
      int *coupling);
00493     Process(int pid, Model *model, Options *opt, FGInput *fgi);
00494     ~Process(void);
00495
00496     void prProcess(void);
00497     void outProcP(FILE *fp);
00498     void prProcessP(const char *fname);
00499     void mkSProcess(void);
00500
00501 };
00502
00503 //*************************************************************
00504 // classes for EGraph
00505 //=============================================================
00506 // Constants
00507
00508 #define GRCC_ED_Undef   0
00509 #define GRCC_ED_Deleted 1
00510 #define GRCC_ED_Extern  2
00511 #define GRCC_ED_Back    3
00512 #define GRCC_ED_Bridge  4
00513 #define GRCC_ED_Inloop  5
00514 #define GRCC_ED_Size    6
00515 #define GRCC_ED_NAMES   {"Undef", "Deleted", "Extern", "Back_Edge", "Bridge", "In_Loop"}
00516
00517 #define GRCC_ND_Undef   0
00518 #define GRCC_ND_Deleted 1
00519 #define GRCC_ND_Initial 2
00520 #define GRCC_ND_Final   3
00521 #define GRCC_ND_CPoint  4
00522 #define GRCC_ND_VBlock  5
00523 #define GRCC_ND_Inblock 6
00524 #define GRCC_ND_Size    7
00525 #define GRCC_ND_NAMES   {"Undef", "Deleted", "Init", "Final", "CPoint", "VBlock", "In_Block"}
00526
00527 //-------------------------------------------------------------
00528 class ENode {
00529   public:
00530     EGraph *egraph;
00531     int    *edges;           // list of edges
00532                              // the value is \pm [(edge index)+1]
00533
00534     int     id;              // id of the enode
00535     int     maxdeg;          // maximum degree
00536     int     deg;             // degree
00537     int     extloop;         // loop inside this node or AT_Initial etc.
00538     int     ndtype;          // type of the node
00539     int     intrct;          // assigned interaction/particle (ext.)
00540
00541     // used in EGraph::biconn()
00542     int    *klow;
00543     int     visited;
00544
00545     int    DUMMYPADDING;
00546
00547     //--------------------------------
00548     // functions
00549     ENode(void);
00550     ENode(EGraph *egrph, int loops, int sdeg);
00551     ~ENode(void);
00552     void initAss(EGraph *egrph, int nid, int sdg);
00553     void setId(EGraph *egrph, const int nid);
00554     void copy(ENode *en);
00555     void setExtern(int typ, int pt);
00556     void setType(int typ);
00557     void print(void);
00558
00559 };
00560
00561 //-------------------------------------------------------------
00562 class EEdge {
00563   // momentum is printed like: ("%s%d", (enode.ext)?"Q":"p", enode.momn)
```

```
00564   public:
00565     EGraph *egraph;        // egraph
00566
00567     int  id;               // id
00568     int  ext;
00569     int  ptcl;             // assigned particle (agraph)
00570     int  deleted;          // deleted edge, if true
00571     int  nodes[2];         // nodes of bothsides
00572     int  nlegs[2];         // nodes of both side (agraph)
00573
00574     // for biconn
00575     int *emom;             // external momenta
00576     int *lmom;             // loop momenta
00577     int *extMom;           // set of external momenta.
00578
00579     // momentum obtain in searchME
00580     ULong  momset;         // set of momenta in bit string (leaf --> root)
00581     int    momdir;         // direction (leg=0 --> leg=1)
00582
00583     Bool cut;
00584     int  visited;
00585     int  conid;            // connected component
00586     int  edtype;           // type
00587     int  opicomp;          // id of 1PI component
00588     int  dir;              // direction of momentum
00589
00590     int      DUMMYPADDING;
00591
00592     //--------------------------------
00593     // functions
00594     EEdge(void);
00595     EEdge(EGraph *egrph, int nedges, int nloops);
00596     ~EEdge(void);
00597     void copy(EEdge *ee);
00598     void setId(EGraph *egrph, const int eid);
00599     void print(void);
00600
00601     void setType(int typ);
00602     void setLMom(int k, int dir);
00603     void setEMom(int nedges, int *extn, int dir);
00604 };
00605
00606 //-------------------------------------------------------------
00607 // type of fermion lines
00608
00609 typedef enum {FL_Open, FL_Closed} FLType;
00610
00611 //-------------------------------------------------------------
00612 class EFLine {
00613   public:
00614     int    elist[GRCC_MAXNODES];     // list of  (\pm [(edge index)+1])
00615     FLType ftype;
00616     int    fkind;
00617     int    nlist;
00618
00619     EFLine(void);
00620     void print(const char *msg);
00621 };
00622
00623 //-------------------------------------------------------------
00624 class EGraph {
00625   public:
00626     Options    *opt;          // table of options
00627     Model      *model;
00628     Process    *proc;
00629     SProcess   *sproc;
00630     MGraph     *mgraph;
00631     MConn      *econn;
00632
00633     ENode **nodes;
00634     EEdge **edges;          // edges[nEdges+1]: index starts from 1
00635
00636     BigInt mId;             // id of mgraph
00637     BigInt aId;             // id of agraph in the same mgraph
00638     BigInt sId;             // sequential no. of agraph
00639     BigInt gSubId;          // ???
00640     Bool   assigned;        // mgraph (False) or agraph (True)
00641
00642     int    fsign;
00643     BigInt nsym, esym;      // symmetry factor with symm. ext.
00644     BigInt nsym1;           // symmetry factor without symm. ext.
00645     BigInt extperm;         // the order of group of symm. ext.
00646     BigInt multp;           // multiplicity of graph in symm. ext.
00647
00648     int    pId;
00649
00650     int   sNodes;           // memory size
```

```
00651     int   sEdges;            // memory size
00652     int   sMaxdeg;           // memory size
00653     int   sLoops;            // memory size
00654
00655     int    nNodes;
00656     int    nEdges;
00657     int    nExtern;
00658     int    maxdeg;            // maximum value of degree of nodes
00659     int    nLoops;
00660     int    totalc;            // total order of coupling constants
00661
00662     // biconnect
00663     int    nopicomp;
00664     int    opi2plp;
00665     int    nopi2p;
00666     int    nadj2ptv;           // the no. of edges connecting 2point vertices
00667
00668     int    DUMMYPADDING;
00669
00670     int   *bidef;
00671     int   *bilow;
00672     int   *extMom;
00673     int    bconn;
00674     int    bicount;
00675     int    loopm;
00676     int    opiCount;
00677
00678     // Fermion lines
00679     EFLine *flines[GRCC_MAXFLINES];
00680     int     nFlines;
00681
00682     int    DUMMYPADDING1;
00683
00684     //-------------------------
00685     // functions
00686     EGraph(int nnodes, int nedges, int mxdeg);
00687     ~EGraph(void);
00688
00689     void copy(EGraph *eg);
00690     void print(void);
00691     void printPy(FILE *fp, long mId);
00692     void fromDGraph(DGraph *dg);
00693     void fromMGraph(MGraph *mgraph);
00694
00695     Bool   optQGrafM(Options *opt);
00696     Bool   optQGrafA(Options *opt);
00697     Bool isOptE(void);
00698
00699     ENode *setExtern(int n0, int pt, int ndtp);
00700     Bool   isExternal(int nd)  { return (nodes[nd]->extloop < 0); };
00701     Bool   isFermion(int nd);
00702
00703     void setExtLoop(int nd, int val);
00704     void endSetExtLoop(void);
00705
00706     int connComp(void);
00707     int connVisit(int nd, int ncc);
00708
00709     void biconnE(void);
00710     void biinitE(void);
00711     void bisearchE(int nd, int *extlst, int *intlst, int *opiext, int *opiloop);
00712
00713     int  findRoot(void);
00714     int  dirEdge(int n, int e);
00715     void extMomConsv(void);
00716     int  cmpMom(int *lm0, int *em0, int *lm1, int *em1);
00717     int  groupLMom(int *grp, int *ed2gr);
00718
00719     void chkMomConsv(void);
00720
00721     void prFLines(void);
00722     void getFLines(void);
00723     int  fltrace(int fk, int nd0, int *fl);
00724     void addFLine(const FLType ft, int fk, int nfl, int *fl);
00725
00726     int  legParticle(int ed, int lg);
00727 };
00728
00729 //************************************************************
00730 //  Symmetry group of graphs
00731 //============================================================
00732 class SGroup {
00733   public:
00734     BigInt  size;              // # of allocated elements
00735     BigInt  nelem;             // # of saved elements
00736     int   **elem;              // saved elements
00737     int     nnodes;            // # of nodes.
```

```
00738     int     neclass;           // # of classes
00739     int     eclass[GRCC_MAXNODES];  // table of classes
00740     int     cgen;              // counter for the generation
00741     int     csav;              // counter for the saved elements
00742     int     permg[GRCC_MAXNODES];   // resulting permutation
00743     int     perms[GRCC_MAXNODES];   // curr. elem of saved ones.
00744
00745     int     pgr[GRCC_MAXNODES];  // work
00746     int     pgq[GRCC_MAXNODES];  // work
00747     int     psr[GRCC_MAXNODES];  // work
00748     int     psq[GRCC_MAXNODES];  // work
00749
00750     //------------------------------
00751     // functions
00752
00753     SGroup(void);
00754     ~SGroup(void);
00755
00756     void  print(void);
00757     void  newGroup(int nelm, int nclss, int *clss);
00758     void  clearGroup(void);
00759     void  delGroup(void);
00760     int  *genNext(void);
00761     void  addGroup(int *p);
00762     BigInt  nElem(void);
00763     int  *nextElem(void);
00764 };
00765
00766 //*************************************************************
00767 // MGraph
00768 //=============================================================
00769 // class of nodes for MGraph
00770
00771 class MNode {
00772   public:
00773     int id;         // node id
00774     int deg;        // degree(node) = the number of legs
00775     int clss;       // initial class number in which the node belongs
00776     int extloop;
00777     int cmindeg;
00778     int cmaxdeg;
00779
00780     int freelg;     // the number of free legs
00781     int visited;
00782
00783     MNode(int id, int clss, NCInput *mgi);
00784     MNode(int id, int deg, int extloop, int clss, int cmind, int cmaxd);
00785 };
00786
00787 //=============================================================
00788 //  class of scalar graph expressed by matrix form
00789 //
00790 //  Input  : the classified set of nodes.
00791 //  Output : control passed to 'EGraph(self)'
00792
00793 class MGraph {
00794
00795   public:
00796
00797     // initial conditions
00798     Options  *opt;          // options
00799
00800     // symmetry group
00801     SGroup   *group;        // symmetry group
00802     MOrbits  *orbits;       // Orbits of nodes with respect to symmetry group
00803
00804     MNode **nodes;          // table of MNode object
00805     BigInt mId;             // process/sprocess ID
00806     int   *clist;           // list of initial classes
00807     int    pId;             // process/sprocess ID
00808     int    nNodes;          // the number of nodes
00809     int    nEdges;          // the number of edges
00810     int    nLoops;          // the number of loops
00811     int    nExtern;         // the number of external nodes
00812     int    nClasses;        // the number of initial classes
00813     int    mindeg;          // minimum value of degree of nodes
00814     int    maxdeg;          // maximum value of degree of nodes
00815
00816     // the current graph
00817     int **adjMat;           // adjacency matrix
00818     MNodeClass *curcl;      // the current 'MNodeClass' object
00819     EGraph *egraph;
00820     BigInt nsym;            // symmetry factor from nodes
00821     BigInt esym;            // symmetry factor from edges
00822
00823     // generated set of graphs
00824     BigInt cDiag;           // the total number of generated graphs
```

```
00825     BigInt c1PI;           // the total number of 1PI graphs
00826     BigInt cNoTadpole;     // the total number of graphs without tadpoles
00827     BigInt cNoTadBlock;    // the total number of graphs without tad-blocks
00828     BigInt c1PINoTadBlock; // the total number of 1PI graphs without tad-blocks
00829     Fraction wscon;         // weighted sum of graphs
00830     Fraction wsopi;         // weighted sum of 1PI graphs
00831
00832     // measures of efficiency
00833     BigInt ngen;              // generated graph before check
00834     BigInt ngconn;            // generated connected graph before check
00835
00836     BigInt nCallRefine;
00837     BigInt discardRefine;
00838     BigInt discardDisc;
00839     BigInt discardIso;
00840
00841     // for options
00842     Bool opi;
00843     Bool opiloop;
00844     Bool extself;
00845     Bool selfloop;
00846     Bool multiedge;
00847     Bool tadpole;
00848     Bool tadblock;
00849     Bool block;
00850     Bool bipart;
00851
00852     int    DUMMYPADDING1;
00853
00854     // table of n edge-connected components
00855     MConn *mconn;
00856
00857     // work space for isomorphism
00858     int **modmat;           // permutated adjacency matrix
00859
00860     // work space for biconnected component
00861     int *bidef;
00862     int *bilow;
00863     int *bicol;
00864     int  bicount;
00865
00866     int    DUMMYPADDING2;
00867
00868     //----------
00869     // functions
00870     MGraph(int pid, int ncl, NCInput *mgi, Options *opt);
00871     MGraph(int pid, int ncl, int *cldeg, int *clnum, int *clexl, int *cmind, int *cmaxd, Options
      *opt);
00872     ~MGraph(void);
00873
00874     void   init(void);
00875     BigInt   generate(void);
00876     Bool   isExternal(int nd)  { return (nodes[nd]->extloop < 0); };
00877     void   printAdjMat(MNodeClass *cl);
00878     void   print(void);
00879     void   printPy(FILE *fp, long mId);
00880
00881     Bool   isConnected(void);
00882     Bool   visit(int nd);
00883     Bool   isIsomorphic(MNodeClass *cl);
00884     void   permMat(int size, int *perm, int **mat0, int **mat1);
00885     int    compMat(int size, int **mat0, int **mat1);
00886     MNodeClass *refineClass(MNodeClass *cl);
00887     void   bisearchME(int nd, int pd, int ned, int col, MCOpi *mopi, MCBlock *mblk, ULong *momset, int
      *next, int *nart);
00888     void   biconnME(void);
00889
00890     Bool   isOptM(void);
00891     void   connectClass(MNodeClass *cl);
00892     void   connectNode(int sc, int ss, MNodeClass *cl);
00893     void   connectLeg(int sc, int sn, int tc, int ts, MNodeClass *cl);
00894     void   newGraph(MNodeClass *cl);
00895 };
00896
00897 //=============================================================
00898 //  class of node-classes for MGraph
00899 //-------------------------------------------------------------
00900 class MNodeClass {
00901   public:
00902     int   clmat[GRCC_MAXNODES][GRCC_MAXNODES];   // matrix used for classification
00903     int   clist[GRCC_MAXNODES];    // the number of nodes in each class
00904     int   ndcl[GRCC_MAXNODES];     // node --> class
00905     int   flist[GRCC_MAXNODES+1];  // the first node in each class
00906     int   clord[GRCC_MAXNODES];    // ordering of classes
00907     int   cmindeg[GRCC_MAXNODES];    // min(deg of connectable node)
00908     int   cmaxdeg[GRCC_MAXNODES];    // max(deg of connectable node)
00909     int   nNodes;          // the number of nodes
```

```
00910     int   nClasses;       // the number of classes
00911     int   maxdeg;         // maximal value of degree(node)
00912
00913     int   flg0;
00914     int   flg1;
00915     int   flg2;
00916
00917     MNodeClass(int nnodes, int nclasses);
00918     ~MNodeClass(void);
00919
00920     void  init(int *cl, int mxdeg, int **adjmat);
00921     void  copy(MNodeClass* mnc);
00922     int   clCmp(int nd0, int nd1, int cn);
00923     void  printMat(void);
00924
00925     void  mkFlist(void);
00926     void  mkNdCl(void);
00927     void  mkClMat(int **adjmat);
00928     void  incMat(int nd, int td, int val);
00929     int   cmpMNCArray(int *a0, int *a1, int ma);
00930     void  reorder(MGraph *mg);
00931 };
00932
00933 //=============================================================
00934 //  class of an edge
00935 //-------------------------------------------------------------
00936 class MCEdge {
00937   public:
00938     Edge2n nodes;  // nodes at the both size of the edge (leaf --> root)
00939     ULong  momset; // set of momenta in bit string
00940     int    momdir; // set of momenta in bit string
00941
00942     int      DUMMYPADDING;
00943
00944     MCEdge(void);
00945     ~MCEdge(void);
00946 };
00947
00948 //=============================================================
00949 //  class of 1PI component
00950 //-------------------------------------------------------------
00951 class MCOpi {
00952   public:
00953     int *nodes;    // array of nodes in
00954     int  nnodes;   // # nodes in the 1PI component
00955     int  nlegs;    // # leg (bridges) of the 1PI component
00956     int  next;     // # external particles of the 1PI comp.
00957     int  nedges;   // # edges in the 1PI comp.
00958     int  loop;     // # loops in the 1PI comp.
00959     int  ctloop;   // # loops in the counter terms in the OP comp.
00960     int  mom0lg;   // # leg (bridges) with 0 momentum
00961
00962     int      DUMMYPADDING;
00963
00964     MCOpi(void);
00965     ~MCOpi(void);
00966
00967     void init(void);
00968 };
00969
00970 //=============================================================
00971 //  class of bridge
00972 //-------------------------------------------------------------
00973 class MCBridge {
00974   public:
00975     Edge2n nodes;  // nodes at the both size of the bridge
00976     int    next;   // # momenta of ext. particles flowing on the bridge
00977
00978     MCBridge(void);
00979     ~MCBridge(void);
00980 };
00981
00982 //=============================================================
00983 //  class of block
00984 //-------------------------------------------------------------
00985 class MCBlock {
00986   public:
00987     Edge2n *edges;   // array of edges in the block
00988     int     nmedges; // # edges in the block
00989     int     nartps;  // # articulation points of the block
00990     int     loop;    // # loop in the block
00991
00992     int    DUMMYPADDING;
00993
00994     MCBlock(void);
00995     ~MCBlock(void);
00996     void init(void);
```

```
00997 };
00998
00999 //============================================================
01000 //  class of table of MConn
01001 //------------------------------------------------------------
01002 class MConn {
01003   public:
01004     // 2-edge connected components
01005     MCEdge    *cedges;      // table of edges
01006     MCOpi     *opics;       // table of n-edge-connected components
01007     MCBridge  *bridges;     // table of bridges
01008     MCBlock   *blocks;      // table of blocks
01009     int       *articuls;    // buffer for nodes for articulation points
01010
01011     int       *opisp;       // buffer for nodes in 1PI components
01012     int       *opistk;      // stack of nodes for 1PI components.
01013     Edge2n    *blksp;       // buffer for edges in blocks
01014     Edge2n    *blkstk;      // stack of edges for blocks
01015     int        snodes;      // # nodes
01016     int        sedges;      // # edges
01017
01018     // opi components (edge-connected)
01019     int        nopic;       // # 1PI components (n1PIComps)
01020     int        nlpopic;     // # looped 1PI components (n1PIComps)
01021     int        nctopic;     // # 1PI components of one counter term.
01022
01023     // edges
01024     int        nbacked;     // # back edges
01025
01026     // bridges
01027     int        nbridges;    // # bridges
01028     int        ne0bridges;  // # bridges whose next=0
01029     int        ne1bridges;  // # bridges whose next=1
01030     int        nselfloops;
01031     int        nmultiedges;
01032
01033     // blocks (node-connected)
01034     int        nblocks;     // # blocks
01035     int        na1blocks;   // # bridges whose next=0
01036     int        narticuls;   // # articulation points
01037     int        neblocks;    // # effective looped blocks
01038
01039     // indices to work spaces
01040     int        nopisp;      // # used in opisp
01041     int        opistkptr;   // # stack pointer of opistk
01042     int        nblksp;      // # used in blksp
01043     int        blkstkptr;   // # stack pointer of tlkstk
01044
01045     int    DUMMYPADDING;
01046
01047     MConn(int nnod, int nedg);
01048     ~MConn(void);
01049
01050     void init(void);
01051     void initCEdges(MGraph *);
01052     void pushNode(int nd);
01053     void pushEdge(int n0, int n1);
01054     void addCEdge(int n0, int n1, ULong momset);
01055     void addOPIc(MCOpi *mopi, int stp);
01056     void addBridge(int n0, int n1, int nex, int nextot);
01057     void addArtic(int nd, int mul);
01058     void addBlock(MCBlock *eblk, int stp);
01059     void addBlockSelf(int nd, int mul);
01060     void print(void);
01061     void prEdges(void);
01062 };
01063
01064 //============================================================
01065 //  class of orbits of nodes
01066 //------------------------------------------------------------
01067 class MOrbits {
01068   // Usage
01069   // 1. node ==> orbit
01070   //    (class) = nd2or[(node)]
01071   //
01072   // 2. class ==> nodes
01073   //    for (c = 0; c < nClass; c++) {
01074   //        for (j = flist[c]; j < flist[c+1]; j++) {
01075   //            (node) = or2nd[c];
01076   //        }
01077   //    }
01078   public:
01079     int nOrbits;
01080     int nNodes;
01081     int nd2or[GRCC_MAXNODES];
01082     int or2nd[GRCC_MAXNODES];
01083     int flist[GRCC_MAXNODES+1];
```

```
01084
01085     MOrbits(void);
01086     ~MOrbits(void);
01087
01088     void print(void);
01089
01090     // construction of orbits
01091     void initPerm(int nnodes);
01092     void fromPerm(int *perm);
01093     void toOrbits(void);
01094 };
01095
01096 //*************************************************************
01097 // Particle assignment
01098 //=============================================================
01099 typedef int CheckPt[2];
01100
01101 typedef enum {
01102     AS_UnAssLegs, AS_Assigned, AS_Assigned0, AS_AssExt, AS_Impossible
01103 } NCandSt;
01104
01105 //=============================================================
01106 class NCand {
01107   // List of candidates for the assignment of interactions to a vertex
01108
01109   public:
01110     int      deg;                    // degree of the node
01111     NCandSt  st;                     // status
01112     int      nilist;                 // length of the list
01113     int      ilist[GRCC_MAXMINTERACT];     // list of candidates
01114
01115     //========
01116     NCand(const NCandSt sta, const int dega, const int nilst, int *ilst);
01117     ~NCand(void);
01118
01119     // print
01120     void prNCand(const char* msg);
01121 };
01122
01123 //=============================================================
01124 class ECand {
01125   // List of candidates for the assignment of particles to an edge
01126
01127   public:
01128     Bool  det;                    // determined or not
01129     int   nplist;                 // size of the list of candidates
01130     int   plist[GRCC_MAXMPARTICLES2];   // list of candidates
01131
01132     ECand(int dt, int nplist, int *plst);
01133     ~ECand(void);
01134
01135     // print
01136     void   prECand(const char *msg);
01137 };
01138
01139 //=============================================================
01140 class ANode {
01141   // Connection information of a node to others
01142   //
01143   //  (this node) -- (adjacent edge) -- (next node)
01144   //    (n0, j)            e               n1
01145   //
01146   // e  = (aedges[j], aelegs[j])
01147   // n1 = anodes[j]
01148
01149   public:
01150     int    deg;              // degree of the node
01151     int    nlegs;            // the number of legs already assigned
01152     int   *anodes;           // anodes[j] = (next node of leg j)
01153     int   *aedges;           // aedges[j] = (next edge of leg j)
01154     int   *aelegs;           // aelegs[j] = (leg of the next edge)
01155     NCand *cand;             // candidate list
01156
01157     ANode(int dg);
01158     ~ANode(void);
01159
01160     int  newleg(void);        // get a new leg
01161 };
01162
01163 //=============================================================
01164 class AEdge {
01165 // Connection information of an edge
01166 // (self) ==> nodes [(nodes[0], nlegs[0]), (nodes[1], nlegs[1])]
01167 // particle 'ptcl' flows from leg=0 to 1.
01168
01169   public:
01170
```

```
01171     ECand *cand;            // candidate
01172     int    nodes[2];        // nodes of both sides of the edge
01173     int    nlegs[2];        // nodes of both sides of the edge
01174     int    ptcl;            // particle defined in the model
01175
01176     int    DUMMYPADDING;
01177
01178     AEdge(int n0, int l0, int n1, int l1);
01179     ~AEdge(void);
01180 };
01181
01182 //============================================================
01183 class Assign {
01184 // Class for particle/interaction assignment.
01185
01186   public:
01187
01188     EGraph     *egraph;       // EGraph object
01189     MGraph     *mgraph;       // MGraph object
01190     SProcess   *sproc;        // Sub-process object
01191     Process    *proc;         // Process object
01192     Model      *model;        // Model object
01193     Options    *opt;          // Option object
01194     AStack     *astack;       // Stack for saving candidates
01195     PNodeClass *pnclass;      // class of nodes
01196     MOrbits    *orbits;       // orbits of nodes by symmetry group
01197
01198     int         nNodes;       // the number of nodes
01199     int         nEdges;       // the number of edges
01200     int         nExtern;      // the number of external particles.
01201     int         nETotal;      // the total number of edges
01202
01203     BigInt      nAGraphs;     // the number of assigned graphs
01204     Fraction    wAGraphs;     // the weighted sum of graphs
01205     BigInt      nAOPI;        // the number of assigned 1PI
01206     Fraction    wAOPI;        // the weighted sum of 1PI
01207
01208     ANode     **nodes;        // table of nodes
01209     AEdge     **edges;        // table of edges
01210
01211     CheckPt     checkpoint0;  // save stack pointers
01212
01213     int         cplleft[GRCC_MAXNCPLG];  // coupling constants left
01214
01215     //-------------------------------------------
01216     Assign(SProcess *sprc, MGraph *mgr, PNodeClass *pnc);
01217     ~Assign(void);
01218
01219     //-------------------------------------------
01220     // print lists of candidates
01221     void    prCand(const char *msg);
01222
01223     // check
01224     void checkAG(const char *msg);
01225
01226     //-------------------------------------------
01227     // control of assignment
01228
01229     // entry point of assignment
01230     Bool    assignAllVertices(void);
01231
01232     // select a source node for assignment
01233     Bool    selectVertex(void);
01234     Bool    selectVertexSimp(int lastv);
01235
01236     // select a source leg of the node for assignment
01237     Bool    selectLeg(int v, int lastlg);
01238
01239     // assign a vertex a interaction
01240     Bool    assignVertex(int v);
01241
01242     // assignment procedure is finished.  Needs check.
01243     Bool    allAssigned(void);
01244
01245     //-------------------------------------------
01246     // Input and output
01247
01248     // Input from mgraph
01249     Bool    fromMGraph(void);
01250
01251     // add an edge
01252     void    addEdge(int n0, int n1, int nplist, int *plist);
01253
01254     // connect nodes and edges.
01255     void    connect(int n0, int l0, int eg, int el, int n1, int l1);
01256
01257     // Output to egraph
```

```
01258     Bool    fillEGraph(int aid, BigInt nsym, BigInt esym, BigInt nsym1);
01259
01260     // reorder legs in accordance with the interaction definition
01261     int    *reordLeg(int n, int *reord, int *plist, int *used);
01262
01263     //=========================================
01264     // direction of particle on an edge and at (node, leg)
01265
01266     // convert particle code
01267     //   at node-leg ('n', 'ln') <==> edge at ('n', 'ln')
01268     int     getLegParticle(int n,  int ln);
01269
01270     // convert particle 'pt' on the edge to ('n', 'ln')
01271     int     legEdgeParticle(int n, int ln, int pt);
01272
01273     // convert candidate list at ('v', 'lg') into in-coming direction
01274     int     legPart(int v, int lg, int nplst, int *plst, int *rlist, const int size);
01275
01276     // convert candidate list at ('v', 'lg') into in-coming direction
01277     int     candPart(int v, int ln, int *plist, const int size);
01278
01279     //=========================================
01280     // assignment
01281
01282     // find a vertex to be assigned
01283     int     selUnAssVertex(void);
01284     int     selUnAssVertexSimp(int lastv);
01285
01286     // find a leg of the vertex to be assigned
01287     int     selUnAssLeg(int v, int lastlg);
01288
01289     // assign the interaction 'ia' to the vertex 'v'.
01290     NCandSt assignIVertex(int v, int ia);
01291
01292     // assign particle 'pt' the the leg 'ln' of the node 'n'.
01293     Bool    assignPLeg(int n, int ln, int pt);
01294     Bool    isOrdPLeg(int n, int ln, int pt);
01295     Bool    detEdge(int e);
01296
01297     //=========================================
01298     // canndidates
01299
01300     // construct lists of assigned / unassigned particles at a node
01301     Bool    candPartClassify(int v, int *npdass, int *pdass, int *npuass, int *puass, const int size);
01302
01303     // update candidate list for a vertex 'v'.
01304     Bool    updateCandNode(int v);
01305
01306     //=========================================
01307     // filter
01308
01309     Bool    checkOrderCpl(void);
01310     Bool    isOrdLegs(void);
01311     Bool    isIsomorphic(MNodeClass *cl, BigInt *nsym, BigInt *esym, BigInt *nsym1);
01312     int     cmpPermGraph(int *p, MNodeClass *cl);
01313     int     cmpNodes(int nd0, int nd1, MNodeClass *cn);
01314     BigInt  edgeSym(void);
01315
01316     void    saveCouple(int *sav);
01317     void    restoreCouple(int *sav);
01318     Bool    subCouple(int *cpl);
01319
01320 #ifdef CHECK
01321     //=========================================
01322     // check
01323     Bool    checkCand(const char *msg);
01324
01325     void    checkNode(int n, const char *msg);
01326 #endif
01327 };
01328
01329 //************************************************************
01330 // Stack of candidate lists
01331 //============================================================
01332 class NStack {
01333 // Stack for backtracking method for a node.
01334
01335   public:
01336     int     noden;
01337     int     deg;
01338     NCandSt st;
01339     int     nilist;
01340     int     ilist[GRCC_MAXMINTERACT];
01341
01342     NStack() { };
01343     ~NStack() { };
01344
```

```
01345     void print(const char *msg);
01346
01347 };
01348
01349 //============================================================
01350 class EStack {
01351 // Stack for backtracking method for an edge.
01352
01353   public:
01354     int    edgen;
01355     int    det;
01356     int    nplist;
01357     int    plist[GRCC_MAXMPARTICLES2];
01358
01359     EStack() { };
01360     ~EStack() { };
01361
01362     void print(const char *msg);
01363 };
01364
01365
01366 //============================================================
01367 class AStack {
01368 // Stack for backtracking method for an edge.
01369
01370   public:
01371     Assign     *agraph;
01372
01373     NStack    **nStack;       // stack for nodes
01374     int         nStackP;      // stack pointer for nodes
01375     int         nSize;        // stack size
01376
01377     EStack    **eStack;       // stack for edges
01378     int         eStackP;      // stack pointer for edges
01379     int         eSize;        // stack size
01380
01381     AStack(int nSize, int eSize);
01382     ~AStack(void);
01383
01384     void setAGraph(Assign *ag);
01385
01386     void checkPoint(CheckPt sav);
01387     void restore(CheckPt sav);
01388     void restoreMsg(CheckPt sav, const char *msg);
01389     void prStack(void);
01390
01391     void pushNode(int n);
01392     void pushEdge(int e);
01393
01394   private:
01395     void restoreNode(int spr);
01396     void restoreEdge(int spr);
01397 };
01398
01399 //============================================================
01400 // end of namespace Grcc
01401 #ifdef GRCC_NAMESPACE
01402 }
01403 #endif
```

4.61 grccparam.h

```
00001 #ifdef GRCC_PARAM_H
00002 #else
00003 #define GRCC_PARAM_H
00004
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 2023-2026 T. Kaneko
00008  *   When using this file you are requested to refer to the publication
00009  *   Comput.Phys.Commun. 92 (1995) 127-152
00010  *
00011  *   This file is part of FORM.
00012  *
00013  *   FORM is free software: you can redistribute it and/or modify it under the
00014  *   terms of the GNU General Public License as published by the Free Software
00015  *   Foundation, either version 3 of the License, or (at your option) any later
00016  *   version.
00017  *
00018  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00019  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00020  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00021  *   details.
```

```
00022  *
00023  *   You should have received a copy of the GNU General Public License along
00024  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00025  */
00026 /* #] License : */
00027
00028 /*============================================================
00029  * Parameters
00030  */
00031 /*
00032 #define GRCC_NAMESPACE
00033 */
00034
00035 #define GRCC_MAXNCPLG         4
00036 #define GRCC_MAXLEGS         10
00037 #define GRCC_MAXMPARTICLES   50
00038 #define GRCC_MAXMINTERACT   500
00039 #define GRCC_MAXSUBPROCS    500
00040 #define GRCC_MAXNODES        20
00041 #define GRCC_MAXEDGES       100
00042 #define GRCC_MAXNSTACK       50
00043 #define GRCC_MAXESTACK       50
00044 #define GRCC_MAXGROUP      1000
00045 #define GRCC_MAXPSLIST      500
00046
00047 #define GRCC_MAXMPARTICLES2  (2*GRCC_MAXMPARTICLES-1)
00048 #define GRCC_MAXFLINES       GRCC_MAXEDGES
00049
00050 #define GRCC_FRACERROR       (1.0e-10)
00051
00052 /* Standard and error message destination */
00053 #define GRCC_Stdout          stdout
00054 #define GRCC_Stderr          stdout
00055
00056 #ifndef NOFORM
00057     #define GRCC_ABORT()      Terminate(-1);
00058 #else
00059     #define GRCC_ABORT()      exit(1)
00060 #endif
00061
00062 /*------------------------------------------------------------
00063  * Constants
00064  */
00065
00066 /* model definition */
00067 #define GRCC_DEFBYNAME   1    /* define interactions by particle name */
00068 #define GRCC_DEFBYCODE   2    /* define interactions by particle code */
00069
00070 /* types of particles */
00071 #define GRCC_PT_Undef      0
00072 #define GRCC_PT_Scalar     1
00073 #define GRCC_PT_Dirac      2
00074 #define GRCC_PT_Majorana   3
00075 #define GRCC_PT_Vector     4
00076 #define GRCC_PT_Ghost      5
00077 #define GRCC_PT_Size       6
00078
00079 #define GRCC_PTS_Undef      "undef"
00080 #define GRCC_PTS_Scalar     "scalar"
00081 #define GRCC_PTS_Dirac      "dirac"
00082 #define GRCC_PTS_Majorana   "majorana"
00083 #define GRCC_PTS_Vector     "vector"
00084 #define GRCC_PTS_Ghost      "ghost"
00085
00086 #define GRCC_PT_Table  { GRCC_PTS_Undef, GRCC_PTS_Scalar, GRCC_PTS_Dirac, GRCC_PTS_Majorana,
      GRCC_PTS_Vector, GRCC_PTS_Ghost}
00087
00088 #define GRCC_PT_GTable { "Undef", "Scalar", "Fermion", "Majorana", "Vector", "Ghost"}
00089
00090 /* for mgraph generation */
00091 #define GRCC_AT_Vertex     0
00092 #define GRCC_AT_External  -1
00093 #define GRCC_AT_Initial   -2
00094 #define GRCC_AT_Final     -3
00095
00096 #define GRCC_AT_NdStr(x)   ((x)>=GRCC_AT_Vertex   ?"Vertex":       \
00097                             ((x)==GRCC_AT_External ?"External":     \
00098                             ((x)==GRCC_AT_Final    ?"Final":        \
00099                             ((x)==GRCC_AT_Initial  ?"Initial":"Undef"))))
00100
00101 /* graph generation */
00102 #define GRCC_MGraph   0       /* mgraph (topology) generation */
00103 #define GRCC_AGraph   1       /* agraph generation */
00104
00105 /* options for graph grenerations */
00106 #define GRCC_OPT_Step            0
00107 #define GRCC_OPT_Outgrf          1
```

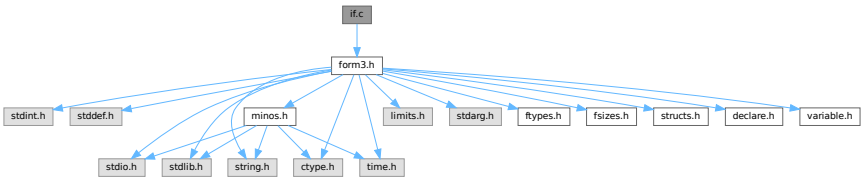
```
00108 #define GRCC_OPT_Outgrp          2
00109 #define GRCC_OPT_1PI             3
00110 #define GRCC_OPT_NoSelfLoop      4
00111 #define GRCC_OPT_NoTadpole       5
00112 #define GRCC_OPT_No1PtBlock      6
00113 #define GRCC_OPT_No2PtL1PI       7
00114 #define GRCC_OPT_NoExtSelf       8
00115 #define GRCC_OPT_NoAdj2PtV       9
00116 #define GRCC_OPT_Block          10
00117 #define GRCC_OPT_NoMultiEdge    11
00118 #define GRCC_OPT_SymmInitial    12
00119 #define GRCC_OPT_SymmFinal      13
00120 #define GRCC_OPT_Size           14
00121
00122 typedef unsigned long ULong;
00123
00124 typedef struct {
00125     const char *name;
00126     const char *mean;
00127     int         defaultv;
00128     int         DUMMYPADDING;
00129 } OptDef;
00130
00131 typedef struct {
00132     const char *name;
00133     const char *cname;
00134     const char *mean;
00135 } OptQGDef;
00136
00137 typedef struct {
00138     const char *name;
00139     int         index;
00140     int         sign;
00141 } OptQGRef;
00142
00143 /*-------------------------------------------------------------*
00144  * QGRAF options
00145  */
00146 #define GRCC_QGRAF_OPT_ONEPI       0
00147 #define GRCC_QGRAF_OPT_ONSHELL     1
00148 #define GRCC_QGRAF_OPT_NOSIGMA     2
00149 #define GRCC_QGRAF_OPT_NOSNAIL     3
00150 #define GRCC_QGRAF_OPT_NOTADPOLE   4
00151 #define GRCC_QGRAF_OPT_SIMPLE      5
00152 #define GRCC_QGRAF_OPT_BIPART      6
00153 #define GRCC_QGRAF_OPT_CYCLI       7
00154 #define GRCC_QGRAF_OPT_FLOOP       8
00155 //TODO what does this do? Does it work?
00156 //#define GRCC_QGRAF_OPT_TOPOL       9
00157
00158 #ifdef GRCC_QGRAF_OPT_TOPOL
00159 #define GRCC_QGRAF_OPT_Size       10
00160 #else
00161 #define GRCC_QGRAF_OPT_Size        9
00162 #endif
00163
00164 /*-------------------------------------------------------------*
00165  * Conversion of
00166  *     eind : index (ed) of EGraph.edges[ed]
00167  *     sind : value (v)  of EGraph.nodes[nd]->edges[j])
00168  */
00169 #define   I2Vedge(eind, sign)    (((sign<=0)?(-1):(1))*((eind)+1))
00170 #define   V2Iedge(sind)          (abs(sind)-1)
00171 #define   V2Ileg(sind)           ((sind)<0 ? 0 : 1)
00172 /*-------------------------------------------------------------*
00173  * data types for model definition
00174  */
00175 typedef struct {
00176     const char *name;              /* name of the model */
00177           int   defpart;           /* particle is defined by name or code */
00178                                    /* GRCC_DEFBYNAME or GRCC_DEFBYCODE */
00179           int   ncouple;           /* No. of coupling constants */
00180     const char *cnamlist[GRCC_MAXNCPLG]; /* Names of coupling constants */
00181 } MInput;
00182
00183 typedef struct {
00184     const char *name;              /* name of particle */
00185     const char *aname;             /* name of anti-particle */
00186           int   pcode;             /* code of particle */
00187           int   acode;             /* code of anti-particle */
00188     const char *ptypen;            /* type : 'GRCC_PTS_Scalar' etc */
00189           int   ptypec;            /* type : 'GRCC_PT_Scalar' etc */
00190           int   extonly;           /* only as external particle */
00191 } PInput;
00192
00193 typedef struct {
00194     const char *name;                /* name of interaction */
```

```
00195           int   icode;               /* code of interaction */
00196           int   nplistn;             /* the number of legs */
00197     const char *plistn[GRCC_MAXLEGS];    /* list of particle names */
00198           int   plistc[GRCC_MAXLEGS];    /* list of particle codes */
00199           int   cvallist[GRCC_MAXNCPLG]; /* coupling constants */
00200 } IInput;
00201
00202 /*---------------------------------------------------------------
00203  * data type for node class input
00204  */
00205 typedef struct {
00206     /* used as input to MGraph() and SProcess() */
00207     int cldeg;
00208     int clnum;
00209     int cltyp;
00210     int cmind;
00211     int cmaxd;
00212
00213     /* used as input to SProcess() */
00214     int ptcl;
00215     int cple;
00216 } NCInput;
00217
00218 /*---------------------------------------------------------------
00219  * data types for process definition
00220  */
00221 typedef struct {
00222     long          ninitl;              /* the number of initial particles */
00223     const char   *initln[GRCC_MAXNODES];    /* list of initial particles (name) */
00224     int           initlc[GRCC_MAXNODES];    /* list of final particles (code) */
00225     long          nfinal;              /* the number of final particles */
00226     const char   *finaln[GRCC_MAXNODES];    /* list of final particles (name) */
00227     int           finalc[GRCC_MAXNODES];    /* list of final particles (code) */
00228     int           coupl[GRCC_MAXNCPLG];     /* list of orders of c. consts */
00229 } FGInput;
00230
00231 /* class of node : input for SProcess */
00232 typedef struct {
00233     int         cdeg;                /* degree of each node */
00234     int         ctyp;                /* type : GRCC_AT_xxx */
00235     int         cnum;                /* the number of nodes in the class */
00236     int         cple;                /* total order of c.c. of each node */
00237                                      /* = 0 for external particle */
00238     const char *pname;               /* particle name defined in the model */
00239                                      /* = NULL for vertex */
00240     long        pcode;               /* particle code defined in the model */
00241                                      /* = 0 for vertex */
00242                                      /* long = int + padding */
00243 } PGInput;
00244
00245 /*---------------------------------------------------------------
00246  * minimum input data to EGraph
00247  */
00248 typedef struct {
00249     int    extloop;     /* external/loop */
00250     int    intrct;      /* particl/interaction */
00251 } DNode;
00252
00253 typedef int  DEdge[2];  /* {node0, node1} */
00254
00255 typedef struct {
00256     int    nnodes;
00257     int    nedges;
00258     DNode *nodes;
00259     DEdge *edges;
00260 } DGraph;
00261
00262 /*---------------------------------------------------------------
00263  */
00264 #endif
```

4.62 if.c File Reference

```
#include "form3.h"
```

Include dependency graph for if.c:



Functions

- WORD GetIfDollarNum (WORD *ifp, WORD *ifstop)
- int FindVar (WORD *v, WORD *term)
- int DoIfStatement (PHEAD WORD *ifcode, WORD *term)
- WORD HowMany (PHEAD WORD *ifcode, WORD *term)
- void DoubleIfBuffers (void)
- int DoSwitch (PHEAD WORD *term, WORD *lhs)
- int DoEndSwitch (PHEAD WORD *term, WORD *lhs)
- SWITCHTABLE * FindCase (WORD nswitch, WORD ncase)
- int DoubleSwitchBuffers (void)
- void SwitchSplitMergeRec (SWITCHTABLE *array, WORD num, SWITCHTABLE *auxarray)
- void SwitchSplitMerge (SWITCHTABLE *array, WORD num)

4.62.1 Detailed Description

Routines for the dealing with if statements.

Definition in file if.c.

4.62.2 Function Documentation

4.62.2.1 GetIfDollarNum()

```
WORD GetIfDollarNum (
            WORD * ifp,
            WORD * ifstop )
```

Definition at line 106 of file if.c.

4.62.2.2 FindVar()

```
int FindVar (
            WORD * v,
            WORD * term )
```

Definition at line 173 of file if.c.

4.62.2.3 DoIfStatement()

```
int DoIfStatement (
            PHEAD WORD * ifcode,
            WORD * term )
```

Definition at line 280 of file if.c.

4.62.2.4 HowMany()

```
WORD HowMany (
            PHEAD WORD * ifcode,
            WORD * term )
```

Definition at line 856 of file if.c.

4.62.2.5 DoubleIfBuffers()

```
void DoubleIfBuffers (
            void  )
```

Definition at line 1049 of file if.c.

4.62.2.6 DoSwitch()

```
int DoSwitch (
            PHEAD WORD * term,
            WORD * lhs )
```

Definition at line 1086 of file if.c.

4.62.2.7 DoEndSwitch()

```
int DoEndSwitch (
            PHEAD WORD * term,
            WORD * lhs )
```

Definition at line 1102 of file if.c.

4.62.2.8 FindCase()

```
SWITCHTABLE * FindCase (
            WORD nswitch,
            WORD ncase )
```

Definition at line 1113 of file if.c.

4.62.2.9 DoubleSwitchBuffers()

```
int DoubleSwitchBuffers (
            void  )
```

Definition at line 1151 of file if.c.

4.62.2.10 SwitchSplitMergeRec()

```
void SwitchSplitMergeRec (
            SWITCHTABLE * array,
            WORD num,
            SWITCHTABLE * auxarray )
```

Definition at line 1200 of file if.c.

4.62.2.11 SwitchSplitMerge()

```
void SwitchSplitMerge (
            SWITCHTABLE * array,
            WORD num )
```

Definition at line 1229 of file if.c.

4.63 if.c

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
00026  *
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
00031 /*
00032     #[ Includes : if.c
00033 */
00034
00035 #include "form3.h"
00036
00037 /*
00038     #] Includes :
00039     #[ If statement :
00040         #[ Syntax :
```

```
00041
00042         The `if' is a conglomerate of statements: if,else,endif
00043
00044         The if consists in principle of:
00045
00046         if ( number );
00047             statements
00048         else;
00049             statements
00050         endif;
00051
00052         The first set is taken when number != 0.
00053         The else is not mandatory.
00054         TRUE = 1 and FALSE = 0
00055
00056         The number can be built up via a logical expression:
00057
00058                     expr1 condition expr2
00059
00060         each expression can be a subexpression again. It has to be
00061         enclosed in parentheses in that case.
00062         Conditions are:
00063             >, >=, <, <=, ==, !=, ||, &&
00064
00065         When Expressions are chained evaluation is from left to right,
00066         independent of whether this indicates nonsense.
00067         if ( a || b || c || d ); is a perfectly normal statement.
00068         if ( a >= b || c == d ); would be messed up. This should be:
00069         if ( ( a >= b ) || ( c == d ) );
00070
00071         The building blocks of the Expressions are:
00072
00073             Match(option,pattern)   The number of times pattern fits in term_
00074             Count(....)             The count value of term_
00075             Coeff[icient]           The coefficient of term_
00076             FindLoop(options)       Are there loops (as in ReplaceLoop).
00077
00078         Implementation for internal notation:
00079
00080         TYPEIF,length,gotolevel(if fail),EXPRTYPE,length,......
00081
00082         EXPRTYPE can be:
00083             SHORTNUMBER         ->,4,sign,size
00084             LONGNUMBER          ->,|ncoef+2|,ncoef,numer,denom
00085             MATCH               ->,patternsiz+3,keyword,pattern
00086             MULTIPLEOF          ->,3,thenumber
00087             COUNT               ->,countsiz+2,countinfo
00088             TYPEFINDLOOP        ->,7 (findloop info)
00089             COEFFICIENT         ->,2
00090             IFDOLLAR            ->,3,dollarnumber
00091             SUBEXPR             ->,size,dummy,size1,EXPRTYPE,length,...
00092                                   ,2,condition1,size2,...
00093                                 This is like functions.
00094
00095         Note that there must be a restriction to the number of nestings
00096         of parentheses in an if statement. It has been set to 10.
00097
00098         The syntax of match corresponds to the syntax of the left side
00099         of an id statement. The only difference is the keyword
00100         MATCH vs TYPEIDNEW.
00101
00102         #] Syntax :
00103         #[ GetIfDollarNum :
00104 */
00105
00106 WORD GetIfDollarNum(WORD *ifp, WORD *ifstop)
00107 {
00108     DOLLARS d;
00109     WORD num, *w;
00110     if ( ifp[2] < 0 ) { return(-ifp[2]-1); }
00111     d = Dollars+ifp[2];
00112     if ( ifp+3 < ifstop && ifp[3] == IFDOLLAREXTRA ) {
00113         if ( d->nfactors == 0 ) {
00114             MLOCK(ErrorMessageLock);
00115             MesPrint("Attempt to use a factor of an unfactored $-variable");
00116             MUNLOCK(ErrorMessageLock);
00117             Terminate(-1);
00118         }
00119         num = GetIfDollarNum(ifp+3,ifstop);
00120         if ( num > d->nfactors ) {
00121             MLOCK(ErrorMessageLock);
00122             MesPrint("Dollar factor number %s out of range",num);
00123             MUNLOCK(ErrorMessageLock);
00124             Terminate(-1);
00125         }
00126         if ( num == 0 ) {
00127             return(d->nfactors);
```

```
00128         }
00129         w = d->factors[num-1].where;
00130         if ( w == 0 ) return(d->factors[num].value);
00131 getnumber:;
00132         if ( *w == 0 ) return(0);
00133         if ( *w == 4 && w[3] == 3 && w[2] == 1 && w[1] < MAXPOSITIVE && w[4] == 0 ) {
00134             return(w[1]);
00135         }
00136         if ( ( w[w[0]] != 0 ) || ( ABS(w[w[0]-1]) != w[0]-1 ) ) {
00137             MLOCK(ErrorMessageLock);
00138             MesPrint("Dollar factor number expected but found expression");
00139             MUNLOCK(ErrorMessageLock);
00140             Terminate(-1);
00141         }
00142         else {
00143             MLOCK(ErrorMessageLock);
00144             MesPrint("Dollar factor number out of range");
00145             MUNLOCK(ErrorMessageLock);
00146             Terminate(-1);
00147         }
00148         return(0);
00149     }
00150 /*
00151     Now we have just a dollar and should evaluate that into a short number
00152 */
00153     if ( d->type == DOLZERO ) {
00154         return(0);
00155     }
00156     else if ( d->type == DOLNUMBER || d->type == DOLTERMS ) {
00157         w = d->where; goto getnumber;
00158     }
00159     else {
00160         MLOCK(ErrorMessageLock);
00161         MesPrint("Dollar factor number is wrong type");
00162         MUNLOCK(ErrorMessageLock);
00163         Terminate(-1);
00164         return(0);
00165     }
00166 }
00167
00168 /*
00169         #] GetIfDollarNum :
00170         #[ FindVar :
00171 */
00172
00173 int FindVar(WORD *v, WORD *term)
00174 {
00175     WORD *t, *tstop, *m, *mstop, *f, *fstop, *a, *astop;
00176     GETSTOP(term,tstop);
00177     t = term+1;
00178     while ( t < tstop ) {
00179         if ( *v == *t && *v < FUNCTION ) {  /* VECTOR, INDEX, SYMBOL, DOTPRODUCT */
00180             switch ( *v ) {
00181                 case SYMBOL:
00182                     m = t+2; mstop = t+t[1];
00183                     while ( m < mstop ) {
00184                         if ( *m == v[1] ) return(1);
00185                         m += 2;
00186                     }
00187                     break;
00188                 case INDEX:
00189                 case VECTOR:
00190 InVe:
00191                     m = t+2; mstop = t+t[1];
00192                     while ( m < mstop ) {
00193                         if ( *m == v[1] ) return(1);
00194                         m++;
00195                     }
00196                     break;
00197                 case DOTPRODUCT:
00198                     m = t+2; mstop = t+t[1];
00199                     while ( m < mstop ) {
00200                         if ( *m == v[1] && m[1] == v[2] ) return(1);
00201                         if ( *m == v[2] && m[1] == v[1] ) return(1);
00202                         m += 3;
00203                     }
00204                     break;
00205             }
00206         }
00207         else if ( *v == VECTOR && *t == INDEX ) goto InVe;
00208         else if ( *v == INDEX && *t == VECTOR ) goto InVe;
00209         else if ( ( *v == VECTOR || *v == INDEX ) && *t == DOTPRODUCT ) {
00210             m = t+2; mstop = t+t[1];
00211             while ( m < mstop ) {
00212                 if ( v[1] == m[0] || v[1] == m[1] ) return(1);
00213                 m += 3;
00214             }
```

```
00215         }
00216         else if ( *t >= FUNCTION ) {
00217             if ( *v == FUNCTION && v[1] == *t ) return(1);
00218             if ( functions[*t-FUNCTION].spec > 0 ) {
00219               if ( *v == VECTOR || *v == INDEX ) { /* we need to check arguments */
00220                 int i;
00221                 for ( i = FUNHEAD; i < t[1]; i++ ) {
00222                     if ( v[1] == t[i] ) return(1);
00223                 }
00224               }
00225             }
00226             else {
00227               fstop = t + t[1]; f = t + FUNHEAD;
00228               while ( f < fstop ) { /* Do the arguments one by one */
00229                 if ( *f <= 0 ) {
00230                     switch ( *f ) {
00231                         case -SYMBOL:
00232                             if ( *v == SYMBOL && v[1] == f[1] ) return(1);
00233                             f += 2;
00234                             break;
00235                         case -VECTOR:
00236                         case -MINVECTOR:
00237                         case -INDEX:
00238                             if ( ( *v == VECTOR || *v == INDEX )
00239                              && ( v[1] == f[1] ) ) return(1);
00240                             f += 2;
00241                             break;
00242                         case -SNUMBER:
00243                             f += 2;
00244                             break;
00245                         default:
00246                             if ( *v == FUNCTION && v[1] == -*f && *f <= -FUNCTION ) return(1);
00247                             if ( *f <= -FUNCTION ) f++;
00248                             else f += 2;
00249                             break;
00250                     }
00251                 }
00252                 else {
00253                     a = f + ARGHEAD; astop = f + *f;
00254                     while ( a < astop ) {
00255                         if ( FindVar(v,a) == 1 ) return(1);
00256                         a += *a;
00257                     }
00258                     f = astop;
00259                 }
00260               }
00261             }
00262         }
00263         t += t[1];
00264     }
00265     return(0);
00266 }
00267
00268 /*
00269         #] FindVar :
00270         #[ DoIfStatement :              WORD DoIfStatement(PHEAD ifcode,term)
00271
00272         The execution time part of the if-statement.
00273         The arguments are a pointer to the TYPEIF and a pointer to the term.
00274         The answer is either 1 (success) or 0 (fail).
00275         The calling routine can figure out where to go in case of failure
00276         by picking up gotolevel.
00277         Note that the whole setup asks for recursions.
00278 */
00279
00280 int DoIfStatement(PHEAD WORD *ifcode, WORD *term)
00281 {
00282     GETBIDENTITY
00283     WORD *ifstop, *ifp;
00284     UWORD *coef1 = 0, *coef2, *coef3, *cc;
00285     WORD ncoef1, ncoef2, ncoef3, i = 0, first, *r, acoef, ismul1, ismul2, j;
00286     UWORD *Spac1, *Spac2;
00287     ifstop = ifcode + ifcode[1];
00288     ifp = ifcode + 3;
00289     if ( ifp >= ifstop ) return(1);
00290     if ( ( ifp + ifp[1] ) >= ifstop ) {
00291         switch ( *ifp ) {
00292             case LONGNUMBER:
00293                 if ( ifp[2] ) return(1);
00294                 else return(0);
00295             case MATCH:
00296             case TYPEIF:
00297                 if ( HowMany(BHEAD ifp,term) ) return(1);
00298                 else return(0);
00299             case TYPEFINDLOOP:
00300                 if ( Lus(term,ifp[3],ifp[4],ifp[5],ifp[6],ifp[2]) ) return(1);
00301                 else return(0);
```

```
00302             case TYPECOUNT:
00303                 if ( CountDo(term,ifp) ) return(1);
00304                 else return(0);
00305             case COEFFI:
00306             case MULTIPLEOF:
00307                 return(1);
00308             case IFDOLLAR:
00309                 {
00310                     DOLLARS d = Dollars + ifp[2];
00311 #ifdef WITHPTHREADS
00312                     int nummodopt, dtype = -1;
00313                     if ( AS.MultiThreaded ) {
00314                         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
00315                             if ( ifp[2] == ModOptdollars[nummodopt].number ) break;
00316                         }
00317                         if ( nummodopt < NumModOptdollars ) {
00318                             dtype = ModOptdollars[nummodopt].type;
00319                             if ( DollarLocalCopy(dtype) ) {
00320                                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
00321                             }
00322                         }
00323                     }
00324                     dtype = d->type;
00325 #else
00326                     int dtype = d->type;  /* We use dtype to make the operation atomic */
00327 #endif
00328                     if ( dtype == DOLZERO ) return(0);
00329                     if ( dtype == DOLUNDEFINED ) {
00330                         if ( AC.UnsureDollarMode == 0 ) {
00331                             MesPrint("$%s is undefined",AC.dollarnames->namebuffer+d->name);
00332                             Terminate(-1);
00333                         }
00334                     }
00335                 }
00336                 return(1);
00337             case IFEXPRESSION:
00338                 r = ifp+2; j = ifp[1] - 2;
00339                 while ( --j >= 0 ) {
00340                     if ( *r == AR.CurExpr ) return(1);
00341                     r++;
00342                 }
00343                 return(0);
00344             case IFISFACTORIZED:
00345                 r = ifp+2; j = ifp[1] - 2;
00346                 if ( j == 0 ) {
00347                     if ( ( Expressions[AR.CurExpr].vflags & ISFACTORIZED ) != 0 )
00348                         return(1);
00349                     else
00350                         return(0);
00351                 }
00352                 while ( --j >= 0 ) {
00353                     if ( ( Expressions[*r].vflags & ISFACTORIZED ) == 0 ) return(0);
00354                     r++;
00355                 }
00356                 return(1);
00357             case IFOCCURS:
00358                 {
00359                     WORD *OccStop = ifp + ifp[1];
00360                     ifp += 2;
00361                     while ( ifp < OccStop ) {
00362                         if ( FindVar(ifp,term) == 1 ) return(1);
00363                         if ( *ifp == DOTPRODUCT ) ifp += 3;
00364                         else ifp += 2;
00365                     }
00366                 }
00367                 return(0);
00368             case IFUSERFLAG:
00369                 if ( ( Expressions[AR.CurExpr].uflags & (1 « ifp[2]) ) != 0 )
00370                     return(1);
00371                 return(0);
00372             default:
00373 /*
00374                 Now we have a subexpression. Test first for one with a single item.
00375 */
00376                 if ( ifp[3] == ( ifp[1] + 3 ) ) return(DoIfStatement(BHEAD ifp,term));
00377                 ifstop = ifp + ifp[1];
00378                 ifp += 3;
00379                 break;
00380         }
00381     }
00382 /*
00383     Here is the composite condition.
00384 */
00385     coef3 = NumberMalloc("DoIfStatement");
00386     Spac1 = NumberMalloc("DoIfStatement");
00387     Spac2 = (UWORD *)(TermMalloc("DoIfStatement"));
00388     ncoef1 = 0; first = 1; ismul1 = 0;
```

```
00389     do {
00390         if ( !first ) {
00391             ifp += 2;
00392             if ( ifp[-2] == ORCOND && ncoef1 ) {
00393                 coef1 = Spac1;
00394                 ncoef1 = 1; coef1[0] = coef1[1] = 1;
00395                 goto SkipCond;
00396             }
00397             if ( ifp[-2] == ANDCOND && !ncoef1 ) goto SkipCond;
00398         }
00399         coef2 = Spac2;
00400         ncoef2 = 1;
00401         ismul2 = 0;
00402         switch ( *ifp ) {
00403             case LONGNUMBER:
00404                 ncoef2 = ifp[2];
00405                 j = 2*(ABS(ncoef2));
00406                 cc = (UWORD *)(ifp + 3);
00407                 for ( i = 0; i < j; i++ ) coef2[i] = cc[i];
00408                 break;
00409 #ifdef WITHFLOAT
00410 /* UNFINISHED_FEATURE_EXCL_START */
00411             case IFFLOATNUMBER:
00412 /*
00413                 The sloppy solution is: Convert to rational.
00414                 This way we can write it over coef2,ncoef2
00415 */
00416                 ncoef2 = FloatFunToRat(BHEAD coef2,ifp);
00417                 break;
00418 /* UNFINISHED_FEATURE_EXCL_STOP */
00419 #endif
00420             case MATCH:
00421             case TYPEIF:
00422                 coef2[0] = HowMany(BHEAD ifp,term);
00423                 coef2[1] = 1;
00424                 if ( coef2[0] == 0 ) ncoef2 = 0;
00425                 break;
00426             case TYPECOUNT:
00427                 acoef = CountDo(term,ifp);
00428                 coef2[0] = ABS(acoef);
00429                 coef2[1] = 1;
00430                 if ( acoef == 0 ) ncoef2 = 0;
00431                 else if ( acoef < 0 ) ncoef2 = -1;
00432                 break;
00433             case TYPEFINDLOOP:
00434                 acoef = Lus(term,ifp[3],ifp[4],ifp[5],ifp[6],ifp[2]);
00435                 coef2[0] = ABS(acoef);
00436                 coef2[1] = 1;
00437                 if ( acoef == 0 ) ncoef2 = 0;
00438                 else if ( acoef < 0 ) ncoef2 = -1;
00439                 break;
00440             case COEFFI:
00441                 r = term + *term;
00442                 ncoef2 = r[-1];
00443                 i = ABS(ncoef2);
00444                 cc = (UWORD *)(r - i);
00445                 if ( ncoef2 < 0 ) ncoef2 = (ncoef2+1)>>1;
00446                 else ncoef2 = (ncoef2-1)>>1;
00447                 i--; for ( j = 0; j < i; j++ ) coef2[j] = cc[j];
00448                 break;
00449             case SUBEXPR:
00450                 ncoef2 = coef2[0] = DoIfStatement(BHEAD ifp,term);
00451                 coef2[1] = 1;
00452                 break;
00453             case MULTIPLEOF:
00454                 ncoef2 = 1;
00455                 coef2[0] = ifp[2];
00456                 coef2[1] = 1;
00457                 ismul2 = 1;
00458                 break;
00459             case IFDOLLAREXTRA:
00460                 break;
00461             case IFDOLLAR:
00462                 {
00463 /*
00464                     We need to abstract a long rational in coef2
00465                     with length ncoef2. What if that cannot be done?
00466 */
00467                     DOLLARS d = Dollars + ifp[2];
00468 #ifdef WITHPTHREADS
00469                     int nummodopt, dtype = -1;
00470                     if ( AS.MultiThreaded ) {
00471                         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
00472                             if ( ifp[2] == ModOptdollars[nummodopt].number ) break;
00473                         }
00474                         if ( nummodopt < NumModOptdollars ) {
00475                             dtype = ModOptdollars[nummodopt].type;
```

```
00476                           if ( DollarLocalCopy(dtype) ) {
00477                               d = ModOptdollars[nummodopt].dstruct+AT.identity;
00478                           }
00479                           else {
00480                               LOCK(d->pthreadslock);
00481                           }
00482                       }
00483                   }
00484 #endif
00485 /*
00486                   We have to pick up the IFDOLLAREXTRA pieces for [1], [$y] etc.
00487 */
00488                   if ( ifp+3 < ifstop && ifp[3] == IFDOLLAREXTRA ) {
00489                       if ( d->nfactors == 0 ) {
00490                           MLOCK(ErrorMessageLock);
00491                           MesPrint("Attempt to use a factor of an unfactored $-variable");
00492                           MUNLOCK(ErrorMessageLock);
00493                           Terminate(-1);
00494                       } {
00495                       WORD num = GetIfDollarNum(ifp+3,ifstop);
00496                       WORD *w;
00497                       while ( ifp+3 < ifstop && ifp[3] == IFDOLLAREXTRA ) ifp += 3;
00498                       if ( num > d->nfactors ) {
00499                           MLOCK(ErrorMessageLock);
00500                           MesPrint("Dollar factor number %s out of range",num);
00501                           MUNLOCK(ErrorMessageLock);
00502                           Terminate(-1);
00503                       }
00504                       if ( num == 0 ) {
00505                           ncoef2 = 1; coef2[0] = d->nfactors; coef2[1] = 1;
00506                           break;
00507                       }
00508                       w = d->factors[num-1].where;
00509                       if ( w == 0 ) {
00510                           if ( d->factors[num-1].value < 0 ) {
00511                               ncoef2 = -1; coef2[0] = -d->factors[num-1].value; coef2[1] = 1;
00512                           }
00513                           else {
00514                               ncoef2 = 1; coef2[0] = d->factors[num-1].value; coef2[1] = 1;
00515                           }
00516                           break;
00517                       }
00518                       if ( w[*w] == 0 ) {
00519                           r = w + *w - 1;
00520                           i = ABS(*r);
00521                           if ( i == ( *w-1 ) ) {
00522                               ncoef2 = (i-1)/2;
00523                               if ( *r < 0 ) ncoef2 = -ncoef2;
00524                               i--; cc = coef2; r = w + 1;
00525                               while ( --i >= 0 ) *cc++ = (UWORD)(*r++);
00526                               break;
00527                           }
00528                       }
00529                       goto generic;
00530                       }
00531                   }
00532                   else {
00533                     switch ( d->type ) {
00534                       case DOLUNDEFINED:
00535                           if ( AC.UnsureDollarMode == 0 ) {
00536 #ifdef WITHPTHREADS
00537                               if ( dtype > 0 && ! DollarLocalCopy(dtype) ) {
00538                                   UNLOCK(d->pthreadslock);
00539                               }
00540 #endif
00541                               MLOCK(ErrorMessageLock);
00542                               MesPrint("$%s is undefined",AC.dollarnames->namebuffer+d->name);
00543                               MUNLOCK(ErrorMessageLock);
00544                               Terminate(-1);
00545                           }
00546                           ncoef2 = 0; coef2[0] = 0; coef2[1] = 1;
00547                           break;
00548                       case DOLZERO:
00549                           ncoef2 = coef2[0] = 0; coef2[1] = 1;
00550                           break;
00551                       case DOLSUBTERM:
00552                           if ( d->where[0] != INDEX || d->where[1] != 3
00553                           || d->where[2] < 0 || d->where[2] >= AM.OffsetIndex ) {
00554                               if ( AC.UnsureDollarMode == 0 ) {
00555 #ifdef WITHPTHREADS
00556                                   if ( dtype > 0 && ! DollarLocalCopy(dtype) ) {
00557                                       UNLOCK(d->pthreadslock);
00558                                   }
00559 #endif
00560                                   MLOCK(ErrorMessageLock);
00561                                   MesPrint("$%s is of wrong
      type",AC.dollarnames->namebuffer+d->name);
```

```
00562                                   MUNLOCK(ErrorMessageLock);
00563                                   Terminate(-1);
00564                               }
00565                               ncoef2 = 0; coef2[0] = 0; coef2[1] = 1;
00566                               break;
00567                           }
00568                           d->index = d->where[2];
00569                           /* fall through */
00570                       case DOLINDEX:
00571                           if ( d->index == 0 ) {
00572                               ncoef2 = coef2[0] = 0; coef2[1] = 1;
00573                           }
00574                           else if ( d->index > 0 && d->index < AM.OffsetIndex ) {
00575                               ncoef2 = 1; coef2[0] = d->index; coef2[1] = 1;
00576                           }
00577                           else if ( AC.UnsureDollarMode == 0 ) {
00578 #ifdef WITHPTHREADS
00579                               if ( dtype > 0 && ! DollarLocalCopy(dtype) ) {
00580                                   UNLOCK(d->pthreadslock);
00581                               }
00582 #endif
00583                               MLOCK(ErrorMessageLock);
00584                               MesPrint("$%s is of wrong type",AC.dollarnames->namebuffer+d->name);
00585                               MUNLOCK(ErrorMessageLock);
00586                               Terminate(-1);
00587                           }
00588                           ncoef2 = coef2[0] = 0; coef2[1] = 1;
00589                           break;
00590                       case DOLWILDARGS:
00591                           if ( d->where[0] <= -FUNCTION ||
00592                           ( d->where[0] < 0 && d->where[2] != 0 )
00593                           || ( d->where[0] > 0 && d->where[d->where[0]] != 0 )
00594                           ) {
00595                               if ( AC.UnsureDollarMode == 0 ) {
00596 #ifdef WITHPTHREADS
00597                                   if ( dtype > 0 && ! DollarLocalCopy(dtype) ) {
00598                                       UNLOCK(d->pthreadslock);
00599                                   }
00600 #endif
00601                                   MLOCK(ErrorMessageLock);
00602                                   MesPrint("$%s is of wrong
      type",AC.dollarnames->namebuffer+d->name);
00603                                   MUNLOCK(ErrorMessageLock);
00604                                   Terminate(-1);
00605                               }
00606                               ncoef2 = coef2[0] = 0; coef2[1] = 1;
00607                               break;
00608                           }
00609                           /* fall through */
00610                       case DOLARGUMENT:
00611                           if ( d->where[0] == -SNUMBER ) {
00612                               if ( d->where[1] == 0 ) {
00613                                   ncoef2 = coef2[0] = 0;
00614                               }
00615                               else if ( d->where[1] < 0 ) {
00616                                   ncoef2 = -1;
00617                                   coef2[0] = -d->where[1];
00618                               }
00619                               else {
00620                                   ncoef2 = 1;
00621                                   coef2[0] = d->where[1];
00622                               }
00623                               coef2[1] = 1;
00624                           }
00625                           else if ( d->where[0] == -INDEX
00626                           && d->where[1] >= 0 && d->where[1] < AM.OffsetIndex ) {
00627                               if ( d->where[1] == 0 ) {
00628                                   ncoef2 = coef2[0] = 0; coef2[1] = 1;
00629                               }
00630                               else {
00631                                   ncoef2 = 1; coef2[0] = d->where[1];
00632                                   coef2[1] = 1;
00633                               }
00634                           }
00635                           else if ( d->where[0] > 0
00636                           && d->where[ARGHEAD] == (d->where[0]-ARGHEAD)
00637                           && ABS(d->where[d->where[0]-1]) ==
00638                                       (d->where[0] - ARGHEAD-1) ) {
00639                               i = d->where[d->where[0]-1];
00640                               ncoef2 = (ABS(i)-1)/2;
00641                               if ( i < 0 ) { ncoef2 = -ncoef2; i = -i; }
00642                               i--; cc = coef2; r = d->where + ARGHEAD+1;
00643                               while ( --i >= 0 ) *cc++ = (UWORD)(*r++);
00644                           }
00645                           else {
00646                               if ( AC.UnsureDollarMode == 0 ) {
00647 #ifdef WITHPTHREADS
```

```
00648                                   if ( dtype > 0 && ! DollarLocalCopy(dtype) ) {
00649                                       UNLOCK(d->pthreadslock);
00650                                   }
00651 #endif
00652                                   MLOCK(ErrorMessageLock);
00653                                   MesPrint("$%s is of wrong
    type",AC.dollarnames->namebuffer+d->name);
00654                                   MUNLOCK(ErrorMessageLock);
00655                                   Terminate(-1);
00656                               }
00657                               ncoef2 = 0; coef2[0] = 0; coef2[1] = 1;
00658                           }
00659                           break;
00660                       case DOLNUMBER:
00661                       case DOLTERMS:
00662                           if ( d->where[d->where[0]] == 0 ) {
00663                               r = d->where + d->where[0]-1;
00664                               i = ABS(*r);
00665                               if ( i == ( d->where[0]-1 ) ) {
00666                                   ncoef2 = (i-1)/2;
00667                                   if ( *r < 0 ) ncoef2 = -ncoef2;
00668                                   i--; cc = coef2; r = d->where + 1;
00669                                   while ( --i >= 0 ) *cc++ = (UWORD)(*r++);
00670                                   break;
00671                               }
00672                           }
00673 generic:;
00674                           if ( AC.UnsureDollarMode == 0 ) {
00675 #ifdef WITHPTHREADS
00676                               if ( dtype > 0 && ! DollarLocalCopy(dtype) ) {
00677                                   UNLOCK(d->pthreadslock);
00678                               }
00679 #endif
00680                               MLOCK(ErrorMessageLock);
00681                               MesPrint("$%s is of wrong type",AC.dollarnames->namebuffer+d->name);
00682                               MUNLOCK(ErrorMessageLock);
00683                               Terminate(-1);
00684                           }
00685                           ncoef2 = 0; coef2[0] = 0; coef2[1] = 1;
00686                           break;
00687                     }
00688                 }
00689 #ifdef WITHPTHREADS
00690                 if ( dtype > 0 && ! DollarLocalCopy(dtype) ) {
00691                     UNLOCK(d->pthreadslock);
00692                 }
00693 #endif
00694             }
00695             break;
00696         case IFEXPRESSION:
00697             r = ifp+2; j = ifp[1] - 2; ncoef2 = 0;
00698             while ( --j >= 0 ) {
00699                 if ( *r == AR.CurExpr ) { ncoef2 = 1; break; }
00700                 r++;
00701             }
00702             coef2[0] = ncoef2;
00703             coef2[1] = 1;
00704             break;
00705         case IFISFACTORIZED:
00706             r = ifp+2; j = ifp[1] - 2;
00707             if ( j == 0 ) {
00708                 ncoef2 = 0;
00709                 if ( ( Expressions[AR.CurExpr].vflags & ISFACTORIZED ) != 0 ) {
00710                     ncoef2 = 1;
00711                 }
00712             }
00713             else {
00714                 ncoef2 = 1;
00715                 while ( --j >= 0 ) {
00716                     if ( ( Expressions[*r].vflags & ISFACTORIZED ) == 0 ) {
00717                         ncoef2 = 0;
00718                         break;
00719                     }
00720                     r++;
00721                 }
00722             }
00723             coef2[0] = ncoef2;
00724             coef2[1] = 1;
00725             break;
00726         case IFOCCURS:
00727             {
00728                 WORD *OccStop = ifp + ifp[1], *ifpp = ifp+2;
00729                 ncoef2 = 0;
00730                 while ( ifpp < OccStop ) {
00731                     if ( FindVar(ifpp,term) == 1 ) {
00732                         ncoef2 = 1; break;
00733                     }
```

```
00734                       if ( *ifpp == DOTPRODUCT ) ifp += 3;
00735                       else ifpp += 2;
00736                   }
00737                   coef2[0] = ncoef2;
00738                   coef2[1] = 1;
00739               }
00740               break;
00741           case IFUSERFLAG:
00742               {
00743                   ncoef2 = 0;
00744                   if ( ( Expressions[AR.CurExpr].uflags & (1 << ifp[2]) ) != 0 )
00745                       ncoef2 = 1;
00746                   coef2[0] = ncoef2;
00747                   coef2[1] = 1;
00748               }
00749               break;
00750           default:
00751               break;
00752       }
00753       if ( !first ) {
00754           if ( ifp[-2] != ORCOND && ifp[-2] != ANDCOND ) {
00755               if ( ( ifp[-2] == EQUAL || ifp[-2] == NOTEQUAL ) &&
00756               ( ismul2 || ismul1 ) ) {
00757                   if ( ismul1 && ismul2 ) {
00758                       if ( coef1[0] == coef2[0] ) i = 1;
00759                       else i = 0;
00760                   }
00761                   else {
00762                       if ( ismul1 ) {
00763                           if ( ncoef2 )
00764                               Divvy(BHEAD coef2,&ncoef2,coef1,ncoef1);
00765                           cc = coef2; ncoef3 = ncoef2;
00766                       }
00767                       else {
00768                           if ( ncoef1 )
00769                               Divvy(BHEAD coef1,&ncoef1,coef2,ncoef2);
00770                           cc = coef1; ncoef3 = ncoef1;
00771                       }
00772                       if ( ncoef3 < 0 ) ncoef3 = -ncoef3;
00773                       if ( ncoef3 == 0 ) {
00774                           if ( ifp[-2] == EQUAL ) i = 1;
00775                           else i = 0;
00776                       }
00777                       else if ( cc[ncoef3] != 1 ) {
00778                           if ( ifp[-2] == EQUAL ) i = 0;
00779                           else i = 1;
00780                       }
00781                       else {
00782                           for ( j = 1; j < ncoef3; j++ ) {
00783                               if ( cc[ncoef3+j] != 0 ) break;
00784                           }
00785                           if ( j < ncoef3 ) {
00786                               if ( ifp[-2] == EQUAL ) i = 0;
00787                               else i = 1;
00788                           }
00789                           else if ( ifp[-2] == EQUAL ) i = 1;
00790                           else i = 0;
00791                       }
00792                   }
00793                   goto donemul;
00794               }
00795               else if ( AddRat(BHEAD coef1,ncoef1,coef2,-ncoef2,coef3,&ncoef3) ) {
00796                   NumberFree(coef3,"DoIfStatement"); NumberFree(Spac1,"DoIfStatement");
      TermFree(Spac2,"DoIfStatement");
00797                   MesCall("DoIfStatement"); return(-1);
00798               }
00799               switch ( ifp[-2] ) {
00800                   case GREATER:
00801                       if ( ncoef3 > 0 ) i = 1;
00802                       else i = 0;
00803                       break;
00804                   case GREATEREQUAL:
00805                       if ( ncoef3 >= 0 ) i = 1;
00806                       else i = 0;
00807                       break;
00808                   case LESS:
00809                       if ( ncoef3 < 0 ) i = 1;
00810                       else i = 0;
00811                       break;
00812                   case LESSEQUAL:
00813                       if ( ncoef3 <= 0 ) i = 1;
00814                       else i = 0;
00815                       break;
00816                   case EQUAL:
00817                       if ( ncoef3 == 0 ) i = 1;
00818                       else i = 0;
00819                       break;
```

```
00820                     case NOTEQUAL:
00821                         if ( ncoef3 != 0 ) i = 1;
00822                         else i = 0;
00823                         break;
00824                 }
00825 donemul:        if ( i ) { ncoef2 = 1; coef2 = Spac2; coef2[0] = coef2[1] = 1; }
00826                 else ncoef2 = 0;
00827                 ismul1 = ismul2 = 0;
00828             }
00829         }
00830         else {
00831             first = 0;
00832         }
00833         coef1 = Spac1;
00834         i = 2*ABS(ncoef2);
00835         for ( j = 0; j < i; j++ ) coef1[j] = coef2[j];
00836         ncoef1 = ncoef2;
00837 SkipCond:
00838         ifp += ifp[1];
00839     } while ( ifp < ifstop );
00840
00841     NumberFree(coef3,"DoIfStatement"); NumberFree(Spac1,"DoIfStatement");
      TermFree(Spac2,"DoIfStatement");
00842     if ( ncoef1 ) return(1);
00843     else return(0);
00844 }
00845
00846 /*
00847         #] DoIfStatement :
00848         #[ HowMany :                    WORD HowMany(ifcode,term)
00849
00850         Returns the number of times that the pattern in ifcode
00851         can be taken out from term. There is a subkey in ifcode[2];
00852         The notation is identical to the lhs of an id statement.
00853         Most of the code comes from TestMatch.
00854 */
00855
00856 WORD HowMany(PHEAD WORD *ifcode, WORD *term)
00857 {
00858     GETBIDENTITY
00859     WORD *m, *t, *r, *w, power, RetVal, i, topje, *newterm;
00860     WORD *OldWork, *ww, *mm;
00861     int *RepSto, RepVal;
00862     int numdollars = 0;
00863     m = ifcode + IDHEAD;
00864     AN.FullProto = m;
00865     AN.WildValue = w = m + SUBEXPSIZE;
00866     m += m[1];
00867     AN.WildStop = m;
00868     OldWork = AT.WorkPointer;
00869     if ( ( ifcode[4] & 1 ) != 0 ) { /* We have at least one dollar in the pattern */
00870         AR.Eside = LHSIDEX;
00871         ww = AT.WorkPointer; i = m[0]; mm = m;
00872         NCOPY(ww,mm,i);
00873         *OldWork += 3;
00874         *ww++ = 1; *ww++ = 1; *ww++ = 3;
00875         AT.WorkPointer = ww;
00876         RepSto = AN.RepPoint;
00877         RepVal = *RepSto;
00878         NewSort(BHEAD0);
00879         if ( Generator(BHEAD OldWork,AR.Cnumlhs) ) {
00880             LowerSortLevel();
00881             *RepSto = RepVal;
00882             AN.RepPoint = RepSto;
00883             AT.WorkPointer = OldWork;
00884             return(-1);
00885         }
00886         AT.WorkPointer = ww;
00887         if ( EndSort(BHEAD ww,0) < 0 ) {}
00888         *RepSto = RepVal;
00889         AN.RepPoint = RepSto;
00890         if ( *ww == 0 || *(ww+*ww) != 0 ) {
00891             if ( AP.lhdollarerror == 0 ) {
00892                 MLOCK(ErrorMessageLock);
00893                 MesPrint("&LHS must be one term");
00894                 MUNLOCK(ErrorMessageLock);
00895                 AP.lhdollarerror = 1;
00896             }
00897             AT.WorkPointer = OldWork;
00898             return(-1);
00899         }
00900         m = ww; AT.WorkPointer = ww = m + *m;
00901         if ( m[*m-1] < 0 ) { m[*m-1] = -m[*m-1]; }
00902         *m -= m[*m-1];
00903         AR.Eside = RHSIDE;
00904     }
00905     else {
```

```
00906         ww = term + *term;
00907         if ( AT.WorkPointer < ww ) AT.WorkPointer = ww;
00908     }
00909     ClearWild(BHEAD0);
00910     while ( w < AN.WildStop ) {
00911         if ( *w == LOADDOLLAR ) numdollars++;
00912         w += w[1];
00913     }
00914     AN.RepFunNum = 0;
00915     AN.RepFunList = AT.WorkPointer;
00916     AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
00917     topje = cbuf[AT.ebufnum].numrhs;
00918     if ( AT.WorkPointer >= AT.WorkTop ) {
00919         MLOCK(ErrorMessageLock);
00920         MesWork();
00921         MUNLOCK(ErrorMessageLock);
00922         return(-1);
00923     }
00924     AN.DisOrderFlag = ifcode[2] & SUBDISORDER;
00925     switch ( ifcode[2] & (~SUBDISORDER) ) {
00926         case SUBONLY :
00927             /* Must be an exact match */
00928             AN.UseFindOnly = 1; AN.ForFindOnly = 0;
00929 /*
00930         Copy the term first to scratchterm. This is needed
00931         because of the Substitute.
00932 */
00933             i = *term;
00934             t = term; newterm = r = AT.WorkPointer;
00935             NCOPY(r,t,i); AT.WorkPointer = r;
00936             RetVal = 0;
00937             if ( FindRest(BHEAD newterm,m) && ( AN.UsedOtherFind ||
00938                 FindOnly(BHEAD newterm,m) ) ) {
00939                 Substitute(BHEAD newterm,m,1);
00940                 if ( numdollars ) {
00941                     WildDollars(BHEAD (WORD *)0);
00942                     numdollars = 0;
00943                 }
00944                 ClearWild(BHEAD0);
00945                 RetVal = 1;
00946             }
00947             else RetVal = 0;
00948             break;
00949         case SUBMANY :
00950 /*
00951         Copy the term first to scratchterm. This is needed
00952         because of the Substitute.
00953 */
00954             i = *term;
00955             t = term; newterm = r = AT.WorkPointer;
00956             NCOPY(r,t,i); AT.WorkPointer = r;
00957             RetVal = 0;
00958             AN.UseFindOnly = 0;
00959             if ( ( power = FindRest(BHEAD newterm,m) ) > 0 ) {
00960                 if ( ( power = FindOnce(BHEAD newterm,m) ) > 0 ) {
00961                     AN.UseFindOnly = 0;
00962                     do {
00963                         Substitute(BHEAD newterm,m,1);
00964                         if ( numdollars ) {
00965                             WildDollars(BHEAD (WORD *)0);
00966                             numdollars = 0;
00967                         }
00968                         ClearWild(BHEAD0);
00969                         RetVal++;
00970                     } while ( FindRest(BHEAD newterm,m) && (
00971                         AN.UsedOtherFind || FindOnce(BHEAD newterm,m) ) );
00972                 }
00973                 else if ( power < 0 ) {
00974                     do {
00975                         Substitute(BHEAD newterm,m,1);
00976                         if ( numdollars ) {
00977                             WildDollars(BHEAD (WORD *)0);
00978                             numdollars = 0;
00979                         }
00980                         ClearWild(BHEAD0);
00981                         RetVal++;
00982                     } while ( FindRest(BHEAD newterm,m) );
00983                 }
00984             }
00985             else if ( power < 0 ) {
00986                 if ( FindOnce(BHEAD newterm,m) ) {
00987                     do {
00988                         Substitute(BHEAD newterm,m,1);
00989                         if ( numdollars ) {
00990                             WildDollars(BHEAD (WORD *)0);
00991                             numdollars = 0;
00992                         }
```

```
00993                       ClearWild(BHEAD0);
00994                   } while ( FindOnce(BHEAD newterm,m) );
00995                   RetVal = 1;
00996               }
00997           }
00998           break;
00999       case SUBONCE :
01000 /*
01001       Copy the term first to scratchterm. This is needed
01002       because of the Substitute.
01003 */
01004           i = *term;
01005           t = term; newterm = r = AT.WorkPointer;
01006           NCOPY(r,t,i); AT.WorkPointer = r;
01007           RetVal = 0;
01008           AN.UseFindOnly = 0;
01009           if ( FindRest(BHEAD newterm,m) && ( AN.UsedOtherFind || FindOnce(BHEAD newterm,m) ) ) {
01010               Substitute(BHEAD newterm,m,1);
01011               if ( numdollars ) {
01012                   WildDollars(BHEAD (WORD *)0);
01013                   numdollars = 0;
01014               }
01015               ClearWild(BHEAD0);
01016               RetVal = 1;
01017           }
01018           else RetVal = 0;
01019           break;
01020       case SUBMULTI :
01021           RetVal = FindMulti(BHEAD term,m);
01022           break;
01023       case SUBVECTOR :
01024           RetVal = 0;
01025           for ( i = 0; i < *term; i++ ) ww[i] = term[i];
01026           while ( ( power = FindAll(BHEAD ww,m,AR.Cnumlhs,ifcode) ) != 0 ) { RetVal += power; }
01027           break;
01028       case SUBSELECT :
01029           ifcode += IDHEAD;   ifcode += ifcode[1];    ifcode += *ifcode;
01030           AN.UseFindOnly = 1; AN.ForFindOnly = ifcode;
01031           if ( FindRest(BHEAD term,m) && ( AN.UsedOtherFind ||
01032                   FindOnly(BHEAD term,m) ) ) RetVal = 1;
01033           else RetVal = 0;
01034           break;
01035       default :
01036           RetVal = 0;
01037           break;
01038   }
01039   AT.WorkPointer = AN.RepFunList;
01040   cbuf[AT.ebufnum].numrhs = topje;
01041   return(RetVal);
01042 }
01043
01044 /*
01045       #] HowMany :
01046       #[ DoubleIfBuffers :
01047 */
01048
01049 void DoubleIfBuffers(void)
01050 {
01051   int newmax, i;
01052   WORD *newsumcheck;
01053   LONG *newheap, *newifcount;
01054   if ( AC.MaxIf == 0 ) newmax = 10;
01055   else                 newmax = 2*AC.MaxIf;
01056   newheap = (LONG *)Malloc1(sizeof(LONG)*(newmax+1),"IfHeap");
01057   newsumcheck = (WORD *)Malloc1(sizeof(WORD)*(newmax+1),"IfSumCheck");
01058   newifcount = (LONG *)Malloc1(sizeof(LONG)*(newmax+1),"IfCount");
01059   if ( AC.MaxIf ) {
01060       for ( i = 0; i < AC.MaxIf; i++ ) {
01061           newheap[i] = AC.IfHeap[i];
01062           newsumcheck[i] = AC.IfSumCheck[i];
01063           newifcount[i] = AC.IfCount[i];
01064       }
01065       AC.IfStack = (AC.IfStack-AC.IfHeap) + newheap;
01066       M_free(AC.IfHeap,"AC.IfHeap");
01067       M_free(AC.IfCount,"AC.IfCount");
01068       M_free(AC.IfSumCheck,"AC.IfSumCheck");
01069   }
01070   else {
01071       AC.IfStack = newheap;
01072   }
01073   AC.IfHeap = newheap;
01074   AC.IfSumCheck = newsumcheck;
01075   AC.IfCount = newifcount;
01076   AC.MaxIf = newmax;
01077 }
01078
01079 /*
```

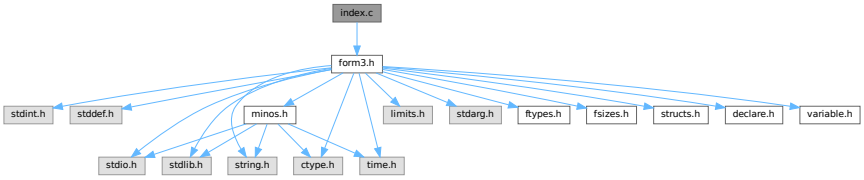
```
01080         #] DoubleIfBuffers :
01081     #] If statement :
01082     #[ Switch statement :
01083         #[ DoSwitch :
01084 */
01085
01086 int DoSwitch(PHEAD WORD *term, WORD *lhs)
01087 {
01088 /*
01089     For the moment we ignore the compiler buffer problems.
01090 */
01091     WORD numdollar = lhs[2];
01092     WORD ncase = DolToNumber(BHEAD numdollar);
01093     SWITCHTABLE *swtab = FindCase(lhs[3],ncase);
01094     return(Generator(BHEAD term,swtab->value));
01095 }
01096
01097 /*
01098         #] DoSwitch :
01099         #[ DoEndSwitch :
01100 */
01101
01102 int DoEndSwitch(PHEAD WORD *term, WORD *lhs)
01103 {
01104     SWITCH *sw = AC.SwitchArray+lhs[2];
01105     return(Generator(BHEAD term,sw->endswitch.value+1));
01106 }
01107
01108 /*
01109         #] DoEndSwitch :
01110         #[ FindCase :
01111 */
01112
01113 SWITCHTABLE *FindCase(WORD nswitch, WORD ncase)
01114 {
01115 /*
01116     First find the switch table and determine how we have to search.
01117 */
01118     SWITCH *sw = AC.SwitchArray+nswitch;
01119     WORD hi, lo, med;
01120     if ( sw->typetable == DENSETABLE ) {
01121         med = ncase - sw->caseoffset;
01122         if ( med >= sw->numcases || med < 0 ) return(&sw->defaultcase);
01123     }
01124     else {
01125 /*
01126         We need a binary search in the table.
01127 */
01128         if ( ncase > sw->maxcase || ncase < sw->mincase ) return(&sw->defaultcase);
01129         hi = sw->numcases-1; lo = 0;
01130         for(;;) {
01131             med = (hi+lo)/2;
01132             if ( ncase == sw->table[med].ncase ) break;
01133             else if ( ncase > sw->table[med].ncase ) {
01134                 lo = med+1;
01135                 if ( lo > hi ) return(&sw->defaultcase);
01136             }
01137             else {
01138                 hi = med-1;
01139                 if ( hi < lo ) return(&sw->defaultcase);
01140             }
01141         }
01142     }
01143     return(&sw->table[med]);
01144 }
01145
01146 /*
01147         #] FindCase :
01148         #[ DoubleSwitchBuffers :
01149 */
01150
01151 int DoubleSwitchBuffers(void)
01152 {
01153     int newmax, i;
01154     SWITCH *newarray;
01155     WORD *newheap;
01156     if ( AC.MaxSwitch == 0 ) newmax = 10;
01157     else                     newmax = 2*AC.MaxSwitch;
01158     newarray = (SWITCH *)Malloc1(sizeof(SWITCH)*(newmax+1),"SwitchArray");
01159     newheap = (WORD *)Malloc1(sizeof(WORD)*(newmax+1),"SwitchHeap");
01160     if ( AC.MaxSwitch ) {
01161         for ( i = 0; i < AC.MaxSwitch; i++ ) {
01162             newarray[i] = AC.SwitchArray[i];
01163             newheap[i] = AC.SwitchHeap[i];
01164         }
01165         M_free(AC.SwitchHeap,"AC.SwitchHeap");
01166         M_free(AC.SwitchArray,"AC.SwitchArray");
```

```
01167     }
01168     for ( i = AC.MaxSwitch; i <= newmax; i++ ) {
01169         newarray[i].table = 0;
01170         newarray[i].tablesize = 0;
01171         newarray[i].defaultcase.ncase = 0;
01172         newarray[i].defaultcase.value = 0;
01173         newarray[i].defaultcase.compbuffer = 0;
01174         newarray[i].endswitch.ncase = 0;
01175         newarray[i].endswitch.value = 0;
01176         newarray[i].endswitch.compbuffer = 0;
01177         newarray[i].typetable = 0;
01178         newarray[i].mincase = 0;
01179         newarray[i].maxcase = 0;
01180         newarray[i].numcases = 0;
01181         newarray[i].caseoffset = 0;
01182         newarray[i].iflevel = 0;
01183         newarray[i].whilelevel = 0;
01184         newarray[i].nestingsum = 0;
01185         newheap[i] = 0;
01186     }
01187     AC.SwitchArray = newarray;
01188     AC.SwitchHeap = newheap;
01189     AC.MaxSwitch = newmax;
01190     return(0);
01191 }
01192
01193 /*
01194         #] DoubleSwitchBuffers :
01195         #[ SwitchSplitMerge :
01196
01197         Sorts an array of WORDs. No adding of equal objects.
01198 */
01199
01200 void SwitchSplitMergeRec(SWITCHTABLE *array,WORD num,SWITCHTABLE *auxarray)
01201 {
01202     WORD n1,n2,i,j,k;
01203     SWITCHTABLE *t1,*t2, t;
01204     if ( num < 2 ) return;
01205     if ( num == 2 ) {
01206         if ( array[0].ncase > array[1].ncase ) {
01207             t = array[0]; array[0] = array[1]; array[1] = t;
01208         }
01209         return;
01210     }
01211     n1 = num/2;
01212     n2 = num - n1;
01213     SwitchSplitMergeRec(array,n1,auxarray);
01214     SwitchSplitMergeRec(array+n1,n2,auxarray);
01215     if ( array[n1-1].ncase <= array[n1].ncase ) return;
01216
01217     t1 = array; t2 = auxarray; i = n1; NCOPY(t2,t1,i);
01218     i = 0; j = n1; k = 0;
01219     while ( i < n1 && j < num ) {
01220         if ( auxarray[i].ncase <= array[j].ncase ) { array[k++] = auxarray[i++]; }
01221         else { array[k++] = array[j++]; }
01222     }
01223     while ( i < n1 ) array[k++] = auxarray[i++];
01224 /*
01225     Remember: remnants of j are still in place!
01226 */
01227 }
01228
01229 void SwitchSplitMerge(SWITCHTABLE *array,WORD num)
01230 {
01231     SWITCHTABLE *auxarray = (SWITCHTABLE *)Malloc1(sizeof(SWITCHTABLE)*num/2,"SwitchSplitMerge");
01232     SwitchSplitMergeRec(array,num,auxarray);
01233     M_free(auxarray,"SwitchSplitMerge");
01234 }
01235
01236 /*
01237         #] SwitchSplitMerge :
01238     #] Switch statement :
01239 */
```

4.64 index.c File Reference

```
#include "form3.h"
```

Include dependency graph for index.c:



Functions

- POSITION * FindBracket (WORD nexp, WORD *bracket)
- void PutBracketInIndex (PHEAD WORD *term, POSITION *newpos)
- void ClearBracketIndex (WORD numexp)
- void OpenBracketIndex (WORD nexpr)
- int PutInside (PHEAD WORD *term, WORD *code)

4.64.1 Detailed Description

The routines that deal with bracket indexing It creates the bracket index and it can find the brackets using the index.

Definition in file index.c.

4.64.2 Function Documentation

4.64.2.1 FindBracket()

```
POSITION * FindBracket (
            WORD nexp,
            WORD * bracket )
```

Definition at line 65 of file index.c.

4.64.2.2 PutBracketInIndex()

```
void PutBracketInIndex (
            PHEAD WORD * term,
            POSITION * newpos )
```

Definition at line 331 of file index.c.

4.64.2.3 ClearBracketIndex()

```
void ClearBracketIndex (
            WORD numexp )
```

Definition at line 548 of file index.c.

4.64.2.4 OpenBracketIndex()

```
void OpenBracketIndex (
            WORD nexpr )
```

Definition at line 580 of file index.c.

4.64.2.5 PutInside()

```
int PutInside (
            PHEAD WORD * term,
            WORD * code )
```

Definition at line 611 of file index.c.

4.65 index.c

Go to the documentation of this file.

```
00001
00008 /* #[ License : */
00009 /*
00010  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00011  *   When using this file you are requested to refer to the publication
00012  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013  *   This is considered a matter of courtesy as the development was paid
00014  *   for by FOM the Dutch physics granting agency and we would like to
00015  *   be able to track its scientific use to convince FOM of its value
00016  *   for the community.
00017  *
00018  *   This file is part of FORM.
00019  *
00020  *   FORM is free software: you can redistribute it and/or modify it under the
00021  *   terms of the GNU General Public License as published by the Free Software
00022  *   Foundation, either version 3 of the License, or (at your option) any later
00023  *   version.
00024  *
00025  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00027  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00028  *   details.
00029  *
00030  *   You should have received a copy of the GNU General Public License along
00031  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00032  */
00033 /* #] License : */
00034
00035 /*
00036     #[ Includes : index.c
00037 */
00038
00039 #include "form3.h"
00040
00041 /*
00042     #] Includes :
00043     #[ syntax and use :
00044
00045     The indexing of brackets is not automatic! It should only be used
00046     when one intends to use the contents of individual brackets.
```

```
00047     This is done with the addition of a + sign to the bracket statement:
00048     B+ a,b,c; or AB+ a,b,c;
00049     It does require resources! The index is kept in memory and is removed
00050     once the expression is treated and passed on to the output with different
00051     or no brackets.
00052     The index is limited to a given amount of space. Hence if there are too
00053     many brackets we will skip some in the index. Skipping goes by space
00054     occupied by the contents. We take the two adjacent bracket(s) with the
00055     least space together and represent them by the first one only. This gives
00056     a new spot.
00057     The expression struct has two pointers:
00058     bracketinfo      for using.
00059     newbracketinfo   for making new index.
00060
00061     #] syntax and use :
00062     #[ FindBracket :
00063 */
00064
00065 POSITION *FindBracket(WORD nexp, WORD *bracket)
00066 {
00067     GETIDENTITY
00068     BRACKETINDEX *bi;
00069     BRACKETINFO *bracketinfo;
00070     EXPRESSIONS e = &(Expressions[nexp]);
00071     LONG hi, low, med;
00072     int i;
00073     WORD oldsorttype = AR.SortType, *t1, *t2, j, bsize, *term, *p, *pstop, *pp;
00074     WORD *tstop, *cp, a[4], *bracketh;
00075     FILEHANDLE *fi;
00076     POSITION auxpos, toppos;
00077     switch ( e->status ) {
00078         case UNHIDELEXPRESSION:
00079         case UNHIDEGEXPRESSION:
00080         case DROPHLEXPRESSION:
00081         case DROPHGEXPRESSION:
00082         case HIDDENLEXPRESSION:
00083         case HIDDENGEXPRESSION:
00084             fi = AR.hidefile;
00085             break;
00086         default:
00087             fi = AR.infile;
00088             break;
00089     }
00090     if ( AT.bracketinfo ) bracketinfo = AT.bracketinfo;
00091     else                  bracketinfo = e->bracketinfo;
00092     hi = bracketinfo->indexfill; low = 0;
00093     if ( hi <= 0 ) return(0);
00094 /*
00095     The next code is needed for a problem with sorting when there are
00096     only functions outside the bracket. This gives ordinarily the wrong
00097     sorting. We solve that by taking HAAKJE along with the outside while
00098     running the Compare. But this means that we need to copy bracket.
00099 */
00100     bracketh = TermMalloc("FindBracket");
00101     i = *bracket; p = bracket; pp = bracketh; NCOPY(pp,p,i)
00102     pp -= 3; *pp++ = HAAKJE; *pp++ = 3; *pp++ = 0; *pp++ = 1; *pp++ = 1; *pp++ = 3;
00103     *bracketh += 3;
00104
00105     AT.fromindex = 1;
00106     AR.SortType = bracketinfo->SortType;
00107     bi = bracketinfo->indexbuffer + hi - 1;
00108     if ( *bracketh == 7 ) {
00109         if ( bracketinfo->bracketbuffer[bi->bracket] == 7 ) i = 0;
00110         else i = -1;
00111     }
00112     else if ( bracketinfo->bracketbuffer[bi->bracket] == 7 ) i = 1;
00113     else i = CompareTerms(BHEAD bracketh,bracketinfo->bracketbuffer+bi->bracket,0);
00114     if ( i < 0 ) {
00115         AR.SortType = oldsorttype;
00116         AT.fromindex = 0;
00117         TermFree(bracketh,"FindBracket");
00118         return(0);
00119     }
00120     else if ( i == 0 ) med = hi-1;
00121     else {
00122         for (;;) {
00123         med = (hi+low)/2;
00124         bi = bracketinfo->indexbuffer + med;
00125         if ( *bracketh == 7 ) {
00126             if ( bracketinfo->bracketbuffer[bi->bracket] == 7 ) i = 0;
00127             else i = -1;
00128         }
00129         else if ( bracketinfo->bracketbuffer[bi->bracket] == 7 ) i = 1;
00130         else i = CompareTerms(BHEAD bracketh,bracketinfo->bracketbuffer+bi->bracket,0);
00131         if ( i == 0 ) { break; }
00132         if ( i > 0 ) {
00133             if ( low == med ) { /* no occurrence */
```

```
00134                 AR.SortType = oldsorttype;
00135                 AT.fromindex = 0;
00136                 TermFree(bracketh,"FindBracket");
00137                 return(0);
00138             }
00139             hi = med;
00140         }
00141         else if ( i < 0 ) {
00142             if ( low == med ) break;
00143             low = med;
00144         }
00145     }}
00146 /*
00147     The bracket is now either bi itself or between bi and the next one
00148     or it is not present at all.
00149 */
00150     AN.theposition = AS.OldOnFile[nexp];
00151     ADD2POS(AN.theposition,bi->start);
00152 /*
00153     The seek will have to move closer to the actual read so that we
00154     can place a lock around the two.
00155     if ( fi->handle >= 0 ) SeekFile(fi->handle,&AN.theposition,SEEK_SET);
00156     else SetScratch(fi,&AN.theposition);
00157 */
00158 /*
00159     Put the bracket in the compress buffer as if it were the last term read.
00160     Have a look at AR.CompressPointer. (set it right)
00161 */
00162     term = AT.WorkPointer;
00163     t1 = bracketinfo->bracketbuffer+bi->bracket;
00164     j = *t1;
00165 /*
00166     Note that in the bracketbuffer, the bracket sits with HAAKJE.
00167
00168     The next is (hopefully) a bug fix. Originally the code read bsize = j
00169     but that overcounts one. We have the part outside the bracket and the
00170     coefficient which is 1,1,3. But we also have the length indicator.
00171     Where we use the variable bsize we do not include the length indicator,
00172     and we have the part outside plus 7,3,0 which is also three words.
00173 */
00174     bsize = j-1;
00175     t2 = AR.CompressPointer;
00176     NCOPY(t2,t1,j)
00177     if ( i == 0 ) { /* We found the proper bracket already */
00178         AR.SortType = oldsorttype;
00179         AT.fromindex = 0;
00180         TermFree(bracketh,"FindBracket");
00181         return(&AN.theposition);
00182     }
00183 /*
00184     Here we have to skip to the bracket if it exists (!)
00185     Let us first look whether the expression is in memory.
00186     If not we have to make a buffer to increase speed..
00187 */
00188     if ( fi->handle < 0 ) {
00189         p = (WORD *)((UBYTE *)(fi->PObuffer)
00190              + BASEPOSITION(AS.OldOnFile[nexp])
00191              + BASEPOSITION(bi->start));
00192         pstop = (WORD *)((UBYTE *)(fi->PObuffer)
00193              + BASEPOSITION(AS.OldOnFile[nexp])
00194              + BASEPOSITION(bi->next));
00195         while ( p < pstop ) {
00196 /*
00197             Check now: if size or part from previous term < size of bracket
00198                     we have to setup the bracket again and test.
00199                     Otherwise, skip immediately to the next term.
00200 */
00201             if ( *p <= -bsize ) {   /* no change of bracket */
00202                 p++; p += *p + 1;
00203             }
00204             else if ( *p < 0 ) {    /* changes bracket */
00205                 pp = p;
00206                 t2 = AR.CompressPointer;
00207                 t1 = t2 - *p++ + 1;
00208                 j = *p++;
00209                 NCOPY(t1,p,j)
00210                 t2++; while ( *t2 != HAAKJE ) t2 += t2[1];
00211                 t2 += t2[1];
00212                 a[1] = t2[0]; a[2] = t2[1]; a[3] = t2[2];
00213                 *t2++ = 1; *t2++ = 1; *t2++ = 3;
00214                 *AR.CompressPointer = t2 - AR.CompressPointer;
00215                 bsize = *AR.CompressPointer - 1;
00216                 if ( *bracketh == 7 ) {
00217                     if ( AR.CompressPointer[0] == 7 ) i = 0;
00218                     else i = -1;
00219                 }
00220                 else if ( AR.CompressPointer[0] == 7 ) i = 1;
```

```
00221                 else i = CompareTerms(BHEAD bracketh,AR.CompressPointer,0);
00222                 t2[-3] = a[1]; t2[-2] = a[2]; t2[-1] = a[3];
00223                 if ( i == 0 ) {
00224                     SETBASEPOSITION(AN.theposition,(pp-fi->PObuffer)*sizeof(WORD));
00225                     fi->POfill = pp;
00226                     goto found;
00227                 }
00228                 if ( i > 0 ) break; /* passed what was possible */
00229             }
00230             else {  /* no compression. We have to check! */
00231                 WORD *oldworkpointer = AT.WorkPointer, *t3, *t4;
00232                 t2 = p + 1; while ( *t2 != HAAKJE ) t2 += t2[1];
00233                 t2 += t2[1];
00234 /*
00235                 Here we need to copy the term. Modifying has proven to
00236                 be NOT threadsafe.
00237 */
00238                 t3 = oldworkpointer; t4 = p;
00239                 while ( t4 < t2 ) *t3++ = *t4++;
00240                 *t3++ = 1; *t3++ = 1; *t3++ = 3;
00241                 *oldworkpointer = t3 - oldworkpointer;
00242                 bsize = *oldworkpointer - 1;
00243                 AT.WorkPointer = t3;
00244                 t3 = oldworkpointer;
00245                 if ( *bracketh == 7 ) {
00246                     if ( t3[0] == 7 ) i = 0;
00247                     else i = -1;
00248                 }
00249                 else if ( t3[0] == 7 ) i = 1;
00250                 else {
00251                     i = CompareTerms(BHEAD bracketh,t3,0);
00252                 }
00253                 AT.WorkPointer = oldworkpointer;
00254                 if ( i == 0 ) {
00255                     SETBASEPOSITION(AN.theposition,(p-fi->PObuffer)*sizeof(WORD));
00256                     fi->POfill = p;
00257                     goto found;
00258                 }
00259                 if ( i > 0 ) break; /* passed what was possible */
00260                 p += *p;
00261             }
00262         }
00263         AR.SortType = oldsorttype;
00264         AT.fromindex = 0;
00265         TermFree(bracketh,"FindBracket");
00266         return(0);  /* Bracket does not exist */
00267     }
00268     else {
00269 /*
00270         In this case we can work with the old representation without HAAKJE.
00271         We stop searching when we reach toppos and we do not call Compare.
00272 */
00273         toppos = AS.OldOnFile[nexp];
00274         ADD2POS(toppos,bi->next);
00275         cp = AR.CompressPointer;
00276         for(;;) {
00277             auxpos = AN.theposition;
00278             GetOneTerm(BHEAD term,fi,&auxpos,0);
00279             if ( *term == 0 ) {
00280                 AR.SortType = oldsorttype;
00281                 AT.fromindex = 0;
00282                 return(0);  /* Bracket does not exist */
00283             }
00284             tstop = term + *term;
00285             tstop -= ABS(tstop[-1]);
00286             t1 = term + 1;
00287             while ( *t1 != HAAKJE && t1 < tstop ) t1 += t1[1];
00288             i = *bracket-4;
00289             if ( t1-term == *bracket-3 ) {
00290                 t1 = term + 1; t2 = bracket+1;
00291                 while ( i > 0 && *t1 == *t2 ) { t1++; t2++; i--; }
00292                 if ( i <= 0 ) {
00293                     AR.CompressPointer = cp;
00294                     goto found;
00295                 }
00296             }
00297             AR.CompressPointer = cp;
00298             AN.theposition = auxpos;
00299 /*
00300             Now check whether we passed the 'point'
00301 */
00302             if ( ISGEPOS(AN.theposition,toppos) ) {
00303                 AR.SortType = oldsorttype;
00304                 AR.CompressPointer = cp;
00305                 AT.fromindex = 0;
00306                 TermFree(bracketh,"FindBracket");
00307                 return(0);  /* Bracket does not exist */
```

```
00308             }
00309         }
00310     }
00311 found:
00312     AR.SortType = oldsorttype;
00313     AT.fromindex = 0;
00314     TermFree(bracketh,"FindBracket");
00315     return(&AN.theposition);
00316 }
00317
00318 /*
00319     #] FindBracket :
00320     #[ PutBracketInIndex :
00321
00322     Call via
00323     if ( AR.BracketOn ) PutBracketInIndex(BHEAD term);
00324
00325     This means that there should be a bracket somewhere
00326     Note that the brackets come in in proper order.
00327
00328     DON'T forget AR.SortType to be put into e->bracketinfo->SortType
00329 */
00330
00331 void PutBracketInIndex(PHEAD WORD *term, POSITION *newpos)
00332 {
00333     GETBIDENTITY
00334     BRACKETINDEX *bi, *b1, *b2, *b3;
00335     BRACKETINFO *b;
00336     POSITION thepos;
00337     EXPRESSIONS e = Expressions + AR.CurExpr;
00338     LONG hi, i, average;
00339     WORD *t, *tstop, *t1, *t2, *oldt, oldsize, a[4];
00340     if ( ( b = e->newbracketinfo ) == 0 ) return;
00341     DIFPOS(thepos,*newpos,e->onfile);
00342     tstop = term + *term;
00343     tstop -= ABS(tstop[-1]);
00344     t = term+1;
00345     while ( *t != HAAKJE && t < tstop ) t += t[1];
00346     if ( t >= tstop ) return; /* no ticket, no laundry */
00347     t += t[1];  /* include HAAKJE for the sorting */
00348     a[0] = t[0]; a[1] = t[1]; a[2] = t[2];
00349     oldt = t; oldsize = *term; *t++ = 1; *t++ = 1; *t++ = 3;
00350     *term = t - term;
00351     AT.fromindex = 1;
00352 /*
00353     Check now with the last bracket in the buffer.
00354     If it is the same we can abort.
00355 */
00356     hi = b->indexfill;
00357     if ( hi > 0 ) {
00358         bi = b->indexbuffer + hi - 1;
00359         bi->next = thepos;
00360         if ( *term == 7 ) {
00361             if ( b->bracketbuffer[bi->bracket] == 7 ) i = 0;
00362             else i = -1;
00363         }
00364         else if ( b->bracketbuffer[bi->bracket] == 7 ) i = 1;
00365         else i = CompareTerms(BHEAD term,b->bracketbuffer+bi->bracket,0);
00366         if ( i == 0 ) { /* still the same bracket */
00367             bi->termsinbracket++;
00368             goto bracketdone;
00369         }
00370         if ( i > 0 ) { /* We have a problem */
00371 /*
00372             There is a special case in which we have only functions and
00373             term is contained completely in the bracket
00374 */
00375 /*
00376             t = term + 1;
00377             tstop = term + *term - 3;
00378             while ( t < tstop && *t > HAAKJE ) t += t[1];
00379             if ( t < tstop ) goto problems;
00380 */
00381             for ( i = 1; i < *term - 3; i++ ) {
00382                 if ( term[i] != b->bracketbuffer[bi->bracket+i] ) break;
00383             }
00384             if ( i < *term - 3 ) {
00385 /*
00386 problems:;
00387 */
00388 /* INTERNAL_ERROR_EXCL_START */
00389                 *term = oldsize; oldt[0] = a[0]; oldt[1] = a[1]; oldt[2] = a[2];
00390                 MLOCK(ErrorMessageLock);
00391                 MesPrint("!>Error! Illegal bracket sequence detected in PutBracketInIndex");
00392 #ifdef WITHPTHREADS
00393                 MesPrint("Worker = %w");
00394 #endif
```

```
00395                 PrintTerm(term,"term into index");
00396                 PrintTerm(b->bracketbuffer+bi->bracket,"Last in index");
00397                 MUNLOCK(ErrorMessageLock);
00398                 AT.fromindex = 0;
00399                 Terminate(-1);
00400 /* INTERNAL_ERROR_EXCL_STOP */
00401             }
00402             i = -1;
00403         }
00404     }
00405 /*
00406     If there is room for more brackets, we add this one.
00407 */
00408     if ( b->bracketfill+*term >= b->bracketbuffersize
00409     && ( b->bracketbuffersize < AM.MaxBracketBufferSize
00410     || ( e->vflags & ISFACTORIZED ) != 0 ) ) {
00411 /*
00412         Enlarge bracket buffer
00413 */
00414         WORD *oldbracketbuffer = b->bracketbuffer;
00415         i = MaX(b->bracketbuffersize * 2, b->bracketfill+*term+1);
00416         if ( i > AM.MaxBracketBufferSize && ( e->vflags & ISFACTORIZED ) == 0 )
00417                 i = AM.MaxBracketBufferSize;
00418         if ( i > b->bracketfill+*term ) {
00419             b->bracketbuffersize = i;
00420             b->bracketbuffer = (WORD *)Malloc1(b->bracketbuffersize*sizeof(WORD),
00421                                 "new bracket buffer");
00422             t1 = b->bracketbuffer; t2 = oldbracketbuffer;
00423             i = b->bracketfill;
00424             NCOPY(t1,t2,i)
00425             if ( oldbracketbuffer ) M_free(oldbracketbuffer,"old bracket buffer");
00426         }
00427     }
00428     if ( b->bracketfill+*term < b->bracketbuffersize ) {
00429         if ( b->indexfill >= b->indexbuffersize ) {
00430 /*
00431             Enlarge index
00432 */
00433             BRACKETINDEX *oldindexbuffer = b->indexbuffer;
00434             b->indexbuffersize *= 2;
00435             b->indexbuffer = (BRACKETINDEX *)
00436                 Malloc1(b->indexbuffersize*sizeof(BRACKETINDEX),"new bracket index");
00437             b1 = b->indexbuffer; b2 = oldindexbuffer;
00438             i = b->indexfill;
00439             NCOPY(b1,b2,i)
00440             if ( oldindexbuffer ) M_free(oldindexbuffer,"old bracket index");
00441         }
00442     }
00443     else {
00444 /*
00445         We have too many brackets in the buffer. Try to improve.
00446         This is the interesting algorithm. We try to eliminate about 1/4 to
00447         1/2 of the brackets from the index. This should be done by size of
00448         the bracket contents to make the searching as fast as possible.
00449         But! Do not touch the last bracket.
00450         Note that we are always filling from the back.
00451         Algorithm: Throw away every second bracket, unless b1+b2 is much longer
00452         than average. How much is something we can tune.
00453 */
00454         average = DIVPOS(thepos,b->indexfill+1);
00455         if ( average <= 0 ) {
00456             MLOCK(ErrorMessageLock);
00457             MesPrint("Problems with bracket buffer. Increase MaxBracketBufferSize in form.set");
00458             MesPrint("Current size is %l",AM.MaxBracketBufferSize*sizeof(WORD));
00459             MUNLOCK(ErrorMessageLock);
00460             Terminate(-1);
00461         }
00462         average *= 4;  /* 2*2: one 2 for much longer, one 2 because we have pairs */
00463         t2 = b->bracketbuffer;
00464         b3 = b1 = b->indexbuffer;
00465         bi = b->indexbuffer + b->indexfill;
00466         b2 = b1+1;
00467         while ( b2+2 < bi ) {
00468             if ( DIFBASE(b2->next,b1->start) > average ) {
00469                 t1 = b->bracketbuffer + b1->bracket;
00470                 b1->bracket = t2 - b->bracketbuffer;
00471                 i = *t1; NCOPY(t2,t1,i)
00472                 *b3++ = *b1;
00473                 t1 = b->bracketbuffer + b2->bracket;
00474                 b2->bracket = t2 - b->bracketbuffer;
00475                 i = *t1; NCOPY(t2,t1,i)
00476                 *b3++ = *b2;
00477                 if ( b3 <= b1 ) {
00478                     PUTZERO(b1->start);
00479                     PUTZERO(b1->next);
00480                     b1->bracket = 0;
00481                     b1->termsinbracket = 0;
```

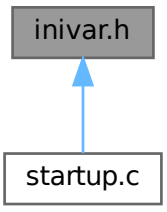
```
00482                 }
00483                 if ( b3 <= b2 ) {
00484                     PUTZERO(b2->start);
00485                     PUTZERO(b2->next);
00486                     b2->bracket = 0;
00487                     b2->termsinbracket = 0;
00488                 }
00489             }
00490             else {
00491                 t1 = b->bracketbuffer + b1->bracket;
00492                 b1->bracket = t2 - b->bracketbuffer;
00493                 i = *t1; NCOPY(t2,t1,i)
00494                 b1->next = b2->next;
00495                 b1->termsinbracket += b2->termsinbracket;
00496                 *b3++ = *b1;
00497                 if ( b3 <= b1 ) {
00498                     PUTZERO(b1->start);
00499                     PUTZERO(b1->next);
00500                     b1->bracket = 0;
00501                     b1->termsinbracket = 0;
00502                 }
00503                 PUTZERO(b2->start);
00504                 PUTZERO(b2->next);
00505                 b2->bracket = 0;
00506                 b2->termsinbracket = 0;
00507             }
00508             b1 += 2; b2 += 2;
00509         }
00510         while ( b1 < bi ) {
00511             t1 = b->bracketbuffer + b1->bracket;
00512             b1->bracket = t2 - b->bracketbuffer;
00513             i = *t1; NCOPY(t2,t1,i)
00514             *b3++ = *b1;
00515             if ( b3 <= b1 ) {
00516                 PUTZERO(b1->start);
00517                 PUTZERO(b1->next);
00518                 b1->bracket = 0;
00519                 b1->termsinbracket = 0;
00520             }
00521             b1++;
00522         }
00523         b->indexfill = b3 - b->indexbuffer;
00524         b->bracketfill = t2 - b->bracketbuffer;
00525     }
00526     bi = b->indexbuffer + b->indexfill;
00527     b->indexfill++;
00528     bi->bracket = b->bracketfill;
00529     bi->start = thepos;
00530     bi->next = thepos;
00531     bi->termsinbracket = 1;
00532 /*
00533     Copy the bracket into the buffer
00534 */
00535     t1 = term; t2 = b->bracketbuffer + bi->bracket; i = *t1;
00536     b->bracketfill += i;
00537     NCOPY(t2,t1,i)
00538 bracketdone:
00539     *term = oldsize; oldt[0] = a[0]; oldt[1] = a[1]; oldt[2] = a[2];
00540     AT.fromindex = 0;
00541 }
00542
00543 /*
00544     #] PutBracketInIndex :
00545     #[ ClearBracketIndex :
00546 */
00547
00548 void ClearBracketIndex(WORD numexp)
00549 {
00550     BRACKETINFO *b;
00551     if ( numexp >= 0 ) {
00552         b = Expressions[numexp].bracketinfo;
00553         Expressions[numexp].bracketinfo = 0;
00554     }
00555     else if ( numexp == -1 ) {
00556         GETIDENTITY
00557         b = AT.bracketinfo;
00558         AT.bracketinfo = 0;
00559     }
00560     else {
00561         numexp = -numexp-2;
00562         b = Expressions[numexp].newbracketinfo;
00563         Expressions[numexp].newbracketinfo = 0;
00564     }
00565     if ( b == 0 ) return;
00566     b->indexfill = b->indexbuffersize = 0;
00567     b->bracketfill = b->bracketbuffersize = 0;
00568     M_free(b->bracketbuffer,"ClearBracketBuffer");
```

```
00569     M_free(b->indexbuffer,"ClearIndexBuffer");
00570     M_free(b,"BracketInfo");
00571 }
00572
00573 /*
00574     #] ClearBracketIndex :
00575     #[ OpenBracketIndex :
00576
00577     Note: This routine is thread-safe
00578 */
00579
00580 void OpenBracketIndex(WORD nexpr)
00581 {
00582     EXPRESSIONS e = Expressions + nexpr;
00583     BRACKETINFO *bi;
00584     LONG i;
00585     bi = (BRACKETINFO *)Malloc1(sizeof(BRACKETINFO),"BracketInfo");
00586     e->newbracketinfo = bi;
00587 /*
00588     i = 20*AM.MaxTer/sizeof(WORD);
00589     if ( i < 1000 ) i = 1000;
00590 */
00591     i = 2000;
00592     bi->bracketbuffer = (WORD *)Malloc1(i*sizeof(WORD),"Bracket Buffer");
00593     bi->bracketbuffersize = i;
00594     bi->bracketfill = 0;
00595     i = 50;
00596     bi->indexbuffer = (BRACKETINDEX *)Malloc1(i*sizeof(BRACKETINDEX),"Bracket Index");
00597     bi->indexbuffersize = i;
00598     bi->indexfill = 0;
00599     bi->SortType = AC.SortType;
00600 }
00601
00602 /*
00603     #] OpenBracketIndex :
00604     #[ PutInside :
00605
00606     Puts a term, or a bracket determined part of a term inside a function.
00607
00608     AT.WorkPointer points at term+*term
00609 */
00610
00611 int PutInside(PHEAD WORD *term, WORD *code)
00612 {
00613     WORD *from, *to, *oldbuf, *tStop, *t, *tt, oldon, oldact, inc, argsize, *termout;
00614     int i, ii, error;
00615
00616     if ( code[1] == 4 && ( code[2] == 0 || code[2] == 1 ) ) {
00617 /*
00618         Put all inside. Move the term by 1+FUNHEAD+ARGHEAD
00619 */
00620         from = term+*term; to = from+1+ARGHEAD+FUNHEAD; i = ii = *term;
00621         to[0] = 1; to[1] = 1; to[2] = 3;
00622         while ( --i >= 0 ) *--to = *--from;
00623         to = term;
00624         *to++ = term[0]+4+ARGHEAD+FUNHEAD;
00625         *to++ = code[3];
00626         *to++ = ii+FUNHEAD+ARGHEAD;
00627         *to++ = 1;  /* set dirty flags, because there could be a fast notation */
00628         FILLFUN3(to)
00629         *to++ = ii+ARGHEAD;
00630         *to++ = 1;
00631         FILLARG(to)
00632         return(0);
00633     }
00634 /*
00635     First we save the old bracket variables. Then we set variables to
00636     influence the PutBracket routine and call it.
00637     After that we set the values back and sort out the results by placing the
00638     inside of the bracket inside the function.
00639 */
00640     termout = AT.WorkPointer;
00641     oldbuf = AT.BrackBuf;
00642     oldon  = AR.BracketOn;
00643     oldact = AT.PolyAct;
00644     AR.BracketOn = -code[2];
00645     AT.BrackBuf  = code+4;
00646     AT.PolyAct = 0;
00647     error = PutBracket(BHEAD term);
00648     AT.PolyAct = oldact;
00649     AT.BrackBuf = oldbuf;
00650     AR.BracketOn = oldon;
00651     if ( error ) return(error);
00652     i = *termout; from = termout; to = term;
00653     NCOPY(to,from,i);
00654     tStop = term +*term; tStop -= tStop[-1];
00655     t = term+1;
```

```
00656     while ( t < tStop && *t != HAAKJE ) t += t[1];
00657     from = term + *term;
00658     inc = FUNHEAD+ARGHEAD-t[1]+1;
00659     tt = t + t[1];
00660     argsize = from-tt+1;
00661     to = from + inc;
00662     to[0] = 1;
00663     to[1] = 1;
00664     to[2] = 3;
00665     while ( from > tt ) *--to = *--from;
00666     *--to = argsize;
00667     *t++ = code[3];
00668     *t++ = argsize+FUNHEAD+ARGHEAD;
00669     *t++ = 1;
00670     FILLFUN3(t);
00671     *t++ = argsize+ARGHEAD;
00672     *t++ = 1;
00673     FILLARG(t);
00674     *term += inc+3;
00675     AT.WorkPointer = term+*term;
00676     if ( Normalize(BHEAD term) ) error = 1;
00677     return(error);
00678 }
00679
00680 /*
00681     #] PutInside :
00682 */
00683
00684 /*
00685     The next routines are for indexing the local output files in a parallel
00686     sort. This indexing is needed to get a fast determination of the
00687     splitting terms needed to divide the terms evenly over the processors.
00688     Actually this method works well for ParFORM, but may not work well
00689     for TFORM.
00690
00691     #[ PutTermInIndex :
00692
00693     Puts a term in the term index.
00694     Action:
00695         if the index hasn't reached its full size
00696             if there is room, put the term
00697             if there is no room: extend the buffer, put the term
00698         else
00699             check if the last term has a number of the type skip*m+1
00700                 if no, overwrite the last term
00701                 if yes, check whether there is room for one more term
00702                     yes: add the term
00703                     no: drop all even terms, compress the list,
00704                         multiply skip by 2, and add this term.
00705
00706 int PutTermInIndex(WORD *term,POSITION *position)
00707 {
00708     return(0);
00709 }
00710
00711     #] PutTermInIndex :
00712 */
```

4.66 inivar.h File Reference

This graph shows which files directly or indirectly include this file:



Data Structures

- struct **fixedfun**

Variables

- FIXEDGLOBALS FG
- ALLGLOBALS A
- FIXEDSET fixedsets [ ]
- UBYTE BufferForOutput [MAXLINELENGTH+14]
- char * setupfilename = "form.set"

4.66.1 Detailed Description

Contains the initialization of a number of structs at compile time This file should only be included in the file startup.c !!!

Definition in file inivar.h.

4.66.2 Variable Documentation

4.66.2.1 FG

```
FIXEDGLOBALS FG
```

Definition at line 34 of file inivar.h.

4.66.2.2 A

```
ALLGLOBALS A
```

Definition at line 133 of file inivar.h.

4.66.2.3 fixedsets

```
FIXEDSET fixedsets[]
```

Initial value:

```
= {
     {"pos_",    "integers > 0",  CSYMBOL, 0}
    ,{"pos0_",   "integers >= 0", CSYMBOL, 0}
    ,{"neg_",    "integers < 0",  CSYMBOL, 0}
    ,{"neg0_",   "integers <= 0", CSYMBOL, 0}
    ,{"even_",   "even integers", CSYMBOL, 0}
    ,{"odd_",    "odd integers",  CSYMBOL, 0}
    ,{"int_",    "all integers",  CSYMBOL, 0}
    ,{"symbol_","only symbols",  CSYMBOL, MAXPOSITIVE}
    ,{"fixed_", "fixed indices", CINDEX, 0}
    ,{"index_", "all indices",    CINDEX, 0}
    ,{"number_","all rationals", CSYMBOL, 0}
    ,{"dummyindices_", "dummy indices", CINDEX, 0}
    ,{"vector_","only vectors",  CVECTOR, 0}
}
```

Definition at line 259 of file inivar.h.

4.66.2.4 BufferForOutput

```
UBYTE BufferForOutput[MAXLINELENGTH+14]
```

Definition at line 275 of file inivar.h.

4.66.2.5 setupfilename

```
char* setupfilename = "form.set"
```

Definition at line 277 of file inivar.h.

4.67 inivar.h

Go to the documentation of this file.

```
00001
00007 /* #[ License : */
00008 /*
00009  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00010  *   When using this file you are requested to refer to the publication
00011  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00012  *   This is considered a matter of courtesy as the development was paid
00013  *   for by FOM the Dutch physics granting agency and we would like to
00014  *   be able to track its scientific use to convince FOM of its value
00015  *   for the community.
00016  *
00017  *   This file is part of FORM.
00018  *
00019  *   FORM is free software: you can redistribute it and/or modify it under the
00020  *   terms of the GNU General Public License as published by the Free Software
00021  *   Foundation, either version 3 of the License, or (at your option) any later
00022  *   version.
00023  *
00024  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00025  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00026  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00027  *   details.
00028  *
00029  *   You should have received a copy of the GNU General Public License along
00030  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00031  */
00032 /* #] License : */
00033
00034 FIXEDGLOBALS FG = {
00035          {Traces                /* Dummy as zeroeth argument */
00036         ,Traces
00037         ,EpfCon
00038         ,RatioGen
00039         ,SymGen
00040         ,TenVec
00041         ,DoSumF1
00042         ,DoSumF2}
00043
00044         ,{TraceFind             /* Dummy as zeroeth argument */
00045         ,TraceFind
00046         ,EpfFind
00047         ,RatioFind
00048         ,SymFind
00049         ,TenVecFind}
00050
00051         ,{"symbol"
00052         ,"index"
00053         ,"vector"
00054         ,"function"
00055         ,"set"
00056         ,"expression"
00057         ,"dotproduct"
00058         ,"number"
00059         ,"subexp"
00060         ,"delta"}
00061
00062         ,{"(local)"
00063         ,"(skip/local)"
```

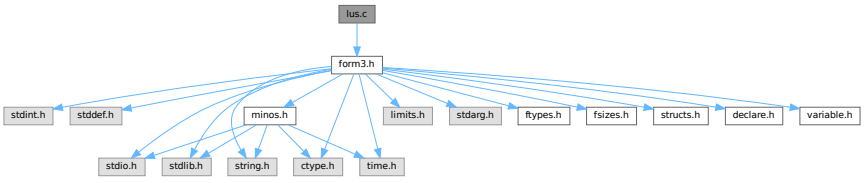
```
00064         ,"(drop/local)"
00065         ,"(dropped)"
00066         ,"(global)"
00067         ,"(skip/global)"
00068         ,"(drop/global)"
00069         ,"(dropped)"
00070         ,"(stored)"
00071         ,"(local-hidden)"
00072         ,"(local-to be hidden)"
00073         ,"(local-hidden-dropped)"
00074         ,"(local-to be unhidden)"
00075         ,"(global-hidden)"
00076         ,"(global-to be hidden)"
00077         ,"(global-hidden-dropped)"
00078         ,"(global-to be unhidden)"
00079         ,"(into-hide-local)"
00080         ,"(into-hide-global)"
00081         ,"(spectator)"
00082         ,"(drop/spectator)"}
00083
00084         ,{" Functions"
00085         ," Commuting Functions"}
00086
00087         ,{"left"
00088         ,"active"
00089         ,"in output"}
00090
00091     ,(char *)0
00092     ,(char *)0
00093     ,(UBYTE *)"1"
00094     ,(WORD)0
00095     ,(WORD)0
00096     ,(WORD)0
00097     ,(WORD)0
00098
00099 /*  ASCII table of character types. Note that on some computers this
00100     table may be different from the ASCII table.
00101
00102     -1  Illegal character
00103      0  Alphabetic character
00104      1  Digit
00105      2  . $ _ ? # or '
00106      3  [,]
00107      4  ( ) = ; or ,
00108      5  + - * % / ^ :
00109      6  blank, tab, linefeed
00110      7  {,|,}
00111      8  ! & < >
00112      9  "
00113     10  The ultimate end.
00114 */
00115     ,{    10,255,255,255,255,255,255,255,255,  6,  6,255,255,  6,255,255,
00116          255,255,255,255,255,255,255,255,255,255, 10,255,255,255,255,255,
00117            6,  8,  9,  2,  2,  5,  8,  2,  4,  4,  5,  5,  4,  5,  2,  5,
00118            1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  5,  4,  8,  4,  8,  2,
00119          255,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
00120            0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  3,255,  3,  5,  2,
00121          255,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
00122            0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  7,  7,  7,255,255,
00123          255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,
00124          255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,
00125          255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,
00126          255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,
00127          255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,
00128          255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,
00129          255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,
00130          255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255 }
00131 };
00132
00133 ALLGLOBALS A;
00134 #ifdef WITHPTHREADS
00135 ALLPRIVATES **AB;
00136 #endif
00137
00138 static struct fixedfun {
00139     char *name;
00140     int      commu ,tensor           ,complx          ,symmetric;
00141 } fixedfunctions[] = {
00142      {"exp_"     ,1 ,0                ,0               ,0} /* EXPONENT */
00143     ,{"denom_"   ,1 ,0                ,0               ,0} /* DENOMINATOR */
00144     ,{"setfun_"  ,1 ,0                ,0               ,0} /* SETFUNCTION */
00145     ,{"g_"       ,1 ,GAMMAFUNCTION    ,VARTYPEIMAGINARY,0} /* GAMMA */
00146     ,{"gi_"      ,1 ,GAMMAFUNCTION    ,VARTYPEIMAGINARY,0} /* GAMMAI */
00147     ,{"g5_"      ,1 ,GAMMAFUNCTION    ,VARTYPEIMAGINARY,0} /* GAMMAFIVE */
00148     ,{"g6_"      ,1 ,GAMMAFUNCTION    ,VARTYPEIMAGINARY,0} /* GAMMASIX */
00149     ,{"g7_"      ,1 ,GAMMAFUNCTION    ,VARTYPEIMAGINARY,0} /* GAMMASEVEN */
00150     ,{"sum_"     ,1 ,0                ,0               ,0} /* SUMF1 */
```

```
00151     ,{"sump_"     ,1 ,0                ,0                ,0} /* SUMF2 */
00152     ,{"dum_"      ,1 ,0                ,0                ,0} /* DUMFUN */
00153     ,{"replace_" ,0 ,0                 ,0                ,0} /* REPLACEMENT */
00154     ,{"reverse_" ,1 ,0                 ,0                ,0} /* REVERSEFUNCTION */
00155     ,{"distrib_" ,1 ,0                 ,0                ,0} /* DISTRIBUTION */
00156     ,{"dd_"       ,0 ,TENSORFUNCTION   ,0                ,0} /* DELTA3 */
00157     ,{"dummy_"    ,0 ,0                ,0                ,0} /* DUMMYFUN */
00158     ,{"dummyten_",0 ,TENSORFUNCTION    ,0                ,0} /* DUMMYTEN */
00159     ,{"e_"        ,0 ,TENSORFUNCTION|1 ,VARTYPEIMAGINARY,ANTISYMMETRIC}
00160                                                             /* LEVICIVITA */
00161     ,{"fac_"      ,0 ,0                ,0                ,0} /* FACTORIAL */
00162     ,{"invfac_"   ,0 ,0                ,0                ,0} /* INVERSEFACTORIAL */
00163     ,{"binom_"    ,0 ,0                ,0                ,0} /* BINOMIAL */
00164     ,{"nargs_"    ,0 ,0                ,0                ,0} /* NUMARGSFUN */
00165     ,{"sign_"     ,0 ,0                ,0                ,0} /* SIGNFUN */
00166     ,{"mod_"      ,0 ,0                ,0                ,0} /* MODFUNCTION */
00167     ,{"mod2_"     ,0 ,0                ,0                ,0} /* MOD2FUNCTION */
00168     ,{"min_"      ,0 ,0                ,0                ,0} /* MINFUNCTION */
00169     ,{"max_"      ,0 ,0                ,0                ,0} /* MAXFUNCTION */
00170     ,{"abs_"      ,0 ,0                ,0                ,0} /* ABSFUNCTION */
00171     ,{"sig_"      ,0 ,0                ,0                ,0} /* SIGFUNCTION */
00172     ,{"integer_" ,0 ,0                 ,0                ,0} /* INTFUNCTION */
00173     ,{"theta_"    ,0 ,0                ,0                ,0} /* THETA */
00174     ,{"thetap_"   ,0 ,0                ,0                ,0} /* THETA2 */
00175     ,{"delta_"    ,0 ,0                ,0                ,0} /* DELTA2 */
00176     ,{"deltap_"   ,0 ,0                ,0                ,0} /* DELTAP */
00177     ,{"bernoulli_",0,0                 ,0                ,0} /* BERNOULLIFUNCTION */
00178     ,{"count_"    ,0 ,0                ,0                ,0} /* COUNTFUNCTION */
00179     ,{"match_"    ,0 ,0                ,0                ,0} /* MATCHFUNCTION */
00180     ,{"pattern_" ,0 ,0                 ,VARTYPECOMPLEX   ,0} /* PATTERNFUNCTION */
00181     ,{"term_"     ,1 ,0                ,0                ,0} /* TERMFUNCTION */
00182     ,{"conjg_"    ,1 ,0                ,VARTYPECOMPLEX   ,0} /* CONJUGATEFUNCTION */
00183     ,{"root_"     ,0 ,0                ,0                ,0} /* ROOTFUNCTION */
00184     ,{"table_"    ,1 ,0                ,0                ,0} /* TABLEFUNCTION */
00185     ,{"firstbracket_",0 ,0             ,0                ,0} /* FIRSTBRACKET */
00186     ,{"termsin_" ,0 ,0                 ,0                ,0} /* TERMSINEXPR */
00187     ,{"nterms_"   ,0 ,0                ,0                ,0} /* NUMTERMSFUN */
00188     ,{"gcd_"      ,0 ,0                ,0                ,0} /* GCDFUNCTION */
00189     ,{"div_"        ,0 ,0              ,0                ,0} /* DIVFUNCTION */
00190     ,{"rem_"        ,0 ,0              ,0                ,0} /* REMFUNCTION */
00191     ,{"maxpowerof_",0 ,0               ,0                ,0} /* MAXPOWEROF */
00192     ,{"minpowerof_",0 ,0               ,0                ,0} /* MINPOWEROF */
00193     ,{"tbl_"      ,0 ,0                ,0                ,0} /* TABLESTUB */
00194     ,{"factorin_",0 ,0                 ,0                ,0} /* FACTORIN */
00195     ,{"termsinbracket_",0 ,0           ,0                ,0} /* TERMSINBRACKET */
00196     ,{"farg_"     ,0 ,0                ,0                ,0} /* WILDARGFUN */
00197 /*
00198      The following names are reserved for the floating point library.
00199      As long as we have no floating point numbers they do not do anything.
00200 */
00201     ,{"sqrt_"     ,0 ,0                ,0                ,0} /* SQRTFUNCTION */
00202     ,{"ln_"       ,0 ,0                ,0                ,0} /* LNFUNCTION */
00203     ,{"sin_"      ,0 ,0                ,0                ,0} /* SINFUNCTION */
00204     ,{"cos_"      ,0 ,0                ,0                ,0} /* COSFUNCTION */
00205     ,{"tan_"      ,0 ,0                ,0                ,0} /* TANFUNCTION */
00206     ,{"asin_"     ,0 ,0                ,0                ,0} /* ASINFUNCTION */
00207     ,{"acos_"     ,0 ,0                ,0                ,0} /* ACOSFUNCTION */
00208     ,{"atan_"     ,0 ,0                ,0                ,0} /* ATANFUNCTION */
00209     ,{"atan2_"    ,0 ,0                ,0                ,0} /* ATAN2FUNCTION */
00210     ,{"sinh_"     ,0 ,0                ,0                ,0} /* SINHFUNCTION */
00211     ,{"cosh_"     ,0 ,0                ,0                ,0} /* COSHFUNCTION */
00212     ,{"tanh_"     ,0 ,0                ,0                ,0} /* TANHFUNCTION */
00213     ,{"asinh_"    ,0 ,0                ,0                ,0} /* ASINHFUNCTION */
00214     ,{"acosh_"    ,0 ,0                ,0                ,0} /* ACOSHFUNCTION */
00215     ,{"atanh_"    ,0 ,0                ,0                ,0} /* ATANHFUNCTION */
00216     ,{"li2_"      ,0 ,0                ,0                ,0} /* LI2FUNCTION */
00217     ,{"lin_"      ,0 ,0                ,0                ,0} /* LINFUNCTION */
00218 /*
00219      From here on we continue with new functions (26-sep-2010)
00220 */
00221     ,{"extrasymbol_",0 ,0              ,0                ,0} /* EXTRASYMFUN */
00222     ,{"random_"     ,0 ,0              ,0                ,0} /* RANDOMFUNCTION */
00223     ,{"ranperm_"    ,1 ,0              ,0                ,0} /* RANPERM */
00224     ,{"numfactors_" ,0 ,0              ,0                ,0} /* NUMFACTORS */
00225     ,{"firstterm_"  ,0 ,0              ,0                ,0} /* FIRSTTERM */
00226     ,{"content_"    ,0 ,0              ,0                ,0} /* CONTENTTERM */
00227     ,{"prime_"      ,0 ,0              ,0                ,0} /* PRIMENUMBER */
00228     ,{"exteuclidean_",0 ,0             ,0                ,0} /* EXTEUCLIDEAN */
00229     ,{"makerational_",0 ,0             ,0                ,0} /* MAKERATIONAL */
00230     ,{"inverse_"    ,0 ,0              ,0                ,0} /* INVERSEFUNCTION */
00231     ,{"id_"         ,1 ,0              ,0                ,0} /* IDFUNCTION */
00232     ,{"putfirst_"   ,1 ,0              ,0                ,0} /* PUTFIRST */
00233     ,{"perm_"       ,1 ,0              ,0                ,0} /* PERMUTATIONS */
00234     ,{"partitions_" ,1 ,0              ,0                ,0} /* PARTITIONS */
00235     ,{"mul_"        ,0 ,0              ,0                ,0} /* MULFUNCTION */
00236     ,{"moebius_"    ,0 ,0              ,0                ,0} /* MOEBIUS */
00237     ,{"topologies_" ,0 ,0              ,0                ,0} /* TOPOLOGIES */
```

```
00238     ,{"diagrams_"   ,0 ,0              ,0              ,0} /* DIAGRAMS */
00239     ,{"topo_"       ,0 ,0              ,0              ,0} /* TOPO */
00240     ,{"node_"       ,0 ,0              ,0              ,0} /* VERTEX */
00241     ,{"edge_"       ,0 ,0              ,0              ,0} /* EDGE */
00242     ,{"sizeof_"     ,0 ,0              ,0              ,0} /* SIZEOFFUNCTION */
00243     ,{"block_"      ,0 ,0              ,0              ,0} /* BLOCK */
00244     ,{"onepi_"      ,0 ,0              ,0              ,0} /* ONEPI */
00245     ,{"phi_"        ,0 ,VERTEXFUNCTION,0              ,0} /* PHI */
00246     ,{"float_"      ,0 ,0              ,0              ,0} /* FLOATFUN */
00247     ,{"tofloat_"    ,0 ,0              ,0              ,0} /* TOFLOAT */
00248     ,{"torat_"      ,0 ,0              ,0              ,0} /* TORAT */
00249     ,{"mzv_"        ,0 ,0              ,0              ,0} /* MZV */
00250     ,{"euler_"      ,0 ,0              ,0              ,0} /* EULER */
00251     ,{"mzvhalf_"    ,0 ,0              ,0              ,0} /* MZVHALF */
00252     ,{"agm_"        ,0 ,0              ,0              ,0} /* AGMFUNCTION */
00253     ,{"gamma_"      ,0 ,0              ,0              ,0} /* GAMMAFUN */
00254     ,{"eexp_"       ,0 ,0              ,0              ,0} /* EXPFUNCTION */
00255     ,{"hpl_"        ,0 ,0              ,0              ,0} /* HPLFUNCTION */
00256     ,{"mpl_"        ,0 ,0              ,0              ,0} /* MPLFUNCTION */
00257 };
00258
00259 FIXEDSET fixedsets[] = {
00260      {"pos_",    "integers > 0",  CSYMBOL, 0}  /* POS_    0  */
00261     ,{"pos0_",   "integers >= 0", CSYMBOL, 0}  /* POS0_   1  */
00262     ,{"neg_",    "integers < 0",  CSYMBOL, 0}  /* NEG_    2  */
00263     ,{"neg0_",   "integers <= 0", CSYMBOL, 0}  /* NEG0_   3  */
00264     ,{"even_",   "even integers", CSYMBOL, 0}  /* EVEN_   4  */
00265     ,{"odd_",    "odd integers",  CSYMBOL, 0}  /* ODD_    5  */
00266     ,{"int_",    "all integers",  CSYMBOL, 0}  /* Z_      6  */
00267     ,{"symbol_","only symbols",  CSYMBOL, MAXPOSITIVE}  /* SYMBOL_ 7  */
00268     ,{"fixed_", "fixed indices", CINDEX, 0}   /* FIXED_  8  */
00269     ,{"index_", "all indices",   CINDEX, 0}   /* INDEX_  9  */
00270     ,{"number_","all rationals", CSYMBOL, 0}  /* Q_     10  */
00271     ,{"dummyindices_", "dummy indices", CINDEX, 0}   /* DUMMYINDEX_ 11  */
00272     ,{"vector_","only vectors",  CVECTOR, 0}  /* VECTOR_ 12  */
00273 };
00274
00275 UBYTE BufferForOutput[MAXLINELENGTH+14];
00276
00277 char *setupfilename = "form.set";
00278
00279 #ifdef WITHPTHREADS
00280 INILOCK(ErrorMessageLock)
00281 INILOCK(FileReadLock)
00282 #endif
```

4.68 lus.c File Reference

`#include "form3.h"`
Include dependency graph for lus.c:



Functions

- int Lus (WORD *term, WORD funnum, WORD loopsize, WORD numargs, WORD outfun, WORD mode)
- int FindLus (int from, int level, int openindex)
- int SortTheList (int *slist, int num)
- int AllLoops (PHEAD WORD *term, WORD level)
- LONG StartLoops (PHEAD WORD *term, WORD level, LONG vert, WORD nvert, WORD *arglist, WORD nargs, WORD *loop, WORD nloop)

- LONG GenLoops (PHEAD WORD *term, WORD level, LONG vert, WORD nvert, WORD *arglist, WORD nargs, WORD *loop, WORD nloop)
- void LoopOutput (PHEAD WORD *term, WORD level, WORD *loop, WORD nloop)
- int AllPaths (PHEAD WORD *term, WORD level)
- LONG GenPaths (PHEAD WORD *term, WORD level, LONG vert, WORD nvert, WORD *arglist, WORD nargs, WORD *path, WORD npath)
- void PathOutput (PHEAD WORD *term, WORD level, WORD *path, WORD npath)
- WORD AllOnePI (WORD *term, WORD level)
- int RemoveBridges (void)
- int TakeOneLine (WORD *term, WORD level)
- int OutputOnePI (PHEAD WORD *term, WORD level)

4.68.1 Detailed Description

Routines to find loops in index contractions. These routines allow for a category of topological statements. They were originally developed for the color library.

Definition in file lus.c.

4.68.2 Function Documentation

4.68.2.1 Lus()

```
int Lus (
            WORD * term,
            WORD funnum,
            WORD loopsize,
            WORD numargs,
            WORD outfun,
            WORD mode )
```

Definition at line 57 of file lus.c.

4.68.2.2 FindLus()

```
int FindLus (
            int from,
            int level,
            int openindex )
```

Definition at line 515 of file lus.c.

4.68.2.3 SortTheList()

```
int SortTheList (
            int * slist,
            int num )
```

Definition at line 578 of file lus.c.

4.68.2.4 AllLoops()

```
int AllLoops (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 650 of file lus.c.

4.68.2.5 StartLoops()

```
LONG StartLoops (
            PHEAD WORD * term,
            WORD level,
            LONG vert,
            WORD nvert,
            WORD * arglist,
            WORD nargs,
            WORD * loop,
            WORD nloop )
```

Definition at line 894 of file lus.c.

4.68.2.6 GenLoops()

```
LONG GenLoops (
            PHEAD WORD * term,
            WORD level,
            LONG vert,
            WORD nvert,
            WORD * arglist,
            WORD nargs,
            WORD * loop,
            WORD nloop )
```

Definition at line 982 of file lus.c.

4.68.2.7 LoopOutput()

```
void LoopOutput (
            PHEAD WORD * term,
            WORD level,
            WORD * loop,
            WORD nloop )
```

Definition at line 1061 of file lus.c.

4.68.2.8 AllPaths()

```
int AllPaths (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 1125 of file lus.c.

4.68.2.9 GenPaths()

```
LONG GenPaths (
             PHEAD WORD * term,
             WORD level,
             LONG vert,
             WORD nvert,
             WORD * arglist,
             WORD nargs,
             WORD * path,
             WORD npath )
```

Definition at line 1415 of file lus.c.

4.68.2.10 PathOutput()

```
void PathOutput (
             PHEAD WORD * term,
             WORD level,
             WORD * path,
             WORD npath )
```

Definition at line 1488 of file lus.c.

4.68.2.11 AllOnePI()

```
WORD AllOnePI (
             WORD * term,
             WORD level )
```

Definition at line 1568 of file lus.c.

4.68.2.12 RemoveBridges()

```
int RemoveBridges (
             void  )
```

Definition at line 1588 of file lus.c.

4.68.2.13 TakeOneLine()

```
int TakeOneLine (
             WORD * term,
             WORD level )
```

Definition at line 1598 of file lus.c.

4.68.2.14 OutputOnePI()

```
int OutputOnePI (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 1610 of file lus.c.

4.69 lus.c

Go to the documentation of this file.

```
00001
00007 /* #[ License : */
00008 /*
00009  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00010  *   When using this file you are requested to refer to the publication
00011  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00012  *   This is considered a matter of courtesy as the development was paid
00013  *   for by FOM the Dutch physics granting agency and we would like to
00014  *   be able to track its scientific use to convince FOM of its value
00015  *   for the community.
00016  *
00017  *   This file is part of FORM.
00018  *
00019  *   FORM is free software: you can redistribute it and/or modify it under the
00020  *   terms of the GNU General Public License as published by the Free Software
00021  *   Foundation, either version 3 of the License, or (at your option) any later
00022  *   version.
00023  *
00024  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00025  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00026  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00027  *   details.
00028  *
00029  *   You should have received a copy of the GNU General Public License along
00030  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00031  */
00032 /* #] License : */
00033 /*
00034     #[ Includes : lus.c
00035 */
00036
00037 #include "form3.h"
00038
00039 /*
00040     #] Includes :
00041     #[ Lus :
00042
00043     Routine to find loops.
00044     Mode: 0: Just tell whether there is such a loop.
00045           1: Take out the functions and replace by outfun with the
00046              remaining arguments of the loop function
00047           > AM.OffsetIndex: This index must be included in the loop.
00048           < -AM.OffsetIndex: This index must be included in the loop. Replace.
00049     Return value: 0: no loop. 1: there is/was such a loop.
00050     funnum: the function(s) in which we look for a loop.
00051     numargs: the number of arguments admissible in the function.
00052     outfun: the output function in case of a substitution.
00053     loopsize: the size of the loop we are looking for.
00054               if < 0 we look for all loops.
00055 */
00056
00057 int Lus(WORD *term, WORD funnum, WORD loopsize, WORD numargs, WORD outfun, WORD mode)
00058 {
00059     GETIDENTITY
00060     WORD *w, *t, *tt, *m, *r, **loc, *tstop, minloopsize;
00061     int nfun, i, j, jj, k, n, sign = 0, action = 0, L, ten, ten2, totnum,
00062     sign2, *alist, *wi, mini, maxi, medi = 0;
00063     if ( numargs <= 1 ) return(0);
00064     GETSTOP(term,tstop);
00065 /*
00066     First count the number of functions with the proper number of arguments.
00067 */
00068     t = term+1; nfun = 0;
00069     if ( ( ten = functions[funnum-FUNCTION].spec ) >= TENSORFUNCTION ) {
00070         while ( t < tstop ) {
00071             if ( *t == funnum && t[1] == FUNHEAD+numargs ) { nfun++; }
```

```
00072             t += t[1];
00073         }
00074     }
00075     else {
00076         while ( t < tstop ) {
00077             if ( *t == funnum ) {
00078                 i = 0; m = t+FUNHEAD; t += t[1];
00079                 while ( m < t ) { i++; NEXTARG(m) }
00080                 if ( i == numargs ) nfun++;
00081             }
00082             else t += t[1];
00083         }
00084     }
00085     if ( loopsize < 0 ) minloopsize = 2;
00086     else                minloopsize = loopsize;
00087     if ( funnum < minloopsize ) return(0); /* quick abort */
00088     if ( ((functions[funnum-FUNCTION].symmetric) & ~REVERSEORDER) == ANTISYMMETRIC ) sign = 1;
00089     if ( mode == 1 || mode < 0 ) {
00090         ten2 = functions[outfun-FUNCTION].spec >= TENSORFUNCTION;
00091     }
00092     else ten2 = -1;
00093 /*
00094     Allocations:
00095 */
00096     if ( AN.numflocs < funnum ) {
00097         if ( AN.funlocs ) M_free(AN.funlocs,"Lus: AN.funlocs");
00098         AN.numflocs = funnum;
00099         AN.funlocs = (WORD **)Malloc1(sizeof(WORD *)*AN.numflocs,"Lus: AN.funlocs");
00100     }
00101     if ( AN.numfargs < funnum*numargs ) {
00102         if ( AN.funargs ) M_free(AN.funargs,"Lus: AN.funargs");
00103         AN.numfargs = funnum*numargs;
00104         AN.funargs = (int *)Malloc1(sizeof(int *)*AN.numfargs,"Lus: AN.funargs");
00105     }
00106 /*
00107     Make a list of relevant indices
00108 */
00109     alist = AN.funargs; loc = AN.funlocs;
00110     t = term+1;
00111     if ( ten >= TENSORFUNCTION ) {
00112         while ( t < tstop ) {
00113             if ( *t == funnum && t[1] == FUNHEAD+numargs ) {
00114                 *loc++ = t;
00115                 t += FUNHEAD;
00116                 j = i = numargs; while ( --i >= 0 ) {
00117                     if ( *t >= AM.OffsetIndex &&
00118                         ( *t >= AM.OffsetIndex+WILDOFFSET ||
00119                         indices[*t-AM.OffsetIndex].dimension != 0 ) ) {
00120                         *alist++ = *t++; j--;
00121                     }
00122                     else t++;
00123                 }
00124                 while ( --j >= 0 ) *alist++ = -1;
00125             }
00126             else t += t[1];
00127         }
00128     }
00129     else {
00130         nfun = 0;
00131         while ( t < tstop ) {
00132             if ( *t == funnum ) {
00133                 w = t;
00134                 i = 0; m = t+FUNHEAD; t += t[1];
00135                 while ( m < t ) { i++; NEXTARG(m) }
00136                 if ( i == numargs ) {
00137                     m = w + FUNHEAD;
00138                     while ( m < t ) {
00139                         if ( *m == -INDEX && m[1] >= AM.OffsetIndex &&
00140                         ( m[1] >= AM.OffsetIndex+WILDOFFSET ||
00141                         indices[m[1]-AM.OffsetIndex].dimension != 0 ) ) {
00142                             *alist++ = m[1]; m += 2; i--;
00143                         }
00144                         else if ( ten2 >= TENSORFUNCTION && *m != -INDEX
00145                          && *m != -VECTOR && *m != -MINVECTOR &&
00146                         ( *m != -SNUMBER || *m < 0 || *m >= AM.OffsetIndex ) ) {
00147                             i = numargs; break;
00148                         }
00149                         else { NEXTARG(m) }
00150                     }
00151                     if ( i < numargs ) {
00152                         *loc++ = w;
00153                         nfun++;
00154                         while ( --i >= 0 ) *alist++ = -1;
00155                     }
00156                 }
00157             }
00158             else t += t[1];
```

```
00159         }
00160         if ( nfun < minloopsize ) return(0);
00161     }
00162 /*
00163     We have now nfun objects. Not all indices may be usable though.
00164     If the list is not long, we use a quadratic algorithm to remove
00165     indices and vertices that cannot be used. If it becomes large we
00166     sort the list of available indices (and their multiplicity) and
00167     work with binary searches.
00168 */
00169     alist = AN.funargs; totnum = numargs*nfun;
00170     if ( nfun > 7 ) {
00171         if ( AN.funisize < totnum ) {
00172             if ( AN.funinds ) M_free(AN.funinds,"AN.funinds");
00173             AN.funisize = (totnum*3)/2;
00174             AN.funinds = (int *)Malloc1(AN.funisize*2*sizeof(int),"AN.funinds");
00175         }
00176         i = totnum; n = 0; wi = AN.funinds;
00177         while ( --i >= 0 ) {
00178             if ( *alist >= 0 ) { n++; *wi++ = *alist; *wi++ = 1; }
00179             alist++;
00180         }
00181         n = SortTheList(AN.funinds,n);
00182         do {
00183             action = 0;
00184             for ( i = 0; i < nfun; i++ ) {
00185                 alist = AN.funargs + i*numargs;
00186                 jj = numargs;
00187                 for ( j = 0; j < jj; j++ ) {
00188                     if ( alist[j] < 0 ) break;
00189                     mini = 0; maxi = n-1;
00190                     while ( mini <= maxi ) {
00191                         medi = (mini + maxi) / 2; k = AN.funinds[2*medi];
00192                         if ( alist[j] > k ) mini = medi + 1;
00193                         else if ( alist[j] < k ) maxi = medi - 1;
00194                         else break;
00195                     }
00196                     if ( AN.funinds[2*medi+1] <= 1 ) {
00197                         (AN.funinds[2*medi+1])--;
00198                         jj--; k = j; while ( k < jj ) { alist[k] = alist[k+1]; k++; }
00199                         alist[jj] = -1; j--;
00200                     }
00201                 }
00202                 if ( jj < 2 ) {
00203                     if ( jj == 1 ) {
00204                         mini = 0; maxi = n-1;
00205                         while ( mini <= maxi ) {
00206                             medi = (mini + maxi) / 2; k = AN.funinds[2*medi];
00207                             if ( alist[0] > k ) mini = medi + 1;
00208                             else if ( alist[0] < k ) maxi = medi - 1;
00209                             else break;
00210                         }
00211                         (AN.funinds[2*medi+1])--;
00212                         if ( AN.funinds[2*medi+1] == 1 ) action++;
00213                     }
00214                     nfun--; totnum -= numargs; AN.funlocs[i] = AN.funlocs[nfun];
00215                     wi = AN.funargs + nfun*numargs;
00216                     for ( j = 0; j < numargs; j++ ) alist[j] = *wi++;
00217                     i--;
00218                 }
00219             }
00220         } while ( action );
00221     }
00222     else {
00223         for ( i = 0; i < totnum; i++ ) {
00224             if ( alist[i] == -1 ) continue;
00225             for ( j = 0; j < totnum; j++ ) {
00226                 if ( alist[j] == alist[i] && j != i ) break;
00227             }
00228             if ( j >= totnum ) alist[i] = -1;
00229         }
00230         do {
00231             action = 0;
00232             for ( i = 0; i < nfun; i++ ) {
00233                 alist = AN.funargs + i*numargs;
00234                 n = numargs;
00235                 for ( k = 0; k < n; k++ ) {
00236                     if ( alist[k] < 0 ) { alist[k--] = alist[--n]; alist[n] = -1; }
00237                 }
00238                 if ( n <= 1 ) {
00239                     if ( n == 1 ) { j = alist[0]; }
00240                     else j = -1;
00241                     nfun--; totnum -= numargs; AN.funlocs[i] = AN.funlocs[nfun];
00242                     wi = AN.funargs + nfun * numargs;
00243                     for ( k = 0; k < numargs; k++ ) alist[k] = wi[k];
00244                     i--;
00245                     if ( j >= 0 ) {
```

```
00246                         for ( k = 0, jj = 0, wi = AN.funargs; k < totnum; k++, wi++ ) {
00247                             if ( *wi == j ) { jj++; if ( jj > 1 ) break; }
00248                         }
00249                         if ( jj <= 1 ) {
00250                             for ( k = 0, wi = AN.funargs; k < totnum; k++, wi++ ) {
00251                                 if ( *wi == j ) { *wi = -1; action = 1; }
00252                             }
00253                         }
00254                     }
00255                 }
00256             }
00257         } while ( action );
00258     }
00259     if ( nfun < minloopsize ) return(0);
00260 /*
00261     Now we have nfun objects, each with at least 2 indices, each of which
00262     occurs at least twice in our list. There will be a loop!
00263 */
00264     if ( mode != 0 && mode != 1 ) {
00265         if ( mode > 0 ) AN.tohunt =  mode - 5;
00266         else            AN.tohunt = -mode - 5;
00267         AN.nargs = numargs; AN.numoffuns = nfun;
00268         i = 0;
00269         if ( loopsize < 0 ) {
00270             if ( loopsize == -1 ) k = nfun;
00271             else { k = -loopsize-1; if ( k > nfun ) k = nfun; }
00272             for ( L = 2; L <= k; L++ ) {
00273                 if ( FindLus(0,L,AN.tohunt) ) goto Success;
00274             }
00275         }
00276         else if ( FindLus(0,loopsize,AN.tohunt) ) { L = loopsize; goto Success; }
00277     }
00278     else {
00279         AN.nargs = numargs; AN.numoffuns = nfun;
00280         if ( loopsize < 0 ) {
00281             jj = 2; k = nfun;
00282             if ( loopsize < -1 ) { k = -loopsize-1; if ( k > nfun ) k = nfun; }
00283         }
00284         else { jj = k = loopsize; }
00285         for ( L = jj; L <= k; L++ ) {
00286             for ( i = 0; i <= nfun-L; i++ ) {
00287                 alist = AN.funargs + i * numargs;
00288                 for ( jj = 0; jj < numargs; jj++ ) {
00289                     if ( alist[jj] < 0 ) continue;
00290                     AN.tohunt = alist[jj];
00291                     for ( j = jj+1; j < numargs; j++ ) {
00292                         if ( alist[j] < 0 ) continue;
00293                         if ( FindLus(i+1,L-1,alist[j]) ) {
00294                             alist[0] = alist[jj];
00295                             alist[1] = alist[j];
00296                             goto Success;
00297                         }
00298                     }
00299                 }
00300             }
00301         }
00302     }
00303     return(0);
00304 Success:;
00305     if ( mode == 0 || mode > 1 ) return(1);
00306 /*
00307     Now we have to make the replacement and fix the potential sign
00308 */
00309     sign2 = 1;
00310     wi = AN.funargs + i*numargs; loc = AN.funlocs + i;
00311     for ( k = 0; k < L; k++ ) *(loc[k]) = -1;
00312     if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
00313     w = AT.WorkPointer + 1;
00314     m = t = term + 1;
00315     while ( t < tstop ) {
00316         if ( *t == -1 ) break;
00317         t += t[1];
00318     }
00319     while ( m < t ) *w++ = *m++;
00320     r = w;
00321     *w++ = outfun;
00322     w++;
00323     *w++ = DIRTYFLAG;
00324     FILLFUN3(w)
00325     if ( functions[outfun-FUNCTION].spec >= TENSORFUNCTION ) {
00326         if ( ten >= TENSORFUNCTION ) {
00327             for ( i = 0; i < L; i++ ) {
00328                 alist = wi + i*numargs;
00329                 m = loc[i] + FUNHEAD;
00330                 for ( k = 0; k < numargs; k++ ) {
00331                     if ( m[k] == alist[0] ) {
00332                         if ( k != 0 ) {
```

```
00333                               jj = m[k]; m[k] = m[0]; m[0] = jj;
00334                               sign = -sign;
00335                           }
00336                           break;
00337                       }
00338                   }
00339                   for ( k = 1; k < numargs; k++ ) {
00340                       if ( m[k] == alist[1] ) {
00341                           if ( k != 1 ) {
00342                               jj = m[k]; m[k] = m[1]; m[1] = jj;
00343                               sign = -sign;
00344                           }
00345                           break;
00346                       }
00347                   }
00348                   m += 2;
00349                   for ( k = 2; k < numargs; k++ ) *w++ = *m++;
00350               }
00351           }
00352           else {
00353               WORD *t1, *t2, *t3;
00354               for ( i = 0; i < L; i++ ) {
00355                   alist = wi + i*numargs;
00356                   tt = loc[i];
00357                   m = tt + FUNHEAD;
00358                   for ( k = 0; k < numargs; k++ ) {
00359                       if ( *m == -INDEX && m[1] == alist[0] ) {
00360                           if ( k != 0 ) {
00361                               if ( ( k & 1 ) != 0 ) sign = -sign;
00362 /*
00363                               now move to position 0
00364 */
00365                               t2 = m+2; t1 = m; t3 = tt+FUNHEAD;
00366                               while ( t1 > t3 ) { *--t2 = *--t1; }
00367                               t3[0] = -INDEX; t3[1] = alist[0];
00368                           }
00369                           break;
00370                       }
00371                       NEXTARG(m)
00372                   }
00373                   m = tt + FUNHEAD + 2;
00374                   for ( k = 1; k < numargs; k++ ) {
00375                       if ( *m == -INDEX && m[1] == alist[1] ) {
00376                           if ( k != 1 ) {
00377                               if ( ( k & 1 ) == 0 ) sign = -sign;
00378 /*
00379                               now move to position 1
00380 */
00381                               t2 = m+2; t1 = m; t3 = tt+FUNHEAD+2;
00382                               while ( t1 > t3 ) { *--t2 = *--t1; }
00383                               t3[0] = -INDEX; t3[1] = alist[1];
00384                           }
00385                           break;
00386                       }
00387                       NEXTARG(m)
00388                   }
00389 /*
00390                   now copy the remaining arguments to w
00391                   keep in mind that the output function is a tensor!
00392 */
00393                   t1 = tt + FUNHEAD + 4;
00394                   t2 = tt + tt[1];
00395                   while ( t1 < t2 ) {
00396                       if ( *t1 == -INDEX || *t1 == -VECTOR ) {
00397                           *w++ = t1[1]; t1 += 2;
00398                       }
00399                       else if ( *t1 == -MINVECTOR ) {
00400                           *w++ = t1[1]; t1 += 2; sign2 = -sign2;
00401                       }
00402                       else if ( ( *t1 == -SNUMBER ) && ( t1[1] >= 0 ) && ( t1[1] < AM.OffsetIndex ) ) {
00403                           *w++ = t1[1]; t1 += 2; sign2 = -sign2;
00404                       }
00405                       else {
00406                           MLOCK(ErrorMessageLock);
00407                           MesPrint("Illegal attempt to use a non-index-like argument in a tensor in
      ReplaceLoop statement");
00408                           MUNLOCK(ErrorMessageLock);
00409                           Terminate(-1);
00410                       }
00411                   }
00412               }
00413           }
00414       }
00415       else {
00416           if ( ten >= TENSORFUNCTION ) {
00417               for ( i = 0; i < L; i++ ) {
00418                   alist = wi + i*numargs;
```

```
00419                 m = loc[i] + FUNHEAD;
00420                 for ( k = 0; k < numargs; k++ ) {
00421                     if ( m[k] == alist[0] ) {
00422                         if ( k != 0 ) {
00423                             jj = m[k]; m[k] = m[0]; m[0] = jj;
00424                             sign = -sign;
00425                             break;
00426                         }
00427                     }
00428                 }
00429                 for ( k = 1; k < numargs; k++ ) {
00430                     if ( m[k] == alist[1] ) {
00431                         if ( k != 1 ) {
00432                             jj = m[k]; m[k] = m[1]; m[1] = jj;
00433                             sign = -sign;
00434                             break;
00435                         }
00436                     }
00437                 }
00438                 m += 2;
00439                 for ( k = 2; k < numargs; k++ ) {
00440                     if ( *m >= AM.OffsetIndex ) { *w++ = -INDEX; }
00441                     else if ( *m < 0 ) { *w++ = -VECTOR; }
00442                     else { *w = -SNUMBER; }
00443                     *w++ = *m++;
00444                 }
00445             }
00446         }
00447         else {
00448             WORD *t1, *t2, *t3;
00449             for ( i = 0; i < L; i++ ) {
00450                 alist = wi + i*numargs;
00451                 tt = loc[i];
00452                 m = tt + FUNHEAD;
00453                 for ( k = 0; k < numargs; k++ ) {
00454                     if ( *m == -INDEX && m[1] == alist[0] ) {
00455                         if ( k != 0 ) {
00456                             if ( ( k & 1 ) != 0 ) sign = -sign;
00457 /*
00458                             now move to position 0
00459 */
00460                             t2 = m+2; t1 = m; t3 = tt+FUNHEAD;
00461                             while ( t1 > t3 ) { *--t2 = *--t1; }
00462                             t3[0] = -INDEX; t3[1] = alist[0];
00463                         }
00464                         break;
00465                     }
00466                     NEXTARG(m)
00467                 }
00468                 m = tt + FUNHEAD + 2;
00469                 for ( k = 1; k < numargs; k++ ) {
00470                     if ( *m == -INDEX && m[1] == alist[1] ) {
00471                         if ( k != 1 ) {
00472                             if ( ( k & 1 ) == 0 ) sign = -sign;
00473 /*
00474                             now move to position 1
00475 */
00476                             t2 = m+2; t1 = m; t3 = tt+FUNHEAD+2;
00477                             while ( t1 > t3 ) { *--t2 = *--t1; }
00478                             t3[0] = -INDEX; t3[1] = alist[1];
00479                         }
00480                         break;
00481                     }
00482                     NEXTARG(m)
00483                 }
00484 /*
00485                 now copy the remaining arguments to w
00486 */
00487                 t1 = tt + FUNHEAD + 4;
00488                 t2 = tt + tt[1];
00489                 while ( t1 < t2 ) *w++ = *t1++;
00490             }
00491         }
00492     }
00493     r[1] = w-r;
00494     while ( t < tstop ) {
00495         if ( *t == -1 ) { t += t[1]; continue; }
00496         i = t[1];
00497         NCOPY(w,t,i)
00498     }
00499     tstop = term + *term;
00500     while ( t < tstop ) *w++ = *t++;
00501     if ( sign < 0 ) w[-1] = -w[-1];
00502     i = w - AT.WorkPointer;
00503     *AT.WorkPointer = i;
00504     t = term; w = AT.WorkPointer;
00505     NCOPY(t,w,i)
```

```
00506     *AN.RepPoint = 1;    /* For Repeat */
00507     return(1);
00508 }
00509
00510 /*
00511     #] Lus :
00512     #[ FindLus :
00513 */
00514
00515 int FindLus(int from, int level, int openindex)
00516 {
00517     GETIDENTITY
00518     int i, j, k, jj, *alist, *blist, *w, *m, partner;
00519     WORD **loc = AN.funlocs, *wor;
00520     if ( level == 1 ) {
00521         for ( i = from; i < AN.numoffuns; i++ ) {
00522             alist = AN.funargs + i*AN.nargs;
00523             for ( j = 0; j < AN.nargs; j++ ) {
00524                 if ( alist[j] == openindex ) {
00525                     for ( k = 0; k < AN.nargs; k++ ) {
00526                         if ( k == j ) continue;
00527                         if ( alist[k] == AN.tohunt ) {
00528                             loc[from] = loc[i];
00529                             alist = AN.funargs + from*AN.nargs;
00530                             alist[0] = openindex; alist[1] = AN.tohunt;
00531                             return(1);
00532                         }
00533                     }
00534                 }
00535             }
00536         }
00537     }
00538     else {
00539         for ( i = from; i < AN.numoffuns; i++ ) {
00540             alist = AN.funargs + i*AN.nargs;
00541             for ( j = 0; j < AN.nargs; j++ ) {
00542                 if ( alist[j] == openindex ) {
00543                     if ( from != i ) {
00544                         wor = loc[i]; loc[i] = loc[from]; loc[from] = wor;
00545                         blist = w = AN.funargs + from*AN.nargs;
00546                         m = alist;
00547                         k = AN.nargs;
00548                         while ( --k >= 0 ) { jj = *m; *m++ = *w; *w++ = jj; }
00549                     }
00550                     else blist = alist;
00551                     for ( k = 0; k < AN.nargs; k++ ) {
00552                         if ( k == j || blist[k] < 0 ) continue;
00553                         partner = blist[k];
00554                         if ( FindLus(from+1,level-1,partner) ) {
00555                             blist[0] = openindex; blist[1] = partner;
00556                             return(1);
00557                         }
00558                     }
00559                     if ( from != i ) {
00560                         wor = loc[i]; loc[i] = loc[from]; loc[from] = wor;
00561                         w = AN.funargs + from*AN.nargs;
00562                         m = alist;
00563                         k = AN.nargs;
00564                         while ( --k >= 0 ) { jj = *m; *m++ = *w; *w++ = jj; }
00565                     }
00566                 }
00567             }
00568         }
00569     }
00570     return(0);
00571 }
00572
00573 /*
00574     #] FindLus :
00575     #[ SortTheList :
00576 */
00577
00578 int SortTheList(int *slist, int num)
00579 {
00580     GETIDENTITY
00581     int i, nleft, nright, *t1, *t2, *t3, *rlist;
00582     if ( num <= 2 ) {
00583         if ( num <=  1 ) return(num);
00584         if ( slist[0] < slist[2] ) return(2);
00585         if ( slist[0] > slist[2] ) {
00586             i = slist[0]; slist[0] = slist[2]; slist[2] = i;
00587             i = slist[1]; slist[1] = slist[3]; slist[3] = i;
00588             return(2);
00589         }
00590         slist[1] += slist[3];
00591         return(1);
00592     }
```

```
00593     else {
00594         nleft = num/2; rlist = slist + 2*nleft;
00595         nright = SortTheList(rlist,num-nleft);
00596         nleft = SortTheList(slist,nleft);
00597         if ( AN.tlistsize < nleft ) {
00598             if ( AN.tlistbuf ) M_free(AN.tlistbuf,"AN.tlistbuf");
00599             AN.tlistsize = (nleft*3)/2;
00600             AN.tlistbuf = (int *)Malloc1(AN.tlistsize*2*sizeof(int),"AN.tlistbuf");
00601         }
00602         i = nleft; t1 = slist; t2 = AN.tlistbuf;
00603         while ( --i >= 0 ) { *t2++ = *t1++; *t2++ = *t1++; }
00604         i = nleft+nright; t1 = AN.tlistbuf; t2 = rlist; t3 = slist;
00605         while ( nleft > 0 && nright > 0 ) {
00606             if ( *t1 < *t2 ) {
00607                 *t3++ = *t1++; *t3++ = *t1++; nleft--;
00608             }
00609             else if ( *t1 > *t2 ) {
00610                 *t3++ = *t2++; *t3++ = *t2++; nright--;
00611             }
00612             else {
00613                 *t3++ = *t1++; t2++; *t3++ = (*t1++) + (*t2++); i--;
00614                 nleft--; nright--;
00615             }
00616         }
00617         while ( --nleft >= 0 ) { *t3++ = *t1++; *t3++ = *t1++; }
00618         while ( --nright >= 0 ) { *t3++ = *t2++; *t3++ = *t2++; }
00619         return(i);
00620     }
00621 }
00622
00623 /*
00624     #] SortTheList :
00625     #[ AllLoops :
00626
00627     Routine finds all possible loops that can be made in the
00628     arguments of the symmetric commuting vertex function v and creates
00629     for each loop a new term which is the original term times the loop.
00630     The occurrences of v can have different numbers of arguments.
00631     Each argument that occurs twice (and in different instances of v)
00632     in total will be considered.
00633
00634     The input parameters are in C->lhs[level] and are
00635         C->lhs[level][0]  TYPEALLLOOPS
00636         C->lhs[level][1]  6
00637         C->lhs[level][2]  Number of function v.
00638         C->lhs[level][3]  Number of the output function loop.
00639         C->lhs[level][4]  option1: the type of argument.
00640         C->lhs[level][5]  option2: 0,1: what to do if no loop.
00641     Eligible arguments must be of the type SYMBOL, VECTOR, INDEX or SNUMBER.
00642     They must all be of the same type, indicated by option1.
00643     Extra restriction: In a loop, each v can be visited at most once.
00644     If there is no loop at all, option2 determines whether the term
00645     remains unmodified, or is replaced by zero.
00646     Function v can be either a regular function or a tensor.
00647     To facilitate this we copy the relevant arguments into the workspace.
00648 */
00649
00650 int AllLoops(PHEAD WORD *term,WORD level)
00651 {
00652     CBUF *C = cbuf+AM.rbufnum;
00653     WORD vcode = C->lhs[level][2];    /* The input function */
00654     WORD option1 = C->lhs[level][4];  /* type of argument */
00655     WORD option2 = C->lhs[level][5];  /* what to do when no loop */
00656     WORD *tstop, *t, *tend, *tstart, *tfrom;
00657     WORD i, j, jj;
00658     WORD *arglist, nargs, *loop, nloop;
00659     WORD *oldworkpointer = AT.WorkPointer;
00660     LONG oldpworkpointer = AT.pWorkPointer;
00661     LONG numgenerated = 0, vert;
00662     WORD *a, *a1, *a2, *a3, *v, *vv, nvert, *to, *from, *tos, action;
00663 /*
00664     Search for the first occurrence of vcode.
00665 */
00666     tstop = term+*term; tstop -=  ABS(tstop[-1]);
00667     t = term + 1;
00668     while ( t < tstop && *t != vcode ) t += t[1];
00669     if ( t == tstop ) {
00670         if ( option2 == 0 ) return(0);
00671         else return(Generator(BHEAD term,level));
00672     }
00673     tstart = t;
00674     nvert = 0;
00675     do {
00676         t += t[1]; nvert++;
00677     } while ( t < tstop && *t == vcode );
00678     tend = t;
00679 /*
```

```
00680     Make room for 2*nvert pointers
00681 */
00682     WantAddPointers(2*nvert);
00683     vert = AT.pWorkPointer;
00684     AT.pWorkPointer += 2*nvert;
00685     nvert = 0;
00686 /*
00687     Next we copy these functions into the workspace, but only the arguments
00688     that agree with option1. Because of the difference between tensors and
00689     regular functions in the copy we strip the type of the argument.
00690 */
00691     to = AT.WorkPointer;
00692     from = tstart;
00693     if ( functions[vcode-FUNCTION].spec == TENSORFUNCTION ) {
00694         from = tstart;
00695         while ( from < tend ) {
00696             tos = to++;
00697             tfrom = from+from[1];
00698             from += FUNHEAD;
00699             while ( from < tfrom ) {
00700                 if ( option1 == -INDEX && *from >= 0 ) {
00701                     *to++ = *from++;
00702                 }
00703                 else if ( option1 == -VECTOR && *from < 0 ) {
00704                     *to++ = *from++ - AM.OffsetVector;
00705                 }
00706                 else {
00707                     from++;
00708                 }
00709             }
00710             *tos = to-from;
00711             if ( *tos < 3 ) to = tos;
00712             else AT.pWorkSpace[vert+nvert++] = tos;
00713         }
00714     }
00715     else if ( functions[vcode-FUNCTION].spec == 0 ) {
00716         from = tstart;
00717         while ( from < tend ) {
00718             tos = to++;
00719             tfrom = from + from[1];
00720             from += FUNHEAD;
00721             while ( from < tfrom ) {
00722                 if ( option1 == -VECTOR
00723                      && ( from[0] == -INDEX || from[0] == -VECTOR )
00724                      && from[1] < 0 ) {
00725                     from++;
00726                     *to++ = *from++ - AM.OffsetVector;
00727                 }
00728                 else if ( option1 == *from ) {
00729                     from++; *to++ = *from++;
00730                 }
00731                 else {
00732                     NEXTARG(from);
00733                 }
00734             }
00735             *tos = to-tos;
00736             if ( *tos < 3 ) to = tos;
00737             else AT.pWorkSpace[vert+nvert++] = tos;
00738         }
00739     }
00740     else {
00741         AT.WorkPointer = oldworkpointer;
00742         AT.pWorkPointer = oldpworkpointer;
00743         if ( option2 == 0 ) return(0);
00744         return(Generator(BHEAD term,level));
00745     }
00746     AT.WorkPointer = to;
00747 /*
00748     Now make a list of all relevant arguments.
00749 */
00750     a = arglist = AT.WorkPointer;
00751     for ( i = 0; i < nvert; i++ ) {
00752         v = AT.pWorkSpace[vert+i];
00753         j = *v-1; v++;
00754         NCOPY(a,v,j);
00755     }
00756     nargs = a-arglist;
00757     AT.WorkPointer = a;
00758 /*
00759     Now sort the list. Bubble type sort.
00760 */
00761     a1 = arglist; a2 = a1+1;
00762     while ( a2 < a ) {
00763         a3 = a2;
00764         while ( a3 > a1 && a3[-1] > a3[0] ) {
00765             EXCH(*a3,a3[-1]);
00766             a3--;
```

```
00767         }
00768         a2++;
00769     }
00770 /*
00771     Now remove the elements from the list that do not occur exactly twice.
00772 */
00773     a1 = arglist; a2 = a1; a3 = a1+nargs;
00774     while ( a2 < a3 ) {
00775         if ( a2+1 == a3 ) { break; }
00776         else if ( a2+2 == a3 ) {
00777             if ( a2[0] == a2[1] ) { *a1++ = a2[0]; }
00778             break;
00779         }
00780         else {
00781             if ( a2[0] != a2[1] ) { a2++; }
00782             else if ( a2[0] != a2[2] ) {
00783                 *a1++ = a2[0]; a2 += 2;
00784             }
00785             else {
00786                 a2++; while ( a2 < a3 && a2[-1] == a2[0] ) a2++;
00787             }
00788         }
00789     }
00790     nargs = a1-arglist;
00791 /*
00792     Now we need to redo the list of vertices and remove the elements
00793     that are not in our arglist.
00794 */
00795     do {
00796       action = 0;
00797       for ( i = 0; i < nvert; i++ ) {
00798         vv = v = AT.pWorkSpace[vert+i];
00799         v++;
00800         for ( j = 1; j < vv[0]; j++ ) {
00801             for ( jj = 0; jj < nargs; jj++ ) {
00802                 if ( *v == arglist[jj] ) break;
00803             }
00804             if ( jj >= nargs ) {    /* was not in the list */
00805                 vv[0] = vv[0]-1;
00806                 for ( jj = j; jj < vv[0]; jj++ ) vv[jj] = vv[jj+1];
00807             }
00808             v++;
00809         }
00810       }
00811 /*
00812       Next we need to remove vertices that have only one object remaining.
00813       Also, the object can be removed from arglist. After that we go back
00814       to clean up the list.
00815 */
00816       for ( i = 0; i < nvert; i++ ) {
00817         vv = AT.pWorkSpace[vert+i];
00818         if ( vv[0] == 1 ) {
00819             AT.pWorkSpace[vert+i] = AT.pWorkSpace[vert+nvert-1];
00820             nvert--; i--; continue;
00821         }
00822         else if ( vv[0] == 2 ) {
00823             for ( j = 0; j < nargs; j++ ) {
00824                 if ( arglist[j] == vv[1] ) break;
00825             }
00826             while ( j < nargs-1 ) arglist[j] = arglist[j+1];
00827             nargs--;
00828             AT.pWorkSpace[vert+i] = AT.pWorkSpace[vert+nvert-1];
00829             nvert--; i--;
00830             action = 1;
00831         }
00832       }
00833     } while ( action );
00834 /*
00835     Because the user can remove tadpoles rather easily as in
00836         id v(?a,i?,?b,i?,?c) = v(?a,?b,?c)
00837     there is no need to avoid them as potential loops.
00838
00839     At this point we are ready to look for loops.
00840     We have
00841         arglist,nargs   list of eligible objects.
00842         vert,nvert      position in pWorkSpace for vertices and their number.
00843                         The vertices themselves are in the WorkSpace.
00844         loop,nloop      The buildup of the loop.
00845 */
00846     loop = AT.WorkPointer;
00847     AT.WorkPointer += nargs;
00848     nloop = 0;
00849     numgenerated += StartLoops(BHEAD term,level,vert,nvert,arglist,nargs,loop,nloop);
00850     AT.WorkPointer = oldworkpointer;
00851     AT.pWorkPointer = oldpworkpointer;
00852
00853     if ( numgenerated == 0 && option2 != 0 ) return(Generator(BHEAD term,level));
```

```
00854     return(0);
00855 }
00856
00857 /*
00858     #] AllLoops :
00859     #[ StartLoops :
00860
00861     Algorithm.
00862     1: have a list of vertices and objects that can be part of the loops.
00863     2: starting with the first element of arglist, look for the two
00864        vertices that contain this object. The first is vstart.
00865     3: At the second, look at all possible objects to continue from here.
00866     4: For each, find the vertex with its partner.
00867     5: if the partner vertex is vstart, we have a loop. --> 9.
00868     6: The partner vertex is not allowed to be a vertex that we passed
00869        in the current build-up of the loop.
00870     7: Put the object in loop and increase nloop.
00871     8: Now sum over all allowed objects in the partner vertex. --> 4.
00872
00873     9: Create the function loop with the arguments from loop,nloop.
00874    10: If a reverse cyclic permutation gives a smaller result, drop this
00875        solution and continue with the last summing over objects at a vertex.
00876    11: If not, output the result by adding the function loop
00877        to the term and call Generator(term,level)
00878
00879    12: when all possibilities with the first element of arglist have been
00880        exhausted, raise arglist and lower nargs by one. --> 2
00881    13: when nargs == 1, we can stop.
00882
00883     The inclusion of a new object when we sum over the possibilities at a
00884     vertex can best be done in a recursion, because we have no idea how
00885     many nested loops we would otherwise need. Of course the recursion can
00886     be emulated, but that makes the algorithm rather messy.
00887
00888     We have two routines: StartLoops and GenLoops.
00889     StartLoops takes care of the first argument (and the summing over who
00890     is the first argument), while GenLoops treats an additional vertex until
00891     a loop is closed. GenLoops also sends completed loops off toe Generator.
00892 */
00893
00894 LONG StartLoops(PHEAD WORD *term,WORD level,LONG vert,WORD nvert,
00895                 WORD *arglist,WORD nargs,WORD *loop,WORD nloop)
00896 {
00897     LONG numgenerated = 0;
00898     WORD *v, *vv, *vstart, istart, *vpartner, ipartner, j;
00899     while ( nargs > 1 ) {
00900 /*
00901         Look for a vertex with arglist[0]. This is our starting vertex.
00902         Next we look for its partner and we put arglist[0] in loop.
00903 */
00904         nloop = 0;
00905         loop[nloop++] = arglist[0];
00906         for ( istart = 0; istart < nvert; istart++ ) {
00907             vstart = AT.pWorkSpace[vert+istart];
00908             v = vstart+1; vv = vstart + *vstart;
00909             do {
00910                 if ( *v == arglist[0] ) goto havestart;
00911                 v++;
00912             } while ( v < vv );
00913         }
00914 /*
00915         If we come here, we have a problem.
00916 */
00917 /* INTERNAL_ERROR_EXCL_START */
00918         MesPrint("!>Internal error in StartLoops. Object not found.");
00919         Terminate(-1);
00920         return(-1);
00921 /* INTERNAL_ERROR_EXCL_STOP */
00922 havestart:
00923         AT.pWorkSpace[vert+nvert] = vstart;
00924 /*
00925         Check for tadpole.
00926 */
00927         v++;
00928         while ( v < vv ) {
00929             if ( *v == arglist[0] ) {   /* tadpole */
00930                 LoopOutput(BHEAD term,level,loop,nloop);
00931                 numgenerated++;
00932                 goto nextarg;
00933             }
00934             v++;
00935         }
00936 /*
00937         Now the partner vertex.
00938 */
00939         for ( ipartner = istart+1; ipartner < nvert; ipartner++ ) {
00940             vpartner = AT.pWorkSpace[vert+ipartner];
```

```
00941             vv = vpartner+*vpartner; v = vpartner+1;
00942             do {
00943                 if ( *v == arglist[0] ) goto havepartner;
00944                 v++;
00945             } while ( v < vv );
00946         }
00947         return(numgenerated);
00948 havepartner:
00949         AT.pWorkSpace[vert+nvert+1] = vpartner;
00950 /*
00951         Now we run through all other possibilities at vpartner.
00952         vv is still OK.
00953 */
00954         v = vpartner+1;
00955         while ( v < vv ) {
00956             if ( *v != arglist[0] ) {
00957                 for ( j = 1; j < nargs; j++ ) {
00958                     if ( *v == arglist[j] ) {
00959                         loop[nloop++] = *v;
00960                         numgenerated += GenLoops(BHEAD term,level,vert,nvert,
00961                                                  arglist,nargs,loop,nloop);
00962                         nloop--;
00963                         break;
00964                     }
00965                 }
00966             }
00967             v++;
00968         }
00969 nextarg:
00970         arglist++; nargs--;
00971     }
00972     return(numgenerated);
00973 }
00974
00975 /*
00976     #] StartLoops :
00977     #[ GenLoops :
00978
00979     We enter with an open line in loop[nloop-1]
00980 */
00981
00982 LONG GenLoops(PHEAD WORD *term,WORD level,LONG vert,WORD nvert,
00983         WORD *arglist,WORD nargs,WORD *loop,WORD nloop)
00984 {
00985     LONG numgenerated = 0;
00986     WORD *vstart, *v, *vv, i, j, *vpartner;
00987 /*
00988     Start with checking whether the partner is in vstart (=vert[nvert])
00989 */
00990     vstart = AT.pWorkSpace[nvert];
00991     vv = vstart + *vstart; v = vstart+1;
00992     while ( v < vv ) {
00993         if ( *v == loop[nloop-1] ) {
00994 /*
00995             This closes the loop.
00996             Now we can output it.
00997 */
00998             LoopOutput(BHEAD term,level,loop,nloop);
00999             numgenerated++;
01000             return(numgenerated);
01001         }
01002         v++;
01003     }
01004 /*
01005     Start with finding the partner.
01006 */
01007     for ( i = 0; i < nvert; i++ ) {
01008         vpartner = AT.pWorkSpace[vert+i];
01009         if ( vpartner == vstart ) continue;
01010         for ( j = 0; j < nloop; j++ ) {
01011             if ( vpartner == AT.pWorkSpace[vert+nvert+j] ) break;
01012         }
01013         if ( j < nloop ) continue;
01014         v = vpartner+1; vv = vpartner + *vpartner;
01015         while ( v < vv ) {
01016             if ( *v == loop[nloop-1] ) {
01017 /*
01018                 Found the partner.
01019                 Now we can sum over the remaining permitted arguments of this vertex.
01020 */
01021                 v = vpartner + 1;
01022                 while ( v < vv ) {
01023 /*
01024                     *v should be in arglist.
01025 */
01026                     for ( j = 0; j < nargs; j++ ) {
01027                         if ( *v == arglist[j] ) break;
```

```
01028                     }
01029                     if ( j >= nargs ) { v++; continue; }
01030 /*
01031                     *v should not be in loops.
01032 */
01033                     for ( j = 0; j < nloop; j++ ) {
01034                         if ( *v == loop[j] ) break;
01035                     }
01036                     if ( j >= nloop ) {
01037                         AT.pWorkSpace[vert+nvert+nloop] = vpartner;
01038                         loop[nloop++] = *v;
01039                         numgenerated += GenLoops(BHEAD term,level,vert,nvert,
01040                                 arglist,nargs,loop,nloop);
01041                         nloop--;
01042                     }
01043                     v++;
01044                 }
01045                 return(numgenerated);
01046             }
01047             v++;
01048         }
01049     }
01050 /*
01051     The partner is in a vertex we have finished before.
01052 */
01053     return(numgenerated);
01054 }
01055
01056 /*
01057     #] GenLoops :
01058     #[ LoopOutput :
01059 */
01060
01061 void LoopOutput(PHEAD WORD *term, WORD level, WORD *loop, WORD nloop)
01062 {
01063     CBUF *C = cbuf+AM.rbufnum;
01064     WORD loopfun = C->lhs[level][3];  /* The output function */
01065     WORD option1 = C->lhs[level][4];  /* type of argument */
01066     WORD *tstop, *tstop1, *t, *tt;
01067     WORD *outterm, *loop1;
01068     WORD i;
01069     tstop1 = term+*term; tstop = tstop1 - ABS(tstop1[-1]);
01070 /*
01071     Construct the rcycle symmetrized version of loop.
01072 */
01073     if ( nloop > 2 ) {
01074         loop1 = AT.WorkPointer;
01075         loop1[0] = loop[0];
01076         for ( i = 1; i < nloop; i++ ) { loop1[i] = loop[nloop-i]; }
01077         if ( loop1[1] < loop[1] ) {
01078             AT.WorkPointer += nloop;
01079             loop = loop1;
01080         }
01081     }
01082     outterm = AT.WorkPointer;
01083     tt = outterm; t = term;
01084     while ( t < tstop ) *tt++ = *t++;
01085     *tt++ = loopfun;
01086     if ( functions[loopfun-FUNCTION].spec == TENSORFUNCTION ) {
01087         *tt++ = FUNHEAD+nloop;
01088         FILLFUN(tt)
01089         if ( option1 == -VECTOR ) {
01090             for ( i = 0; i < nloop; i++ ) *tt++ = loop[i]+AM.OffsetVector;
01091         }
01092         else {
01093             for ( i = 0; i < nloop; i++ ) *tt++ = loop[i];
01094         }
01095     }
01096     else {
01097         *tt++ = FUNHEAD+nloop*2;
01098         FILLFUN(tt)
01099         for ( i = 0; i< nloop; i++ ) {
01100             *tt++ = option1;
01101             if ( option1 == -VECTOR ) *tt++ = loop[i] + AM.OffsetVector;
01102             else *tt++ = loop[i];
01103         }
01104     }
01105     while ( t < tstop1 ) *tt++ = *t++;
01106     *outterm = tt - outterm;
01107     AT.WorkPointer = tt;
01108     if ( Generator(BHEAD outterm,level) ) {
01109         MesCall("LoopOutput");
01110         Terminate(-1);
01111     }
01112     AT.WorkPointer = outterm;
01113 }
01114
```

```
01115 /*
01116     #] LoopOutput :
01117     #[ AllPaths :
01118
01119     This routine has many similarities with AllLoops.
01120     In AllLoops we have startingpoint and endpoint at the same vertex.
01121     In AllPaths the startingpoint and the endpoints are different and
01122     given in advance. This endfun can occur only twice.
01123 */
01124
01125 int AllPaths(PHEAD WORD *term,WORD level)
01126 {
01127     CBUF *C = cbuf+AM.rbufnum;
01128     WORD endcode = C->lhs[level][2];    /* The endpoint function */
01129     WORD vcode   = C->lhs[level][3];    /* The intermediate function */
01130     WORD option1 = C->lhs[level][5];    /* type of argument */
01131     WORD option2 = C->lhs[level][6];    /* what to do when no loop */
01132     WORD *t, *tstop, *tend1, *tend2, *tstart, *tend, numend, nvert, npass;
01133     WORD *tfrom, *to, *tos, *from;
01134     WORD *arglist, nargs, *path, npath, *a, *a1, *a2, *a3;
01135     WORD i, j, jj, *v, *vv, action;
01136     LONG vert,vert1; /* ,vert2; */
01137     WORD numgenerated = 0;
01138     WORD *oldworkpointer = AT.WorkPointer;
01139     LONG oldpworkpointer = AT.pWorkPointer;
01140 /*
01141     Search for the two occurrences of endcode.
01142 */
01143     tstop = term+*term; tstop -=  ABS(tstop[-1]);
01144     t = term + 1;
01145     while ( t < tstop && *t != endcode ) t += t[1];
01146     if ( t == tstop ) { /* no endpoints */
01147         if ( option2 == 0 ) return(0);
01148         else return(Generator(BHEAD term,level));
01149     }
01150     tend1 = t;
01151     numend = 0;
01152     while ( t < tstop && *t == endcode ) { tend2 = t; t += t[1]; numend++; }
01153     if ( numend != 2 ) { /* not 2 endpoints -> no path */
01154         if ( option2 == 0 ) return(0);
01155         else return(Generator(BHEAD term,level));
01156     }
01157 /*
01158     Search for the intermediate functions.
01159 */
01160     t = term + 1;
01161     nvert = 0;
01162     while ( t < tstop && *t != vcode ) t += t[1];
01163     tstart = t;
01164     while ( t < tstop && *t == vcode ) {
01165         t += t[1]; nvert++;
01166     }
01167     tend = t;
01168 /*
01169     Next we copy the relevant content of the various functions into the
01170     WorkSpace. We keep pointers to the functions in pWorkSpace.
01171     First the two endpoints and then the intermediate ones.
01172 */
01173     WantAddPointers((2*nvert+8));
01174     vert = AT.pWorkPointer+2;
01175     vert1 = AT.pWorkPointer;
01176 /*  vert2 = AT.pWorkPointer+1; */
01177     AT.pWorkPointer += 2*nvert+8;
01178     nvert = 0;
01179 /*
01180     Copy the endpoints.
01181 */
01182     to = AT.WorkPointer;
01183
01184     if ( functions[endcode-FUNCTION].spec == TENSORFUNCTION ) {
01185         from = tend1; npass = 0;
01186 redo1:
01187         tos = to++;
01188         tfrom = from+from[1];
01189         from += FUNHEAD;
01190         while ( from < tfrom ) {
01191             if ( option1 == -INDEX && *from >= 0 ) {
01192                 *to++ = *from++;
01193             }
01194             else if ( option1 == -VECTOR && *from < 0 ) {
01195                 *to++ = *from++ - AM.OffsetVector;
01196             }
01197             else {
01198                 from++;
01199             }
01200         }
01201         *tos = to-tos;
```

```
01202         if ( *tos < 2 ) to = tos;
01203         else AT.pWorkSpace[vert1+npass] = tos;
01204         npass++;
01205         if ( from == tend2 ) goto redo1;
01206     }
01207     else if ( functions[endcode-FUNCTION].spec == 0 ) {
01208         from = tend1;
01209         npass = 0;
01210 redo2:
01211         tos = to++;
01212         tfrom = from + from[1];
01213         from += FUNHEAD;
01214         while ( from < tfrom ) {
01215             if ( option1 == -VECTOR
01216                  && ( from[0] == -INDEX || from[0] == -VECTOR )
01217                  && from[1] < 0 ) {
01218                 from++;
01219                 *to++ = *from++ - AM.OffsetVector;
01220             }
01221             else if ( option1 == *from ) {
01222                 from++; *to++ = *from++;
01223             }
01224             else {
01225                 NEXTARG(from);
01226             }
01227         }
01228         *tos = to-tos;
01229         if ( *tos < 2 ) to = tos;
01230         else AT.pWorkSpace[vert1+npass] = tos;
01231         npass++;
01232         if ( from == tend2 ) goto redo2;
01233     }
01234     else {
01235         AT.WorkPointer = oldworkpointer;
01236         AT.pWorkPointer = oldpworkpointer;
01237         if ( option2 == 0 ) return(0);
01238         return(Generator(BHEAD term,level));
01239     }
01240 /*
01241     And now the intermediate functions
01242 */
01243     from = tstart;
01244     if ( functions[vcode-FUNCTION].spec == TENSORFUNCTION ) {
01245         from = tstart;
01246         while ( from < tend ) {
01247             tos = to++;
01248             tfrom = from+from[1];
01249             from += FUNHEAD;
01250             while ( from < tfrom ) {
01251                 if ( option1 == -INDEX && *from >= 0 ) {
01252                     *to++ = *from++;
01253                 }
01254                 else if ( option1 == -VECTOR && *from < 0 ) {
01255                     *to++ = *from++ - AM.OffsetVector;
01256                 }
01257                 else {
01258                     from++;
01259                 }
01260             }
01261             *tos = to-tos;
01262             if ( *tos < 3 ) to = tos;
01263             else AT.pWorkSpace[vert+nvert++] = tos;
01264         }
01265     }
01266     else if ( functions[vcode-FUNCTION].spec == 0 ) {
01267         from = tstart;
01268         while ( from < tend ) {
01269             tos = to++;
01270             tfrom = from + from[1];
01271             from += FUNHEAD;
01272             while ( from < tfrom ) {
01273                 if ( option1 == -VECTOR
01274                      && ( from[0] == -INDEX || from[0] == -VECTOR )
01275                      && from[1] < 0 ) {
01276                     from++;
01277                     *to++ = *from++ - AM.OffsetVector;
01278                 }
01279                 else if ( option1 == *from ) {
01280                     from++; *to++ = *from++;
01281                 }
01282                 else {
01283                     NEXTARG(from);
01284                 }
01285             }
01286             *tos = to-tos;
01287             if ( *tos < 3 ) to = tos;
01288             else AT.pWorkSpace[vert+nvert++] = tos;
```

```
01289         }
01290     }
01291     else {
01292         AT.WorkPointer = oldworkpointer;
01293         AT.pWorkPointer = oldpworkpointer;
01294         if ( option2 == 0 ) return(0);
01295         return(Generator(BHEAD term,level));
01296     }
01297     AT.WorkPointer = to;
01298 /*
01299     Now make a list of all relevant arguments.
01300 */
01301     a = arglist = AT.WorkPointer;
01302     for ( i = -2; i < nvert; i++ ) {
01303         v = AT.pWorkSpace[vert+i];
01304         j = *v-1; v++;
01305         NCOPY(a,v,j);
01306     }
01307     nargs = a-arglist;
01308     AT.WorkPointer = a;
01309 /*
01310     Now sort the list. Bubble type sort.
01311 */
01312     a1 = arglist; a2 = a1+1;
01313     while ( a2 < a ) {
01314         a3 = a2;
01315         while ( a3 > a1 && a3[-1] > a3[0] ) {
01316             EXCH(*a3,a3[-1]);
01317             a3--;
01318         }
01319         a2++;
01320     }
01321 /*
01322     Now remove the elements from the list that do not occur exactly twice.
01323 */
01324     a1 = arglist; a2 = a1; a3 = a1+nargs;
01325     while ( a2 < a3 ) {
01326         if ( a2+1 == a3 ) { break; }
01327         else if ( a2+2 == a3 ) {
01328             if ( a2[0] == a2[1] ) { *a1++ = a2[0]; }
01329             break;
01330         }
01331         else {
01332             if ( a2[0] != a2[1] ) { a2++; }
01333             else if ( a2[0] != a2[2] ) {
01334                 *a1++ = a2[0]; a2 += 2;
01335             }
01336             else {
01337                 a2++; while ( a2 < a3 && a2[-1] == a2[0] ) a2++;
01338             }
01339         }
01340     }
01341     nargs = a1-arglist;
01342 /*
01343     Now we need to redo the list of vertices and remove the elements
01344     that are not in our arglist.
01345 */
01346     do {
01347       action = 0;
01348       for ( i = -2; i < nvert; i++ ) {
01349         vv = v = AT.pWorkSpace[vert+i];
01350         v++;
01351         for ( j = 1; j < vv[0]; j++ ) {
01352             for ( jj = 0; jj < nargs; jj++ ) {
01353                 if ( *v == arglist[jj] ) break;
01354             }
01355             if ( jj >= nargs ) {    /* was not in the list */
01356                 vv[0] = vv[0]-1;
01357                 for ( jj = j; jj < vv[0]; jj++ ) vv[jj] = vv[jj+1];
01358             }
01359             v++;
01360         }
01361       }
01362 /*
01363       Next we need to remove vertices that have only one object remaining.
01364       Also, the object can be removed from arglist. After that we go back
01365       to clean up the list.
01366 */
01367       for ( i = -2; i < nvert; i++ ) {
01368         vv = AT.pWorkSpace[vert+i];
01369         if ( vv[0] == 1 ) {
01370             AT.pWorkSpace[vert+i] = AT.pWorkSpace[vert+nvert-1];
01371             nvert--; i--; continue;
01372         }
01373         else if ( vv[0] == 2 && i >= 0 ) {
01374             for ( j = 0; j < nargs; j++ ) {
01375                 if ( arglist[j] == vv[1] ) break;
```

```
01376             }
01377             while ( j < nargs-1 ) arglist[j] = arglist[j+1];
01378             nargs--;
01379             AT.pWorkSpace[vert+i] = AT.pWorkSpace[vert+nvert-1];
01380             nvert--; i--;
01381             action = 1;
01382         }
01383       }
01384     } while ( action );
01385 /*
01386     Now we have a clean list of connections and we can start building
01387     the path. We start with summing over all objects in vert1.
01388 */
01389     path = AT.WorkPointer; npath = 0;
01390     AT.WorkPointer += nvert+8;
01391
01392     t = AT.pWorkSpace[vert1];
01393     if ( *t >= 2 ) {
01394         for ( i = 1; i < *t; i++ ) {
01395             AT.pWorkSpace[vert+nvert] = t;
01396             path[npath++] = t[i];
01397             numgenerated += GenPaths(BHEAD term,level,vert,nvert,arglist,nargs,path,npath);
01398             npath--;
01399         }
01400     }
01401     AT.WorkPointer = oldworkpointer;
01402     AT.pWorkPointer = oldpworkpointer;
01403     if ( numgenerated == 0 && option2 != 0 ) return(Generator(BHEAD term,level));
01404     return(0);
01405 }
01406
01407 /*
01408     #] AllPaths :
01409     #[ GenPaths :
01410
01411     We enter with an open line in path[npath-1]
01412     We traverse though the vertices until we reach vert-1, the endpoint.
01413 */
01414
01415 LONG GenPaths(PHEAD WORD *term, WORD level, LONG vert, WORD nvert,
01416         WORD *arglist, WORD nargs, WORD *path, WORD npath)
01417 {
01418     LONG numgenerated = 0;
01419     WORD *t, *vpartner, *v, *vv;
01420     WORD i, j;
01421 /*
01422     Check whether path[npath-1] is part of the endpoint.
01423 */
01424     t = AT.pWorkSpace[vert-1];
01425     for ( i = 1; i < *t; i++ ) {
01426         if ( t[i] == path[npath-1] ) { /* Got a path! */
01427             PathOutput(BHEAD term,level,path,npath);
01428             numgenerated++;
01429             return(numgenerated);
01430         }
01431     }
01432 /*
01433     Start with finding the partner.
01434 */
01435     for ( i = 0; i < nvert; i++ ) {
01436         vpartner = AT.pWorkSpace[vert+i];
01437         for ( j = 0; j < npath; j++ ) {
01438             if ( vpartner == AT.pWorkSpace[vert+nvert+j] ) break;
01439         }
01440         if ( j < npath ) continue;
01441         v = vpartner+1; vv = vpartner + *vpartner;
01442         while ( v < vv ) {
01443             if ( *v == path[npath-1] ) {
01444 /*
01445                 Found the partner.
01446                 Now we can sum over the remaining permitted arguments of this vertex.
01447 */
01448                 v = vpartner + 1;
01449                 while ( v < vv ) {
01450 /*
01451                     *v should be in arglist.
01452 */
01453                     for ( j = 0; j < nargs; j++ ) {
01454                         if ( *v == arglist[j] ) break;
01455                     }
01456                     if ( j >= nargs ) { v++; continue; }
01457 /*
01458                     *v should not be in path.
01459 */
01460                     for ( j = 0; j < npath; j++ ) {
01461                         if ( *v == path[j] ) break;
01462                     }
```

```
01463                     if ( j >= npath ) {
01464                         AT.pWorkSpace[vert+nvert+npath] = vpartner;
01465                         path[npath++] = *v;
01466                         numgenerated += GenPaths(BHEAD term,level,vert,nvert,
01467                                 arglist,nargs,path,npath);
01468                         npath--;
01469                     }
01470                     v++;
01471                 }
01472                 return(numgenerated);
01473             }
01474             v++;
01475         }
01476     }
01477 /*
01478     The partner is in a vertex we have finished before.
01479 */
01480     return(numgenerated);
01481 }
01482
01483 /*
01484     #] GenPaths :
01485     #[ PathOutput :
01486 */
01487
01488 void PathOutput(PHEAD WORD *term, WORD level, WORD *path, WORD npath)
01489 {
01490     CBUF *C = cbuf+AM.rbufnum;
01491     WORD pathfun = C->lhs[level][4];  /* The output function */
01492     WORD option1 = C->lhs[level][5];  /* type of argument */
01493     WORD *tstop, *tstop1, *t, *tt;
01494     WORD *outterm;
01495     WORD i;
01496     tstop1 = term+*term; tstop = tstop1 - ABS(tstop1[-1]);
01497     outterm = AT.WorkPointer;
01498     tt = outterm; t = term;
01499     while ( t < tstop ) *tt++ = *t++;
01500     *tt++ = pathfun;
01501     if ( functions[pathfun-FUNCTION].spec == TENSORFUNCTION ) {
01502         *tt++ = FUNHEAD+npath;
01503         FILLFUN(tt)
01504         if ( option1 == -VECTOR ) {
01505             for ( i = 0; i < npath; i++ ) *tt++ = path[i]+AM.OffsetVector;
01506         }
01507         else {
01508             for ( i = 0; i < npath; i++ ) *tt++ = path[i];
01509         }
01510     }
01511     else {
01512         *tt++ = FUNHEAD+npath*2;
01513         FILLFUN(tt)
01514         for ( i = 0; i< npath; i++ ) {
01515             *tt++ = option1;
01516             if ( option1 == -VECTOR ) *tt++ = path[i] + AM.OffsetVector;
01517             else *tt++ = path[i];
01518         }
01519     }
01520     while ( t < tstop1 ) *tt++ = *t++;
01521     *outterm = tt - outterm;
01522     AT.WorkPointer = tt;
01523     if ( Generator(BHEAD outterm,level) ) {
01524         MesCall("PathOutput");
01525         Terminate(-1);
01526     }
01527     AT.WorkPointer = outterm;
01528 }
01529
01530
01531
01532 /*
01533     #] PathOutput :
01534     #[ AllOnePI :
01535
01536     We assume a graph that has loops and is OnePI.
01537     This routine initializes the recursion that creates all onePI subgraphs.
01538
01539     Algorithm:
01540         a: have a routine that removes all bridges: RemoveBridges
01541         b: output an empty diagram.
01542         c: output the diagram itself
01543         d: cut a line, followed by removing all bridges.
01544         e: if the diagram still has lines, go to c, else go to the next line in d.
01545     In order to avoid factorial blowup, we need to prune the tree as fast as possible.
01546
01547     1: list of lines to be cut.
01548     2: once we have tried a line and removed its bridges, we do not have to
01549        try this line deeper in the tree, neither its bridges.
```

```
01550
01551
01552      1 2 3
01553         cut 1.                        x
01554             cut 2 -> bridge 3          x
01555         cut 2.                        x
01556             cut 3 -> bridge 1   ????   mag niet
01557         cut 3.                        x
01558     Hence  1,2,3,4,5,6,7
01559         1 still to go 2,3,4,5,6,7
01560         2 still to go 3,4,5,6,7
01561         3 still to go 4,5,6,7
01562         etc.
01563     bridges: if x is bridge, we take x from the list_to_go.
01564     If we have a bridge that is a number lower than the list_to_go we skip this possibility.
01565     We reach the end when the list_to_go is empty.
01566 */
01567
01568 WORD AllOnePI(WORD *term,WORD level)
01569 {
01570     CBUF *C = cbuf+AM.rbufnum;
01571     WORD vcode   = C->lhs[level][2];    /* The vertex function */
01572     WORD option1 = C->lhs[level][4];    /* type of argument */
01573 /*
01574     First we have to collect all relevant information about the diagram.
01575     We should start with removing bridges to make the real starting point onePI.
01576 */
01577     DUMMYUSE(term)
01578     DUMMYUSE(vcode)
01579     DUMMYUSE(option1)
01580     return(0);
01581 }
01582
01583 /*
01584     #] AllOnePI :
01585     #[ RemoveBridges :
01586 */
01587
01588 int RemoveBridges(void)
01589 {
01590     return(0);
01591 }
01592
01593 /*
01594     #] RemoveBridges :
01595     #[ TakeOneLine :
01596 */
01597
01598 int TakeOneLine(WORD*term,WORD level)
01599 {
01600     DUMMYUSE(term)
01601     DUMMYUSE(level)
01602     return(0);
01603 }
01604
01605 /*
01606     #] TakeOneLine :
01607     #[ OutputOnePI :
01608 */
01609
01610 int OutputOnePI(PHEAD WORD *term,WORD level)
01611 {
01612     return(Generator(BHEAD term,level));
01613 }
01614
01615 /*
01616     #] OutputOnePI :
01617 */
01618
```

4.70 mallocprotect.h

```
00001 #ifndef MALLOCPROTECT_H
00002 #define MALLOCPROTECT_H
00003
00010 /* #[ License : */
00011 /*
00012  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00013  *   When using this file you are requested to refer to the publication
00014  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00015  *   This is considered a matter of courtesy as the development was paid
00016  *   for by FOM the Dutch physics granting agency and we would like to
00017  *   be able to track its scientific use to convince FOM of its value
```

```
00018  *   for the community.
00019  *
00020  *   This file is part of FORM.
00021  *
00022  *   FORM is free software: you can redistribute it and/or modify it under the
00023  *   terms of the GNU General Public License as published by the Free Software
00024  *   Foundation, either version 3 of the License, or (at your option) any later
00025  *   version.
00026  *
00027  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00028  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00029  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00030  *   details.
00031  *
00032  *   You should have received a copy of the GNU General Public License along
00033  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00034  */
00035 /* #] License : */
00036
00037 /*
00038     #[ Documentation :
00039
00040 The enhanced malloc debugger. It intercepts access (both write AND
00041 read) to the memory outside the allocated chunk in the following
00042 manner: it prints out (to the stderr) the information about the event
00043 and goes to the spilock. The user is able to attach the debugger to
00044 the running process, investigate the problem and continue evaluation.
00045
00046 In case of error, the debugger will print to the stderr the following
00047 information:
00048 "***** PID: ###, Addr: ###, signal ### (code ###)"
00049 "Attach gdb -p ### and set var loopForever=0 in a corresponding frame to continue"
00050 or for TFORM:
00051 "Attach gdb -p ### and set var loopForever=0 in a corresponding frame
00052    of a thread with LPW id ### to continue"
00053
00054 and go to the spilock. The user may attach gdb by the command
00055 gdb -p ### and type "where" (in case of TFORM, the user should first
00056 switch to the proper thread).  Then investigation of the corresponding
00057 frames should clarify the situation. In order to continue the program,
00058 the user might set (in the corresponding frame of the corresponding
00059 thread) the variable loopForever to 0. The debugger will try to remove
00060 local problem lead to the exception.
00061
00062 To activate the debugger, the macro MALLOCPROTECT should be
00063 defined. There are three possible values:
00064 #define MALLOCPROTECT -1 (any integer <0)
00065 #define MALLOCPROTECT 0
00066 #define MALLOCPROTECT 1 (any integer>0)
00067
00068 If MALLOCPROTECT < 0, the debugger will intercept any access to the
00069 memory before the allocated chunk, even one byte.
00070
00071 If MALLOCPROTECT == 0, the debugger will intercept any access to the
00072 memory before the allocated chunk, even one byte, and access to the
00073 memory page next to the allocated one.
00074
00075 If MALLOCPROTECT > 0, the debugger will intercept any access to the
00076 memory after the allocated chunk, even one byte, and access to the
00077 memory page before the allocated one.
00078
00079 The original FORM malloc debugger is able to catch errors when the
00080 allocated memory is freed improper, or if some small portion OUT of
00081 the allocated chunk is written. This debugger is a complementary
00082 one. It permits to catch situation when some small portion of memory
00083 before or after the allocated chunk is written or even just read.
00084
00085 The idea is to protect the beginning or the end of the allocated chunk
00086 for any kind of access and install the SIGSEGV handler in order to
00087 intercept the access to this memory. Moreover, the allocated memory is
00088 always immediately returned to the system so if the user tries to
00089 access the memory after it is freed then the handler is triggered,
00090 also.
00091
00092 The problem here is that we are able to allocate / protect only the
00093 whole page. So the user has to run the debugger twice: one time with
00094 the left alignment (#define MALLOCPROTECT <0), and then with the right
00095 alignment (#define MALLOCPROTECT >0). During the first run the possible
00096 errors like x[-1] will be caught, and the second run will manifest
00097 reading over the allocated ares.
00098
00099 The leftmost extra page is always allocated and protected. The size
00100 of the allocated chunk is stored in the beginning of this page.
00101
00102     #] Documentation :
00103     #[ Includes :
00104 */
```

```
00105
00106 #include <stdio.h>
00107 #include <stdlib.h>
00108 #include <unistd.h>
00109 #include <signal.h>
00110 #include <sys/mman.h>
00111
00112 #ifdef WITHPTHREADS
00113 #ifdef LINUX
00114 #include <sys/syscall.h>
00115 #endif
00116 #endif
00117
00118 /*
00119     #] Includes :
00120 */
00121
00122 static int pageSize=4096;/*the default value*/
00123
00124
00125 /*
00126     #[ segv_handler :
00127 */
00128
00129 /*The handler will be invoked on some signal (usually SIGSEGV). It
00130 will print some diagnostics to stderr and hands up in infinite loop
00131 waiting the user for debugging:*/
00132 static void segv_handler(int sig, siginfo_t *sip, void *xxx) {
00133     char *vadr;
00134     char *errStr;
00135     int actionBeforeExit=0;/* < 0 - unprotect, > 0 - map */
00136     switch(sip->si_signo){/* All POSIX signals for which the field siginfo_t.si_addr is defined:*/
00137         case SIGILL:
00138             switch(sip->si_code){
00139             case ILL_ILLOPC:
00140                 errStr="SIGILL: Illegal opcode";
00141                 break;
00142             case ILL_ILLOPN:
00143                 errStr="SIGILL: Illegal operand";
00144                 break;
00145             case ILL_ILLADR:
00146                 errStr="SIGILL: Illegal addressing mode";
00147                 break;
00148             case ILL_ILLTRP:
00149                 errStr="SIGILL: Illegal trap";
00150                 break;
00151             case ILL_PRVOPC:
00152                 errStr="SIGILL: Privileged opcode";
00153                 break;
00154             case ILL_PRVREG:
00155                 errStr="SIGILL: Privileged register";
00156                 break;
00157             case ILL_COPROC:
00158                 errStr="SIGILL: Coprocessor error";
00159                 break;
00160             case ILL_BADSTK:
00161                 errStr="SIGILL: Internal stack error";
00162                 break;
00163             default:
00164                 errStr="SIGILL: Unknown signal code";
00165         }/*case SIGILL:*/
00166         break;
00167         case SIGFPE:
00168             switch(sip->si_code){
00169                 case FPE_INTDIV:
00170                     errStr="SIGFPE: Integer divide-by-zero";
00171                     break;
00172                 case FPE_INTOVF:
00173                     errStr="SIGFPE: Integer overflow";
00174                     break;
00175                 case FPE_FLTDIV:
00176                     errStr="SIGFPE: Floating point divide-by-zero";
00177                     break;
00178                 case FPE_FLTOVF:
00179                     errStr="SIGFPE: Floating point overflow";
00180                     break;
00181                 case FPE_FLTUND:
00182                     errStr="SIGFPE: Floating point underflow";
00183                     break;
00184                 case FPE_FLTRES:
00185                     errStr="SIGFPE: Floating point inexact result";
00186                     break;
00187                 case FPE_FLTINV:
00188                     errStr="SIGFPE: Invalid floating point operation";
00189                     break;
00190                 case FPE_FLTSUB:
00191                     errStr="SIGFPE: Subscript out of range";
```

```
00192                     break;
00193                 default:
00194                     errStr="SIGFPE: Unknown signal code";
00195             }/*switch(sip->si_code)*/
00196             break;
00197         case SIGSEGV:
00198             switch(sip->si_code){
00199                 case SEGV_MAPERR:
00200                     errStr="SIGSEGV: Address not mapped";
00201                     actionBeforeExit = 1;
00202                     break;
00203                 case SEGV_ACCERR:
00204                     errStr="SIGSEGV: Invalid permissions";
00205                     actionBeforeExit = -1;
00206                     break;
00207                 default:
00208                     errStr="SIGSEGV: Unknown signal code";
00209             }/*switch(sip->si_code)*/
00210             break;
00211         case SIGBUS:
00212             switch(sip->si_code){
00213                 case BUS_ADRALN:
00214                     errStr="SIGBUS: Invalid address alignment";
00215                     break;
00216                 case BUS_ADRERR:
00217                     errStr="SIGBUS: Non-existent physical address";
00218                     break;
00219                 case BUS_OBJERR:
00220                     errStr="SIGBUS: Object-specific hardware error";
00221                     break;
00222                 default:
00223                     errStr="SIGBUS: Unknown signal code";
00224             }/*switch(sip->si_code)*/
00225             break;
00226         default:
00227             errStr="Unknown signal";
00228     }/*switch(sip->si_signo)*/
00229
00230     vadr = (caddr_t)sip->si_addr;
00231     fprintf(stderr, "\n***** PID: %ld, Addr: %p, signal %s (code %d)\n",
00232         (long)getpid(), vadr, errStr,sip->si_code);
00233
00234     {/*Block*/
00235         /*The process hangs up at this block.
00236         Attach gdb to investigate and continue:
00237         */
00238         volatile int loopForever=1;
00239         size_t alignedAdr=((size_t)vadr) & (~(pageSize-1));
00240         fprintf(stderr, "   Attach gdb -p %ld and set var loopForever=0 in a corresponding frame",
00241             (long)getpid());
00242 #ifdef CORRECTCODE
00243 #ifdef WITHPTHREADS
00244 #ifdef LINUX
00245         fprintf(stderr, "\n   of a thread with LPW id %ld",(long)syscall(SYS_gettid));
00246 #else
00247         /*If the compiler fails here, just remove the next line:*/
00248         fprintf(stderr, "\n   of thread with LPW id %ld",(long)gettid());
00249 #endif
00250 #endif
00251 #endif
00252         fprintf(stderr, " to continue\n");
00253
00254
00255         while(loopForever)
00256             sleep(1);
00257
00258         if(actionBeforeExit<0)/*After changing loopForever=0, unprotect the page to continue:*/
00259             mprotect((char*)alignedAdr, pageSize, PROT_READ | PROT_WRITE);
00260         if(actionBeforeExit>0)/*After changing loopForever=0, map the page to continue:*/
00261 /*          mmap((void*)alignedAdr,pageSize,PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, 0, 0); */
00262             mmap((void*)alignedAdr,pageSize,PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, 0, 0);
00263     }/*Block*/
00264 }/*segv_handler*/
00265
00266 /*
00267     #] segv_handler :
00268 */
00269
00270 /*
00271     #[ mprotectInit :
00272 */
00273
00274 static FORM_INLINE int mprotectInit(void)
00275 {
00276     struct sigaction sa;
00277     pageSize = getpagesize();
00278     sa.sa_sigaction = &segv_handler;
```

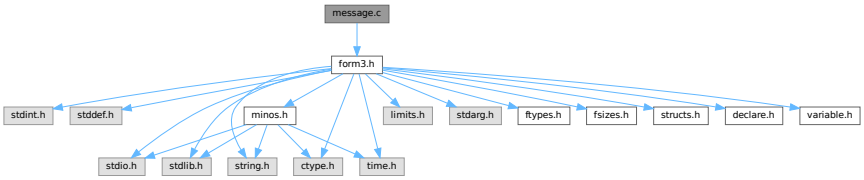
```
00279
00280     sigemptyset(&sa.sa_mask);
00281     sigaddset(&sa.sa_mask, SIGIO);
00282     sigaddset(&sa.sa_mask, SIGALRM);
00283
00284     sa.sa_flags = SA_SIGINFO;
00285
00286     if (sigaction(SIGILL, &sa, NULL)) {
00287         fprintf(stderr,"Error on assigning %s.\n","SIGILL");
00288         return SIGILL;
00289     }
00290     if (sigaction(SIGFPE, &sa, NULL)) {
00291         fprintf(stderr,"Error on assigning %s.\n","SIGFPE");
00292         return SIGFPE;
00293     }
00294     if (sigaction(SIGSEGV, &sa, NULL)) {
00295         fprintf(stderr,"Error on assigning %s.\n","SIGSEGV");
00296         return SIGSEGV;
00297     }
00298     if (sigaction(SIGBUS, &sa, NULL)) {
00299         fprintf(stderr,"Error on assigning %s.\n","SIGBUS");
00300         return SIGBUS;
00301     }
00302     return 0;
00303 }/*mprotectInit*/
00304
00305 /*
00306     #] mprotectInit :
00307 */
00308
00309
00310 /*
00311     #[ mprotectMalloc :
00312 */
00313
00314 static void *mprotectMalloc(size_t theSize)
00315 {
00316 #if MALLOCPROTECT < 0
00317     /*Only one side is protected*/
00318     size_t nPages=1;
00319 #else
00320     /*Both sides are protected*/
00321     size_t nPages=2;
00322 #if MALLOCPROTECT > 0
00323     /*will need the original theSize*/
00324     size_t requestedSize=theSize;
00325 #endif
00326 #endif
00327
00328     char *ret=NULL;
00329     size_t *ptr;
00330
00331
00332
00333     /*Align required size to the pagesize:*/
00334     if(theSize % pageSize)
00335         nPages++;
00336     theSize= (theSize/pageSize+nPages)*pageSize;
00337
00338 /*  ret=(char*)mmap(0,theSize,PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, 0, 0); */
00339     ret=(char*)mmap(0,theSize,PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, 0, 0);
00340     if(ret == MAP_FAILED)
00341     return NULL;
00342     ptr=(size_t *)ret;
00343     *ptr=theSize;
00344     if(mprotect(ret, pageSize, PROT_NONE))return NULL;
00345 #if MALLOCPROTECT < 0
00346     return ret+pageSize;
00347 #else
00348     if(mprotect(ret+(theSize-pageSize), pageSize, PROT_NONE))return NULL;
00349 #if MALLOCPROTECT ==0
00350     return ret+pageSize;
00351 #endif
00352 #endif
00353     /*MALLOCPROTECT > 0, but we need conditional compilation
00354     since the variable requestedSize:*/
00355 #if MALLOCPROTECT > 0
00356
00357     /*Potential problems with alignment if the requested size is not
00358     a multiple of items. But no problems on x86-64:*/
00359     return ret+ (theSize-pageSize-requestedSize);
00360 #endif
00361 }/*mprotectMalloc*/
00362 /*
00363     #] mprotectMalloc :
00364 */
00365
```

```
00366 /*
00367     #[ mprotectFree :
00368 */
00369
00370 static void mprotectFree(char *ptr)
00371 {
00372     size_t theSize;
00373     if(ptr==NULL) return;
00374 #if MALLOCPROTECT > 0
00375     /*The memory block was moved to the right, find the left page boundary:*/
00376     {/*Block*/
00377         size_t alignedPtr=((size_t)ptr) & (~(pageSize-1));
00378         ptr=(char*)alignedPtr;
00379     }/*Block*/
00380 #endif
00381
00382     ptr-=pageSize;
00383     mprotect(ptr, pageSize, PROT_READ);
00384     theSize=*((size_t*)ptr);
00385     munmap(ptr,theSize);
00386 }/*mprotectFree*/
00387
00388 /*
00389     #] mprotectFree :
00390 */
00391
00392 #endif
```

4.71 message.c File Reference

`#include "form3.h"`

Include dependency graph for message.c:



Functions

- NORETURN void Error0 (char *s)
- NORETURN void Error1 (char *s, UBYTE *t)
- NORETURN void Error2 (char *s1, char *s2, UBYTE *t)
- NORETURN void MesWork (void)
- int MesPrint (const char *fmt,...)
- void Warning (char *s)
- void HighWarning (char *s)
- int MesCall (char *s)
- int MesCerr (char *s, UBYTE *t)
- int MesComp (char *s, UBYTE *p, UBYTE *q)
- void PrintTerm (WORD *term, char *where)
- void PrintTermC (WORD *term, char *where)
- void PrintSubTerm (WORD *term, char *where)
- void PrintWords (WORD *buffer, LONG number)
- void PrintSeq (WORD *a, char *text)

4.71.1 Detailed Description

Contains the routines that write messages. This includes the very important routine MesPrint which is the FORM equivalent of printf but then with escape sequences that are relevant for symbolic manipulation. The FORM statement Print "...." is passed almost literally to MesPrint.

Definition in file message.c.

4.71.2 Function Documentation

4.71.2.1 Error0()

```
NORETURN void Error0 (
            char * s )
```

Definition at line 56 of file message.c.

4.71.2.2 Error1()

```
NORETURN void Error1 (
            char * s,
            UBYTE * t )
```

Definition at line 67 of file message.c.

4.71.2.3 Error2()

```
NORETURN void Error2 (
            char * s1,
            char * s2,
            UBYTE * t )
```

Definition at line 78 of file message.c.

4.71.2.4 MesWork()

```
NORETURN void MesWork (
            void  )
```

Definition at line 89 of file message.c.

4.71.2.5 MesPrint()

```
int MesPrint (
            const char * fmt,
             ...  )
```

Definition at line 136 of file message.c.

4.71.2.6 Warning()

```
void Warning (
            char * s )
```

Definition at line 794 of file message.c.

4.71.2.7 HighWarning()

```
void HighWarning (
            char * s )
```

Definition at line 806 of file message.c.

4.71.2.8 MesCall()

```
int MesCall (
            char * s )
```

Definition at line 818 of file message.c.

4.71.2.9 MesCerr()

```
int MesCerr (
            char * s,
            UBYTE * t )
```

Definition at line 828 of file message.c.

4.71.2.10 MesComp()

```
int MesComp (
            char * s,
            UBYTE * p,
            UBYTE * q )
```

Definition at line 847 of file message.c.

4.71.2.11 PrintTerm()

```
void PrintTerm (
            WORD * term,
            char * where )
```

Definition at line 861 of file message.c.

4.71.2.12 PrintTermC()

```
void PrintTermC (
            WORD * term,
            char * where )
```

Definition at line 891 of file message.c.

4.71.2.13 PrintSubTerm()

```
void PrintSubTerm (
            WORD * term,
            char * where )
```

Definition at line 925 of file message.c.

4.71.2.14 PrintWords()

```
void PrintWords (
            WORD * buffer,
            LONG number )
```

Definition at line 947 of file message.c.

4.71.2.15 PrintSeq()

```
void PrintSeq (
            WORD * a,
            char * text )
```

Definition at line 965 of file message.c.

4.72 message.c

Go to the documentation of this file.

```
00001
00009 /* #[ License : */
00010 /*
00011  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00012  *   When using this file you are requested to refer to the publication
00013  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00014  *   This is considered a matter of courtesy as the development was paid
00015  *   for by FOM the Dutch physics granting agency and we would like to
00016  *   be able to track its scientific use to convince FOM of its value
00017  *   for the community.
00018  *
00019  *   This file is part of FORM.
00020  *
00021  *   FORM is free software: you can redistribute it and/or modify it under the
00022  *   terms of the GNU General Public License as published by the Free Software
00023  *   Foundation, either version 3 of the License, or (at your option) any later
00024  *   version.
00025  *
00026  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00027  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00028  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00029  *   details.
00030  *
```

```
00031  *   You should have received a copy of the GNU General Public License along
00032  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00033  */
00034 /* #] License : */
00035 /*
00036     #[ Includes :
00037
00038     The static variables for the messages can remain as such also for
00039     the parallel version as messages are to be locked to avoid problems
00040     with simultaneous messages.
00041 */
00042
00043 #include "form3.h"
00044
00045 static int iswarning = 0;
00046
00047 static char hex[] = {'0','1','2','3','4','5','6','7','8','9',
00048                      'A','B','C','D','E','F'};
00049
00050 /*
00051     #] Includes :
00052     #[ exit :
00053         #[ Error0 :
00054 */
00055
00056 NORETURN void Error0(char *s)
00057 {
00058     MesPrint("=== %s",s);
00059     Terminate(-1);
00060 }
00061
00062 /*
00063         #] Error0 :
00064         #[ Error1 :
00065 */
00066
00067 NORETURN void Error1(char *s, UBYTE *t)
00068 {
00069     MesPrint("@%s %s",s,t);
00070     Terminate(-1);
00071 }
00072
00073 /*
00074         #] Error1 :
00075         #[ Error2 :
00076 */
00077
00078 NORETURN void Error2(char *s1, char *s2, UBYTE *t)
00079 {
00080     MesPrint("@%s%s %s",s1,s2,t);
00081     Terminate(-1);
00082 }
00083
00084 /*
00085         #] Error2 :
00086         #[ MesWork :
00087 */
00088
00089 NORETURN void MesWork(void)
00090 {
00091     MesPrint("=== Workspace overflow. %l bytes is not enough.",AM.WorkSize);
00092     MesPrint("=== Change parameter WorkSpace in %s",setupfilename);
00093     Terminate(-1);
00094 }
00095
00096 /*
00097         #] MesWork :
00098         #[ MesPrint :
00099
00100     Kind of a printf function for simple messages.
00101     The main concern is getting the arguments in a portable way.
00102     Note: many compilers have errors when sizeof(WORD) < sizeof(int)
00103     %a  array of size n WORDs (two parameters, first is int, second WORD *)
00104     %b  array of size n UBYTEs (two parameters, first is int, second UBYTE *)
00105     %C  array of size n chars (two parameters, first is int, second char *)
00106     %d  word;
00107     %l  long;
00108     %L  long long *;
00109     %s  string;
00110     %#i unsigned word filled
00111     %#d word positioned
00112     %#l long word positioned.
00113     %#L long long word * positioned.
00114     %#s string positioned.
00115     %#p position in file.
00116     %r  The current term in raw format (internal representation)
00117     %t  The current term (AN.currentTerm)
```

```
00118     %T  The current term (AN.currentTerm) with its sign
00119     %w  Number of the thread(worker)
00120     %$  The next $ in AN.listinprint
00121     %x  hexadecimal. Takes 8 places. Mainly for debugging.
00122     %%  %
00123     %#  #
00124     #   " ==> "
00125     @   " ==> "   Preprocessor error
00126     &   ' --> '   Regular compiler error
00127     !>  ' ~~> '   Internal error
00128     Each call is terminated with a new line.
00129     Put a % at the end of the string to suppress the new line.
00130
00131     New feature (7-dec-2011): The & will only work when we do not block it
00132     from the execution of the print statement because we need the & also for
00133     the tabulator in the print "" statement.
00134 */
00135
00136 int MesPrint(const char *fmt, ... )
00137 {
00138     GETIDENTITY
00139     char Out[MAXLINELENGTH+14], *stopper, *t, *s, *u, c, *carray;
00140     UBYTE extrabuffer[MAXLINELENGTH+14];
00141     int w, x, i, specialerror = 0;
00142     LONG num, y;
00143     WORD *array;
00144     UBYTE *oldoutfill = AO.OutputLine, *barray;
00145     /*[19apr2004 mt]:*/
00146     LONG (*OldWrite)(int handle, UBYTE *buffer, LONG size) = WriteFile;
00147     /*:[19apr2004 mt]*/
00148     va_list ap;
00149     va_start(ap,fmt);
00150     s = (char *)fmt;
00151 #ifdef WITHMPI
00152     /*
00153      * On slaves, if AS.printflag is
00154      *   = 0 : print nothing.
00155      *   > 0 : synchronized output. All text will be sent to the master
00156      *         in the next MUNLOCK().
00157      *   < 0 : normal output.
00158      */
00159     if ( PF.me != MASTER && AS.printflag == 0 ) return(0);
00160     if ( PF.me == MASTER || AS.printflag < 0 )
00161 #endif
00162     FLUSHCONSOLE;
00163     /*
00164      * MesPrints() never prints a message to an external channel even if
00165      * WriteFile is set to &WriteToExternalChannel.
00166      */
00167 #ifdef WITHMPI
00168     WriteFile = PF.me == MASTER || AS.printflag > 0 ? &PF_WriteFileToFile : &WriteFileToFile;
00169 #else
00170     WriteFile = &WriteFileToFile;
00171 #endif
00172     AO.OutputLine = extrabuffer;
00173     t = Out;
00174     stopper = Out + AC.LineLength;
00175     while ( *s ) {
00176         if ( ( *s == '&' || *s == '@' || *s == '#' || ( *s == '!' && s[1] == '>' ) ) && AO.ErrorBlock
    == 0 && AC.CurrentStream != 0 ) {
00177             u = (char *)AC.CurrentStream->name;
00178             while ( *u ) {
00179                 *t++ = *u++;
00180                 if ( t >= stopper ) {
00181                     num = t - Out;
00182                     WriteString(ERROROUT,(UBYTE *)Out,num);
00183                     num = 0; t = Out;
00184                 }
00185             }
00186             *t++ = ' ';
00187             if ( t+20 >= stopper ) {
00188                 num = t - Out;
00189                 WriteString(ERROROUT,(UBYTE *)Out,num);
00190                 num = 0; t = Out;
00191             }
00192             *t++ = 'L'; *t++ = 'i'; *t++ = 'n'; *t++ = 'e'; *t++ = ' ';
00193             if ( *s == '&' ) y = AC.CurrentStream->prevline;
00194             else             y = AC.CurrentStream->linenumber;
00195             t = LongCopy(y,t);
00196             if ( !iswarning && ( *s == '&' || *s == '@' ) ) {
00197                 for ( i = 0; i < NumDoLoops; i++ ) DoLoops[i].errorsinloop = 1;
00198             }
00199         }
00200         if ( ( *s == '&' && AO.ErrorBlock == 0 ) ) {
00201             *t++ = ' '; *t++ = '-'; *t++ = '-'; *t++ = '>'; *t++ = ' '; s++;
00202         }
00203         else if ( ( *s == '@' || *s == '#' ) && AO.ErrorBlock == 0 ) {
```

```
00204             *t++ = ' '; *t++ = '='; *t++ = '='; *t++ = '>'; *t++ = ' '; s++;
00205         }
00206         else if ( *s == '!' && s[1] == '>' && AO.ErrorBlock == 0 ) {
00207             const char *m = " ~~> Internal error, please report with the following message:";
00208             while ( *m ) {
00209                 *t++ = *m++;
00210                 if ( t >= stopper ) {
00211                     num = t - Out;
00212                     WriteString(ERROROUT,(UBYTE *)Out,num);
00213                     num = 0; t = Out;
00214                 }
00215             }
00216             num = t - Out;
00217             WriteString(ERROROUT,(UBYTE *)Out,num);
00218             num = 0; t = Out;
00219             s += 2;
00220         }
00221 /*
00222         else if ( *s == '&' && AO.ErrorBlock == 1 ) {
00223
00224         }
00225 */
00226         else if ( *s != '%' ) {
00227             *t++ = *s++;
00228             if ( t >= stopper ) {
00229                 num = t - Out;
00230                 WriteString(ERROROUT,(UBYTE *)Out,num);
00231                 num = 0; t = Out;
00232             }
00233         }
00234         else {
00235             s++;
00236             if ( *s == 'd' ) {
00237                 if ( ( w = va_arg(ap, int) ) < 0 ) { *t++ = '-'; w = -w; }
00238                 t = (char *)NumCopy(w,(UBYTE *)t);
00239             }
00240             else if ( *s == 'l' ) {
00241                 if ( ( y = va_arg(ap, LONG) ) < 0 ) { *t++ = '-'; y = -y; }
00242                 t = LongCopy(y,t);
00243             }
00244 /*  #ifdef __GLIBC_HAVE_LONG_LONG */
00245             else if ( *s == 'p' ) {
00246                 POSITION *pp;
00247                 off_t ly;
00248                 pp = va_arg(ap, POSITION *);
00249                 ly = BASEPOSITION(*pp);
00250                 if ( ly < 0 ) { *t++ = '-'; ly = -ly; }
00251 /*----change 10-feb-2003 did not have & */
00252                 t = LongLongCopy(&(ly),t);
00253             }
00254 /*  #endif  */
00255             else if ( *s == 'c' ) {
00256                 c = (char)(va_arg(ap, int));
00257                 *t++ = c; *t = 0;
00258             }
00259             else if ( *s == 'a' ) {
00260                 w = va_arg(ap, int);
00261                 array = va_arg(ap,WORD *);
00262                 while ( w > 0 ) {
00263                     t = (char *)NumCopy(*array,(UBYTE *)t);
00264                     if ( t >= stopper ) {
00265                         num = t - Out;
00266                         WriteString(ERROROUT,(UBYTE *)Out,num);
00267                         t = Out;
00268                         *t++ = ' ';
00269                     }
00270                     *t++ = ' ';
00271                     w--; array++;
00272                 }
00273             }
00274             else if ( *s == 'b' ) {
00275                 w = va_arg(ap, int);
00276                 barray = va_arg(ap,UBYTE *);
00277                 while ( w > 0 ) {
00278                     *t++ = hex[((*barray)»4)&0xF];
00279                     *t++ = hex[(*barray)&0xF];
00280                     *t = 0;
00281                     if ( t >= stopper ) {
00282                         num = t - Out;
00283                         WriteString(ERROROUT,(UBYTE *)Out,num);
00284                         t = Out;
00285                         *t++ = ' ';
00286                     }
00287                     *t++ = ' ';
00288                     w--; barray++;
00289                 }
00290             }
```

```
00291             else if ( *s == 'C' ) {
00292                 w = va_arg(ap, int);
00293                 carray = va_arg(ap,char *);
00294                 while ( w > 0 ) {
00295                     if ( *carray < 32 ) *t++ = '^';
00296                     else *t++ = *carray;
00297                     *t = 0;
00298                     if ( t >= stopper ) {
00299                         num = t - Out;
00300                         WriteString(ERROROUT,(UBYTE *)Out,num);
00301                         t = Out;
00302                         *t++ = ' ';
00303                     }
00304                     w--; carray++;
00305                 }
00306             }
00307             else if ( *s == 'I' ) {
00308                 int *iarray;
00309                 w = va_arg(ap, int);
00310                 iarray = va_arg(ap,int *);
00311                 while ( w > 0 ) {
00312                     t = (char *)LongCopy((LONG)(*iarray),(char *)t);
00313                     if ( t >= stopper ) {
00314                         num = t - Out;
00315                         WriteString(ERROROUT,(UBYTE *)Out,num);
00316                         t = Out;
00317                         *t++ = ' ';
00318                     }
00319                     *t++ = ' ';
00320                     w--; array++;
00321                 }
00322             }
00323             else if ( *s == 'E' ) {
00324                 LONG *larray;
00325                 w = va_arg(ap, int);
00326                 larray = va_arg(ap,LONG *);
00327                 while ( w > 0 ) {
00328                     t = (char *)LongCopy(*larray,(char *)t);
00329                     if ( t >= stopper ) {
00330                         num = t - Out;
00331                         WriteString(ERROROUT,(UBYTE *)Out,num);
00332                         t = Out;
00333                         *t++ = ' ';
00334                     }
00335                     *t++ = ' ';
00336                     w--; array++;
00337                 }
00338             }
00339             else if ( *s == 's' ) {
00340                 u = va_arg(ap,char *);
00341                 while ( *u ) {
00342                     if ( t >= stopper ) {
00343                         num = t - Out;
00344                         WriteString(ERROROUT,(UBYTE *)Out,num);
00345                         t = Out;
00346                     }
00347                     *t++ = *u++;
00348                 }
00349                 *t = 0;
00350             }
00351             else if ( *s == 't' || *s == 'T' ) {
00352                 WORD oldskip = AO.OutSkip, noleadsign;
00353                 WORD oldmode = AC.OutputMode;
00354                 WORD oldbracket = AO.IsBracket;
00355                 WORD oldlength = AC.LineLength;
00356                 UBYTE *oldStop = AO.OutStop;
00357                 if ( AN.currentTerm ) {
00358                     if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
00359                     AO.IsBracket = 0;
00360                     AO.OutSkip = 1;
00361                     AC.OutputMode = 0;
00362                     AO.OutFill = AO.OutputLine;
00363                     AO.OutStop = AO.OutputLine + AC.LineLength;
00364                     *t = 0;
00365                     AddToLine((UBYTE *)Out);
00366                     if ( *s == 'T' ) noleadsign = 1;
00367                     else noleadsign = 0;
00368                     if ( WriteInnerTerm(AN.currentTerm,noleadsign) ) Terminate(-1);
00369                     t = Out;
00370                     u = (char *)AO.OutputLine;
00371                     *(AO.OutFill) = 0;
00372                     while ( u < (char *)(AO.OutFill) ) *t++ = *u++;
00373                     *t = 0;
00374                     AO.OutSkip = oldskip;
00375                     AC.OutputMode = oldmode;
00376                     AO.IsBracket = oldbracket;
00377                     AC.LineLength = oldlength;
```

```
00378                     AO.OutStop = oldStop;
00379                 }
00380             }
00381             else if ( *s == 'r' ) {
00382                 WORD oldskip = AO.OutSkip;
00383                 WORD oldmode = AC.OutputMode;
00384                 WORD oldbracket = AO.IsBracket;
00385                 WORD oldlength = AC.LineLength;
00386                 UBYTE *oldStop = AO.OutStop;
00387                 if ( AN.currentTerm ) {
00388                     WORD *tt = AN.currentTerm;
00389                     if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
00390                     AO.IsBracket = 0;
00391                     AO.OutSkip = 1;
00392                     AC.OutputMode = 0;
00393                     AO.OutFill = AO.OutputLine;
00394                     AO.OutStop = AO.OutputLine + AC.LineLength;
00395                     *t = 0;
00396                     i = *tt;
00397                     while ( --i >= 0 ) {
00398                         t = (char *)NumCopy(*tt,(UBYTE *)t);
00399                         tt++;
00400                         if ( t >= stopper ) {
00401                             num = t - Out;
00402                             WriteString(ERROROUT,(UBYTE *)Out,num);
00403                             num = 0; t = Out;
00404                         }
00405                         *t++ = ' '; *t++ = ' ';
00406                     }
00407                     *t = 0;
00408                     AO.OutSkip = oldskip;
00409                     AC.OutputMode = oldmode;
00410                     AO.IsBracket = oldbracket;
00411                     AC.LineLength = oldlength;
00412                     AO.OutStop = oldStop;
00413                 }
00414             }
00415             else if ( *s == '$' ) {
00416 /*
00417             #[ dollars :
00418 */
00419                 WORD oldskip = AO.OutSkip;
00420                 WORD oldmode = AC.OutputMode;
00421                 WORD oldbracket = AO.IsBracket;
00422                 WORD oldlength = AC.LineLength;
00423                 UBYTE *oldStop = AO.OutStop;
00424                 WORD *term, indsubterm[3], *tt;
00425                 WORD value[5], first, num;
00426                 if ( *AN.listinprint != DOLLAREXPRESSION ) {
00427                     specialerror = 1;
00428                 }
00429                 else {
00430                     DOLLARS d = Dollars + AN.listinprint[1];
00431 #ifdef WITHPTHREADS
00432                     int nummodopt, dtype;
00433                     dtype = -1;
00434                     if ( AS.MultiThreaded ) {
00435                         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
00436                             if ( AN.listinprint[1] == ModOptdollars[nummodopt].number ) break;
00437                         }
00438                         if ( nummodopt < NumModOptdollars ) {
00439                             dtype = ModOptdollars[nummodopt].type;
00440                             if ( DollarLocalCopy(dtype) ) {
00441                                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
00442                             }
00443                             else {
00444                                 LOCK(d->pthreadslock);
00445                             }
00446                         }
00447                     }
00448 #endif
00449                     AO.IsBracket = 0;
00450                     AO.OutSkip = 0;
00451                     AC.OutputMode = 0;
00452                     AO.OutFill = AO.OutputLine;
00453                     AO.OutStop = AO.OutputLine + AC.LineLength;
00454                     *t = 0;
00455                     AddToLine((UBYTE *)Out);
00456                     if ( d->nfactors >= 1 && AN.listinprint[2] == DOLLAREXPR2 ) {
00457                         if ( d->type == 0 ||
00458                          ( d->factors == 0 && d->nfactors != 1 ) ) goto dollarzero;
00459                         num = EvalDoLoopArg(BHEAD AN.listinprint+2,-1);
00460                         if ( num == 0 ) {
00461                             value[0] = 4; value[1] = d->nfactors; value[2] = 1; value[3] = 3; value[4]
      = 0;
00462                             term = value; goto printterms;
00463                         }
```

```
00464                         if ( num == 1 && d->nfactors == 1 ) {
00465                             term = d->where;
00466                             if ( *term == 0 ) goto dollarzero;
00467                             goto printterms;
00468                         }
00469                         if ( num > d->nfactors ) {
00470                             MesPrint("\nFactor number for dollar is too large.");
00471                             Terminate(-1);
00472                         }
00473                         term = d->factors[num-1].where;
00474                         if ( term == 0 ) {
00475                             if ( d->factors[num-1].value < 0 ) {
00476                                 value[0] = 4; value[1] = -d->factors[num-1].value; value[2] = 1;
      value[3] = -3; value[4] = 0;
00477                             }
00478                             else {
00479                                 value[0] = 4; value[1] = d->factors[num-1].value; value[2] = 1;
      value[3] = 3; value[4] = 0;
00480                             }
00481                             term = value;
00482                         }
00483                         goto printterms;
00484                     }
00485                     if ( d->type == DOLTERMS || d->type == DOLNUMBER ) {
00486                         term = d->where;
00487 printterms:             first = 1;
00488                         do {
00489                             if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
00490                             AO.IsBracket = 0;
00491                             AO.OutSkip = 1;
00492                             AC.OutputMode = 0;
00493                             AO.OutFill = AO.OutputLine;
00494                             AO.OutStop = AO.OutputLine + AC.LineLength;
00495                             *t = 0;
00496                             AddToLine((UBYTE *)Out);
00497                             if ( WriteInnerTerm(term,first) ) {
00498 #ifdef WITHPTHREADS
00499                                 if ( dtype > 0 && ! DollarLocalCopy(dtype) ) {
      UNLOCK(d->pthreadslock); }
00500 #endif
00501                                 Terminate(-1);
00502                             }
00503                             first = 0;
00504                             t = Out;
00505                             u = (char *)AO.OutputLine;
00506                             *(AO.OutFill) = 0;
00507                             while ( u < (char *)(AO.OutFill) ) *t++ = *u++;
00508                             *t = 0;
00509                             AO.OutSkip = oldskip;
00510                             AC.OutputMode = oldmode;
00511                             AO.IsBracket = oldbracket;
00512                             AC.LineLength = oldlength;
00513                             AO.OutStop = oldStop;
00514                             term += *term;
00515                         } while ( *term );
00516                         AO.OutSkip = oldskip;
00517                     }
00518                     else if ( d->type == DOLSUBTERM ) {
00519                         tt = d->where;
00520 dosubterm:              if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
00521                         AO.IsBracket = 0;
00522                         AO.OutSkip = 1;
00523                         AC.OutputMode = 0;
00524                         AO.OutFill = AO.OutputLine;
00525                         AO.OutStop = AO.OutputLine + AC.LineLength;
00526                         *t = 0;
00527                         AddToLine((UBYTE *)Out);
00528                         if ( WriteSubTerm(tt,1) ) {
00529 #ifdef WITHPTHREADS
00530                             if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
00531 #endif
00532                             Terminate(-1);
00533                         }
00534                         t = Out;
00535                         u = (char *)AO.OutputLine;
00536                         *(AO.OutFill) = 0;
00537                         while ( u < (char *)(AO.OutFill) ) *t++ = *u++;
00538                         *t = 0;
00539                         AO.OutSkip = oldskip;
00540                         AC.OutputMode = oldmode;
00541                         AO.IsBracket = oldbracket;
00542                         AC.LineLength = oldlength;
00543                         AO.OutStop = oldStop;
00544                     }
00545                     else if ( d->type == DOLUNDEFINED ) {
00546                         *t++ = '*'; *t++ = '*'; *t++ = '*'; *t = 0;
00547                     }
```

```
00548                     else if ( d->type == DOLZERO ) {
00549 dollarzero:            *t++ = '0'; *t = 0;
00550                     }
00551                     else if ( d->type == DOLINDEX ) {
00552                         tt = indsubterm; *tt = INDEX;
00553                         tt[1] = 3; tt[2] = d->index;
00554                         goto dosubterm;
00555                     }
00556                     else if ( d->type == DOLARGUMENT ) {
00557                         if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
00558                         AO.IsBracket = 0;
00559                         AO.OutSkip = 1;
00560                         AC.OutputMode = 0;
00561                         AO.OutFill = AO.OutputLine;
00562                         AO.OutStop = AO.OutputLine + AC.LineLength;
00563                         *t = 0;
00564                         AddToLine((UBYTE *)Out);
00565                         WriteArgument(d->where);
00566                         t = Out;
00567                         u = (char *)AO.OutputLine;
00568                         *(AO.OutFill) = 0;
00569                         while ( u < (char *)(AO.OutFill) ) *t++ = *u++;
00570                         *t = 0;
00571                         AO.OutSkip = oldskip;
00572                         AC.OutputMode = oldmode;
00573                         AO.IsBracket = oldbracket;
00574                         AC.LineLength = oldlength;
00575                         AO.OutStop = oldStop;
00576                     }
00577                     else if ( d->type == DOLWILDARGS ) {
00578                         tt = d->where;
00579                         if ( *tt == 0 ) { tt++;
00580                          while ( *tt ) {
00581                           if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
00582                           AO.IsBracket = 0;
00583                           AO.OutSkip = 1;
00584                           AC.OutputMode = 0;
00585                           AO.OutFill = AO.OutputLine;
00586                           AO.OutStop = AO.OutputLine + AC.LineLength;
00587                           *t = 0;
00588                           AddToLine((UBYTE *)Out);
00589                           WriteArgument(tt);
00590                           NEXTARG(tt);
00591                           if ( *tt ) TokenToLine((UBYTE *)",");
00592                           t = Out;
00593                           u = (char *)AO.OutputLine;
00594                           *(AO.OutFill) = 0;
00595                           while ( u < (char *)(AO.OutFill) ) *t++ = *u++;
00596                           *t = 0;
00597                           AO.OutSkip = oldskip;
00598                           AC.OutputMode = oldmode;
00599                           AO.IsBracket = oldbracket;
00600                           AC.LineLength = oldlength;
00601                           AO.OutStop = oldStop;
00602                          }
00603                         }
00604                         else if ( *tt > 0 ) {   /* Tensor arguments */
00605                             i = *tt++;
00606                             while ( --i >= 0 ) {
00607                                 indsubterm[0] = INDEX;
00608                                 indsubterm[1] = 3;
00609                                 indsubterm[2] = *tt++;
00610                                 if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
00611                                 AO.IsBracket = 0;
00612                                 AO.OutSkip = 1;
00613                                 AC.OutputMode = 0;
00614                                 AO.OutFill = AO.OutputLine;
00615                                 AO.OutStop = AO.OutputLine + AC.LineLength;
00616                                 *t = 0;
00617                                 AddToLine((UBYTE *)Out);
00618                                 if ( WriteSubTerm(indsubterm,1) ) Terminate(-1);
00619                                 if ( i > 0 ) TokenToLine((UBYTE *)",");
00620                                 t = Out;
00621                                 u = (char *)AO.OutputLine;
00622                                 *(AO.OutFill) = 0;
00623                                 while ( u < (char *)(AO.OutFill) ) *t++ = *u++;
00624                                 *t = 0;
00625                                 AO.OutSkip = oldskip;
00626                                 AC.OutputMode = oldmode;
00627                                 AO.IsBracket = oldbracket;
00628                                 AC.LineLength = oldlength;
00629                                 AO.OutStop = oldStop;
00630                             }
00631                         }
00632                     }
00633 #ifdef WITHPTHREADS
00634                     if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
```

```
00635 #endif
00636                     AN.listinprint += 2;
00637                     while ( AN.listinprint[0] == DOLLAREXPR2 ) AN.listinprint += 2;
00638                 }
00639 /*
00640             #] dollars :
00641 */
00642             }
00643 #ifdef WITHPTHREADS
00644             else if ( *s == 'W' ) { /* number of the thread with time */
00645                 LONG millitime;
00646                 WORD timepart;
00647                 t = (char *)NumCopy(identity,(UBYTE *)t);
00648                 millitime = TimeCPU(1);
00649                 timepart = (WORD)(millitime%1000);
00650                 millitime /= 1000;
00651                 timepart /= 10;
00652                 *t++ = '('; *t = 0;
00653                 t = (char *)LongCopy(millitime,(char *)t);
00654                 *t++ = '.'; *t = 0;
00655                 t = (char *)NumCopy(timepart,(UBYTE *)t);
00656                 *t++ = ')'; *t = 0;
00657                 if ( t >= stopper ) {
00658                     num = t - Out;
00659                     WriteString(ERROROUT,(UBYTE *)Out,num);
00660                     num = 0; t = Out;
00661                 }
00662             }
00663             else if ( *s == 'w' ) { /* number of the thread */
00664                 t = (char *)NumCopy(identity,(UBYTE *)t);
00665             }
00666 #elif defined(WITHMPI)
00667             else if ( *s == 'W' ) { /* number of the thread with time */
00668                 LONG millitime;
00669                 WORD timepart;
00670                 t = (char *)NumCopy(PF.me,(UBYTE *)t);
00671                 millitime = TimeCPU(1);
00672                 timepart = (WORD)(millitime%1000);
00673                 millitime /= 1000;
00674                 timepart /= 10;
00675                 *t++ = '('; *t = 0;
00676                 t = (char *)LongCopy(millitime,(char *)t);
00677                 *t++ = '.'; *t = 0;
00678                 t = (char *)NumCopy(timepart,(UBYTE *)t);
00679                 *t++ = ')'; *t = 0;
00680                 if ( t >= stopper ) {
00681                     num = t - Out;
00682                     WriteString(ERROROUT,(UBYTE *)Out,num);
00683                     num = 0; t = Out;
00684                 }
00685             }
00686             else if ( *s == 'w' ) { /* number of the thread */
00687                 t = (char *)NumCopy(PF.me,(UBYTE *)t);
00688             }
00689 #else
00690             else if ( *s == 'w' ) { }
00691             else if ( *s == 'W' ) { }
00692 #endif
00693             else if ( FG.cTable[(int)*s] == 1 ) {
00694                 x = *s++ - '0';
00695                 while ( FG.cTable[(int)*s] == 1 )
00696                     x = 10 * x + *s++ - '0';
00697
00698                 if ( *s == 'l' || *s == 'd' ) {
00699                     if ( *s == 'l' ) { y = va_arg(ap,LONG); }
00700                     else { y = va_arg(ap,int); }
00701                     if ( y < 0 ) { y = -y; w = 1; }
00702                     else w = 0;
00703                     u = t + x;
00704                     do { *--u = y%10+'0'; y /= 10; } while ( y && u > t );
00705                     if ( w && u > t ) *--u = '-';
00706                     while ( --u >= t ) *u = ' ';
00707                     t += x;
00708                 }
00709                 else if ( *s == 's' ) {
00710                     u = va_arg(ap,char *);
00711                     i = 0;
00712                     while ( *u ) { i++; u++; }
00713                     if ( i > x ) i = x;
00714                     while ( x > i ) { *t++ = ' '; x--; }
00715                     t += x;
00716                     while ( --i >= 0 ) { *--t = *--u; }
00717                     t += x;
00718                 }
00719                 else if ( *s == 'p' ) {
00720                     POSITION *pp;
00721 /*#ifdef __GLIBC_HAVE_LONG_LONG */
```

```
00722                     off_t ly;
00723 /*
00724 #else
00725                     LONG ly;
00726 #endif
00727 */
00728                     pp = va_arg(ap,POSITION *);
00729                     ly = BASEPOSITION(*pp);
00730                     u = t + x;
00731                     do { *--u = ly%10+'0'; ly /= 10; } while ( ly && u > t );
00732                     while ( --u >= t ) *u = ' ';
00733                     t += x;
00734                 }
00735                 else if ( *s == 'i' ) {
00736                     w = va_arg(ap, int);
00737                     u = t + x;
00738                     do { *--u = (char)(w%10+'0'); w /= 10; } while ( u > t );
00739                     t += x;
00740                 }
00741                 else {
00742                     w = va_arg(ap, int);
00743                     u = t + x;
00744                     do { *--u = (char )(w%10+'0'); w /= 10; } while ( w && u > t );
00745                     while ( --u >= t ) *u = ' ';
00746                     t += x;
00747                 }
00748             }
00749             else if ( *s == 'x' ) {
00750                 char ccc;
00751                 y = va_arg(ap, LONG);
00752                 i = 2*sizeof(LONG);
00753                 while ( --i > 0 ) {
00754                     ccc = ( y » (i*4) ) & 0xF;
00755                     if ( ccc ) break;
00756                 }
00757                 do {
00758                     ccc = ( y » (i*4) ) & 0xF;
00759                     *t++ = hex[(int)ccc];
00760                 } while ( --i >= 0 );
00761             }
00762             else if ( *s == '#' ) *t++ = *s;
00763             else if ( *s == '%' ) *t++ = *s;
00764             else if ( *s == 0 ) { *t++ = 0; break; }
00765             else if ( *s == '&' ) {
00766                 *t++ = *s;
00767             }
00768             else {
00769                 *t++ = '%';
00770                 s--;
00771             }
00772             s++;
00773         }
00774     }
00775     num = t - Out;
00776     WriteString(ERROROUT,(UBYTE *)Out,num);
00777     va_end(ap);
00778     if ( specialerror == 1 ) {
00779         MesPrint("!!!Wrong object in Print statement!!!");
00780         MesPrint("!!!Object encountered is of a different type as in the format specifier");
00781     }
00782     AO.OutputLine = oldoutfill;
00783     /*[19apr2004 mt]:*/
00784     WriteFile=OldWrite;
00785     /*:[19apr2004 mt]*/
00786     return(-1);
00787 }
00788
00789 /*
00790         #] MesPrint :
00791         #[ Warning :
00792 */
00793
00794 void Warning(char *s)
00795 {
00796     iswarning = 1;
00797     if ( AC.WarnFlag ) MesPrint("&Warning: %s",s);
00798     iswarning = 0;
00799 }
00800
00801 /*
00802         #] Warning :
00803         #[ HighWarning :
00804 */
00805
00806 void HighWarning(char *s)
00807 {
00808     iswarning = 1;
```

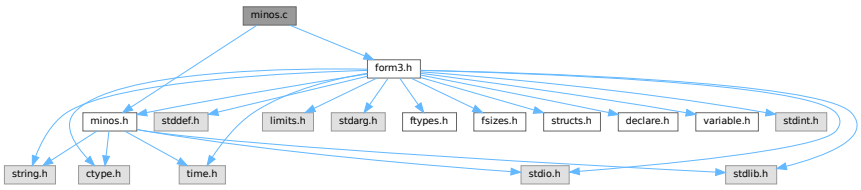
```
00809     if ( AC.WarnFlag >= 2 ) MesPrint("&Warning: %s",s);
00810     iswarning = 0;
00811 }
00812
00813 /*
00814         #] HighWarning :
00815         #[ MesCall :
00816 */
00817
00818 int MesCall(char *s)
00819 {
00820     return(MesPrint((char *)"Called from %s",s));
00821 }
00822
00823 /*
00824         #] MesCall :
00825         #[ MesCerr :
00826 */
00827
00828 int MesCerr(char *s, UBYTE *t)
00829 {
00830     UBYTE *u, c;
00831     WORD i = 11;
00832     u = t;
00833     while ( *u && --i >= 0 ) u--;
00834     u++;
00835     c = *++t;
00836     *t = 0;
00837     MesPrint("&Illegal %s: %s",s,u);
00838     *t = c;
00839     return(-1);
00840 }
00841
00842 /*
00843         #] MesCerr :
00844         #[ MesComp :
00845 */
00846
00847 int MesComp(char *s, UBYTE *p, UBYTE *q)
00848 {
00849     UBYTE c;
00850     c = *++q; *q = 0;
00851     MesPrint("&%s: %s",s,p);
00852     *q = c;
00853     return(-1);
00854 }
00855
00856 /*
00857         #] MesComp :
00858         #[ PrintTerm :
00859 */
00860
00861 void PrintTerm(WORD *term, char *where)
00862 {
00863     UBYTE OutBuf[140];
00864     WORD *t, x;
00865     int i;
00866     AO.OutFill = AO.OutputLine = OutBuf;
00867     t = term;
00868     AO.OutSkip = 3;
00869     FiniLine();
00870     TokenToLine((UBYTE *)where);
00871     TokenToLine((UBYTE *)": ");
00872     i = *t;
00873     while ( --i >= 0 ) {
00874         x = *t++;
00875         if ( x < 0 ) {
00876             x = -x;
00877             TokenToLine((UBYTE *)"-");
00878         }
00879         TalToLine((UWORD)(x));
00880         TokenToLine((UBYTE *)"  ");
00881     }
00882     AO.OutSkip = 0;
00883     FiniLine();
00884 }
00885
00886 /*
00887         #] PrintTerm :
00888         #[ PrintTermC :
00889 */
00890
00891 void PrintTermC(WORD *term, char *where)
00892 {
00893     UBYTE OutBuf[140];
00894     WORD *t, x;
00895     int i;
```

```
00896     if ( *term >= 0 ) {
00897         PrintTerm(term,where);
00898         return;
00899     }
00900     AO.OutFill = AO.OutputLine = OutBuf;
00901     t = term;
00902     AO.OutSkip = 3;
00903     FiniLine();
00904     TokenToLine((UBYTE *)where);
00905     TokenToLine((UBYTE *)": ");
00906     i = t[1]+2;
00907     while ( --i >= 0 ) {
00908         x = *t++;
00909         if ( x < 0 ) {
00910             x = -x;
00911             TokenToLine((UBYTE *)"-");
00912         }
00913         TalToLine((UWORD)(x));
00914         TokenToLine((UBYTE *)"  ");
00915     }
00916     AO.OutSkip = 0;
00917     FiniLine();
00918 }
00919
00920 /*
00921         #] PrintTermC :
00922         #[ PrintSubTerm :
00923 */
00924
00925 void PrintSubTerm(WORD *term, char *where)
00926 {
00927     UBYTE OutBuf[140];
00928     WORD *t;
00929     int i;
00930     AO.OutFill = AO.OutputLine = OutBuf;
00931     t = term;
00932     AO.OutSkip = 3;
00933     FiniLine();
00934     TokenToLine((UBYTE *)where);
00935     TokenToLine((UBYTE *)": ");
00936     i = t[1];
00937     while ( --i >= 0 ) { TalToLine((UWORD)(*t++)); TokenToLine((UBYTE *)"  "); }
00938     AO.OutSkip = 0;
00939     FiniLine();
00940 }
00941
00942 /*
00943         #] PrintSubTerm :
00944         #[ PrintWords :
00945 */
00946
00947 void PrintWords(WORD *buffer, LONG number)
00948 {
00949     UBYTE OutBuf[140];
00950     WORD *t;
00951     AO.OutFill = AO.OutputLine = OutBuf;
00952     t = buffer;
00953     AO.OutSkip = 3;
00954     FiniLine();
00955     while ( --number >= 0 ) { TalToLine((UWORD)(*t++)); TokenToLine((UBYTE *)"  "); }
00956     AO.OutSkip = 0;
00957     FiniLine();
00958 }
00959
00960 /*
00961         #] PrintWords :
00962         #[ PrintSeq :
00963 */
00964
00965 void PrintSeq(WORD *a,char *text)
00966 {
00967     MesPrint(" %s:",text);
00968     while ( *a ) {
00969         MesPrint("     %a",a[0],a);
00970         a += *a;
00971     }
00972 }
00973
00974 /*
00975         #] PrintSeq :
00976     #] exit :
00977 */
```

4.73 minos.c File Reference

```
#include "form3.h"
#include "minos.h"
```

Include dependency graph for minos.c:



Macros

- #define CFD(y, s, type, x, j)
- #define CTD(y, s, type, x, j)

Functions

- int minosread (FILE *f, char *buffer, MLONG size)
- int minoswrite (FILE *f, char *buffer, MLONG size)
- char * str_dup (char *str)
- void convertblock (INDEXBLOCK *in, INDEXBLOCK *out, int mode)
- void convertnamesblock (NAMESBLOCK *in, NAMESBLOCK *out, int mode)
- void convertiniinfo (INIIINFO *in, INIIINFO *out, int mode)
- FILE * LocateBase (char **name, char **newname, char *iomode)
- int ReadIndex (DBASE *d)
- int WriteIndexBlock (DBASE *d, MLONG num)
- int WriteNamesBlock (DBASE *d, MLONG num)
- int WriteIndex (DBASE *d)
- int WriteIniInfo (DBASE *d)
- int ReadIniInfo (DBASE *d)
- DBASE * GetDbase (char *filename, MLONG rwmode)
- DBASE * NewDbase (char *name, MLONG number)
- void FreeTableBase (DBASE *d)
- int ComposeTableNames (DBASE *d)
- DBASE * OpenDbase (char *filename)
- MLONG AddTableName (DBASE *d, char *name, TABLES T)
- MLONG GetTableName (DBASE *d, char *name)
- int PutTableNames (DBASE *d)
- int AddToIndex (DBASE *d, MLONG number)
- MLONG AddObject (DBASE *d, MLONG tablenumber, char *arguments, char *rhs)
- MLONG FindTableNumber (DBASE *d, char *name)
- int WriteObject (DBASE *d, MLONG tablenumber, char *arguments, char *rhs, MLONG number)
- char * ReadObject (DBASE *d, MLONG tablenumber, char *arguments)
- char * ReadijObject (DBASE *d, MLONG i, MLONG j, char *arguments)
- int ExistsObject (DBASE *d, MLONG tablenumber, char *arguments)
- int DeleteObject (DBASE *d, MLONG tablenumber, char *arguments)

Variables

- int withoutflush = 0

4.73.1 Detailed Description

These are the low level functions for the database part of the tablebases. These routines have been copied (and then adapted) from the minos database program. This file goes together with minos.h

Definition in file minos.c.

4.73.2 Macro Definition Documentation

4.73.2.1 CFD

```
#define CFD(
                y,
                s,
                type,
                x,
                j )
```

Value:

```
    for(x=0,j=0;j<((int)sizeof(type));j++) \
    {x=(x<<8)+((*s++)&0x00FF);} y=x;
```

Definition at line 55 of file minos.c.

4.73.2.2 CTD

```
#define CTD(
                y,
                s,
                type,
                x,
                j )
```

Value:

```
                    x=y;for(j=sizeof(type)-1;j>=0;j--){s[j]=x&0xFF; \
                    x>>=8;} s += sizeof(type);
```

Definition at line 57 of file minos.c.

4.73.3 Function Documentation

4.73.3.1 minosread()

```
int minosread (
              FILE * f,
              char * buffer,
              MLONG size )
```

Definition at line 66 of file minos.c.

4.73.3.2 minoswrite()

```
int minoswrite (
            FILE * f,
            char * buffer,
            MLONG size )
```

Definition at line 83 of file minos.c.

4.73.3.3 str_dup()

```
char * str_dup (
            char * str )
```

Definition at line 101 of file minos.c.

4.73.3.4 convertblock()

```
void convertblock (
            INDEXBLOCK * in,
            INDEXBLOCK * out,
            int mode )
```

Definition at line 120 of file minos.c.

4.73.3.5 convertnamesblock()

```
void convertnamesblock (
            NAMESBLOCK * in,
            NAMESBLOCK * out,
            int mode )
```

Definition at line 171 of file minos.c.

4.73.3.6 convertiniinfo()

```
void convertiniinfo (
            INIINFO * in,
            INIINFO * out,
            int mode )
```

Definition at line 197 of file minos.c.

4.73.3.7 LocateBase()

```
FILE * LocateBase (
            char ** name,
            char ** newname,
            char * iomode )
```

Definition at line 225 of file minos.c.

4.73.3.8 ReadIndex()

```
int ReadIndex (
            DBASE * d )
```

Definition at line 288 of file minos.c.

4.73.3.9 WriteIndexBlock()

```
int WriteIndexBlock (
            DBASE * d,
            MLONG num )
```

Definition at line 409 of file minos.c.

4.73.3.10 WriteNamesBlock()

```
int WriteNamesBlock (
            DBASE * d,
            MLONG num )
```

Definition at line 434 of file minos.c.

4.73.3.11 WriteIndex()

```
int WriteIndex (
            DBASE * d )
```

Definition at line 461 of file minos.c.

4.73.3.12 WriteIniInfo()

```
int WriteIniInfo (
            DBASE * d )
```

Definition at line 532 of file minos.c.

4.73.3.13 ReadIniInfo()

```
int ReadIniInfo (
            DBASE * d )
```

Definition at line 551 of file minos.c.

4.73.3.14 GetDbase()

```
DBASE * GetDbase (
            char * filename,
            MLONG rwmode )
```

Definition at line 578 of file minos.c.

4.73.3.15 NewDbase()

```
DBASE * NewDbase (
            char * name,
            MLONG number )
```

Definition at line 634 of file minos.c.

4.73.3.16 FreeTableBase()

```
void FreeTableBase (
            DBASE * d )
```

Definition at line 781 of file minos.c.

4.73.3.17 ComposeTableNames()

```
int ComposeTableNames (
            DBASE * d )
```

Definition at line 813 of file minos.c.

4.73.3.18 OpenDbase()

```
DBASE * OpenDbase (
            char * filename )
```

Definition at line 862 of file minos.c.

4.73.3.19 AddTableName()

```
MLONG AddTableName (
            DBASE * d,
            char * name,
            TABLES T )
```

Definition at line 896 of file minos.c.

4.73.3.20 GetTableName()

```
MLONG GetTableName (
            DBASE * d,
            char * name )
```

Definition at line 979 of file minos.c.

4.73.3.21 PutTableNames()

```
int PutTableNames (
            DBASE * d )
```

Definition at line 1011 of file minos.c.

4.73.3.22 AddToIndex()

```
int AddToIndex (
            DBASE * d,
            MLONG number )
```

Definition at line 1109 of file minos.c.

4.73.3.23 AddObject()

```
MLONG AddObject (
            DBASE * d,
            MLONG tablenumber,
            char * arguments,
            char * rhs )
```

Definition at line 1204 of file minos.c.

4.73.3.24 FindTableNumber()

```
MLONG FindTableNumber (
            DBASE * d,
            char * name )
```

Definition at line 1218 of file minos.c.

4.73.3.25 WriteObject()

```
int WriteObject (
            DBASE * d,
            MLONG tablenumber,
            char * arguments,
            char * rhs,
            MLONG number )
```

Definition at line 1246 of file minos.c.

4.73.3.26 ReadObject()

```
char * ReadObject (
             DBASE * d,
             MLONG tablenumber,
             char * arguments )
```

Definition at line 1350 of file minos.c.

4.73.3.27 ReadijObject()

```
char * ReadijObject (
             DBASE * d,
             MLONG i,
             MLONG j,
             char * arguments )
```

Definition at line 1435 of file minos.c.

4.73.3.28 ExistsObject()

```
int ExistsObject (
             DBASE * d,
             MLONG tablenumber,
             char * arguments )
```

Definition at line 1491 of file minos.c.

4.73.3.29 DeleteObject()

```
int DeleteObject (
             DBASE * d,
             MLONG tablenumber,
             char * arguments )
```

Definition at line 1523 of file minos.c.

4.73.4 Variable Documentation

4.73.4.1 withoutflush

```
int withoutflush = 0
```

Definition at line 45 of file minos.c.

4.74 minos.c

Go to the documentation of this file.

```
00001
00007 /* #[ License : */
00008 /*
00009  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00010  *   When using this file you are requested to refer to the publication
00011  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00012  *   This is considered a matter of courtesy as the development was paid
00013  *   for by FOM the Dutch physics granting agency and we would like to
00014  *   be able to track its scientific use to convince FOM of its value
00015  *   for the community.
00016  *
00017  *   This file is part of FORM.
00018  *
00019  *   FORM is free software: you can redistribute it and/or modify it under the
00020  *   terms of the GNU General Public License as published by the Free Software
00021  *   Foundation, either version 3 of the License, or (at your option) any later
00022  *   version.
00023  *
00024  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00025  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00026  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00027  *   details.
00028  *
00029  *   You should have received a copy of the GNU General Public License along
00030  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00031  */
00032 /* #] License : */
00033 /*
00034     #[ Includes :
00035
00036     File contains the lowlevel routines for the management of primitive
00037     database-like files. The structures are explained in the file manage.h
00038     Original file for minos made by J.Vermaseren, april-1994.
00039
00040     Note: the minos primitives for writing are never invoked in parallel.
00041 */
00042
00043 #include "form3.h"
00044 #include "minos.h"
00045 int withoutflush = 0;
00046
00047 /*
00048     #] Includes :
00049     #[ Variables :
00050 */
00051
00052 static INDEXBLOCK scratchblock;
00053 static NAMESBLOCK scratchnamesblock;
00054
00055 #define CFD(y,s,type,x,j) for(x=0,j=0;j<((int)sizeof(type));j++) \
00056     {x=(x<<8)+((*s++)&0x00FF);} y=x;
00057 #define CTD(y,s,type,x,j) x=y;for(j=sizeof(type)-1;j>=0;j--){s[j]=x&0xFF; \
00058                         x>>=8;} s += sizeof(type);
00059
00060 /*
00061     #] Variables :
00062     #[ Utilities :
00063         #[ minosread :
00064 */
00065
00066 int minosread(FILE *f,char *buffer,MLONG size)
00067 {
00068     MLONG x;
00069     while ( size > 0 ) {
00070         x = fread(buffer,sizeof(char),size,f);
00071         if ( x <= 0 ) return(-1);
00072         buffer += x;
00073         size -= x;
00074     }
00075     return(0);
00076 }
00077
00078 /*
00079         #] minosread :
00080         #[ minoswrite :
00081 */
00082
00083 int minoswrite(FILE *f,char *buffer,MLONG size)
00084 {
00085     MLONG x;
00086     while ( size > 0 ) {
00087         x = fwrite(buffer,sizeof(char),size,f);
```

```
00088         if ( x <= 0 ) return(-1);
00089         buffer += x;
00090         size -= x;
00091     }
00092     if ( withoutflush == 0 ) fflush(f);
00093     return(0);
00094 }
00095
00096 /*
00097         #] minoswrite :
00098         #[ str_dup :
00099 */
00100
00101 char *str_dup(char *str)
00102 {
00103     char *s, *t;
00104     int i;
00105     s = str;
00106     while ( *s ) s++;
00107     i = s - str + 1;
00108     if ( ( s = (char *)Malloc1((size_t)i,"a string copy") ) == 0 ) return(0);
00109     t = s;
00110     while ( *str ) *t++ = *str++;
00111     *t = 0;
00112     return(s);
00113 }
00114
00115 /*
00116         #] str_dup :
00117         #[ convertblock :
00118 */
00119
00120 void convertblock(INDEXBLOCK *in,INDEXBLOCK *out,int mode)
00121 {
00122     char *s,*t;
00123     MLONG i, x;
00124     int j;
00125     OBJECTS *obj;
00126     switch ( mode ) {
00127     case TODISK:
00128         s = (char *)out;
00129         CTD(in->flags,s,MLONG,x,j)
00130         CTD(in->previousblock,s,MLONG,x,j)
00131         CTD(in->position,s,MLONG,x,j)
00132         for ( i = 0, obj = in->objects; i < NUMOBJECTS; i++, obj++ ) {
00133             CTD(obj->position,s,MLONG,x,j)
00134             CTD(obj->size,s,MLONG,x,j)
00135             CTD(obj->date,s,MLONG,x,j)
00136             CTD(obj->tablenumber,s,MLONG,x,j)
00137             CTD(obj->uncompressed,s,MLONG,x,j)
00138             CTD(obj->spare1,s,MLONG,x,j)
00139             CTD(obj->spare2,s,MLONG,x,j)
00140             CTD(obj->spare3,s,MLONG,x,j)
00141             t = obj->element;
00142             for ( j = 0; j < ELEMENTSIZE; j++ ) *s++ = *t++;
00143         }
00144         break;
00145     case FROMDISK:
00146         s = (char *)in;
00147         CFD(out->flags,s,MLONG,x,j)
00148         CFD(out->previousblock,s,MLONG,x,j)
00149         CFD(out->position,s,MLONG,x,j)
00150         for ( i = 0, obj = out->objects; i < NUMOBJECTS; i++, obj++ ) {
00151             CFD(obj->position,s,MLONG,x,j)
00152             CFD(obj->size,s,MLONG,x,j)
00153             CFD(obj->date,s,MLONG,x,j)
00154             CFD(obj->tablenumber,s,MLONG,x,j)
00155             CFD(obj->uncompressed,s,MLONG,x,j)
00156             CFD(obj->spare1,s,MLONG,x,j)
00157             CFD(obj->spare2,s,MLONG,x,j)
00158             CFD(obj->spare3,s,MLONG,x,j)
00159             t = obj->element;
00160             for ( j = 0; j < ELEMENTSIZE; j++ ) *t++ = *s++;
00161         }
00162         break;
00163     }
00164 }
00165
00166 /*
00167         #] convertblock :
00168         #[ convertnamesblock :
00169 */
00170
00171 void convertnamesblock(NAMESBLOCK *in,NAMESBLOCK *out,int mode)
00172 {
00173     char *s;
00174     MLONG x;
```

```
00175     int j;
00176     switch ( mode ) {
00177     case TODISK:
00178         s = (char *)out;
00179         CTD(in->previousblock,s,MLONG,x,j)
00180         CTD(in->position,s,MLONG,x,j)
00181         for ( j = 0; j < NAMETABLESIZE; j++ ) out->names[j] = in->names[j];
00182         break;
00183     case FROMDISK:
00184         s = (char *)in;
00185         CFD(out->previousblock,s,MLONG,x,j)
00186         CFD(out->position,s,MLONG,x,j)
00187         for ( j = 0; j < NAMETABLESIZE; j++ ) out->names[j] = in->names[j];
00188         break;
00189     }
00190 }
00191
00192 /*
00193         #] convertnamesblock :
00194         #[ convertiniinfo :
00195 */
00196
00197 void convertiniinfo(INIINFO *in,INIINFO *out,int mode)
00198 {
00199     char *s;
00200     MLONG i, x, *y;
00201     int j;
00202     switch ( mode ) {
00203     case TODISK:
00204         s = (char *)out; y = (MLONG *)in;
00205         for ( i = sizeof(INIINFO)/sizeof(MLONG); i > 0; i-- ) {
00206             CTD(*y,s,MLONG,x,j)
00207             y++;
00208         }
00209         break;
00210     case FROMDISK:
00211         s = (char *)in; y = (MLONG *)out;
00212         for ( i = sizeof(INIINFO)/sizeof(MLONG); i > 0; i-- ) {
00213             CFD(*y,s,MLONG,x,j)
00214             y++;
00215         }
00216         break;
00217     }
00218 }
00219
00220 /*
00221         #] convertiniinfo :
00222         #[ LocateBase :
00223 */
00224
00225 FILE *LocateBase(char **name, char **newname, char *iomode)
00226 {
00227     FILE *handle;
00228     int  namesize, i;
00229     UBYTE *s, *to, *u1, *u2, *indir;
00230
00231     if ( ( handle = fopen(*name,iomode) ) != 0 ) {
00232         *newname = (char *)strDup1((UBYTE *)(*name),"LocateBase");
00233         return(handle);
00234     }
00235     namesize = 2; s = (UBYTE *)(*name);
00236     while ( *s ) { s++; namesize++; }
00237     indir = AM.IncDir;
00238     if ( indir ) {
00239         s = indir; i = 0;
00240         while ( *s ) { s++; i++; }
00241         *newname = (char *)Malloc1(namesize+i,"LocateBase");
00242         s = indir; to = (UBYTE *)(*newname);
00243         while ( *s ) *to++ = *s++;
00244         if ( to > (UBYTE *)(*newname) && to[-1] != SEPARATOR ) *to++ = SEPARATOR;
00245         s = (UBYTE *)(*name);
00246         while ( *s ) *to++ = *s++;
00247         *to = 0;
00248         if ( ( handle = fopen(*newname,iomode) ) != 0 ) {
00249             return(handle);
00250         }
00251         M_free(*newname,"LocateBase, incdir/file");
00252     }
00253     if ( AM.Path ) {
00254         u1 = AM.Path;
00255         while ( *u1 ) {
00256             u2 = u1; i = 0;
00257             while ( *u1 && *u1 != ':' ) {
00258                 if ( *u1 == '\\' ) u1++;
00259                 u1++; i++;
00260             }
00261             *newname = (char *)Malloc1(namesize+i,"LocateBase");
```

```
00262             s = u2; to = (UBYTE *)(*newname);
00263             while ( s < u1 ) {
00264                 if ( *s == '\\' ) s++;
00265                 *to++ = *s++;
00266             }
00267             if ( to > (UBYTE *)(*newname) && to[-1] != SEPARATOR ) *to++ = SEPARATOR;
00268             s = (UBYTE *)(*name);
00269             while ( *s ) *to++ = *s++;
00270             *to = 0;
00271             if ( ( handle = fopen(*newname,iomode) ) != 0 ) {
00272                 return(handle);
00273             }
00274             M_free(*newname,"LocateBase Path/file");
00275             if ( *u1 ) u1++;
00276         }
00277     }
00278 /*  Error1("LocateBase: Cannot find file",*name); */
00279     return(0);
00280 }
00281
00282 /*
00283         #] LocateBase :
00284     #] Utilities :
00285     #[ ReadIndex :
00286 */
00287
00288 int ReadIndex(DBASE *d)
00289 {
00290     MLONG i;
00291     INDEXBLOCK **ib;
00292     NAMESBLOCK **ina;
00293     MLONG position, size;
00294 /*
00295     Allocate the pieces one by one (makes it easier to give it back)
00296 */
00297     if ( d->info.numberofindexblocks <= 0 ) return(0);
00298     if ( sizeof(INDEXBLOCK)*d->info.numberofindexblocks > MAXINDEXSIZE ) {
00299 /* INTERNAL_ERROR_EXCL_START */
00300         MesPrint("!>We need more than %ld bytes for the index.\n",MAXINDEXSIZE);
00301         MesPrint("The file %s may not be a proper database\n",d->name);
00302         return(-1);
00303 /* INTERNAL_ERROR_EXCL_STOP */
00304     }
00305     size = sizeof(INDEXBLOCK *)*d->info.numberofindexblocks;
00306     if ( ( ib = (INDEXBLOCK **)Malloc1(size,"tb,index") ) == 0 ) return(-1);
00307     for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
00308         if ( ( ib[i] = (INDEXBLOCK *)Malloc1(sizeof(INDEXBLOCK),"index block") ) == 0 ) {
00309             for ( --i; i >= 0; i-- ) M_free(ib[i],"tb,indexblock");
00310             M_free(ib,"tb,index");
00311             return(-1);
00312         }
00313     }
00314     size = sizeof(NAMESBLOCK *)*d->info.numberofnamesblocks;
00315     if ( ( ina = (NAMESBLOCK **)Malloc1(size,"tb,indexnames") ) == 0 ) return(-1);
00316     for ( i = 0; i < d->info.numberofnamesblocks; i++ ) {
00317         if ( ( ina[i] = (NAMESBLOCK *)Malloc1(sizeof(NAMESBLOCK),"index names block") ) == 0 ) {
00318             for ( --i; i >= 0; i-- ) M_free(ina[i],"index names block");
00319             M_free(ina,"tb,indexnames");
00320             for ( i = 0; i < d->info.numberofindexblocks; i++ ) M_free(ib[i],"tb,indexblock");
00321             M_free(ib,"tb,index");
00322             return(-1);
00323         }
00324     }
00325 /*
00326     Read the index blocks, from the back to the front. The links are only
00327     reliable that way.
00328 */
00329     position = d->info.lastindexblock;
00330     for ( i = d->info.numberofindexblocks - 1; i >= 0; i-- ) {
00331         fseek(d->handle,position,SEEK_SET);
00332         if ( minosread(d->handle,(char *)(&scratchblock),sizeof(INDEXBLOCK)) ) {
00333 /* INTERNAL_ERROR_EXCL_START */
00334             MesPrint("!>Error while reading file %s\n",d->name);
00335 thisiswrong:
00336             for ( i = 0; i < d->info.numberofnamesblocks; i++ ) M_free(ina[i],"index names block");
00337             M_free(ina,"tb,indexnames");
00338             for ( i = 0; i < d->info.numberofindexblocks; i++ ) M_free(ib[i],"tb,indexblock");
00339             M_free(ib,"tb,index");
00340             return(-1);
00341 /* INTERNAL_ERROR_EXCL_STOP */
00342         }
00343         convertblock(&scratchblock,ib[i],FROMDISK);
00344         if ( ib[i]->position != position ||
00345         ( ib[i]->previousblock <= 0 && i > 0 ) ) {
00346 /* INTERNAL_ERROR_EXCL_START */
00347             MesPrint("!>File %s has inconsistent contents\n",d->name);
00348             goto thisiswrong;
```

```
00349 /* INTERNAL_ERROR_EXCL_STOP */
00350         }
00351         position = ib[i]->previousblock;
00352     }
00353     d->info.firstindexblock = ib[0]->position;
00354     for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
00355         ib[i]->flags &= MCLEANFLAG;
00356     }
00357 /*
00358     Read the names blocks, from the back to the front. The links are only
00359     reliable that way.
00360 */
00361     position = d->info.lastnameblock;
00362     for ( i = d->info.numberofnamesblocks - 1; i >= 0; i-- ) {
00363         fseek(d->handle,position,SEEK_SET);
00364         if ( minosread(d->handle,(char *)(&scratchnamesblock),sizeof(NAMESBLOCK)) ) {
00365 /* INTERNAL_ERROR_EXCL_START */
00366             MesPrint("!>Error while reading file %s\n",d->name);
00367             goto thisiswrong;
00368 /* INTERNAL_ERROR_EXCL_STOP */
00369         }
00370         convertnamesblock(&scratchnamesblock,ina[i],FROMDISK);
00371         if ( ina[i]->position != position ||
00372         ( ina[i]->previousblock <= 0 && i > 0 ) ) {
00373 /* INTERNAL_ERROR_EXCL_START */
00374             MesPrint("!>File %s has inconsistent contents\n",d->name);
00375             goto thisiswrong;
00376 /* INTERNAL_ERROR_EXCL_STOP */
00377         }
00378         position = ina[i]->previousblock;
00379     }
00380     d->info.firstnameblock = ina[0]->position;
00381 /*
00382     Give the old info back to the system.
00383 */
00384     if ( d->iblocks ) {
00385         for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
00386             if ( d->iblocks[i] ) M_free(d->iblocks[i],"d->iblocks[i]");
00387         }
00388         M_free(d->iblocks,"d->iblocks");
00389     }
00390     if ( d->nblocks ) {
00391         for ( i = 0; i < d->info.numberofnamesblocks; i++ ) {
00392             if ( d->nblocks[i] ) M_free(d->nblocks[i],"d->nblocks[i]");
00393         }
00394         M_free(d->nblocks,"d->nblocks");
00395     }
00396 /*
00397     And substitute the new blocks
00398 */
00399     d->iblocks = ib;
00400     d->nblocks = ina;
00401     return(0);
00402 }
00403
00404 /*
00405     #] ReadIndex :
00406     #[ WriteIndexBlock :
00407 */
00408
00409 int WriteIndexBlock(DBASE *d,MLONG num)
00410 {
00411     if ( num >= d->info.numberofindexblocks ) {
00412 /* INTERNAL_ERROR_EXCL_START */
00413         MesPrint("!>Illegal number specified for number of index blocks\n");
00414         return(-1);
00415 /* INTERNAL_ERROR_EXCL_STOP */
00416     }
00417     fseek(d->handle,d->iblocks[num]->position,SEEK_SET);
00418     convertblock(d->iblocks[num],&scratchblock,TODISK);
00419     if ( minoswrite(d->handle,(char *)(&scratchblock),sizeof(INDEXBLOCK)) ) {
00420 /* INTERNAL_ERROR_EXCL_START */
00421         MesPrint("!>Error while writing an index block in file %s\n",d->name);
00422         MesPrint("File may be unreliable now\n");
00423         return(-1);
00424 /* INTERNAL_ERROR_EXCL_STOP */
00425     }
00426     return(0);
00427 }
00428
00429 /*
00430     #] WriteIndexBlock :
00431     #[ WriteNamesBlock :
00432 */
00433
00434 int WriteNamesBlock(DBASE *d,MLONG num)
00435 {
```

```
00436     if ( num >= d->info.numberofnamesblocks ) {
00437 /* INTERNAL_ERROR_EXCL_START */
00438         MesPrint("!>Illegal number specified for number of names blocks\n");
00439         return(-1);
00440 /* INTERNAL_ERROR_EXCL_STOP */
00441     }
00442     fseek(d->handle,d->nblocks[num]->position,SEEK_SET);
00443     convertnamesblock(d->nblocks[num],&scratchnamesblock,TODISK);
00444     if ( minoswrite(d->handle,(char *)(&scratchnamesblock),sizeof(NAMESBLOCK)) ) {
00445 /* INTERNAL_ERROR_EXCL_START */
00446         MesPrint("!>Error while writing a names block in file %s\n",d->name);
00447         MesPrint("File may be unreliable now\n");
00448         return(-1);
00449 /* INTERNAL_ERROR_EXCL_STOP */
00450     }
00451     return(0);
00452 }
00453
00454 /*
00455     #] WriteNamesBlock :
00456     #[ WriteIndex :
00457
00458     Problem here is to get the links right.
00459 */
00460
00461 int WriteIndex(DBASE *d)
00462 {
00463     MLONG i, position;
00464     if ( d->iblocks == 0 ) return(0);
00465     if ( d->nblocks == 0 ) return(0);
00466     for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
00467         if ( d->iblocks[i] == 0 ) {
00468 /* INTERNAL_ERROR_EXCL_START */
00469             MesPrint("!>Error: unassigned index blocks. Cannot write\n");
00470             return(-1);
00471 /* INTERNAL_ERROR_EXCL_STOP */
00472         }
00473     }
00474     for ( i = 0; i < d->info.numberofnamesblocks; i++ ) {
00475         if ( d->nblocks[i] == 0 ) {
00476 /* INTERNAL_ERROR_EXCL_START */
00477             MesPrint("!>Error: unassigned names blocks. Cannot write\n");
00478             return(-1);
00479 /* INTERNAL_ERROR_EXCL_STOP */
00480         }
00481     }
00482     d->info.lastindexblock = -1;
00483     for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
00484         position = d->iblocks[i]->position;
00485         if ( position <= 0 ) {
00486             fseek(d->handle,0,SEEK_END);
00487             position = ftell(d->handle);
00488             d->iblocks[i]->position = position;
00489             if ( i <= 0 ) d->iblocks[i]->previousblock = -1;
00490             else d->iblocks[i]->previousblock = d->iblocks[i-1]->position;
00491         }
00492         else fseek(d->handle,position,SEEK_SET);
00493         convertblock(d->iblocks[i],&scratchblock,TODISK);
00494         if ( minoswrite(d->handle,(char *)(&scratchblock),sizeof(INDEXBLOCK)) ) {
00495 /* INTERNAL_ERROR_EXCL_START */
00496             MesPrint("!>Error while writing index of file %s",d->name);
00497             d->iblocks[i]->position = -1;
00498             return(-1);
00499 /* INTERNAL_ERROR_EXCL_STOP */
00500         }
00501         d->info.lastindexblock = position;
00502     }
00503     d->info.lastnameblock = -1;
00504     for ( i = 0; i < d->info.numberofnamesblocks; i++ ) {
00505         position = d->nblocks[i]->position;
00506         if ( position <= 0 ) {
00507             fseek(d->handle,0,SEEK_END);
00508             position = ftell(d->handle);
00509             d->nblocks[i]->position = position;
00510             if ( i <= 0 ) d->nblocks[i]->previousblock = -1;
00511             else d->nblocks[i]->previousblock = d->nblocks[i-1]->position;
00512         }
00513         else fseek(d->handle,position,SEEK_SET);
00514         convertnamesblock(d->nblocks[i],&scratchnamesblock,TODISK);
00515         if ( minoswrite(d->handle,(char *)(&scratchnamesblock),sizeof(NAMESBLOCK)) ) {
00516 /* INTERNAL_ERROR_EXCL_START */
00517             MesPrint("!>Error while writing index of file %s",d->name);
00518             d->nblocks[i]->position = -1;
00519             return(-1);
00520 /* INTERNAL_ERROR_EXCL_STOP */
00521         }
00522         d->info.lastnameblock = position;
```

```
00523     }
00524     return(0);
00525 }
00526 
00527 /*
00528     #] WriteIndex :
00529     #[ WriteIniInfo :
00530 */
00531 
00532 int WriteIniInfo(DBASE *d)
00533 {
00534     INIINFO inf;
00535     fseek(d->handle,0,SEEK_SET);
00536     convertiniinfo(&(d->info),&inf,TODISK);
00537     if ( minoswrite(d->handle,(char *)(&inf),sizeof(INIINFO)) ) {
00538 /* INTERNAL_ERROR_EXCL_START */
00539         MesPrint("!>Error while writing masterindex of file %s",d->name);
00540         return(-1);
00541 /* INTERNAL_ERROR_EXCL_STOP */
00542     }
00543     return(0);
00544 }
00545 
00546 /*
00547     #] WriteIniInfo :
00548     #[ ReadIniInfo :
00549 */
00550 
00551 int ReadIniInfo(DBASE *d)
00552 {
00553     INIINFO inf;
00554     fseek(d->handle,0,SEEK_SET);
00555     if ( minosread(d->handle,(char *)(&inf),sizeof(INIINFO)) ) {
00556 /* INTERNAL_ERROR_EXCL_START */
00557         MesPrint("!>Error while reading masterindex of file %s",d->name);
00558         return(-1);
00559 /* INTERNAL_ERROR_EXCL_STOP */
00560     }
00561     convertiniinfo(&inf,&(d->info),FROMDISK);
00562     if ( d->info.entriesinindex < 0
00563     || d->info.numberofindexblocks < 0
00564     || d->info.lastindexblock < 0 ) {
00565 /* INTERNAL_ERROR_EXCL_START */
00566         MesPrint("!>The file %s is not a proper database\n",d->name);
00567         return(-1);
00568 /* INTERNAL_ERROR_EXCL_STOP */
00569     }
00570     return(0);
00571 }
00572 
00573 /*
00574     #] ReadIniInfo :
00575     #[ GetDbase :
00576 */
00577 
00578 DBASE *GetDbase(char *filename, MLONG rwmode)
00579 {
00580     FILE *f;
00581     DBASE *d;
00582     char *newname;
00583     if ( rwmode == 0 ) {
00584         if ( ( f = LocateBase(&filename,&newname,"rb") ) == 0 ) {
00585 
00586             MesPrint("&Trying to open non-existent TableBase in readonly mode: %s", filename);
00587             Terminate(-1);
00588         }
00589     } else {
00590         if ( ( f = LocateBase(&filename,&newname,"r+b") ) == 0 ) {
00591 
00592             return(NewDbase(filename,0));
00593         }
00594     }
00595 
00596 /*  setbuf(f,0); */
00597     d = (DBASE *)From0List(&(AC.TableBaseList));
00598     d->mode = 0;
00599     d->tablenamessize = 0;
00600     d->topnumber = 0;
00601     d->tablenamefill = 0;
00602     d->iblocks = 0;
00603     d->nblocks = 0;
00604     d->tablenames = 0;
00605     d->rwmode = rwmode;
00606 
00607     d->info.entriesinindex = 0;
00608     d->info.numberofindexblocks = 0;
00609     d->info.firstindexblock = 0;
```

```
00610     d->info.lastindexblock = 0;
00611     d->info.numberoftables = 0;
00612     d->info.numberofnamesblocks = 0;
00613     d->info.firstnameblock = 0;
00614     d->info.lastnameblock = 0;
00615 
00616     d->name = str_dup(filename); /* For the moment just for the error messages */
00617     d->handle = f;
00618     if ( ReadIniInfo(d) || ReadIndex(d) ) { M_free(d,"index-d"); fclose(f); return(0); }
00619     if ( ComposeTableNames(d) < 0 ) { FreeTableBase(d); fclose(f); return(0); }
00620     // free allocation from previous str_dup
00621     M_free(d->name, "from str_dup");
00622     d->name = str_dup(filename);
00623     d->fullname = newname;
00624     return(d);
00625 }
00626 
00627 /*
00628     #] GetDbase :
00629     #[ NewDbase :
00630 
00631     Creates a new database with 'number' entries in the index.
00632 */
00633 
00634 DBASE *NewDbase(char *name,MLONG number)
00635 {
00636     FILE *f;
00637     DBASE *d;
00638     MLONG numblocks, numnameblocks, i;
00639     char *s;
00640 /*----------change 10-feb-2003 */
00641     int j, jj;
00642     MLONG t = (MLONG)(time(0));
00643 /*-----------------------------*/
00644     if ( number < 0 ) number = 0;
00645     if ( ( f = fopen(name,"w+b") ) == 0 ) {
00646         MesPrint("Could not create a new file with name %s\n",name);
00647         return(0);
00648     }
00649     numblocks = (number+NUMOBJECTS-1)/NUMOBJECTS;
00650     numnameblocks = 1;
00651     if ( numblocks <= 0 ) numblocks = 1;
00652     if ( numnameblocks <= 0 ) numnameblocks = 1;
00653     d = (DBASE *)From0List(&(AC.TableBaseList));
00654     if ( ( d->iblocks = (INDEXBLOCK **)Malloc1(numblocks*sizeof(INDEXBLOCK *),
00655     "new database") ) == 0 ) {
00656         NumTableBases--;
00657         fclose(f);
00658         return(0);
00659     }
00660     d->tablenames = 0;
00661     d->tablenamessize = 0;
00662     d->topnumber = 0;
00663     d->tablenamefill = 0;
00664     d->rwmode = 1;
00665 
00666     d->mode = 0;
00667     if ( ( d->nblocks = (NAMESBLOCK **)Malloc1(sizeof(NAMESBLOCK *)*numnameblocks,
00668     "new database") ) == 0 ) {
00669         M_free(d->iblocks,"new database");
00670         NumTableBases--;
00671         fclose(f);
00672         return(0);
00673     }
00674     if ( ( f = fopen(name,"w+b") ) == 0 ) {
00675         MesPrint("Could not create new file %s\n",name);
00676         NumTableBases--;
00677         return(0);
00678     }
00679 /*  setbuf(f,0); */
00680     d->name = str_dup(name);
00681     d->fullname = str_dup(name);
00682     d->handle = f;
00683 
00684     d->info.entriesinindex = number;
00685     d->info.numberofindexblocks = numblocks;
00686     d->info.numberofnamesblocks = numnameblocks;
00687     d->info.firstindexblock = 0;
00688     d->info.lastindexblock = 0;
00689     d->info.numberoftables = 0;
00690     d->info.firstnameblock = 0;
00691     d->info.lastnameblock = 0;
00692 
00693     if ( WriteIniInfo(d) ) {
00694 getout:
00695 /* INTERNAL_ERROR_EXCL_START */
00696         fclose(f);
```

```
00697         remove(d->fullname);
00698         if ( d->name ) { M_free(d->name,"name tablebase"); d->name = 0; }
00699         if ( d->fullname ) { M_free(d->fullname,"fullname tablebase"); d->fullname = 0; }
00700         M_free(d->nblocks,"new database");
00701         M_free(d->iblocks,"new database");
00702         NumTableBases--;
00703         return(0);
00704 /* INTERNAL_ERROR_EXCL_STOP */
00705     }
00706     for ( i = 0; i < numblocks; i++ ) {
00707         if ( ( d->iblocks[i] = (INDEXBLOCK *)Malloc1(sizeof(INDEXBLOCK),
00708         "index blocks of new database") ) == 0 ) {
00709 /* INTERNAL_ERROR_EXCL_START */
00710             while ( --i >= 0 ) M_free(d->iblocks[i],"index blocks of new database");
00711             goto getout;
00712 /* INTERNAL_ERROR_EXCL_STOP */
00713         }
00714         if ( i > 0 ) d->iblocks[i]->previousblock = d->iblocks[i-1]->position;
00715         else d->iblocks[i]->previousblock = -1;
00716         d->iblocks[i]->position = ftell(f);
00717         // Initialise, to keep valgrind happy
00718         d->iblocks[i]->flags = -1;
00719 /*----------change 10-feb-2003 */
00720 /*
00721             Zero things properly. We don't want garbage in the file.
00722 */
00723             for ( j = 0; j < NUMOBJECTS; j++ ) {
00724                 d->iblocks[i]->objects[j].date = t;
00725                 d->iblocks[i]->objects[j].size = 0;
00726                 d->iblocks[i]->objects[j].position = -1;
00727                 d->iblocks[i]->objects[j].tablenumber = 0;
00728                 d->iblocks[i]->objects[j].uncompressed = 0;
00729                 d->iblocks[i]->objects[j].spare1 = 0;
00730                 d->iblocks[i]->objects[j].spare2 = 0;
00731                 d->iblocks[i]->objects[j].spare3 = 0;
00732                 for ( jj = 0; jj < ELEMENTSIZE; jj++ ) d->iblocks[i]->objects[j].element[jj] = 0;
00733             }
00734         convertblock(d->iblocks[i],&scratchblock,TODISK);
00735         if ( minoswrite(d->handle,(char *)(&scratchblock),sizeof(INDEXBLOCK)) ) {
00736 /* INTERNAL_ERROR_EXCL_START */
00737             MesPrint("!>Error while writing new index blocks\n");
00738             goto getout;
00739 /* INTERNAL_ERROR_EXCL_STOP */
00740         }
00741     }
00742     for ( i = 0; i < numnameblocks; i++ ) {
00743         if ( ( d->nblocks[i] = (NAMESBLOCK *)Malloc1(sizeof(NAMESBLOCK),
00744         "names blocks of new database") ) == 0 ) {
00745             while ( --i >= 0 ) { M_free(d->nblocks[i],"names blocks of new database"); }
00746             for ( i = 0; i < numblocks; i++ ) M_free(d->iblocks[i],"index blocks of new database");
00747             goto getout;
00748         }
00749         if ( i > 0 ) d->nblocks[i]->previousblock = d->nblocks[i-1]->position;
00750         else d->nblocks[i]->previousblock = -1;
00751         d->nblocks[i]->position = ftell(f);
00752         s = d->nblocks[i]->names;
00753         for ( j = 0; j < NAMETABLESIZE; j++ ) *s++ = 0;
00754         convertnamesblock(d->nblocks[i],&scratchnamesblock,TODISK);
00755         if ( minoswrite(d->handle,(char *)(&scratchnamesblock),sizeof(NAMESBLOCK)) ) {
00756 /* INTERNAL_ERROR_EXCL_START */
00757             MesPrint("!>Error while writing new names blocks\n");
00758             for ( i = 0; i < numnameblocks; i++ ) M_free(d->nblocks[i],"names blocks of new
    database");
00759             for ( i = 0; i < numblocks; i++ ) M_free(d->iblocks[i],"index blocks of new database");
00760             goto getout;
00761 /* INTERNAL_ERROR_EXCL_STOP */
00762         }
00763     }
00764     d->info.firstindexblock = d->iblocks[0]->position;
00765     d->info.lastindexblock = d->iblocks[numblocks-1]->position;
00766     d->info.firstnameblock = d->nblocks[0]->position;
00767     d->info.lastnameblock = d->nblocks[numnameblocks-1]->position;
00768     if ( WriteIniInfo(d) ) {
00769         for ( i = 0; i < numnameblocks; i++ ) M_free(d->nblocks[i],"names blocks of new database");
00770         for ( i = 0; i < numblocks; i++ ) M_free(d->iblocks[i],"index blocks of new database");
00771         goto getout;
00772     }
00773     return(d);
00774 }
00775
00776 /*
00777     #] NewDbase :
00778     #[ FreeTableBase :
00779 */
00780
00781 void FreeTableBase(DBASE *d)
00782 {
```

```
00783     int i, j, *old, *newL;
00784     LIST *L;
00785     for ( i = 0; i < d->info.numberofnamesblocks; i++ ) M_free(d->nblocks[i],"nblocks[i]");
00786     for ( i = 0; i < d->info.numberofindexblocks; i++ ) M_free(d->iblocks[i],"iblocks[i]");
00787     M_free(d->nblocks,"nblocks");
00788     M_free(d->iblocks,"iblocks");
00789     if ( d->tablenames ) M_free(d->tablenames,"d->tablenames");
00790     if ( d->name ) M_free(d->name,"d->name");
00791     if ( d->fullname ) M_free(d->fullname,"d->fullname");
00792     i = d - tablebases;
00793     if ( i < ( NumTableBases - 1 ) ) {
00794         L = &(AC.TableBaseList);
00795         j = ( ( NumTableBases - i - 1 ) * L->size ) / sizeof(int);
00796         old = (int *)d; newL = (int *)(d+1);
00797         while ( --j >= 0 ) *newL++ = *old++;
00798         j = L->size / sizeof(int);
00799         while ( --j >= 0 ) *newL++ = 0;
00800     }
00801     NumTableBases--;
00802     M_free(d,"tb,d");
00803 }
00804
00805 /*
00806     #] FreeTableBase :
00807     #[ ComposeTableNames :
00808
00809         The nameblocks are supposed to be in memory.
00810         Hence we have to go through them
00811 */
00812
00813 int ComposeTableNames(DBASE *d)
00814 {
00815     MLONG nsize = 0;
00816     int i, j, k;
00817     char *s, *t, *ss;
00818     d->topnumber = 0;
00819     i = 0; s = d->nblocks[i]->names; j = NAMETABLESIZE;
00820     while ( *s ) {
00821       if ( *s ) d->topnumber++;
00822       for ( k = 0; k < 2; k++ ) { /* name and argtail */
00823         while ( *s ) {
00824             j--;
00825             if ( j <= 0 ) {
00826                 i++; if ( i >= d->info.numberofnamesblocks ) goto gotall;
00827                 s = d->nblocks[i]->names; j = NAMETABLESIZE;
00828             }
00829             else s++;
00830         }
00831         j--;
00832         if ( j <= 0 ) {
00833             i++; if ( i >= d->info.numberofnamesblocks ) goto gotall;
00834             s = d->nblocks[i]->names; j = NAMETABLESIZE;
00835         }
00836         else s++;
00837       }
00838     }
00839 gotall:;
00840     nsize = (d->info.numberofnamesblocks-1)*NAMETABLESIZE +
00841                 (s-d->nblocks[i]->names)+1;
00842     if ( ( d->tablenames = (char *)Malloc1((2*nsize+30)*sizeof(char),"tablenames") )
00843         == 0 ) { return(-1); }
00844     t = d->tablenames;
00845     d->tablenamessize = 2*nsize+30;
00846     d->tablenamefill = nsize-1;
00847     for ( k = 0; k < i; k++ ) {
00848         ss = d->nblocks[k]->names;
00849         for ( j = 0; j < NAMETABLESIZE; j++ ) *t++ = *ss++;
00850     }
00851     ss = d->nblocks[i]->names;
00852     while ( ss < s ) *t++ = *ss++;
00853     *t = 0;
00854     return(0);
00855 }
00856
00857 /*
00858     #] ComposeTableNames :
00859     #[ OpenDbase :
00860 */
00861
00862 DBASE *OpenDbase(char *filename)
00863 {
00864     FILE *f;
00865     DBASE *d;
00866     char *newname;
00867     if ( ( f = LocateBase(&filename,&newname,"r+b") ) == 0 ) {
00868         MesPrint("Cannot open file %s\n",filename);
00869         return(0);
```

```
00870     }
00871 /*  setbuf(f,0); */
00872     d = (DBASE *)From0List(&(AC.TableBaseList));
00873     d->name = filename; /* For the moment just for the error messages */
00874     d->handle = f;
00875     if ( ReadIniInfo(d) || ReadIndex(d) ) { M_free(d,"OpenDbase"); fclose(f); return(0); }
00876     if ( ComposeTableNames(d) ) {
00877         FreeTableBase(d);
00878         fclose(f);
00879         return(0);
00880     }
00881     d->name = str_dup(filename);
00882     d->fullname = newname;
00883     return(d);
00884 }
00885
00886 /*
00887     #] OpenDbase :
00888     #[ AddTableName :
00889
00890     Adds a name of a table. Writes the namelist to disk.
00891     Returns the number of this tablename in the database.
00892     If the name was already in the table we return its value in negative.
00893     Zero is an error!
00894 */
00895
00896 MLONG AddTableName(DBASE *d,char *name,TABLES T)
00897 {
00898     char *s, *t, *tt;
00899     int namesize, tailsize;
00900     MLONG newsize, i, num;
00901 /*
00902     First search for the name in what we have already
00903 */
00904     if ( d->tablenames ) {
00905         num = 0;
00906         s = d->tablenames;
00907         while ( *s ) {
00908             num++;
00909             t = name;
00910             while ( ( *s == *t ) && *t ) { s++; t++; }
00911             if ( *s == *t ) { return(-num); }
00912             while ( *s ) s++;
00913             s++;
00914             while ( *s ) s++;
00915             s++;
00916         }
00917     }
00918 /*
00919     This name has to be added
00920 */
00921     MesPrint("We add the name %s\n",name);
00922     t = name;
00923     while ( *t ) { t++; }
00924     namesize = t-name;
00925     if ( ( t = (char *)(T->argtail) ) != 0 ) {
00926         while ( *t ) { t++; }
00927         tailsize = t - (char *)(T->argtail);
00928     }
00929     else { tailsize = 0; }
00930     if ( d->tablenames == 0 ) {
00931         if ( ComposeTableNames(d) ) {
00932             FreeTableBase(d);
00933             M_free(d,"AddTableName");
00934             return(0);
00935         }
00936     }
00937     d->info.numberoftables++;
00938     while ( ( d->tablenamefill+namesize+tailsize+3 > d->tablenamessize )
00939                     || ( d->tablenames == 0 ) ) {
00940         newsize = 2*d->tablenamessize + 2*namesize + 2*tailsize + 6;
00941         if ( ( t = (char *)Malloc1(newsize*sizeof(char),"AddTableName") ) == 0 )
00942             return(0);
00943         tt = t;
00944         if ( d->tablenames ) {
00945             s = d->tablenames;
00946             for ( i = 0; i < d->tablenamefill; i++ ) *t++ = *s++;
00947             *t = 0;
00948             M_free(d->tablenames,"d->tablenames");
00949         }
00950         d->tablenames = tt;
00951         d->tablenamessize = newsize;
00952     }
00953     s = d->tablenames + d->tablenamefill;
00954     t = name;
00955     while ( *t ) *s++ = *t++;
00956     *s++ = 0;
```

```
00957     t = (char *)(T->argtail);
00958     while ( *t ) *s++ = *t++;
00959     *s++ = 0;
00960     *s = 0;
00961     d->tablenamefill = s - d->tablenames;
00962     d->topnumber++;
00963 /*
00964     Now we have to synchronize
00965 */
00966     if ( PutTableNames(d) ) return(0);
00967     return(d->topnumber);
00968 }
00969
00970 /*
00971     #] AddTableName :
00972     #[ GetTableName :
00973
00974     Gets a name of a table.
00975     Returns the number of this tablename in the database.
00976     Zero -> error
00977 */
00978
00979 MLONG GetTableName(DBASE *d,char *name)
00980 {
00981     char *s, *t;
00982     MLONG num;
00983 /*
00984     search for the name in what we have
00985 */
00986     if ( d->tablenames ) {
00987         num = 0;
00988         s = d->tablenames;
00989         while ( *s ) {
00990             num++;
00991             t = name;
00992             while ( ( *s == *t ) && *t ) { s++; t++; }
00993             if ( *s == *t ) { return(num); }
00994             while ( *s ) s++;
00995             s++;
00996             while ( *s ) s++;
00997             s++;
00998         }
00999     }
01000     return(0);
01001 }
01002
01003 /*
01004     #] GetTableName :
01005     #[ PutTableNames :
01006
01007     Takes the names string in d->tablenames and puts it in the nblocks
01008     pieces. Writes what has been changed to disk.
01009 */
01010
01011 int PutTableNames(DBASE *d)
01012 {
01013     NAMESBLOCK **nnew;
01014     int i, j, firstdif;
01015     MLONG m;
01016     char *s, *t;
01017 /*
01018     Determine how many blocks are needed.
01019 */
01020     MLONG numblocks = d->tablenamefill/NAMETABLESIZE + 1;
01021     if ( d->info.numberofnamesblocks < numblocks ) {
01022 /*
01023         We need more blocks. First make sure of the space for nblocks.
01024 */
01025         if ( ( nnew = (NAMESBLOCK **)Malloc1(sizeof(NAMESBLOCK *)*numblocks,
01026             "new names block") ) == 0 ) {
01027             return(-1);
01028         }
01029         for ( i = 0; i < d->info.numberofnamesblocks; i++ ) {
01030             nnew[i] = d->nblocks[i];
01031         }
01032         free(d->nblocks);
01033         d->nblocks = nnew;
01034         for ( ; i < numblocks; i++ ) {
01035             if ( ( d->nblocks[i] = (NAMESBLOCK *)Malloc1(sizeof(NAMESBLOCK),
01036             "additional names blocks ") ) == 0 ) {
01037                 FreeTableBase(d);
01038                 return(-1);
01039             }
01040             d->nblocks[i]->previousblock = -1;
01041             d->nblocks[i]->position = -1;
01042             s = d->nblocks[i]->names;
01043             for ( j = 0; j < NAMETABLESIZE; j++ ) *s++ = 0;
```

```
01044         }
01045         d->info.numberofnamesblocks = numblocks;
01046     }
01047 /*
01048     Now look till where the new contents agree with the old.
01049 */
01050     firstdif = 0;
01051     i = 0; t = d->nblocks[i]->names; j = 0; s = d->tablenames;
01052     for ( m = 0; m < d->tablenamefill; m++ ) {
01053         if ( *s == *t ) {
01054             s++; t++; j++;
01055             if ( j >= NAMETABLESIZE ) {
01056                 i++;
01057                 t = d->nblocks[i]->names;
01058                 j = 0;
01059             }
01060         }
01061         else {
01062             firstdif = i;
01063             for ( ; m < d->tablenamefill; m++ ) {
01064                 *t++ = *s++; j++;
01065                 if ( j >= NAMETABLESIZE ) {
01066                     i++;
01067                     t = d->nblocks[i]->names;
01068                     j = 0;
01069                 }
01070             }
01071             *t = 0;
01072             break;
01073         }
01074     }
01075     for ( i = 0; i < d->info.numberofnamesblocks; i++ ) {
01076         if ( i == firstdif ) break;
01077         if ( d->nblocks[i]->position < 0 ) { firstdif = i; break; }
01078     }
01079 /*
01080     Now we have to (re)write the blocks, starting at firstdif.
01081 */
01082     for ( i = firstdif; i < d->info.numberofnamesblocks; i++ ) {
01083         if ( i > 0 ) d->nblocks[i]->previousblock = d->nblocks[i-1]->position;
01084         else d->nblocks[i]->previousblock = -1;
01085         if ( d->nblocks[i]->position < 0 ) {
01086             fseek(d->handle,0,SEEK_END);
01087             d->nblocks[i]->position = ftell(d->handle);
01088         }
01089         else fseek(d->handle,d->nblocks[i]->position,SEEK_SET);
01090         convertnamesblock(d->nblocks[i],&scratchnamesblock,TODISK);
01091         if ( minoswrite(d->handle,(char *)(&scratchnamesblock),sizeof(NAMESBLOCK)) ) {
01092 /* INTERNAL_ERROR_EXCL_START */
01093             MesPrint("!>Error while writing names blocks\n");
01094             FreeTableBase(d);
01095             return(-1);
01096 /* INTERNAL_ERROR_EXCL_STOP */
01097         }
01098     }
01099     d->info.lastnameblock = d->nblocks[d->info.numberofnamesblocks-1]->position;
01100     d->info.firstnameblock = d->nblocks[0]->position;
01101     return(WriteIniInfo(d));
01102 }
01103
01104 /*
01105     #] PutTableNames :
01106     #[ AddToIndex :
01107 */
01108
01109 int AddToIndex(DBASE *d,MLONG number)
01110 {
01111     MLONG i, oldnumofindexblocks = d->info.numberofindexblocks;
01112     MLONG j, newnumofindexblocks, jj;
01113     INDEXBLOCK **ib;
01114     MLONG t = (MLONG)(time(0));
01115     if ( number == 0 ) return(0);
01116     else if ( number < 0 ) {
01117         if ( d->info.entriesinindex < -number ) {
01118 /* INTERNAL_ERROR_EXCL_START */
01119             MesPrint("!>There are only %ld entries in the index of file %s\n",
01120             d->info.entriesinindex,d->name);
01121             return(-1);
01122 /* INTERNAL_ERROR_EXCL_STOP */
01123         }
01124         d->info.entriesinindex += number;
01125 dowrite:
01126         if ( WriteIniInfo(d) ) {
01127 /* INTERNAL_ERROR_EXCL_START */
01128             d->info.entriesinindex -= number;
01129             MesPrint("!>File may be corrupted\n");
01130             return(-1);
```

```
01131 /* INTERNAL_ERROR_EXCL_STOP */
01132         }
01133     }
01134     else if ( d->info.entriesinindex+number <=
01135     NUMOBJECTS*d->info.numberofindexblocks ) {
01136         d->info.entriesinindex += number;
01137         goto dowrite;
01138     }
01139     else {
01140         d->info.entriesinindex += number;
01141         newnumofindexblocks = d->info.numberofindexblocks + ((number -
01142             (NUMOBJECTS*d->info.numberofindexblocks - d->info.entriesinindex))
01143             +NUMOBJECTS-1)/NUMOBJECTS;
01144         if ( ( ib = (INDEXBLOCK **)Malloc1(sizeof(INDEXBLOCK *)*newnumofindexblocks,
01145         "index") ) == 0 ) return(-1);
01146         for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01147             ib[i] = d->iblocks[i];
01148         }
01149         for ( i = d->info.numberofindexblocks; i < newnumofindexblocks; i++ ) {
01150             if ( ( ib[i] = (INDEXBLOCK *)Malloc1(sizeof(INDEXBLOCK),"index block") ) == 0 ) {
01151                 FreeTableBase(d);
01152                 return(-1);
01153             }
01154             if ( i > 0 ) ib[i]->previousblock = ib[i-1]->position;
01155             else ib[i]->previousblock = -1;
01156 /*
01157             Zero things properly. We don't want garbage in the file.
01158 */
01159             for ( j = 0; j < NUMOBJECTS; j++ ) {
01160                 ib[i]->objects[j].date = t;
01161                 ib[i]->objects[j].size = 0;
01162                 ib[i]->objects[j].position = -1;
01163                 ib[i]->objects[j].tablenumber = 0;
01164                 ib[i]->objects[j].uncompressed = 0;
01165                 ib[i]->objects[j].spare1 = 0;
01166                 ib[i]->objects[j].spare2 = 0;
01167                 ib[i]->objects[j].spare3 = 0;
01168                 for ( jj = 0; jj < ELEMENTSIZE; jj++ ) ib[i]->objects[j].element[jj] = 0;
01169             }
01170             fseek(d->handle,0,SEEK_END);
01171             ib[i]->position = ftell(d->handle);
01172             convertblock(ib[i],&scratchblock,TODISK);
01173             if ( minoswrite(d->handle,(char *)(&scratchblock),sizeof(INDEXBLOCK)) ) {
01174 /* INTERNAL_ERROR_EXCL_START */
01175                 MesPrint("!>Error while writing new index of file %s",d->name);
01176                 FreeTableBase(d);
01177                 return(-1);
01178 /* INTERNAL_ERROR_EXCL_STOP */
01179             }
01180         }
01181         d->info.lastindexblock = ib[newnumofindexblocks-1]->position;
01182         d->info.firstindexblock = ib[0]->position;
01183         d->info.numberofindexblocks = newnumofindexblocks;
01184         if ( WriteIniInfo(d) ) {
01185 /* INTERNAL_ERROR_EXCL_START */
01186             d->info.numberofindexblocks = oldnumofindexblocks;
01187             d->info.entriesinindex -= number;
01188             MesPrint("!>File may be corrupted\n");
01189             FreeTableBase(d);
01190             return(-1);
01191 /* INTERNAL_ERROR_EXCL_STOP */
01192         }
01193         M_free(d->iblocks,"AddToIndex");
01194         d->iblocks = ib;
01195     }
01196     return(0);
01197 }
01198
01199 /*
01200     #] AddToIndex :
01201     #[ AddObject :
01202 */
01203
01204 MLONG AddObject(DBASE *d,MLONG tablenumber,char *arguments,char *rhs)
01205 {
01206     MLONG number;
01207     number = d->info.entriesinindex;
01208     if ( AddToIndex(d,1) ) return(-1);
01209     if ( WriteObject(d,tablenumber,arguments,rhs,number) ) return(-1);
01210     return(number);
01211 }
01212
01213 /*
01214     #] AddObject :
01215     #[ FindTableNumber :
01216 */
01217
```

```
01218 MLONG FindTableNumber(DBASE *d,char *name)
01219 {
01220     char *s = d->tablenames, *t, *ss;
01221     MLONG num = 0;
01222     ss = d->tablenames + d->tablenamefill;
01223     while ( s < ss ) {
01224         num++;
01225         t = name;
01226         while ( *s == *t && *t ) {
01227             s++; t++;
01228         }
01229         if ( *s == 0 && *t == 0 ) return(num);
01230         while ( *s ) s++;
01231         s++;
01232 /*
01233         Skip also the argument tail
01234 */
01235         while ( *s ) s++;
01236         s++;
01237     }
01238     return(-1);     /* Name not found */
01239 }
01240
01241 /*
01242     #] FindTableNumber :
01243     #[ WriteObject :
01244 */
01245
01246 int WriteObject(DBASE *d,MLONG tablenumber,char *arguments,char *rhs,MLONG number)
01247 {
01248     char *s, *a;
01249 #ifdef WITHZLIB
01250     char *buffer = 0;
01251     uLongf newsize = 0, oldsize = 0;
01252     uLong ssize;
01253     int error = 0;
01254 #endif
01255     MLONG i, j, position, size, n;
01256     OBJECTS *obj;
01257     if ( ( d->mode & INPUTONLY ) == INPUTONLY ) {
01258         MesPrint("Not allowed to write to input\n");
01259         return(-1);
01260     }
01261     if ( number >= d->info.entriesinindex ) {
01262         MesPrint("Reference to non-existing object number %ld\n",number+1);
01263         return(0);
01264     }
01265     j = number/NUMOBJECTS;
01266     i = number%NUMOBJECTS;
01267     obj = &(d->iblocks[j]->objects[i]);
01268     a = arguments;
01269     while ( *a ) a++;
01270     a++; n = a - arguments;
01271     if ( n > ELEMENTSIZE ) {
01272         MesPrint("Table element %s has more than %ld characters.\n",arguments,
01273             (MLONG)ELEMENTSIZE);
01274         return(-1);
01275     }
01276     s = obj->element;
01277     a = arguments;
01278     while ( *a ) *s++ = *a++;
01279     *s++ = 0;
01280     while ( n < ELEMENTSIZE ) { *s++ = 0; n++; }
01281     obj->spare1 = obj->spare2 = obj->spare3 = 0;
01282
01283     fseek(d->handle,0,SEEK_END);
01284     position = ftell(d->handle);
01285     s = rhs;
01286     while ( *s ) s++;
01287     s++;
01288     size = s - rhs;
01289 #ifdef WITHZLIB
01290     if ( ( d->mode & NOCOMPRESS ) == 0 ) {
01291         newsize = size + size/1000 + 20;
01292         if ( ( buffer = (char *)Malloc1(newsize*sizeof(char),"compress buffer") )
01293             == 0 ) {
01294             MesPrint("No compress used for element %s in file %s\n",arguments,d->name);
01295         }
01296     }
01297     else buffer = 0;
01298     if ( buffer ) {
01299         ssize = size;
01300 #ifdef WITHZSTD
01301         // Force the use of zlib for compressed Tablebase entries, so that tablebases created
01302         // with zstd-supported FORM builds can be used by zstd-unsupported FORM builds.
01303         const int old_isUsingZSTDcompression = ZWRAP_isUsingZSTDcompression();
01304         ZWRAP_useZSTDcompression(0);
```

```
01305 #endif
01306         if ( ( error = compress((Bytef *)buffer,&newsize,(Bytef *)rhs,ssize) ) != Z_OK ) {
01307 /* INTERNAL_ERROR_EXCL_START */
01308             MesPrint("!>Due to error no compress used for element %s in file %s\n",arguments,d->name);
01309             MesPrint("Error = %d\n",error);
01310             M_free(buffer,"tb,WriteObject");
01311             buffer = 0;
01312 /* INTERNAL_ERROR_EXCL_STOP */
01313         }
01314 #ifdef WITHZSTD
01315         ZWRAP_useZSTDcompression(old_isUsingZSTDcompression);
01316 #endif
01317     }
01318     if ( buffer ) {
01319         rhs = buffer;
01320         oldsize = size;
01321         size = newsize;
01322     }
01323 #endif
01324     if ( minoswrite(d->handle,rhs,size) ) {
01325 /* INTERNAL_ERROR_EXCL_START */
01326         MesPrint("!>Error while writing rhs\n");
01327         return(-1);
01328 /* INTERNAL_ERROR_EXCL_STOP */
01329     }
01330     obj->position = position;
01331     obj->size = size;
01332     obj->date = (MLONG)(time(0));
01333     obj->tablenumber = tablenumber;
01334 #ifdef WITHZLIB
01335     obj->uncompressed = oldsize;
01336     if ( buffer ) M_free(buffer,"tb,WriteObject");
01337 #else
01338     obj->uncompressed = 0;
01339 #endif
01340     return(WriteIndexBlock(d,j));
01341 }
01342
01343 /*
01344     #] WriteObject :
01345     #[ ReadObject :
01346
01347     Returns a pointer to the proper rhs
01348 */
01349
01350 char *ReadObject(DBASE *d,MLONG tablenumber,char *arguments)
01351 {
01352     OBJECTS *obj;
01353     MLONG i, j;
01354     char *buffer1, *s, *t;
01355 #ifdef WITHZLIB
01356     char *buffer2 = 0;
01357     uLongf finallength = 0;
01358 #endif
01359     if ( tablenumber > d->topnumber ) {
01360         MesPrint("Reference to non-existing table number in tablebase %s: %ld\n",
01361                 d->name,tablenumber);
01362         return(0);
01363     }
01364 /*
01365     Start looking for the object
01366 */
01367     for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01368         for ( j = 0; j < NUMOBJECTS; j++ ) {
01369             if ( d->iblocks[i]->objects[j].tablenumber != tablenumber ) continue;
01370             s = arguments; t = d->iblocks[i]->objects[j].element;
01371             while ( *s == *t && *s ) { s++; t++; }
01372             if ( *t == 0 && *s == 0 ) goto foundelement;
01373         }
01374     }
01375     s = d->tablenames; i = 1;
01376     while ( *s ) {
01377         if ( i == tablenumber ) break;
01378         while ( *s ) s++;
01379         s++;
01380         while ( *s ) s++;
01381         s++;
01382         i++;
01383     }
01384     MesPrint("%s(%s) not found in tablebase %s\n",s,arguments,d->name);
01385     return(0);
01386
01387 foundelement:;
01388     obj = &(d->iblocks[i]->objects[j]);
01389     fseek(d->handle,obj->position,SEEK_SET);
01390     if ( ( buffer1 = (char *)Malloc1(obj->size,"reading rhs buffer1") ) == 0 ) {
01391         return(0);
```

```
01392     }
01393 #ifdef WITHZLIB
01394     if ( obj->uncompressed > 0 ) {
01395         if ( ( buffer2 = (char *)Malloc1(obj->uncompressed,"reading rhs buffer2") ) == 0 ) {
01396             return(0);
01397         }
01398     }
01399     else buffer2 = 0;
01400 #endif
01401     if ( minosread(d->handle,buffer1,obj->size) ) {
01402 /* INTERNAL_ERROR_EXCL_START */
01403         MesPrint("!>Could not read rhs %s in file %s\n",arguments,d->name);
01404         M_free(buffer1,"tb,ReadObject");
01405 #ifdef WITHZLIB
01406         if ( buffer2 ) M_free(buffer2,"tb,ReadObject");
01407 #endif
01408         return(0);
01409 /* INTERNAL_ERROR_EXCL_STOP */
01410     }
01411 #ifdef WITHZLIB
01412     if ( buffer2 == 0 ) return(buffer1);
01413     finallength = obj->uncompressed;
01414     if ( uncompress((Bytef *)buffer2,&finallength,(Bytef *)buffer1,obj->size) != Z_OK ) {
01415 /* INTERNAL_ERROR_EXCL_START */
01416         MesPrint("!>Cannot uncompress element %s in file %s\n",arguments,d->name);
01417         M_free(buffer1,"tb,ReadObject"); M_free(buffer2,"tb,ReadObject");
01418         return(0);
01419 /* INTERNAL_ERROR_EXCL_STOP */
01420     }
01421     M_free(buffer1,"tb,ReadObject");
01422     return(buffer2);
01423 #else
01424     return(buffer1);
01425 #endif
01426 }
01427
01428 /*
01429     #] ReadObject :
01430     #[ ReadijObject :
01431
01432     Returns a pointer to the proper rhs
01433 */
01434
01435 char *ReadijObject(DBASE *d,MLONG i,MLONG j,char *arguments)
01436 {
01437     OBJECTS *obj;
01438     char *buffer1;
01439 #ifdef WITHZLIB
01440     char *buffer2 = 0;
01441     uLongf finallength = 0;
01442 #endif
01443     obj = &(d->iblocks[i]->objects[j]);
01444     fseek(d->handle,obj->position,SEEK_SET);
01445     if ( ( buffer1 = (char *)Malloc1(obj->size,"reading rhs buffer1") ) == 0 ) {
01446         return(0);
01447     }
01448 #ifdef WITHZLIB
01449     if ( obj->uncompressed > 0 ) {
01450         if ( ( buffer2 = (char *)Malloc1(obj->uncompressed,"reading rhs buffer2") ) == 0 ) {
01451             return(0);
01452         }
01453     }
01454     else buffer2 = 0;
01455 #endif
01456     if ( minosread(d->handle,buffer1,obj->size) ) {
01457 /* INTERNAL_ERROR_EXCL_START */
01458         MesPrint("!>Could not read rhs %s in file %s\n",arguments,d->name);
01459         if ( buffer1 ) M_free(buffer1,"rhs buffer1");
01460 #ifdef WITHZLIB
01461         if ( buffer2 ) M_free(buffer2,"rhs buffer2");
01462 #endif
01463         return(0);
01464 /* INTERNAL_ERROR_EXCL_STOP */
01465     }
01466 #ifdef WITHZLIB
01467     if ( buffer2 == 0 ) return(buffer1);
01468     finallength = obj->uncompressed;
01469     if ( uncompress((Bytef *)buffer2,&finallength,(Bytef *)buffer1,obj->size) != Z_OK ) {
01470 /* INTERNAL_ERROR_EXCL_START */
01471         MesPrint("!>Cannot uncompress element %s in file %s\n",arguments,d->name);
01472         if ( buffer1 ) M_free(buffer1,"rhs buffer1");
01473         if ( buffer2 ) M_free(buffer2,"rhs buffer2");
01474         return(0);
01475 /* INTERNAL_ERROR_EXCL_STOP */
01476     }
01477     M_free(buffer1,"rhs buffer1");
01478     return(buffer2);
```

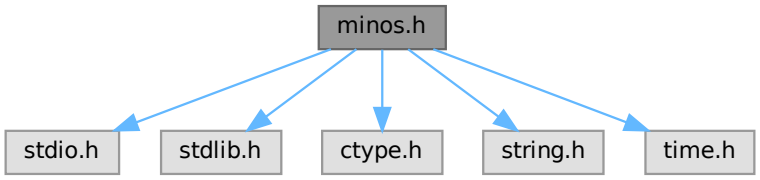
```
01479 #else
01480     return(buffer1);
01481 #endif
01482 }
01483
01484 /*
01485     #] ReadijObject :
01486     #[ ExistsObject :
01487
01488     Returns 1 if Object exists
01489 */
01490
01491 int ExistsObject(DBASE *d,MLONG tablenumber,char *arguments)
01492 {
01493     MLONG i, j;
01494     char *s, *t;
01495     if ( tablenumber > d->topnumber ) {
01496         MesPrint("Reference to non-existing table number in tablebase %s: %ld\n",
01497                 d->name,tablenumber);
01498         return(0);
01499     }
01500 /*
01501     Start looking for the object
01502 */
01503     for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01504         for ( j = 0; j < NUMOBJECTS; j++ ) {
01505             if ( d->iblocks[i]->objects[j].tablenumber != tablenumber ) continue;
01506             s = arguments; t = d->iblocks[i]->objects[j].element;
01507             while ( *s == *t && *s ) { s++; t++; }
01508             if ( *t == 0 && *s == 0 ) return(1);
01509         }
01510     }
01511     return(0);
01512 }
01513
01514 /*
01515     #] ExistsObject :
01516     #[ DeleteObject :
01517
01518     Returns 1 if Object has been deleted.
01519     We leave a hole. Actually the object is still there but has been
01520     inactivated. It can be reactivated by calling this routine again.
01521 */
01522
01523 int DeleteObject(DBASE *d,MLONG tablenumber,char *arguments)
01524 {
01525     MLONG i, j;
01526     char *s, *t;
01527     if ( tablenumber > d->topnumber ) {
01528         MesPrint("Reference to non-existing table number in tablebase %s: %ld\n",
01529                 d->name,tablenumber);
01530         return(0);
01531     }
01532 /*
01533     Start looking for the object
01534 */
01535     for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01536         for ( j = 0; j < NUMOBJECTS; j++ ) {
01537             if ( d->iblocks[i]->objects[j].tablenumber != tablenumber ) continue;
01538             s = arguments; t = d->iblocks[i]->objects[j].element;
01539             while ( *s == *t && *s ) { s++; t++; }
01540             if ( *t == 0 && *s == 0 ) {
01541                 d->iblocks[i]->objects[j].tablenumber =
01542                     -d->iblocks[i]->objects[j].tablenumber - 1;
01543                 return(1);
01544             }
01545         }
01546     }
01547     return(0);
01548 }
01549
01550 /*
01551     #] DeleteObject :
01552 */
```

4.75 minos.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
```

```
#include <string.h>
#include <time.h>
```

Include dependency graph for minos.h:



This graph shows which files directly or indirectly include this file:

Data Structures

- struct iniinfo
- struct objects
- struct indexblock
- struct nameblock
- struct dbase

Macros

- #define TODISK 0
- #define FROMDISK 1
- #define MDIRTYFLAG 1
- #define MCLEANFLAG (~MDIRTYFLAG)
- #define INANDOUT 0
- #define INPUTONLY 1
- #define OUTPUTONLY 2
- #define NOCOMPRESS 4

Typedefs

- typedef struct iniinfo **INIIINFO**
- typedef struct objects **OBJECTS**
- typedef struct indexblock **INDEXBLOCK**
- typedef struct nameblock **NAMESBLOCK**
- typedef struct dbase **DBASE**

Functions

- int minosread (FILE *f, char *buffer, MLONG size)
- int minoswrite (FILE *f, char *buffer, MLONG size)

Variables

- int withoutflush

4.75.1 Detailed Description

Contains all needed declarations and definitions for the tablebase low level file routines. These have been taken from the minos database system and modified somewhat.

!!!CAUTION!!! Changes in this file will most likely have consequences for the recovery mechanism (see checkpoint.c). You need to care for the code in checkpoint.c as well and modify the code there accordingly!

Definition in file minos.h.

4.75.2 Macro Definition Documentation

4.75.2.1 TODISK

```
#define TODISK 0
```

Definition at line 145 of file minos.h.

4.75.2.2 FROMDISK

```
#define FROMDISK 1
```

Definition at line 146 of file minos.h.

4.75.2.3 MDIRTYFLAG

```
#define MDIRTYFLAG 1
```

Definition at line 147 of file minos.h.

4.75.2.4 MCLEANFLAG

```
#define MCLEANFLAG (~MDIRTYFLAG)
```

Definition at line 148 of file minos.h.

4.75.2.5 INANDOUT

```
#define INANDOUT 0
```

Definition at line 149 of file minos.h.

4.75.2.6 INPUTONLY

```
#define INPUTONLY 1
```

Definition at line 150 of file minos.h.

4.75.2.7 OUTPUTONLY

```
#define OUTPUTONLY 2
```

Definition at line 151 of file minos.h.

4.75.2.8 NOCOMPRESS

```
#define NOCOMPRESS 4
```

Definition at line 152 of file minos.h.

4.75.3 Function Documentation

4.75.3.1 minosread()

```
int minosread (
            FILE * f,
            char * buffer,
            MLONG size )
```

Definition at line 66 of file minos.c.

4.75.3.2 minoswrite()

```
int minoswrite (
            FILE * f,
            char * buffer,
            MLONG size )
```

Definition at line 83 of file minos.c.

4.75.4 Variable Documentation

4.75.4.1 withoutflush

```
int withoutflush  [extern]
```

Definition at line 45 of file minos.c.

4.76 minos.h

Go to the documentation of this file.

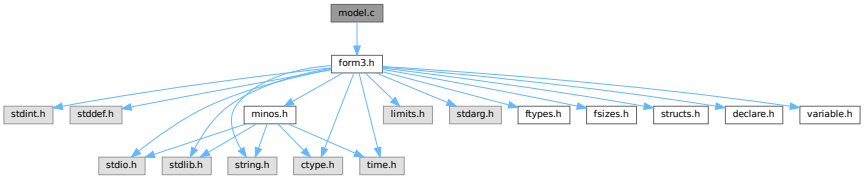
```
00001 #ifndef __MANAGE_H__
00002
00003 #define __MANAGE_H__
00004
00017 /* #[ License : */
00018 /*
00019  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00020  *   When using this file you are requested to refer to the publication
00021  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00022  *   This is considered a matter of courtesy as the development was paid
00023  *   for by FOM the Dutch physics granting agency and we would like to
00024  *   be able to track its scientific use to convince FOM of its value
00025  *   for the community.
00026  *
00027  *   This file is part of FORM.
00028  *
00029  *   FORM is free software: you can redistribute it and/or modify it under the
00030  *   terms of the GNU General Public License as published by the Free Software
00031  *   Foundation, either version 3 of the License, or (at your option) any later
00032  *   version.
00033  *
00034  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00035  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00036  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00037  *   details.
00038  *
00039  *   You should have received a copy of the GNU General Public License along
00040  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00041  */
00042 /* #] License : */
00043
00044 #include <stdio.h>
00045 #include <stdlib.h>
00046 #include <ctype.h>
00047 #include <string.h>
00048 #include <time.h>
00049
00050 /*
00051     The following typedef has been moved to form3.h where all the sizes
00052     are defined for the various memory models.
00053     We want MLONG to have a more or less fixed size.
00054     In form3.h we try to fix it at 8 bytes.
00055     This should make files exchangeable
00056     between various 32-bits and 64-bits systems. At 4 bytes it might have
00057     problems with files of more than 2 Gbytes.
00058
00059 typedef long MLONG;
00060 */
00061
00062 #if BITSINWORD == 32
00063     #define NUMOBJECTS 1024
00064     #define MAXINDEXSIZE 1000000000L
00065     #define NAMETABLESIZE 1008
00066     #define ELEMENTSIZE 256
00067 #elif BITSINWORD == 16
00068     #define NUMOBJECTS 100
00069     #define MAXINDEXSIZE 33000000L
00070     #define NAMETABLESIZE 1008
00071     #define ELEMENTSIZE 128
00072 #else
00073     #error Only 64-bit and 32-bit platforms are supported.
00074 #endif
00075
00076
00077 int minosread(FILE *f,char *buffer,MLONG size);
00078 int minoswrite(FILE *f,char *buffer,MLONG size);
```

```
00079
00080 /*
00081     ELEMENTSIZE should make a nice number of sizeof(OBJECTS)
00082     Usually this will be much too much, but there are cases.....
00083 */
00084
00085 typedef struct iniinfo {
00086 /* should contains only MLONG variables or convertiniinfo should be modified */
00087     MLONG   entriesinindex;
00088     MLONG   numberofindexblocks;
00089     MLONG   firstindexblock;
00090     MLONG   lastindexblock;
00091     MLONG   numberoftables;
00092     MLONG   numberofnamesblocks;
00093     MLONG   firstnameblock;
00094     MLONG   lastnameblock;
00095 } INIINFO;
00096
00097 typedef struct objects {
00098 /* if any changes, convertblock should be adapted too!!!! */
00099     MLONG position;             /* position of RHS= */
00100     MLONG size;                 /* size on disk (could be compressed) */
00101     MLONG date;                 /* Time stamp */
00102     MLONG tablenumber;          /* Number of table. Refers to name in special index */
00103     MLONG uncompressed;         /* uncompressed size if compressed. If not: 0 */
00104     MLONG spare1;
00105     MLONG spare2;
00106     MLONG spare3;
00107     char element[ELEMENTSIZE];  /* table element in character form */
00108 } OBJECTS;
00109
00110 typedef struct indexblock {
00111     MLONG   flags;
00112     MLONG   previousblock;
00113     MLONG   position;
00114     OBJECTS  objects[NUMOBJECTS];
00115 } INDEXBLOCK;
00116
00117 typedef struct nameblock {
00118     MLONG   previousblock;
00119     MLONG   position;
00120     char    names[NAMETABLESIZE];
00121 } NAMESBLOCK;
00122
00123 typedef struct dbase {
00124     INIINFO      info;
00125     MLONG       mode;
00126     MLONG       tablenamessize;
00127     MLONG       topnumber;
00128     MLONG       tablenamefill;
00129     MLONG       rwmode;
00130     INDEXBLOCK    **iblocks;
00131     NAMESBLOCK  **nblocks;
00132     FILE        *handle;
00133     char        *name;
00134     char        *fullname;
00135     char        *tablenames;
00136 } DBASE;
00137
00138 /*
00139 typedef int (*SFUN)(char *);
00140 typedef struct compile {
00141     char *keyword;
00142     SFUN func;
00143 } MCFUNCTION;
00144  */
00145 #define TODISK 0
00146 #define FROMDISK 1
00147 #define MDIRTYFLAG 1
00148 #define MCLEANFLAG (~MDIRTYFLAG)
00149 #define INANDOUT 0
00150 #define INPUTONLY 1
00151 #define OUTPUTONLY 2
00152 #define NOCOMPRESS 4
00153
00154 extern int withoutflush;
00155
00156 #endif
```

4.77 model.c File Reference

```
#include "form3.h"
```

Include dependency graph for model.c:



Functions

- VERTEX * CreateVertex (MODEL *m)
- UBYTE * ReadParticle (UBYTE *s, VERTEX *v, MODEL *m, int par)
- int CoModel (UBYTE *s)
- int CoParticle (UBYTE *s)
- int CoVertex (UBYTE *s)
- int CoEndModel (UBYTE *s)

4.77.1 Detailed Description

Implementation of "Model", required by the diagrams_ function.

Definition in file model.c.

4.77.2 Function Documentation

4.77.2.1 CreateVertex()

```
VERTEX * CreateVertex (
            MODEL * m )
```

Definition at line 76 of file model.c.

4.77.2.2 ReadParticle()

```
UBYTE * ReadParticle (
            UBYTE * s,
            VERTEX * v,
            MODEL * m,
            int par )
```

Definition at line 106 of file model.c.

4.77.2.3 CoModel()

```
int CoModel (
            UBYTE * s )
```

Definition at line 149 of file model.c.

4.77.2.4 CoParticle()

```
int CoParticle (
            UBYTE * s )
```

Definition at line 224 of file model.c.

4.77.2.5 CoVertex()

```
int CoVertex (
            UBYTE * s )
```

Definition at line 306 of file model.c.

4.77.2.6 CoEndModel()

```
int CoEndModel (
            UBYTE * s )
```

Definition at line 418 of file model.c.

4.78 model.c

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
00026  *
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
00031 /*
00032     #[ Explanations :
```

```
00033
00034     This is a second generation much simplified version of model.c
00035     Syntax:
00036         Model QED;
00037             Particle electron,positron,-2;
00038             Particle photon,+3;
00039             Vertex electron,positron,photon: g;
00040         EndModel;
00041     or:
00042         Model QCD;
00043             Particle quark,antiquark,-2;
00044             Particle ghost,antighost,-1;
00045             Particle gluon,+3;
00046             Vertex quark,antiquark,gluon: g;
00047             Vertex ghost,antighost,gluon: g;
00048             Vertex gluon,gluon,gluon: g;
00049             Vertex gluon,gluon,gluon,gluon: g^2;
00050         EndModel;
00051     Remarks:
00052     1: if no second particle is given, a particle and its antiparticle are the same
00053     2: Spin statistics is by dimension of SU(2) representation with a sign.
00054     3: In a vertex we need to mention the coupling constants.
00055     4: Internally a particle gives a vertex with only two lines.
00056     5: All particles must have been declared before any vertex.
00057
00058     The diagrams should be provided as a collection of nodes and edges
00059     with momenta with a direction substituted. The other parameters (like
00060     Lorentz indices, color indices etc.) can then be provided in a procedure
00061     that should go with this mode definition.
00062     In principle the edges are not needed if there are no 'insertions', but
00063     because we would like to have the insertions as well we need the edges.
00064
00065     #] Explanations :
00066     #[ Includes : model.c
00067 */
00068
00069 #include "form3.h"
00070
00071 /*
00072     #] Includes :
00073     #[ CreateVertex :
00074 */
00075
00076 VERTEX *CreateVertex(MODEL *m)
00077 {
00078     VERTEX *v, **new;
00079     WORD newsize;
00080     int i;
00081     if ( m->invertices >= m->sizevertices ) {
00082         if ( m->sizevertices == 0 ) newsize = 20;
00083         else newsize = 2*m->sizevertices;
00084         new = (VERTEX **)Malloc1(newsize*sizeof(VERTEX *),"m->vertices");
00085         for ( i = 0; i < m->sizevertices; i++ ) new[i] = m->vertices[i];
00086         if ( m->sizevertices > 0 ) M_free(m->vertices,"m->vertices");
00087         m->vertices = new;
00088         m->sizevertices = newsize;
00089     }
00090     v = (VERTEX *)Malloc1(sizeof(VERTEX),"VERTEX");
00091     m->vertices[m->invertices++] = v;
00092     v->nparticles = 0;
00093     v->ncouplings = 0;
00094     v->type = 0;
00095     v->error = 0;
00096     v->externonly = 0;
00097     v->spare = 0;
00098     return(v);
00099 }
00100
00101 /*
00102     #] CreateVertex :
00103     #[ ReadParticle :
00104 */
00105
00106 UBYTE *ReadParticle(UBYTE *s, VERTEX *v, MODEL *m, int par)
00107 {
00108     UBYTE *name, c;
00109     PARTICLE *p = v->particles + v->nparticles++;
00110     WORD type, funnum;
00111     int i, j;
00112     SKIPBLANKS(s)
00113     name = s; s = SkipName(s); c = *s; *s = 0;
00114     if ( GetVar(name,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND ) {
00115         p->number = AddFunction(name,0,VERTEXFUNCTION,0,0,0,-1,-1) + FUNCTION;
00116     }
00117     else if ( par == 1 && type == CFUNCTION && functions[funnum].spec == VERTEXFUNCTION ) {
00118         p->number = funnum+FUNCTION;
00119     }
```

```
00120     else if ( par == 0 && type == CFUNCTION && functions[funnum].spec == VERTEXFUNCTION ) {
00121 /*
00122         We should check whether this particle exists already in this model.
00123         If so, this will be an error.
00124 */
00125         for ( i = 0; i < m->nparticles-1; i++ ) {
00126           for ( j = 0; j < m->vertices[i]->nparticles; j++ ) {
00127             if ( m->vertices[i]->particles[j].number == funnum ) {
00128                 MesPrint("&Illegal attempt to redefine a particle in the same model: %s",name);
00129                 v->error = 1;
00130             }
00131           }
00132         }
00133         p->number = funnum+FUNCTION;
00134     }
00135     else {
00136         MesPrint("&Name of particle previously declared as another variable: %s",name);
00137         v->error = 1;
00138     }
00139     *s = c;
00140     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
00141     return(s);
00142 }
00143
00144 /*
00145     #] ReadParticle :
00146     #[ CoModel :
00147 */
00148
00149 int CoModel(UBYTE *s)
00150 {
00151     UBYTE *name, c;
00152     MODEL *m = (MODEL *)Malloc1(sizeof(MODEL),"Model");
00153     WORD numberofset, *e;
00154     SETS set;
00155     SKIPBLANKS(s)
00156     AC.ModelLevel++;
00157     int i;
00158 /*
00159     We now have to add the model to the list of models.
00160 */
00161     if ( AC.modelspace == 0 || AC.nummodels >= AC.modelspace ) {
00162         int newspace = 2*AC.modelspace+2, i;
00163         MODEL **models = (MODEL **)Malloc1((sizeof(MODEL *))*newspace,"AC.models");
00164         for ( i = 0; i < AC.modelspace; i++ ) models[i] = AC.models[i];
00165         if ( AC.models ) M_free(AC.models,"AC.models");
00166         AC.modelspace = newspace;
00167         AC.models = models;
00168     }
00169
00170     AC.models[AC.nummodels++] = m;
00171
00172     m->vertices = 0;
00173     m->couplings = 0;
00174     m->nparticles = m->nvertices = m->sizevertices = m->invertices = 0;
00175     m->sizecouplings = 0;
00176     m->error = 0;
00177     m->grccmodel = NULL;
00178
00179     name = s;
00180     if ( FG.cTable[*s] == 0 ) {
00181         while ( FG.cTable[*s] <= 1 ) s++;
00182         c = *s; *s = 0;
00183         m->name = strDup1(name,"Model name");
00184         *s = c;
00185         SKIPBLANKS(s)
00186         if ( *s != 0 ) {
00187             MesPrint("&Illegal option in model statement: %s",s);
00188             return(1);
00189         }
00190     }
00191     else {
00192         m->name = strDup1((UBYTE *)"---","Model name");
00193         m->error = 1;
00194         MesPrint("&Illegal name for model: %s",name);
00195         return(1);
00196     }
00197 /*
00198     Now make it a set with one element. The type of the set is rather special
00199 */
00200     if ( GetName(AC.varnames,m->name,&numberofset,NOAUTO) != NAMENOTFOUND ) {
00201         MesPrint("&Name conflict with name %s of model",m->name);
00202         return(1);
00203     }
00204     numberofset = AddSet(name,0);
00205     set = Sets + numberofset;
00206     set->type = CMODEL;
```

```
00207     e = (WORD *)FromVarList(&AC.SetElementList);
00208     *e = AC.nummodels-1;
00209     set->last++;
00210     AC.SetList.numtemp = AC.SetList.num;
00211     AC.SetElementList.numtemp = AC.SetElementList.num;
00212
00213     for ( i = 0; i <= MAXLEGS; i++ ) m->legcouple[i] = 0;
00214     m->legcouple[2] = 1;
00215
00216     return(0);
00217 }
00218
00219 /*
00220     #] CoModel :
00221     #[ CoParticle :
00222 */
00223
00224 int CoParticle(UBYTE *s)
00225 {
00226     MODEL *m;
00227     VERTEX *v;
00228     int i;
00229     if ( AC.nummodels <= 0 ) {
00230         MesPrint("&No open model for particle statement");
00231         return(1);
00232     }
00233     m = AC.models[AC.nummodels-1];
00234     if ( m->nvertices > 0 ) {
00235         MesPrint("&In a model description Particle statements should be before Vertex statements.");
00236         m->error = 1;
00237         return(1);
00238     }
00239     v = CreateVertex(m);
00240     m->nparticles++;
00241     s = ReadParticle(s,v,m,0);
00242     v->particles[0].type = 1;
00243     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
00244     if ( v->error == 0 ) {
00245         if ( FG.cTable[*s] == 0 ) {
00246             s = ReadParticle(s,v,m,0);
00247             v->particles[1].type = -1;
00248             if ( v->particles[0].number == v->particles[1].number ) {
00249                 v->particles[0].type = 0;
00250                 v->particles[1].type = 0;
00251             }
00252         }
00253         else {  /* Particle = antiparticle */
00254             v->particles[1] = v->particles[0];
00255             v->nparticles++;
00256             v->particles[0].type = 0;
00257             v->particles[1].type = 0;
00258         }
00259     }
00260     if ( v->error == 0 && ( *s == '+' || *s == '-' ) ) {
00261         int x = 0, sign = ( *s ==  '-' ) ? -1: 1;
00262         s++;
00263         while ( FG.cTable[*s] == 1 ) { x = 10*x + *s++ - '0'; }
00264         if ( x == 0 ) {
00265             v->error = 1;
00266             MesPrint("&Spin goes by dimension of SU(2) representation. Zero is not allowed.");
00267         }
00268         else { v->particles[0].spin = v->particles[1].spin = sign*x; }
00269         SKIPBLANKS(s)
00270     }
00271     else if ( v->error == 0 ) {
00272         v->particles[0].spin = v->particles[1].spin = 1;
00273         v->particles[0].type = 0;
00274         v->particles[1].type = 0;
00275     }
00276 /*
00277     Here will come future options
00278 */
00279     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
00280     while ( v->error == 0 && *s != 0 && FG.cTable[*s] == 0 ) {
00281         UBYTE *opt = s, c;
00282         while ( FG.cTable[*s] == 0 ) s++;
00283         c = *s; *s = 0;
00284         if ( StrICont(opt,(UBYTE *)"external") == 0 ) { v->externonly = 1; }
00285         else {
00286             MesPrint("&Unrecognized option %s in particle statement.",opt);
00287             v->error = 1;
00288         }
00289         *s = c;
00290         while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
00291     }
00292 /*
00293     The couplings array will be used for registering in what kind of vertices the
```

```
00294     particle is involved. (example: in QCD the quarks only in 2 and 3-point vertices)
00295 */
00296     for ( i = 0; i < MAXPARTICLES; i++ ) v->couplings[i] = 0;
00297     v->couplings[2] = 1;
00298     return(v->error);
00299 }
00300
00301 /*
00302     #] CoParticle :
00303     #[ CoVertex :
00304 */
00305
00306 int CoVertex(UBYTE *s)
00307 {
00308     UBYTE *ss, *name, c;
00309     WORD type;
00310     MODEL *m;
00311     VERTEX *v;
00312     if ( AC.ModelLevel <= 0 ) {
00313         MesPrint("&No open model for vertex statement");
00314         return(1);
00315     }
00316     m = AC.models[AC.nummodels-1];
00317 /*
00318     Get an object of type VERTEX
00319 */
00320     v = CreateVertex(m);
00321     m->nvertices++;
00322     SKIPBLANKS(s);
00323     while ( *s && *s != ':' ) {
00324         if ( v->error == 0 ) {
00325             s = ReadParticle(s,v,m,1);
00326             if ( v->error ) m->error = 1;
00327         }
00328         else {
00329             while ( *s && *s != ':' ) {
00330                 if ( *s == '[' ) { SKIPBRA1(s) }
00331                 else if ( *s == '(' ) { SKIPBRA3(s) }
00332                 else if ( *s == 0 ) {
00333                     v->error = 1;
00334                     MesPrint("&No coupling constant in vertex statement.");
00335                     break;
00336                 }
00337             }
00338             break;
00339         }
00340     }
00341     if ( v->error == 0 ) {
00342       if ( *s == ':' ) { /* read symbols and powers */
00343         s++; SKIPBLANKS(s);
00344         while ( *s ) {
00345             if ( v->ncouplings >= 2*MAXCOUPLINGS ) {
00346                 MesPrint("&More than %d coupling constants in vertex.",(WORD)MAXCOUPLINGS);
00347                 MesPrint("    Recompile with a larger value for MAXCOUPLINGS.");
00348                 return(1);
00349             }
00350             ss = s; s = SkipName(s); c = *s; *s = 0; name = ConstructName(ss,0);
00351             // Forbid couplings which have no symbol or an overal numerical factor.
00352             if ( *name == 0 ) {
00353                 v->error = 1;
00354                 MesPrint("&Invalid coupling constant in vertex statement.");
00355             }
00356             if ( GetVar(name,&type,&v->couplings[v->ncouplings],CSYMBOL,WITHAUTO)
00357                  == NAMENOTFOUND ) {
00358                 WORD minpow = -MAXPOWER;
00359                 WORD maxpow =  MAXPOWER;
00360                 WORD cplx = 0, dim = 0;
00361                 v->couplings[v->ncouplings] = AddSymbol(name,minpow,maxpow,cplx,dim);
00362             }
00363             *s = c;
00364             v->ncouplings++;
00365 /*
00366             Now possibly a power
00367 */
00368             if ( *s == '^' ) { /* we need an integer */
00369                 WORD x = 0; WORD sign = 1;
00370                 s++;
00371                 while ( *s == '-' || *s == '+' ) {
00372                     if ( *s == '-' ) sign = -sign;
00373                     s++;
00374                 }
00375                 if ( sign == -1 ) {
00376                     v->error = 1;
00377                     MesPrint("&Invalid negative power of coupling constant.");
00378                 }
00379                 while ( FG.cTable[*s] == 1 ) x = 10*x + *s++ - '0';
00380                 v->couplings[v->ncouplings++] = x;
```

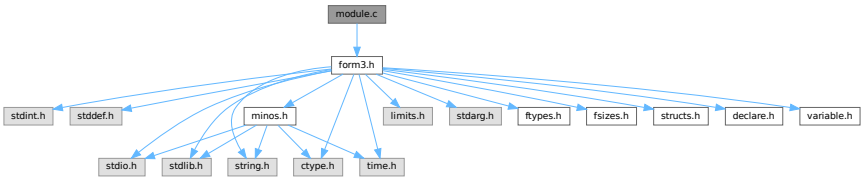
```
00381             }
00382             else {
00383                 v->couplings[v->ncouplings++] = 1;
00384             }
00385             SKIPBLANKS(s);
00386             if ( *s == '*' ) { s++; continue; }
00387             if ( *s != ',' ) break;
00388             s++;
00389             SKIPBLANKS(s);
00390           }
00391         }
00392         else {
00393           v->error = 1;
00394           MesPrint("&A vertex statement needs at least one coupling constant.");
00395         }
00396       }
00397       if ( v->error == 0 ) {
00398 /*
00399         Register which types of vertices are possible for each particle.
00400 */
00401         int i, j;
00402         for ( i = 0; i < v->nparticles; i++ ) {
00403             for ( j = 0; j < m->nparticles; j++ ) {
00404                 if ( m->vertices[j]->particles[0].number == v->particles[i].number ) break;
00405                 if ( m->vertices[j]->particles[1].number == v->particles[i].number ) break;
00406             }
00407             m->vertices[j]->couplings[v->nparticles] = 1;
00408         }
00409       }
00410       return(v->error);
00411 }
00412
00413 /*
00414       #] CoVertex :
00415       #[ CoEndModel :
00416 */
00417
00418 int CoEndModel(UBYTE *s)
00419 {
00420       int i, j, k, kk;
00421       WORD csize = 0, *newcouplings, newsize;
00422       MODEL *m;
00423       VERTEX *v;
00424       if ( AC.ModelLevel <= 0 ) {
00425           MesPrint("&EndModel statement without matching Model statement");
00426           return(1);
00427       }
00428       m = AC.models[AC.nummodels-1];
00429 /*
00430       Now create a list of all coupling constants.
00431       Note that we do not expect an astronomical number of them and hence
00432       we use a simple insertion algorithm.
00433 */
00434       m->ncouplings = 0;
00435       for ( i = 0; i < m->nvertices; i++ ) {
00436           v = m->vertices[i+m->nparticles];
00437           m->legcouple[v->nparticles] = 1;
00438           if ( m->ncouplings + v->ncouplings > csize ) {
00439               if ( csize == 0 ) newsize = v->ncouplings + 10;
00440               else newsize = 2*csize;
00441               newcouplings = (WORD *)Malloc1(newsize*sizeof(WORD),"m->couplings");
00442               for ( k = 0; k < m->ncouplings; k++ ) newcouplings[k] = m->couplings[k];
00443               if ( csize > 0 ) M_free(m->couplings,"m->couplings");
00444               m->couplings = newcouplings;
00445               csize = newsize;
00446           }
00447           for ( j = 0; j < v->ncouplings; j += 2 ) {
00448               WORD sym = v->couplings[j];
00449               for ( k = 0; k < m->ncouplings; k++ ) {
00450                   if ( sym == m->couplings[k] ) break;
00451                   if ( sym < m->couplings[k] ) {
00452                       for ( kk = m->ncouplings; kk > k; k-- )
00453                           m->couplings[kk] = m->couplings[kk-1];
00454                       m->couplings[k] = sym;
00455                       m->ncouplings++;
00456                       break;
00457                   }
00458               }
00459               if ( k >= m->ncouplings ) m->couplings[m->ncouplings++] = sym;
00460           }
00461       }
00462       AC.ModelLevel--;
00463       DUMMYUSE(s)
00464       return(LoadModel(m));
00465 }
00466
00467 /*
```

```
00468      #] CoEndModel :
00469 */
```

4.79 module.c File Reference

```
#include "form3.h"
```

Include dependency graph for module.c:



Functions

- int ModuleInstruction (int *moduletype, int *specialtype)
- int CoModuleOption (UBYTE *s)
- int CoModOption (UBYTE *s)
- void SetSpecialMode (int moduletype, int specialtype)
- int ExecModule (int moduletype)
- int ExecStore (void)
- void FullCleanUp (void)
- int DoPolyfun (UBYTE *s)
- int DoPolyratfun (UBYTE *s)
- int DoNoParallel (UBYTE *s)
- int DoParallel (UBYTE *s)
- int DoModSum (UBYTE *s)
- int DoModMax (UBYTE *s)
- int DoModMin (UBYTE *s)
- int DoModLocal (UBYTE *s)
- int DoProcessBucket (UBYTE *s)
- UBYTE * DoModDollar (UBYTE *s, int type)
- int DoinParallel (UBYTE *s)
- int DonotinParallel (UBYTE *s)
- int DoExecStatement (void)
- int DoPipeStatement (void)

4.79.1 Detailed Description

A number of routines that deal with the moduleoption statement and the execution of modules. Additionally there are the execution of the exec and pipe instructions.

Definition in file module.c.

4.79.2 Function Documentation

4.79.2.1 ModuleInstruction()

```
int ModuleInstruction (
            int * moduletype,
            int * specialtype )
```

Definition at line 76 of file module.c.

4.79.2.2 CoModuleOption()

```
int CoModuleOption (
            UBYTE * s )
```

Definition at line 143 of file module.c.

4.79.2.3 CoModOption()

```
int CoModOption (
            UBYTE * s )
```

Definition at line 213 of file module.c.

4.79.2.4 SetSpecialMode()

```
void SetSpecialMode (
            int moduletype,
            int specialtype )
```

Definition at line 258 of file module.c.

4.79.2.5 ExecModule()

```
int ExecModule (
            int moduletype )
```

Definition at line 275 of file module.c.

4.79.2.6 ExecStore()

```
int ExecStore (
            void  )
```

Definition at line 300 of file module.c.

4.79.2.7 FullCleanUp()

```
void FullCleanUp (
            void  )
```

Definition at line 315 of file module.c.

4.79.2.8 DoPolyfun()

```
int DoPolyfun (
            UBYTE * s )
```

Definition at line 398 of file module.c.

4.79.2.9 DoPolyratfun()

```
int DoPolyratfun (
            UBYTE * s )
```

Definition at line 461 of file module.c.

4.79.2.10 DoNoParallel()

```
int DoNoParallel (
            UBYTE * s )
```

Definition at line 532 of file module.c.

4.79.2.11 DoParallel()

```
int DoParallel (
            UBYTE * s )
```

Definition at line 547 of file module.c.

4.79.2.12 DoModSum()

```
int DoModSum (
            UBYTE * s )
```

Definition at line 562 of file module.c.

4.79.2.13 DoModMax()

```
int DoModMax (
            UBYTE * s )
```

Definition at line 582 of file module.c.

4.79.2.14 DoModMin()

```
int DoModMin (
            UBYTE * s )
```

Definition at line 602 of file module.c.

4.79.2.15 DoModLocal()

```
int DoModLocal (
            UBYTE * s )
```

Definition at line 622 of file module.c.

4.79.2.16 DoProcessBucket()

```
int DoProcessBucket (
            UBYTE * s )
```

Definition at line 642 of file module.c.

4.79.2.17 DoModDollar()

```
UBYTE * DoModDollar (
            UBYTE * s,
            int type )
```

Definition at line 660 of file module.c.

4.79.2.18 DoinParallel()

```
int DoinParallel (
            UBYTE * s )
```

Definition at line 775 of file module.c.

4.79.2.19 DonotinParallel()

```
int DonotinParallel (
            UBYTE * s )
```

Definition at line 785 of file module.c.

4.79.2.20 DoExecStatement()

```
int DoExecStatement (
            void  )
```

Definition at line 797 of file module.c.

4.79.2.21 DoPipeStatement()

```
int DoPipeStatement (
            void  )
```

Definition at line 814 of file module.c.

4.80 module.c

Go to the documentation of this file.

```
00001
00007 /* #[ License : */
00008 /*
00009  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00010  *   When using this file you are requested to refer to the publication
00011  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00012  *   This is considered a matter of courtesy as the development was paid
00013  *   for by FOM the Dutch physics granting agency and we would like to
00014  *   be able to track its scientific use to convince FOM of its value
00015  *   for the community.
00016  *
00017  *   This file is part of FORM.
00018  *
00019  *   FORM is free software: you can redistribute it and/or modify it under the
00020  *   terms of the GNU General Public License as published by the Free Software
00021  *   Foundation, either version 3 of the License, or (at your option) any later
00022  *   version.
00023  *
00024  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00025  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00026  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00027  *   details.
00028  *
00029  *   You should have received a copy of the GNU General Public License along
00030  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00031  */
00032 /* #] License : */
00033 /*
00034     #[ Includes :
00035 */
00036
00037 #include "form3.h"
00038
00039 /*
00040     #] Includes :
00041     #[ Modules :
00042         #[ ModuleInstruction :
00043
00044         Enters after the . of .sort etc
00045         We have word[[(options)][:commentary];]
00046         Word is one of 'clear end global sort store'
00047         Options are 'polyfun, endloopifunchanged, endloopifzero'
00048         Additions for solving equations can be added to 'word'
00049
00050         The flag in the moduleoptions is for telling whether the current
00051         option can be followed by others. If it is > 0 it cannot.
00052 */
00053
00054 static KEYWORD ModuleWords[] = {
00055       {"clear",   (TFUN)0,    CLEARMODULE,    0}
00056     ,{"end",     (TFUN)0,    ENDMODULE,      0}
00057     ,{"global",  (TFUN)0,    GLOBALMODULE,   0}
00058     ,{"sort",    (TFUN)0,    SORTMODULE,     0}
00059     ,{"store",   (TFUN)0,    STOREMODULE,    0}
00060 };
00061
00062 static KEYWORD ModuleOptions[] = {
00063       {"inparallel",         DoinParallel,   1,              1}
00064     ,{"local",               DoModLocal,     MODLOCAL,       0}
00065     ,{"maximum",             DoModMax,       MODMAX,         0}
00066     ,{"minimum",             DoModMin,       MODMIN,         0}
00067     ,{"noparallel",          DoNoParallel,   NOPARALLEL_USER,0}
00068     ,{"notinparallel",       DonotinParallel,0,              1}
00069     ,{"parallel",            DoParallel,     PARALLELFLAG,   0}
00070     ,{"polyfun",             DoPolyfun,      POLYFUN,        0}
00071     ,{"polyratfun",          DoPolyratfun,   POLYFUN,        0}
00072     ,{"processbucketsize",   DoProcessBucket,0,              0}
00073     ,{"sum",                 DoModSum,       MODSUM,         0}
```

```
00074 };
00075
00076 int ModuleInstruction(int *moduletype, int *specialtype)
00077 {
00078     UBYTE *t, *s, *u, c;
00079     KEYWORD *key;
00080     int addit = 0, error = 0, i, j;
00081     DUMMYUSE(specialtype);
00082     LoadInstruction(0);
00083     AC.firstctypemessage = 0;
00084     s = AP.preStart; SKIPBLANKS(s)
00085     t = EndOfToken(s); c = *t; *t = 0;
00086     AC.origin = FROMPOINTINSTRUCTION;
00087     key = FindKeyWord(AP.preStart,ModuleWords,sizeof(ModuleWords)/sizeof(KEYWORD));
00088     if ( key == 0 ) {
00089         MesPrint("@Unrecognized module terminator: %s",s);
00090         error = 1;
00091         key = ModuleWords;
00092         while ( StrCmp((UBYTE *)key->name,(UBYTE *)"end") ) key++;
00093     }
00094     *t = c;
00095     *moduletype = key->type;
00096     SKIPBLANKS(t);
00097     while ( *t == '(' ) {   /* There are options */
00098         s = t+1; SKIPBRA3(t)
00099         if ( *t == 0 ) {
00100             MesPrint("@Improper options field in . instruction");
00101             error = 1;
00102         }
00103         else {
00104             *t = 0;
00105             if ( CoModOption(s) ) error = 1;
00106             *t++ = ')';
00107         }
00108     }
00109     if ( *t == ':' ) {  /* There is an 'advertisement' */
00110         t++;
00111         SKIPBLANKS(t)
00112         s = t; i = 0;
00113         while ( *t && *t != ';' ) {
00114             if ( *t == '\\' ) t++;
00115             t++; i++;
00116         }
00117         u = t;
00118         while ( u > s && u[-1] == ' ' ) { u--; i--; }
00119         if ( *u == '\\' ) { u++; i++; }
00120         for ( j = COMMERCIALSIZE-1; j >= 0; j-- ) {
00121             if ( i <= 0 ) break;
00122             AC.Commercial[j] = *--u; i--;
00123             if ( u > s && u[-1] == '\\' ) u--;
00124         }
00125         for ( ; j >= 0; j-- ) AC.Commercial[j] = ' ';
00126         AC.Commercial[COMMERCIALSIZE] = 0;
00127         addit += 2;
00128     }
00129     if ( addit && *t != ';' ) {
00130         MesPrint("@Improper ending of . instruction");
00131         error = -1;
00132     }
00133     return(error);
00134 }
00135
00136 /*
00137         #] ModuleInstruction :
00138         #[ CoModuleOption :
00139
00140     ModuleOption, options;
00141 */
00142
00143 int CoModuleOption(UBYTE *s)
00144 {
00145     UBYTE *t,*tt,c;
00146     KEYWORD *option;
00147     int error = 0, polyflag = 0;
00148     AC.origin = FROMMODULEOPTION;
00149     if ( *s ) do {
00150         s = ToToken(s);
00151         t = EndOfToken(s);
00152         c = *t; *t = 0;
00153         option = FindKeyWord(s,ModuleOptions,
00154             sizeof(ModuleOptions)/sizeof(KEYWORD));
00155         if ( option == 0 ) {
00156             if ( polyflag ) {
00157                 *t = c; t++; s = SkipAName(t);
00158                 polyflag = 0;
00159                 continue;
00160             }
```

```
00161             else {
00162                 MesPrint("@Unrecognized module option: %s",s);
00163                 error = 1;
00164                 polyflag = 0;
00165                 *t = c;
00166             }
00167         }
00168         else {
00169             *t = c;
00170             SKIPBLANKS(t)
00171             if ( (option->func)(t) ) error = 1;
00172         }
00173         if ( error ) return(error);
00174         if ( StrCmp((UBYTE *)(option->name),(UBYTE *)("polyfun")) == 0
00175          || StrCmp((UBYTE *)(option->name),(UBYTE *)("polyratfun")) == 0 ) {
00176             polyflag = 1;
00177             t++;
00178             t = SkipAName(t);
00179         }
00180         else polyflag = 0;
00181         if ( option->flags > 0 ) return(error);
00182         while ( *t ) {
00183             if ( *t == ',' ) {
00184                 tt = t+1;
00185                 while ( *tt == ',' ) tt++;
00186                 if ( *tt != '$' ) break;
00187                 t = tt+1;
00188             }
00189             if ( *t == ')' ) break;
00190             if ( *t == '(' ) SKIPBRA3(t)
00191             else if ( *t == '{' ) SKIPBRA2(t)
00192             else if ( *t == '[' ) SKIPBRA1(t)
00193             t++;
00194         }
00195         s = t;
00196     } while ( *s == ',' );
00197     if ( *s ) {
00198         MesPrint("@Unrecognized module option: %s",s);
00199         error = 1;
00200     }
00201     return(error);
00202 }
00203
00204 /*
00205         #] CoModuleOption :
00206         #[ CoModOption :
00207
00208     To be called from a .instruction.
00209     Only recognizes polyfun. The newer ones should be via the
00210     ModuleOption statement.
00211 */
00212
00213 int CoModOption(UBYTE *s)
00214 {
00215     UBYTE *t,c;
00216     int error = 0;
00217     AC.origin = FROMPOINTINSTRUCTION;
00218     if ( *s ) do {
00219         s = ToToken(s);
00220         t = EndOfToken(s);
00221         c = *t; *t = 0;
00222         if ( StrICmp(s,(UBYTE *)"polyfun") == 0 ) {
00223             *t = c;
00224             SKIPBLANKS(t)
00225             if ( DoPolyfun(t) ) error = 1;
00226         }
00227         else if ( StrICmp(s,(UBYTE *)"polyratfun") == 0 ) {
00228             *t = c;
00229             SKIPBLANKS(t)
00230             if ( DoPolyratfun(t) ) error = 1;
00231         }
00232         else {
00233             MesPrint("@Unrecognized module option in .instruction: %s",s);
00234             error = 1;
00235             *t = c;
00236         }
00237         while ( *t ) {
00238             if ( *t == ',' || *t == ')' ) break;
00239             if ( *t == '(' ) SKIPBRA3(t)
00240             else if ( *t == '{' ) SKIPBRA2(t)
00241             else if ( *t == '[' ) SKIPBRA1(t)
00242             t++;
00243         }
00244         s = t;
00245     } while ( *s == ',' );
00246     if ( *s ) {
00247         MesPrint("@Unrecognized module option in .instruction: %s",s);
```

```
00248         error = 1;
00249     }
00250     return(error);
00251 }
00252 
00253 /*
00254         #] CoModOption :
00255         #[ SetSpecialMode :
00256 */
00257 
00258 void SetSpecialMode(int moduletype, int specialtype)
00259 {
00260     DUMMYUSE(moduletype); DUMMYUSE(specialtype);
00261 }
00262 
00263 /*
00264         #] SetSpecialMode :
00265         #[ MakeGlobal :
00266 
00267 void MakeGlobal()
00268 {
00269 }
00270 
00271         #] MakeGlobal :
00272         #[ ExecModule :
00273 */
00274 
00275 int ExecModule(int moduletype)
00276 {
00277 /*
00278     Here we check module options and other things that should
00279     be done at the start of a module.
00280 */
00281 #ifdef WITHFLOAT
00282     if ( ( AC.tMaxWeight != 0 && AC.tMaxWeight != AC.MaxWeight )
00283       || ( AC.tDefaultPrecision != 0 && AC.tDefaultPrecision != AC.DefaultPrecision ) ) {
00284         AC.DefaultPrecision = AC.tDefaultPrecision;
00285         AC.tDefaultPrecision = 0;
00286         AC.MaxWeight = AC.tMaxWeight;
00287         AC.tMaxWeight = 0;
00288         ClearMZVTables();
00289         SetupMZVTables();
00290     }
00291 #endif
00292     return(DoExecute(moduletype,0));
00293 }
00294 
00295 /*
00296         #] ExecModule :
00297         #[ ExecStore :
00298 */
00299 
00300 int ExecStore(void)
00301 {
00302     return(0);
00303 }
00304 
00305 /*
00306         #] ExecStore :
00307         #[ FullCleanUp :
00308 
00309         Remark 27-oct-2005 by JV
00310         This routine (and CleanUp in startup.c) may still need some work:
00311             What to do with preprocessor variables
00312             What to do with files we write to
00313 */
00314 
00315 void FullCleanUp(void)
00316 {
00317     int j;
00318 
00319     while ( AC.CurrentStream->previous >= 0 )
00320         AC.CurrentStream = CloseStream(AC.CurrentStream);
00321     AP.PreSwitchLevel = AP.PreIfLevel = 0;
00322 
00323     for ( j = NumProcedures-1; j >= 0; j-- ) {
00324         if ( Procedures[j].name ) M_free(Procedures[j].name,"name of procedure");
00325         if ( Procedures[j].p.buffer ) M_free(Procedures[j].p.buffer,"buffer of procedure");
00326     }
00327     NumProcedures = 0;
00328 
00329     while ( NumPre > AP.gNumPre ) {
00330         NumPre--;
00331         M_free(PreVar[NumPre].name,"PreVar[NumPre].name");
00332         PreVar[NumPre].name = PreVar[NumPre].value = 0;
00333     }
00334 
```

```
00335     AC.DidClean = 0;
00336     for ( j = 0; j < NumExpressions; j++ ) {
00337         AC.exprnames->namenode[Expressions[j].node].type = CDELETE;
00338         AC.DidClean = 1;
00339     }
00340
00341     CompactifyTree(AC.exprnames,EXPRNAMES);
00342
00343     for ( j = AO.NumDictionaries-1; j >= 0; j-- ) {
00344         RemoveDictionary(AO.Dictionaries[j]);
00345         M_free(AO.Dictionaries[j],"Dictionary");
00346     }
00347     AO.NumDictionaries = AO.gNumDictionaries = 0;
00348     M_free(AO.Dictionaries,"Dictionaries");
00349     AO.Dictionaries = 0;
00350     AO.SizeDictionaries = 0;
00351     AP.OpenDictionary = 0;
00352     AO.CurrentDictionary = 0;
00353
00354     AP.ComChar = AP.cComChar;
00355     if ( AP.procedureExtension ) M_free(AP.procedureExtension,"procedureextension");
00356     AP.procedureExtension = strDup1(AP.cprocedureExtension,"procedureextension");
00357
00358     AC.StatsFlag = AM.gStatsFlag = AM.ggStatsFlag;
00359     AC.extrasymbols = AM.gextrasymbols = AM.ggextrasymbols;
00360     AC.extrasym[0] = AM.gextrasym[0] = AM.ggextrasym[0] = 'Z';
00361     AC.extrasym[1] = AM.gextrasym[1] = AM.ggextrasym[1] = 0;
00362     AO.NoSpacesInNumbers = AM.gNoSpacesInNumbers = AM.ggNoSpacesInNumbers;
00363     AO.IndentSpace = AM.gIndentSpace = AM.ggIndentSpace;
00364     AC.ThreadStats = AM.gThreadStats = AM.ggThreadStats;
00365     AC.OldFactArgFlag = AM.gOldFactArgFlag = AM.ggOldFactArgFlag;
00366     AC.FinalStats = AM.gFinalStats = AM.ggFinalStats;
00367     AC.OldGCDflag = AM.gOldGCDflag = AM.ggOldGCDflag;
00368     AC.ThreadsFlag = AM.gThreadsFlag = AM.ggThreadsFlag;
00369     if ( AC.ThreadsFlag && AM.totalnumberofthreads > 1 ) AS.MultiThreaded = 1;
00370     AC.ThreadBucketSize = AM.gThreadBucketSize = AM.ggThreadBucketSize;
00371     AC.ThreadBalancing = AM.gThreadBalancing = AM.ggThreadBalancing;
00372     AC.ThreadSortFileSynch = AM.gThreadSortFileSynch = AM.ggThreadSortFileSynch;
00373     AC.ShortStatsMax = AM.gShortStatsMax = AM.ggShortStatsMax;
00374     AC.SizeCommuteInSet = AM.gSizeCommuteInSet = 0;
00375 #ifdef WITHFLOAT
00376     AC.MaxWeight = AM.gMaxWeight = AM.ggMaxWeight;
00377     AC.DefaultPrecision = AM.gDefaultPrecision = AM.ggDefaultPrecision;
00378 #endif
00379
00380     NumExpressions = 0;
00381     if ( DeleteStore(0) < 0 ) {
00382 /* INTERNAL_ERROR_EXCL_START */
00383         MesPrint("!>Cannot restart the storage file");
00384         Terminate(-1);
00385 /* INTERNAL_ERROR_EXCL_STOP */
00386     }
00387     RemoveDollars();
00388     CleanUp(1);
00389     ResetVariables(2);
00390     IniVars();
00391 }
00392
00393 /*
00394         #] FullCleanUp :
00395         #[ DoPolyfun :
00396 */
00397
00398 int DoPolyfun(UBYTE *s)
00399 {
00400     GETIDENTITY
00401     UBYTE *t, c;
00402     WORD funnum, eqsign = 0;
00403     if ( AC.origin == FROMPOINTINSTRUCTION ) {
00404         if ( *s == 0 || *s == ',' || *s == ')' ) {
00405             AR.PolyFun = 0; AR.PolyFunType = 0;
00406             return(0);
00407         }
00408         if ( *s != '=' ) {
00409             MesPrint("@Proper use in point instructions is: PolyFun[=functionname]");
00410             return(-1);
00411         }
00412         eqsign = 1;
00413     }
00414     else {
00415         if ( *s == 0 ) {
00416             AR.PolyFun = 0; AR.PolyFunType = 0;
00417             return(0);
00418         }
00419         if ( *s != '=' && *s != ',' ) {
00420             MesPrint("@Proper use is: PolyFun[{ ,=}functionname]");
00421             return(-1);
```

```
00422         }
00423         if ( *s == '=' ) eqsign = 1;
00424     }
00425     s++;
00426     SKIPBLANKS(s)
00427     t = EndOfToken(s);
00428     c = *t; *t = 0;
00429
00430     if ( GetName(AC.varnames,s,&funnum,WITHAUTO) != CFUNCTION ) {
00431         if ( AC.origin != FROMPOINTINSTRUCTION && eqsign == 0 ) {
00432             AR.PolyFun = 0; AR.PolyFunType = 0;
00433             return(0);
00434         }
00435         MesPrint("@ %s is not a properly declared function",s);
00436         *t = c;
00437         return(-1);
00438     }
00439     if ( functions[funnum].spec > 0 || functions[funnum].commute != 0 ) {
00440         MesPrint("@The PolyFun must be a regular commuting function!");
00441         *t = c;
00442         return(-1);
00443     }
00444     AR.PolyFun = funnum+FUNCTION; AR.PolyFunType = 1;
00445     *t = c;
00446     SKIPBLANKS(t)
00447     if ( *t && *t != ',' && *t != ')' ) {
00448         t++; c = *t; *t = 0;
00449         MesPrint("@Improper ending of end-of-module instruction: %s",s);
00450         *t = c;
00451         return(-1);
00452     }
00453     return(0);
00454 }
00455
00456 /*
00457         #] DoPolyfun :
00458         #[ DoPolyratfun :
00459 */
00460
00461 int DoPolyratfun(UBYTE *s)
00462 {
00463     GETIDENTITY
00464     UBYTE *t, c;
00465     WORD funnum;
00466     if ( AC.origin == FROMPOINTINSTRUCTION ) {
00467         if ( *s == 0 || *s == ',' || *s == ')' ) {
00468             AR.PolyFun = 0; AR.PolyFunType = 0; AR.PolyFunInv = 0; AR.PolyFunExp = 0;
00469             return(0);
00470         }
00471         if ( *s != '=' ) {
00472             MesPrint("@Proper use in point instructions is:
      PolyRatFun[=functionname[+functionname]]");
00473             return(-1);
00474         }
00475     }
00476     else {
00477         if ( *s == 0 ) {
00478             AR.PolyFun = 0; AR.PolyFunType = 0; AR.PolyFunInv = 0; AR.PolyFunExp = 0;
00479             return(0);
00480         }
00481         if ( *s != '=' && *s != ',' ) {
00482             MesPrint("@Proper use is: PolyRatFun[{ ,=}functionname[+functionname]]");
00483             return(-1);
00484         }
00485     }
00486     s++;
00487     SKIPBLANKS(s)
00488     t = EndOfToken(s);
00489     c = *t; *t = 0;
00490
00491     if ( GetName(AC.varnames,s,&funnum,WITHAUTO) != CFUNCTION ) {
00492 Error1:;
00493         MesPrint("@ %s is not a properly declared function",s);
00494         *t = c;
00495         return(-1);
00496     }
00497     if ( functions[funnum].spec > 0 || functions[funnum].commute != 0 ) {
00498 Error2:;
00499         MesPrint("@The PolyRatFun must be a regular commuting function!");
00500         *t = c;
00501         return(-1);
00502     }
00503     AR.PolyFun = funnum+FUNCTION; AR.PolyFunType = 2;
00504     AR.PolyFunInv = 0;
00505     AR.PolyFunExp = 0;
00506     AC.PolyRatFunChanged = 1;
00507     *t = c;
```

```
00508     if ( *t == '+' ) {
00509         t++; s = t;
00510         t = EndOfToken(s);
00511         c = *t; *t = 0;
00512         if ( GetName(AC.varnames,s,&funnum,WITHAUTO) != CFUNCTION ) goto Error1;
00513         if ( functions[funnum].spec > 0 || functions[funnum].commute != 0 ) goto Error2;
00514         AR.PolyFunInv = funnum+FUNCTION;
00515         *t = c;
00516     }
00517     SKIPBLANKS(t)
00518     if ( *t && *t != ',' && *t != ')' ) {
00519         t++; c = *t; *t = 0;
00520         MesPrint("@Improper ending of end-of-module instruction: %s",s);
00521         *t = c;
00522         return(-1);
00523     }
00524     return(0);
00525 }
00526
00527 /*
00528         #] DoPolyratfun :
00529         #[ DoNoParallel :
00530 */
00531
00532 int DoNoParallel(UBYTE *s)
00533 {
00534     if ( *s == 0 || *s == ',' || *s == ')' ) {
00535         AC.mparallelflag |= NOPARALLEL_USER;
00536         return(0);
00537     }
00538     MesPrint("@NoParallel should not have extra parameters");
00539     return(-1);
00540 }
00541
00542 /*
00543         #] DoNoParallel :
00544         #[ DoParallel :
00545 */
00546
00547 int DoParallel(UBYTE *s)
00548 {
00549     if ( *s == 0 || *s == ',' || *s == ')' ) {
00550         AC.mparallelflag &= ~NOPARALLEL_USER;
00551         return(0);
00552     }
00553     MesPrint("@Parallel should not have extra parameters");
00554     return(-1);
00555 }
00556
00557 /*
00558         #] DoParallel :
00559         #[ DoModSum :
00560 */
00561
00562 int DoModSum(UBYTE *s)
00563 {
00564     while ( *s == ',' ) s++;
00565     if ( *s != '$' ) {
00566         MesPrint("@Module Sum should mention which $-variables");
00567         return(-1);
00568     }
00569     s = DoModDollar(s,MODSUM);
00570     if ( s && *s != 0 && *s != ')' ) {
00571         MesPrint("@Irregular end of Sum option of Module statement");
00572         return(-1);
00573     }
00574     return(0);
00575 }
00576
00577 /*
00578         #] DoModSum :
00579         #[ DoModMax :
00580 */
00581
00582 int DoModMax(UBYTE *s)
00583 {
00584     while ( *s == ',' ) s++;
00585     if ( *s != '$' ) {
00586         MesPrint("@Module Maximum should mention which $-variables");
00587         return(-1);
00588     }
00589     s = DoModDollar(s,MODMAX);
00590     if ( s && *s != 0 ) {
00591         MesPrint("@Irregular end of Maximum option of Module statement");
00592         return(-1);
00593     }
00594     return(0);
```

```
00595 }
00596
00597 /*
00598         #] DoModMax :
00599         #[ DoModMin :
00600 */
00601
00602 int DoModMin(UBYTE *s)
00603 {
00604     while ( *s == ',' ) s++;
00605     if ( *s != '$' ) {
00606         MesPrint("@Module Minimum should mention which $-variables");
00607         return(-1);
00608     }
00609     s = DoModDollar(s,MODMIN);
00610     if ( s && *s != 0 ) {
00611         MesPrint("@Irregular end of Minimum option of Module statement");
00612         return(-1);
00613     }
00614     return(0);
00615 }
00616
00617 /*
00618         #] DoModMin :
00619         #[ DoModLocal :
00620 */
00621
00622 int DoModLocal(UBYTE *s)
00623 {
00624     while ( *s == ',' ) s++;
00625     if ( *s != '$' ) {
00626         MesPrint("@ModuleOption Local should mention which $-variables");
00627         return(-1);
00628     }
00629     s = DoModDollar(s,MODLOCAL);
00630     if ( s && *s != 0 ) {
00631         MesPrint("@Irregular end of Local option of ModuleOption statement");
00632         return(-1);
00633     }
00634     return(0);
00635 }
00636
00637 /*
00638         #] DoModLocal :
00639         #[ DoProcessBucket :
00640 */
00641
00642 int DoProcessBucket(UBYTE *s)
00643 {
00644     LONG x;
00645     while ( *s == ',' || *s == '=' ) s++;
00646     ParseNumber(x,s)
00647     if ( *s && *s != ' ' && *s != '\t' ) {
00648         MesPrint("&Numerical value expected for ProcessBucketSize");
00649         return(1);
00650     }
00651     AC.mProcessBucketSize = x;
00652     return(0);
00653 }
00654
00655 /*
00656         #] DoProcessBucket :
00657         #[ DoModDollar :
00658 */
00659
00660 UBYTE * DoModDollar(UBYTE *s, int type)
00661 {
00662     UBYTE *name, c;
00663     WORD number;
00664     MODOPTDOLLAR *md;
00665     int nummodopt;
00666     while ( *s == '$' ) {
00667 /*
00668         Read the name of the dollar
00669         Mark the type
00670 */
00671         s++;
00672         name = s;
00673         if ( FG.cTable[*s] == 0 ) {
00674             while ( FG.cTable[*s] == 0 || FG.cTable[*s] == 1 ) s++;
00675             c = *s; *s = 0;
00676             number = GetDollar(name);
00677             if ( number < 0 ) {
00678                 UBYTE *s1;
00679                 number = AddDollar(s,0,0,0);
00680                 s1 = strDup1((UBYTE *)"Undefined $-variable in module option; option ignored:
      $","Undefined $-variable in ModuleOption");
```

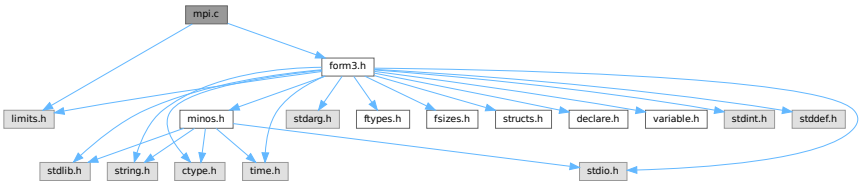
```
00681                 s1 =  AddToString(s1,name,0);
00682                 Warning((char *)s1);
00683                 M_free(s1,"Undefined $-variable in ModuleOption");
00684             }
00685             for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
00686                 if ( number == ModOptdollars[nummodopt].number && type !=
    ModOptdollars[nummodopt].type ) {
00687                     UBYTE *s1;
00688                     s1 = strDup1((UBYTE *)"Conflicting module options for $-variable; later option
    ignored: $","Conflicting $-variable in ModuleOption");
00689                     s1 = AddToString(s1,name,0);
00690                     Warning((char *)s1);
00691                     M_free(s1,"Conflicting $-variable in ModuleOption");
00692                     break;
00693                 }
00694             }
00695             md = (MODOPTDOLLAR *)FromList(&AC.ModOptDolList);
00696             md->number = number;
00697             md->type = type;
00698 #ifdef WITHPTHREADS
00699             if ( DollarLocalCopy(type) ) {
00700                 int j, i;
00701                 DOLLARS dglobal, dlocal;
00702                 md->dstruct = (DOLLARS)Malloc1(
00703                     sizeof(struct DoLlArS)*AM.totalnumberofthreads,"Local DOLLARS");
00704 /*
00705                 Now copy the global dollar into the local copies.
00706                 This can be nontrivial if the value needs an allocation.
00707                 We don't really need the locks.
00708 */
00709                 dglobal = Dollars + number;
00710                 for ( j = 0; j < AM.totalnumberofthreads; j++ ) {
00711                     dlocal = md->dstruct + j;
00712                     dlocal->index = dglobal->index;
00713                     dlocal->node = dglobal->node;
00714                     dlocal->type = dglobal->type;
00715                     dlocal->name = dglobal->name;
00716                     dlocal->size = dglobal->size;
00717                     dlocal->where = dglobal->where;
00718                     if ( dlocal->size > 0 ) {
00719                         dlocal->where = (WORD *)Malloc1((dlocal->size+1)*sizeof(WORD),"Local dollar
    value");
00720                         for ( i = 0; i < dlocal->size; i++ )
00721                             dlocal->where[i] = dglobal->where[i];
00722                         dlocal->where[dlocal->size] = 0;
00723                     }
00724                     INIRECLOCK(dlocal->pthreadslock);
00725                     dlocal->nfactors = dglobal->nfactors;
00726                     if ( dglobal->nfactors > 1 ) {
00727                         int nsize;
00728                         WORD *t, *m;
00729                         dlocal->factors = (FACDOLLAR
    *)Malloc1(dglobal->nfactors*sizeof(FACDOLLAR),"Dollar factors");
00730                         for ( i = 0; i < dglobal->nfactors; i++ ) {
00731                             nsize = dglobal->factors[i].size;
00732                             dlocal->factors[i].type = DOLUNDEFINED;
00733                             dlocal->factors[i].value = dglobal->factors[i].value;
00734                             if ( ( dlocal->factors[i].size = nsize ) > 0 ) {
00735                                 dlocal->factors[i].where = t = (WORD
    *)Malloc1(sizeof(WORD)*(nsize+1),"DollarCopyFactor");
00736                                 m = dglobal->factors[i].where;
00737                                 NCOPY(t,m,nsize);
00738                                 *t = 0;
00739                             }
00740                             else {
00741                                 dlocal->factors[i].where = 0;
00742                             }
00743                         }
00744                     }
00745                     else { dlocal->factors = 0; }
00746                 }
00747             }
00748             else {
00749                 md->dstruct = 0;
00750             }
00751 #endif
00752             *s = c;
00753         }
00754         else {
00755             MesPrint("&Illegal name for $-variable in module option");
00756             while ( *s != ',' && *s != 0 && *s != ')' ) s++;
00757         }
00758         while ( *s == ',' ) s++;
00759     }
00760     return(s);
00761 }
00762
```

```
00763 /*
00764         #] DoModDollar :
00765         #[ DoinParallel :
00766
00767         The idea is that we should have the commands
00768         ModuleOption,InParallel;
00769         ModuleOption,InParallel,name1,name2,...,namen;
00770         ModuleOption,NotInParallel,name1,name2,...,namen;
00771         The advantage over the InParallel statement is that this statement
00772         comes after the definition of the expressions.
00773 */
00774
00775 int DoinParallel(UBYTE *s)
00776 {
00777     return(DoInParallel(s,1));
00778 }
00779
00780 /*
00781         #] DoinParallel :
00782         #[ DonotinParallel :
00783 */
00784
00785 int DonotinParallel(UBYTE *s)
00786 {
00787     return(DoInParallel(s,0));
00788 }
00789
00790 /*
00791         #] DonotinParallel :
00792     #] Modules :
00793     #[ External :
00794         #[ DoExecStatement :
00795 */
00796
00797 int DoExecStatement(void)
00798 {
00799 #ifdef WITHSYSTEM
00800     FLUSHCONSOLE;
00801     if ( system((char *)(AP.preStart)) ) return(-1);
00802     return(0);
00803 #else
00804     Error0("External programs not implemented on this computer/system");
00805     return(-1);
00806 #endif
00807 }
00808
00809 /*
00810         #] DoExecStatement :
00811         #[ DoPipeStatement :
00812 */
00813
00814 int DoPipeStatement(void)
00815 {
00816 #ifdef WITHPIPE
00817     FLUSHCONSOLE;
00818     if ( OpenStream(AP.preStart,PIPESTREAM,0,PRENOACTION) == 0 ) return(-1);
00819     return(0);
00820 #else
00821     Error0("Pipes not implemented on this computer/system");
00822     return(-1);
00823 #endif
00824 }
00825
00826 /*
00827         #] DoPipeStatement :
00828     #] External :
00829 */
```

4.81 mpi.c File Reference

```
#include <limits.h>
#include "form3.h"
```

Include dependency graph for mpi.c:



Data Structures

- struct longMultiStruct

Macros

- #define PF_PACKSIZE 1600
- #define MPI_ERRCODE_CHECK(err)

Typedefs

- typedef struct longMultiStruct **PF_LONGMULTI**

Functions

- LONG PF_RealTime (int i)
- int PF_LibInit (int *argcp, char ***argvp)
- int PF_LibTerminate (int error)
- int PF_Probe (int *src)
- int PF_ISendSbuf (int to, int tag)
- int PF_RecvWbuf (WORD *b, LONG *s, int *src)
- int PF_IRecvRbuf (PF_BUFFER *r, int bn, int from)
- int PF_WaitRbuf (PF_BUFFER *r, int bn, LONG *size)
- int PF_Bcast (void *buffer, int count)
- int PF_Reduce (const void *sendbuf, void *recvbuf, int count, MPI_Datatype type, MPI_Op op, int root)
- int PF_RawSend (int dest, void *buf, LONG l, int tag)
- LONG PF_RawRecv (int *src, void *buf, LONG thesize, int *tag)
- int PF_RawProbe (int *src, int *tag, int *bytesize)
- int PF_RawIsend (int dest, const void *buf, int count, MPI_Datatype type, int tag, MPI_Request *request)
- int PF_RawWaitAll (int count, MPI_Request *request, MPI_Status *status)
- int PF_Discard (int *src, int *tag)
- int PF_PrintPackBuf (char *s, int size)
- int PF_PreparePack (void)
- int PF_Pack (const void *buffer, size_t count, MPI_Datatype type)
- int PF_Unpack (void *buffer, size_t count, MPI_Datatype type)
- int PF_PackString (const UBYTE *str)
- int PF_UnpackString (UBYTE *str)
- int PF_Send (int to, int tag)
- int PF_Receive (int src, int tag, int *psrc, int *ptag)
- int PF_Broadcast (void)

- int PF_PrepareLongSinglePack (void)
- int PF_LongSinglePack (const void *buffer, size_t count, MPI_Datatype type)
- int PF_LongSingleUnpack (void *buffer, size_t count, MPI_Datatype type)
- int PF_LongSingleSend (int to, int tag)
- int PF_LongSingleReceive (int src, int tag, int *psrc, int *ptag)
- int PF_PrepareLongMultiPack (void)
- int PF_LongMultiPackImpl (const void *buffer, size_t count, size_t eSize, MPI_Datatype type)
- int PF_LongMultiUnpackImpl (void *buffer, size_t count, size_t eSize, MPI_Datatype type)
- int PF_LongMultiBroadcast (void)

Variables

- LONG PF_maxDollarChunkSize = 0

4.81.1 Detailed Description

MPI dependent functions of parform

This file contains all the functions for the parallel version of form3 that explicitly need to call mpi routines. This is the only file that really needs to be linked to the mpi-library!

Definition in file mpi.c.

4.81.2 Macro Definition Documentation

4.81.2.1 PF_PACKSIZE

```
#define PF_PACKSIZE 1600
```

Definition at line 63 of file mpi.c.

4.81.2.2 MPI_ERRCODE_CHECK

```
#define MPI_ERRCODE_CHECK(
              err )
```

Value:

```
do { \
    int _tmp_err = (err); \
    if ( _tmp_err != MPI_SUCCESS ) return _tmp_err != 0 ? _tmp_err : -1; \
} while (0)
```

A macro which exits the caller with a non-zero return value if *err* is not MPI_SUCCESS.

Parameters

err	The return value of a MPI function to be checked.

Remarks

The MPI standard defines MPI_SUCCESS == 0. Then (_tmp_err == 0) appears twice and we can expect the second evaluation will be eliminated by the compiler optimization.

Definition at line 89 of file mpi.c.

4.81.3 Function Documentation

4.81.3.1 PF_RealTime()

```
LONG PF_RealTime (
            int i )
```

Returns the realtime in 1/100 sec. as a LONG.

Parameters

i	the timer will be reset if i == 0.

Returns

the real elapsed time in 1/100 second.

Definition at line 106 of file mpi.c.

Referenced by PF_Init().

4.81.3.2 PF_LibInit()

```
int PF_LibInit (
            int * argcp,
            char *** argvp )
```

Performs all library dependent initializations.

Parameters

argcp	pointer to the number of arguments.
argvp	pointer to the arguments.

Returns

0 if OK, nonzero on error.

Definition at line 128 of file mpi.c.

Referenced by PF_Init().

4.81.3.3 PF_LibTerminate()

```
int PF_LibTerminate (
            int error )
```

Exits mpi, when there is an error either indicated or happening, returnvalue is 1, else returnvalue is 0.

Parameters

error	an error code.

Returns

0 if OK, nonzero on error.

Definition at line 214 of file mpi.c.

Referenced by PF_Terminate().

4.81.3.4 PF_Probe()

```
int PF_Probe (
            int * src )
```

Probes the next incoming message. If src == PF_ANY_SOURCE this operation is blocking, otherwise nonblocking.

Parameters

in,out	*src*	the source process number. In output, the process number of actual found source.

Returns

the tag value of the next incoming message if found, 0 if a nonblocking probe (input src != PF_ANY_SOURCE) did not find any messages. A negative returned value indicates an error.

Definition at line 235 of file mpi.c.

4.81.3.5 PF_ISendSbuf()

```
int PF_ISendSbuf (
            int to,
            int tag )
```

Nonblocking send operation. It sends everything from `buff` to `fill` of the active buffer. Depending on *tag* it also can do waiting for other sends to finish or set the active buffer to the next one.

Parameters

to	the destination process number.
tag	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 266 of file mpi.c.

Referenced by FlushOut(), PF_Processor(), and PutOut().

4.81.3.6 PF_RecvWbuf()

```
int PF_RecvWbuf (
            WORD * b,
            LONG * s,
            int * src )
```

Blocking receive of a `WORD` buffer.

Parameters

`out`	*b*	the buffer to store the received data.
`in,out`	*s*	the size of the buffer. The output value is the actual size of the received data.
`in,out`	*src*	the source process number. The output value is the process number of actual source.

Returns

the received message tag. A negative value indicates an error.

Definition at line 342 of file mpi.c.

References PF_ReceiveRuntimeError().

Here is the call graph for this function:

4.81.3.7 PF_IRecvRbuf()

```
int PF_IRecvRbuf (
            PF_BUFFER * r,
            int bn,
            int from )
```

Posts nonblocking receive for the active receive buffer. The buffer is filled from `full` to `stop`.

Parameters

r	the PF_BUFFER struct for the nonblocking receive.
bn	the index of the cyclic buffer.
from	the source process number.

Returns

0 if OK, nonzero on error.

Definition at line 378 of file mpi.c.

4.81.3.8 PF_WaitRbuf()

```
int PF_WaitRbuf (
            PF_BUFFER * r,
            int bn,
            LONG * size )
```

Waits for the buffer *bn* to finish a pending nonblocking receive. It returns the received tag and in `*size` the number of field received. If the receive is already finished, just return the flag and size, else wait for it to finish, but also check for other pending receives.

Parameters

	r	the PF_BUFFER struct for the pending nonblocking receive.
	bn	the index of the cyclic buffer.
out	*size*	the actual size of received data.

Returns

the received message tag. A negative value indicates an error.

Definition at line 412 of file mpi.c.

4.81.3.9 PF_Bcast()

```
int PF_Bcast (
            void * buffer,
            int count )
```

Broadcasts a message from the master to slaves.

Parameters

in,out	*buffer*	the starting address of buffer. The contents in this buffer on the master will be transferred to those on the slaves.
	count	the length of the buffer in bytes.

Returns

0 if OK, nonzero on error.

Definition at line 452 of file mpi.c.

Referenced by PF_BroadcastBuffer(), and PF_BroadcastNumber().

4.81.3.10 PF_Reduce()

```
int PF_Reduce (
              const void * sendbuf,
              void * recvbuf,
              int count,
              MPI_Datatype type,
              MPI_Op op,
              int root )
```

Performs a reduce operation across all processes.

Parameters

in	*sendbuf*	the buffer containing the data to be reduced.
out	*recvbuf*	the buffer to store the reduced result (only for the root process).
	count	the number of elements in the buffers.
	type	the datatype of the elements.
	op	the operation to apply.
	root	the root process number.

Returns

0 if OK, nonzero on error.

Definition at line 475 of file mpi.c.

Referenced by PF_GetSlaveTimes().

4.81.3.11 PF_RawSend()

```
int PF_RawSend (
              int dest,
              void * buf,
              LONG l,
              int tag )
```

Sends *l* bytes from *buf* to *dest*. Returns 0 on success, or -1.

Parameters

dest	the destination process number.
buf	the send buffer.
l	the size of data in the send buffer in bytes.
tag	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 497 of file mpi.c.

Referenced by PF_MUnlock(), PF_ReceiveRuntimeError(), and PF_SendFile().

4.81.3.12 PF_RawRecv()

```
LONG PF_RawRecv (
            int * src,
            void * buf,
            LONG thesize,
            int * tag )
```

Receives not more than *thesize* bytes from *src*, returns the actual number of received bytes, or -1 on failure.

Parameters

in,out	*src*	the source process number. In output, that of the actual received message.
out	*buf*	the receive buffer.
	thesize	the size of the receive buffer in bytes.
out	*tag*	the message tag of the actual received message.

Returns

the actual sizeof received data in bytes, or -1 on failure.

Definition at line 518 of file mpi.c.

Referenced by PF_ReceiveRuntimeError(), and PF_RecvFile().

4.81.3.13 PF_RawProbe()

```
int PF_RawProbe (
            int * src,
            int * tag,
            int * bytesize )
```

Probes an incoming message.

Parameters

in,out	*src*	the source process number. In output, that of the actual received message.
in,out	*tag*	the message tag. In output, that of the actual received message.
out	*bytesize*	the size of incoming data in bytes.

Returns

0 if OK, nonzero on error.

Definition at line 542 of file mpi.c.

4.81.3.14 PF_RawIsend()

```
int PF_RawIsend (
            int dest,
            const void * buf,
            int count,
            MPI_Datatype type,
            int tag,
            MPI_Request * request )
```

Performs a nonblocking send.

Parameters

	dest	the destination process number.
in	*buf*	the send buffer.
	count	the number of elements in the send buffer.
	type	the datatype of the data in the send buffer.
	tag	the message tag.
out	*request*	the request handle for the nonblocking operation.

Returns

0 if OK, nonzero on error.

Definition at line 574 of file mpi.c.

4.81.3.15 PF_RawWaitAll()

```
int PF_RawWaitAll (
            int count,
            MPI_Request * request,
            MPI_Status * status )
```

Waits for all the given requests to complete.

Parameters

	count	the number of requests.
in,out	*request*	the array of request handles.
out	*status*	the array of status objects to store the status of each completed request.

Returns

0 if OK, nonzero on error.

Definition at line 594 of file mpi.c.

4.81.3.16 PF_Discard()

```
int PF_Discard (
            int * src,
            int * tag )
```

Discards an incoming message.

Parameters

in,out	*src*	the source process number. On output, that of the actual received message.
in,out	*tag*	the message tag. On output, that of the actual received message.

Returns

0 if OK, nonzero on error.

Definition at line 615 of file mpi.c.

4.81.3.17 PF_PrintPackBuf()

```
int PF_PrintPackBuf (
            char * s,
            int size )
```

Prints the contents in the pack buffer.

Parameters

s	a message to be shown.
size	the length of the buffer to be shown.

Returns

0 if OK, nonzero on error.

Definition at line 707 of file mpi.c.

4.81.3.18 PF_PreparePack()

```
int PF_PreparePack (
            void  )
```

Prepares for the next pack operations on the sender.

Returns

0 if OK, nonzero on error.

Definition at line 736 of file mpi.c.

Referenced by Optimize(), PF_BroadcastPreDollar(), PF_BroadcastString(), and PF_Init().

4.81.3.19 PF_Pack()

```
int PF_Pack (
            const void * buffer,
            size_t count,
            MPI_Datatype type )
```

Adds data into the pack buffer.

Parameters

buffer	the pointer to the buffer storing the data to be packed.
count	the number of elements in the buffer.
type	the data type of elements in the buffer.

Returns

0 if OK, nonzero on error.

Definition at line 754 of file mpi.c.

References MPI_ERRCODE_CHECK.

Referenced by Optimize(), PF_BroadcastPreDollar(), and PF_Init().

4.81.3.20 PF_Unpack()

```
int PF_Unpack (
            void * buffer,
            size_t count,
            MPI_Datatype type )
```

Retrieves the next data in the pack buffer.

Parameters

out	*buffer*	the pointer to the buffer to store the unpacked data.
	count	the number of elements of data to be received.
	type	the data type of elements of data to be received.

Returns

0 if OK, nonzero on error.

Definition at line 783 of file mpi.c.

References MPI_ERRCODE_CHECK.

Referenced by Optimize(), PF_BroadcastPreDollar(), and PF_Init().

4.81.3.21 PF_PackString()

```
int PF_PackString (
            const UBYTE * str )
```

Packs a string *str* into the packed buffer PF_packbuf, including the trailing zero.

The first element (PF_INT) is the length of the packed portion of the string. If the string does not fit to the buffer PF_packbuf, the function packs only the initial portion. It returns the number of packed bytes, so if (str[length-1]=='\0') then the whole string fits to the buffer, if not, then the rest (str+length) must be packed and send again. On error, the function returns the negative error code.

One exception: the string "\0!\0" is used as an image of the NULL, so all 3 characters will be packed.

Parameters

str	a string to be packed.

Returns

the number of packed bytes, or a negative value on failure.

Definition at line 818 of file mpi.c.

Referenced by PF_BroadcastString().

4.81.3.22 PF_UnpackString()

```
int PF_UnpackString (
            UBYTE * str )
```

Unpacks a string to *str* from the packed buffer PF_packbuf, including the trailing zero.

It returns the number of unpacked bytes, so if (str[length-1]=='\0') then the whole string was unpacked, if not, then the rest must be appended to (str+length). On error, the function returns the negative error code.

Parameters

out	*str*	the buffer to store the unpacked string

Returns

the number of unpacked bytes, or a negative value on failure.

Definition at line 886 of file mpi.c.

Referenced by PF_BroadcastString().

4.81.3.23 PF_Send()

```
int PF_Send (
              int to,
              int tag )
```

Sends the contents in the pack buffer to the process specified by *to*.

Example:

```
if ( PF.me == SRC ) {
  PF_PreparePack();
  // Packing operations here...
  PF_Send(DEST, TAG);
}
else if ( PF.me == DEST ) {
  PF_Receive(SRC, TAG, &actual_src, &actual_tag);
  // Unpacking operations here...
}
```

Parameters

to	the destination process number.
tag	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 933 of file mpi.c.

References MPI_ERRCODE_CHECK.

4.81.3.24 PF_Receive()

```
int PF_Receive (
              int src,
              int tag,
              int * psrc,
              int * ptag )
```

Receives data into the pack buffer from the process specified by *src*. This function allows &src == psrc or &tag == ptag. Either *psrc* or *ptag* can be NULL.

See the example of PF_Send().

Parameters

	src	the source process number (can be PF_ANY_SOURCE).
	tag	the source message tag (can be PF_ANY_TAG).
out	*psrc*	the actual source process number of received message.
out	*ptag*	the received message tag.

Returns

0 if OK, nonzero on error.

Definition at line 959 of file mpi.c.

References MPI_ERRCODE_CHECK.

4.81.3.25 PF_Broadcast()

```
int PF_Broadcast (
            void  )
```

Broadcasts the contents in the pack buffer on the master to those on the slaves.

Example:

```
if ( PF.me == MASTER ) {
  PF_PreparePack();
  // Packing operations here...
}
PF_Broadcast();
if ( PF.me != MASTER ) {
  // Unpacking operations here...
}
```

Returns

0 if OK, nonzero on error.

Definition at line 994 of file mpi.c.

References MPI_ERRCODE_CHECK.

Referenced by Optimize(), PF_BroadcastPreDollar(), PF_BroadcastString(), and PF_Init().

4.81.3.26 PF_PrepareLongSinglePack()

```
int PF_PrepareLongSinglePack (
            void  )
```

Prepares for the next long-single-pack operations on the sender.

Returns

0 if OK, nonzero on error.

Definition at line 1561 of file mpi.c.

Referenced by PF_CollectModifiedDollars(), and PF_Processor().

4.81.3.27 PF_LongSinglePack()

```
int PF_LongSinglePack (
            const void * buffer,
            size_t count,
            MPI_Datatype type )
```

Adds data into the "long single" pack buffer.

Parameters

buffer	the pointer to the buffer storing the data to be packed.
count	the number of elements in the buffer.
type	the data type of elements in the buffer.

Returns

0 if OK, nonzero on error.

Definition at line 1579 of file mpi.c.

Referenced by PF_CollectModifiedDollars(), and PF_Processor().

4.81.3.28 PF_LongSingleUnpack()

```
int PF_LongSingleUnpack (
            void * buffer,
            size_t count,
            MPI_Datatype type )
```

Retrieves the next data in the "long single" pack buffer.

Parameters

out	*buffer*	the pointer to the buffer to store the unpacked data.
	count	the number of elements of data to be received.
	type	the data type of elements of data to be received.

Returns

0 if OK, nonzero on error.

Definition at line 1613 of file mpi.c.

Referenced by PF_CollectModifiedDollars(), and PF_Processor().

4.81.3.29 PF_LongSingleSend()

```
int PF_LongSingleSend (
            int to,
            int tag )
```

Sends the contents in the "long single" pack buffer to the process specified by *to*.

Example:

```
if ( PF.me == SRC ) {
  PF_PrepareLongSinglePack();
  // Packing operations here...
  PF_LongSingleSend(DEST, TAG);
}
else if ( PF.me == DEST ) {
  PF_LongSingleReceive(SRC, TAG, &actual_src, &actual_tag);
  // Unpacking operations here...
}
```

Parameters

to	the destination process number.
tag	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 1650 of file mpi.c.

Referenced by PF_CollectModifiedDollars(), and PF_Processor().

4.81.3.30 PF_LongSingleReceive()

```
int PF_LongSingleReceive (
            int src,
            int tag,
            int * psrc,
            int * ptag )
```

Receives data into the "long single" pack buffer from the process specified by *src*. This function allows &src == psrc or &tag == ptag. Either *psrc* or *ptag* can be NULL.

See the example of PF_LongSingleSend().

Parameters

	src	the source process number (can be PF_ANY_SOURCE).
	tag	the source message tag (can be PF_ANY_TAG).
out	*psrc*	the actual source process number of received message.
out	*ptag*	the received message tag.

Returns

0 if OK, nonzero on error.

Definition at line 1693 of file mpi.c.

Referenced by PF_CollectModifiedDollars(), and PF_Processor().

4.81.3.31 PF_PrepareLongMultiPack()

```
int PF_PrepareLongMultiPack (
            void  )
```

Prepares for the next long-multi-pack operations on the sender.

Returns

0 if OK, nonzero on error.

Definition at line 1752 of file mpi.c.

Referenced by Optimize(), PF_BroadcastCBuf(), PF_BroadcastExpFlags(), PF_BroadcastModifiedDollars(), and PF_BroadcastRedefinedPreVars().

4.81.3.32 PF_LongMultiPackImpl()

```
int PF_LongMultiPackImpl (
            const void * buffer,
            size_t count,
            size_t eSize,
            MPI_Datatype type )
```

Adds data into the "long multi" pack buffer.

Parameters

buffer	the pointer to the buffer storing the data to be packed.
count	the number of elements in the buffer.
eSize	the byte size of each element of data.
type	the data type of elements in the buffer.

Returns

0 if OK, nonzero on error.

Definition at line 1771 of file mpi.c.

References PF_LongMultiPackImpl().

Referenced by PF_LongMultiPackImpl().

Here is the call graph for this function:

4.81.3.33 PF_LongMultiUnpackImpl()

```
int PF_LongMultiUnpackImpl (
            void * buffer,
            size_t count,
            size_t eSize,
            MPI_Datatype type )
```

Retrieves the next data in the "long multi" pack buffer.

Parameters

out	*buffer*	the pointer to the buffer to store the unpacked data.
	count	the number of elements of data to be received.
	eSize	the byte size of each element of data.
	type	the data type of elements of data to be received.

Returns

0 if OK, nonzero on error.

Definition at line 1830 of file mpi.c.

References PF_LongMultiUnpackImpl().

Referenced by PF_LongMultiUnpackImpl().

Here is the call graph for this function:

4.81.3.34 PF_LongMultiBroadcast()

```
int PF_LongMultiBroadcast (
              void  )
```

Broadcasts the contents in the "long multi" pack buffer on the master to those on the slaves.

Example:

```
if ( PF.me == MASTER ) {
  PF_PrepareLongMultiPack();
  // Packing operations here...
}
PF_LongMultiBroadcast();
if ( PF.me != MASTER ) {
  // Unpacking operations here...
}
```

Returns

0 if OK, nonzero on error.

Definition at line 1916 of file mpi.c.

Referenced by Optimize(), PF_BroadcastCBuf(), PF_BroadcastExpFlags(), PF_BroadcastModifiedDollars(), and PF_BroadcastRedefinedPreVars().

4.81.4 Variable Documentation

4.81.4.1 PF_maxDollarChunkSize

```
LONG PF_maxDollarChunkSize = 0
```

Definition at line 74 of file mpi.c.

4.82 mpi.c

Go to the documentation of this file.

```
00001
00009 /* #[ License : */
00010 /*
00011  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00012  *   When using this file you are requested to refer to the publication
00013  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00014  *   This is considered a matter of courtesy as the development was paid
00015  *   for by FOM the Dutch physics granting agency and we would like to
00016  *   be able to track its scientific use to convince FOM of its value
00017  *   for the community.
00018  *
00019  *   This file is part of FORM.
00020  *
00021  *   FORM is free software: you can redistribute it and/or modify it under the
00022  *   terms of the GNU General Public License as published by the Free Software
00023  *   Foundation, either version 3 of the License, or (at your option) any later
00024  *   version.
00025  *
00026  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00027  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00028  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00029  *   details.
00030  *
00031  *   You should have received a copy of the GNU General Public License along
00032  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00033  */
00034 /* #] License : */
00035 /*
00036     #[ Includes and variables :
00037 */
00038
00039 /*
00040 #define MPIDEBUGGING
00041 #define MPIDEBUGGING_DELAY_US 10000
00042 */
00043
00044 #include <limits.h>
00045 #include "form3.h"
00046
00047 #ifdef MPICH_PROFILING
00048 # include "mpe.h"
00049 #endif
00050
00051 #ifdef MPIDEBUGGING
00052 #include "mpidbg.h"
00053 #endif
00054
00055 /*[12oct2005 mt]:*/
00056 /*
00057     Today there was some cleanup, some stuff is moved into another place
00058     in this file, and PF.packsize is removed and PF_packsize is used
00059     instead. It is rather difficult to proper comment it, so not all these
00060     changing are marked by "[12oct2005 mt]"
00061 */
00062
00063 #define PF_PACKSIZE 1600
00064
00065 /*
00066     Size in bytes, will be initialized soon as
00067     PF_packsize=PF_PACKSIZE/sizeof(int)*sizeof(int); for possible
00068     future developing we prefer to do this initialization not here,
00069     but in PF_LibInit:
00070 */
00071
00072 static int PF_packsize = 0;
00073 static MPI_Status PF_status;
```

```
00074 LONG PF_maxDollarChunkSize = 0;      /*:[04oct2005 mt]*/
00075
00076 static int PF_ShortPackInit(void);
00077 static int PF_longPackInit(void);      /*:[12oct2005 mt]*/
00078
00089 #define MPI_ERRCODE_CHECK(err) \
00090     do { \
00091         int _tmp_err = (err); \
00092         if ( _tmp_err != MPI_SUCCESS ) return _tmp_err != 0 ? _tmp_err : -1; \
00093     } while (0)
00094
00095 /*
00096     #] Includes and variables :
00097     #[ PF_RealTime :
00098 */
00099
00106 LONG PF_RealTime(int i)
00107 {
00108     static double starttime;
00109     if ( i == PF_RESET ) {
00110         starttime = MPI_Wtime();
00111         return((LONG)0);
00112     }
00113     return((LONG)( 100. * (MPI_Wtime() - starttime) ) );
00114 }
00115
00116 /*
00117     #] PF_RealTime :
00118     #[ PF_LibInit :
00119 */
00120
00128 int PF_LibInit(int *argcp, char ***argvp)
00129 {
00130     int ret;
00131     ret = MPI_Init(argcp,argvp);
00132     if ( ret != MPI_SUCCESS ) return(ret);
00133     ret = MPI_Comm_rank(PF_COMM,&PF.me);
00134     if ( ret != MPI_SUCCESS ) return(ret);
00135     ret = MPI_Comm_size(PF_COMM,&PF.numtasks);
00136     if ( ret != MPI_SUCCESS ) return(ret);
00137
00138     /* Initialization of packed communications. */
00139     PF_packsize = PF_PACKSIZE/sizeof(int)*sizeof(int);
00140     if ( PF_ShortPackInit() ) return -1;
00141     if ( PF_longPackInit() ) return -1;
00142
00143     {/*Block*/
00144         int bytes, totalbytes=0;
00145 /*
00146             There is one problem with maximal possible packing: there is no API to
00147             convert bytes to the record number. So, here we calculate the buffer
00148             size needed for storing dollarvars:
00149
00150             LONG PF_maxDollarChunkSize is the size for the portion of the dollar
00151             variable buffer suitable for broadcasting. This variable should be
00152             visible from parallel.c
00153
00154             Evaluate PF_Pack(numterms,1,PF_INT):
00155 */
00156         if ( ( ret = MPI_Pack_size(1,PF_INT,PF_COMM,&bytes) )!=MPI_SUCCESS )
00157             return(ret);
00158
00159         totalbytes+=bytes;
00160 /*
00161             Evaluate PF_Pack( newsize,1,PF_LONG):
00162 */
00163         if ( ( ret = MPI_Pack_size(1,PF_LONG,PF_COMM,&bytes) )!=MPI_SUCCESS )
00164             return(ret);
00165
00166         totalbytes += bytes;
00167 /*
00168             Now available room is PF_packsize-totalbytes
00169 */
00170         totalbytes = PF_packsize-totalbytes;
00171 /*
00172             Now totalbytes is the size of chunk in bytes.
00173             Evaluate this size in number of records:
00174
00175             Rough estimate:
00176 */
00177         PF_maxDollarChunkSize=totalbytes/sizeof(WORD);
00178 /*
00179             Go to the up limit:
00180 */
00181         do {
00182             if ( ( ret = MPI_Pack_size(
00183                         ++PF_maxDollarChunkSize,PF_WORD,PF_COMM,&bytes) )!=MPI_SUCCESS )
```

```
00184                 return(ret);
00185         } while ( bytes<totalbytes );
00186 /*
00187             Now the chunk size is too large
00188             And now evaluate the exact value:
00189 */
00190         do {
00191             if ( ( ret = MPI_Pack_size(
00192                         --PF_maxDollarChunkSize,PF_WORD,PF_COMM,&bytes) )!=MPI_SUCCESS )
00193                 return(ret);
00194         } while ( bytes>totalbytes );
00195 /*
00196             Now PF_maxDollarChunkSize is the size of chunk of PF_WORD fitting the
00197             buffer <= (PF_packsize-PF_INT-PF_LONG)
00198 */
00199     }/*Block*/
00200     return(0);
00201 }
00202 /*
00203     #] PF_LibInit :
00204     #[ PF_LibTerminate :
00205 */
00206
00214 int PF_LibTerminate(int error)
00215 {
00216     DUMMYUSE(error);
00217     return(MPI_Finalize());
00218 }
00219
00220 /*
00221     #] PF_LibTerminate :
00222     #[ PF_Probe :
00223 */
00224
00235 int PF_Probe(int *src)
00236 {
00237     int ret, flag;
00238     if ( *src == PF_ANY_SOURCE ) { /*Blocking call*/
00239         ret = MPI_Probe(*src,MPI_ANY_TAG,PF_COMM,&PF_status);
00240         flag = 1;
00241     }
00242     else { /*Non-blocking call*/
00243         ret = MPI_Iprobe(*src,MPI_ANY_TAG,PF_COMM,&flag,&PF_status);
00244     }
00245     *src = PF_status.MPI_SOURCE;
00246     if ( ret != MPI_SUCCESS ) { if ( ret > 0 ) ret *= -1; return(ret); }
00247     if ( !flag ) return(0);
00248     return(PF_status.MPI_TAG);
00249 }
00250
00251 /*
00252     #] PF_Probe :
00253     #[ PF_ISendSbuf :
00254 */
00255
00266 int PF_ISendSbuf(int to, int tag)
00267 {
00268     PF_BUFFER *s = PF.sbuf;
00269     int a = s->active;
00270     int size = s->fill[a] - s->buff[a];
00271     int r = 0;
00272
00273     static int finished;
00274
00275     s->fill[a] = s->buff[a];
00276     if ( s->numbufs == 1 ) {
00277         r = MPI_Ssend(s->buff[a],size,PF_WORD,MASTER,tag,PF_COMM);
00278         if ( r != MPI_SUCCESS ) {
00279             fprintf(stderr,"[%d|%d] PF_ISendSbuf: MPI_Ssend returns: %d \n",
00280                     PF.me,(int)AC.CModule,r);
00281             fflush(stderr);
00282             return(r);
00283         }
00284         return(0);
00285     }
00286
00287     switch ( tag ) { /* things to do before sending */
00288         case PF_TERM_MSGTAG:
00289             if ( PF.sbuf->request[to] != MPI_REQUEST_NULL)
00290                 r = MPI_Wait(&PF.sbuf->request[to],&PF.sbuf->retstat[to]);
00291             if ( r != MPI_SUCCESS ) return(r);
00292             break;
00293         default:
00294             break;
00295     }
00296
00297     r = MPI_Isend(s->buff[a],size,PF_WORD,to,tag,PF_COMM,&s->request[a]);
```

```
00298
00299     if ( r != MPI_SUCCESS ) return(r);
00300
00301     switch ( tag ) { /* things to do after initialising sending */
00302         case PF_TERM_MSGTAG:
00303             finished = 0;
00304             break;
00305         case PF_ENDSORT_MSGTAG:
00306             if ( ++finished == PF.numtasks - 1 )
00307                 r = MPI_Waitall(s->numbufs,s->request,s->status);
00308             if ( r != MPI_SUCCESS ) return(r);
00309             break;
00310         case PF_BUFFER_MSGTAG:
00311             if ( ++s->active >= s->numbufs ) s->active = 0;
00312             while ( s->request[s->active] != MPI_REQUEST_NULL ) {
00313                 r = MPI_Waitsome(s->numbufs,s->request,&size,s->index,s->retstat);
00314                 if ( r != MPI_SUCCESS ) return(r);
00315             }
00316             break;
00317         case PF_ENDBUFFER_MSGTAG:
00318             if ( ++s->active >= s->numbufs ) s->active = 0;
00319             r = MPI_Waitall(s->numbufs,s->request,s->status);
00320             if ( r != MPI_SUCCESS ) return(r);
00321             break;
00322         default:
00323             return(-99);
00324             break;
00325     }
00326     return(0);
00327 }
00328
00329 /*
00330     #] PF_ISendSbuf :
00331     #[ PF_RecvWbuf :
00332 */
00333
00342 int PF_RecvWbuf(WORD *b, LONG *s, int *src)
00343 {
00344     int i, r = 0;
00345
00346     for (;;) {
00347         r = MPI_Probe(*src,PF_ANY_MSGTAG,PF_COMM,&PF_status);
00348         if ( r != MPI_SUCCESS ) { if ( r > 0 ) r *= -1; return(r); }
00349         if ( PF_status.MPI_TAG != PF_RUNTIME_ERROR_MSGTAG ) break;
00350         PF_ReceiveRuntimeError();
00351     }
00352
00353     r = MPI_Recv(b,(int)*s,PF_WORD,PF_status.MPI_SOURCE,PF_status.MPI_TAG,PF_COMM,&PF_status);
00354     if ( r != MPI_SUCCESS ) { if ( r > 0 ) r *= -1; return(r); }
00355
00356     r = MPI_Get_count(&PF_status,PF_WORD,&i);
00357     if ( r != MPI_SUCCESS ) { if ( r > 0 ) r *= -1; return(r); }
00358
00359     *s = (LONG)i;
00360     *src = PF_status.MPI_SOURCE;
00361     return(PF_status.MPI_TAG);
00362 }
00363
00364 /*
00365     #] PF_RecvWbuf :
00366     #[ PF_IRecvRbuf :
00367 */
00368
00378 int PF_IRecvRbuf(PF_BUFFER *r, int bn, int from)
00379 {
00380     int ret;
00381     r->type[bn] = PF_WORD;
00382
00383     if ( r->numbufs == 1 ) {
00384         r->tag[bn] = MPI_ANY_TAG;
00385         r->from[bn] = from;
00386     }
00387     else {
00388         ret = MPI_Irecv(r->full[bn],(int)(r->stop[bn] - r->full[bn]),PF_WORD,from,
00389                         MPI_ANY_TAG,PF_COMM,&r->request[bn]);
00390         if (ret != MPI_SUCCESS) { if(ret > 0) ret *= -1; return(ret); }
00391     }
00392     return(0);
00393 }
00394
00395 /*
00396     #] PF_IRecvRbuf :
00397     #[ PF_WaitRbuf :
00398 */
00399
00412 int PF_WaitRbuf(PF_BUFFER *r, int bn, LONG *size)
00413 {
```

```
00414     int ret, rsize;
00415
00416     if ( r->numbufs == 1 ) {
00417         *size = r->stop[bn] - r->full[bn];
00418         ret = MPI_Recv(r->full[bn],(int)*size,r->type[bn],r->from[bn],r->tag[bn],
00419                        PF_COMM,&(r->status[bn]));
00420         if ( ret != MPI_SUCCESS ) { if ( ret > 0 ) ret *= -1; return(ret); }
00421         ret = MPI_Get_count(&(r->status[bn]),r->type[bn],&rsize);
00422         if ( ret != MPI_SUCCESS ) { if ( ret > 0 ) ret *= -1; return(ret); }
00423         if ( rsize > *size ) return(-99);
00424         *size = (LONG)rsize;
00425     }
00426     else {
00427         while ( r->request[bn] != MPI_REQUEST_NULL ) {
00428             ret = MPI_Waitsome(r->numbufs,r->request,&rsize,r->index,r->retstat);
00429             if ( ret != MPI_SUCCESS ) { if ( ret > 0 ) ret *= -1; return(ret); }
00430             while ( --rsize >= 0 ) r->status[r->index[rsize]] = r->retstat[rsize];
00431         }
00432         ret = MPI_Get_count(&(r->status[bn]),r->type[bn],&rsize);
00433         if ( ret != MPI_SUCCESS ) { if ( ret > 0 ) ret *= -1; return(ret); }
00434         *size = (LONG)rsize;
00435     }
00436     return(r->status[bn].MPI_TAG);
00437 }
00438
00439 /*
00440     #] PF_WaitRbuf :
00441     #[ PF_Bcast :
00442 */
00443
00452 int PF_Bcast(void *buffer, int count)
00453 {
00454     if ( MPI_Bcast(buffer,count,MPI_BYTE,MASTER,PF_COMM) != MPI_SUCCESS )
00455         return(-1);
00456     return(0);
00457 }
00458
00459 /*
00460     #] PF_Bcast :
00461     #[ PF_Reduce :
00462 */
00463
00475 int PF_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype type, MPI_Op op, int root)
00476 {
00477     if ( MPI_Reduce(sendbuf,recvbuf,count,type,op,root,PF_COMM) != MPI_SUCCESS )
00478         return(-1);
00479     return(0);
00480 }
00481
00482 /*
00483     #] PF_Reduce :
00484     #[ PF_RawSend :
00485 */
00486
00497 int PF_RawSend(int dest, void *buf, LONG l, int tag)
00498 {
00499     int ret=MPI_Ssend(buf,(int)l,MPI_BYTE,dest,tag,PF_COMM);
00500     if ( ret != MPI_SUCCESS ) return(-1);
00501     return(0);
00502 }
00503 /*
00504     #] PF_RawSend :
00505     #[ PF_RawRecv :
00506 */
00507
00518 LONG PF_RawRecv(int *src,void *buf,LONG thesize,int *tag)
00519 {
00520     MPI_Status stat;
00521     int ret=MPI_Recv(buf,(int)thesize,MPI_BYTE,*src,MPI_ANY_TAG,PF_COMM,&stat);
00522     if ( ret != MPI_SUCCESS ) return(-1);
00523     if ( MPI_Get_count(&stat,MPI_BYTE,&ret) != MPI_SUCCESS ) return(-1);
00524     *tag = stat.MPI_TAG;
00525     *src = stat.MPI_SOURCE;
00526     return(ret);
00527 }
00528
00529 /*
00530     #] PF_RawRecv :
00531     #[ PF_RawProbe :
00532 */
00533
00542 int PF_RawProbe(int *src, int *tag, int *bytesize)
00543 {
00544     MPI_Status stat;
00545     int srcval = src != NULL ? *src : PF_ANY_SOURCE;
00546     int tagval = tag != NULL ? *tag : PF_ANY_MSGTAG;
00547     int ret = MPI_Probe(srcval, tagval, PF_COMM, &stat);
```

```
00548     if ( ret != MPI_SUCCESS ) return -1;
00549     if ( src != NULL ) *src = stat.MPI_SOURCE;
00550     if ( tag != NULL ) *tag = stat.MPI_TAG;
00551     if ( bytesize != NULL ) {
00552         ret = MPI_Get_count(&stat, MPI_BYTE, bytesize);
00553         if ( ret != MPI_SUCCESS ) return -1;
00554     }
00555     return 0;
00556 }
00557
00558 /*
00559     #] PF_RawProbe :
00560     #[ PF_RawIsend :
00561 */
00562
00574 int PF_RawIsend(int dest, const void *buf, int count, MPI_Datatype type, int tag, MPI_Request
      *request)
00575 {
00576     int ret = MPI_Isend(buf, count, type, dest, tag, PF_COMM, request);
00577     if ( ret != MPI_SUCCESS ) return(-1);
00578     return(0);
00579 }
00580
00581 /*
00582     #] PF_RawIsend :
00583     #[ PF_RawWaitAll :
00584 */
00585
00594 int PF_RawWaitAll(int count, MPI_Request *request, MPI_Status *status)
00595 {
00596     int ret = MPI_Waitall(count, request, status);
00597     if ( ret != MPI_SUCCESS ) return(-1);
00598     return(0);
00599 }
00600
00601 /*
00602     #] PF_RawWaitAll :
00603     #[ PF_Discard :
00604 */
00605
00615 int PF_Discard(int *src, int *tag)
00616 {
00617     enum { DEFAULT_BUF_SIZE = 1024 };
00618     MPI_Status stat;
00619     int count;
00620     void *buf;
00621     char default_buf[DEFAULT_BUF_SIZE];
00622     int srcval = src != NULL ? *src : PF_ANY_SOURCE;
00623     int tagval = tag != NULL ? *tag : PF_ANY_MSGTAG;
00624     int ret = MPI_Probe(srcval, tagval, PF_COMM, &stat);
00625     if ( ret != MPI_SUCCESS ) return -1;
00626     if ( src != NULL ) *src = stat.MPI_SOURCE;
00627     if ( tag != NULL ) *tag = stat.MPI_TAG;
00628     ret = MPI_Get_count(&stat, MPI_BYTE, &count);
00629     if ( ret != MPI_SUCCESS ) return -1;
00630     buf = count <= DEFAULT_BUF_SIZE ? default_buf : Malloc1(count, "PF_Discard");
00631     ret = MPI_Recv(buf, count, MPI_BYTE, stat.MPI_SOURCE, stat.MPI_TAG, PF_COMM, MPI_STATUS_IGNORE);
00632     if ( buf != default_buf ) M_free(buf, "PF_Discard");
00633     if ( ret != MPI_SUCCESS ) return -1;
00634     return 0;
00635 }
00636
00637 /*
00638     #] PF_Discard :
00639     #[ The pack buffer :
00640         #[ Variables :
00641 */
00642
00643 /*
00644  * The pack buffer with the fixed size (= PF_packsize).
00645  */
00646 static UBYTE *PF_packbuf  = NULL;
00647 static UBYTE *PF_packstop = NULL;
00648 static int    PF_packpos  = 0;
00649
00650 /*
00651         #] Variables :
00652         #[ PF_ShortPackInit :
00653 */
00654
00661 static int PF_ShortPackInit(void)
00662 {
00663     PF_packbuf = (UBYTE *)Malloc1(sizeof(UBYTE) * PF_packsize, "PF_ShortPackInit");
00664     if ( PF_packbuf == NULL ) return -1;
00665     PF_packstop = PF_packbuf + PF_packsize;
00666     return 0;
00667 }
```

```
00668
00669 /*
00670         #] PF_ShortPackInit :
00671         #[ PF_InitPackBuf :
00672 */
00673
00679 static inline int PF_InitPackBuf(void)
00680 {
00681 /*
00682         This is definitely not the best place for allocating the
00683         buffer! Moved to PF_LibInit():
00684
00685     if ( PF_packbuf == 0 ) {
00686         PF_packbuf = (UBYTE *)Malloc1(sizeof(UBYTE)*PF.packsize,"PF_InitPackBuf");
00687         if ( PF_packbuf == 0 ) return(-1);
00688         PF_packstop = PF_packbuf + PF.packsize;
00689     }
00690 */
00691     PF_packpos = 0;
00692     return(0);
00693 }
00694
00695 /*
00696         #] PF_InitPackBuf :
00697         #[ PF_PrintPackBuf :
00698 */
00699
00707 int PF_PrintPackBuf(char *s, int size)
00708 {
00709 #ifdef NOMESPRINTYET
00710 /*
00711         The use of printf should be discouraged. The results are flushed to
00712         the output at unpredictable moments. We should use printf only
00713         during startup when MesPrint doesn't have its buffers and output
00714         channels initialized.
00715 */
00716     int i;
00717     printf("[%d] %s: ",PF.me,s);
00718     for(i=0;i<size;i++) printf("%d ",PF_packbuf[i]);
00719     printf("\n");
00720 #else
00721     MesPrint("[%d] %s: %a",PF.me,s,size,(WORD *)(PF_packbuf));
00722 #endif
00723     return(0);
00724 }
00725
00726 /*
00727         #] PF_PrintPackBuf :
00728         #[ PF_PreparePack :
00729 */
00730
00736 int PF_PreparePack(void)
00737 {
00738     return PF_InitPackBuf();
00739 }
00740
00741 /*
00742         #] PF_PreparePack :
00743         #[ PF_Pack :
00744 */
00745
00754 int PF_Pack(const void *buffer, size_t count, MPI_Datatype type)
00755 {
00756     int err, bytes;
00757
00758     if ( count > INT_MAX ) return -99;
00759
00760     err = MPI_Pack_size((int)count, type, PF_COMM, &bytes);
00761     MPI_ERRCODE_CHECK(err);
00762     if ( PF_packpos + bytes > PF_packstop - PF_packbuf ) return -99;
00763
00764     err = MPI_Pack((void *)buffer, (int)count, type, PF_packbuf, PF_packsize, &PF_packpos, PF_COMM);
00765     MPI_ERRCODE_CHECK(err);
00766
00767     return 0;
00768 }
00769
00770 /*
00771         #] PF_Pack :
00772         #[ PF_Unpack :
00773 */
00774
00783 int PF_Unpack(void *buffer, size_t count, MPI_Datatype type)
00784 {
00785     int err;
00786
00787     if ( count > INT_MAX ) return -99;
```

```
00788
00789     err = MPI_Unpack(PF_packbuf, PF_packsize, &PF_packpos, buffer, (int)count, type, PF_COMM);
00790     MPI_ERRCODE_CHECK(err);
00791
00792     return 0;
00793 }
00794
00795 /*
00796         #] PF_Unpack :
00797         #[ PF_PackString :
00798 */
00799
00818 int PF_PackString(const UBYTE *str)
00819 {
00820     int ret,buflength,bytes,length;
00821 /*
00822         length will be packed in the beginning.
00823         Decrement buffer size by the length of the field "length":
00824 */
00825     if ( ( ret = MPI_Pack_size(1,PF_INT,PF_COMM,&bytes) ) != MPI_SUCCESS )
00826         return(ret);
00827     buflength = PF_packsize - bytes;
00828 /*
00829         Calculate the string length (INCLUDING the trailing zero!):
00830 */
00831     for ( length = 0; length < buflength; length++ ) {
00832         if ( str[length] == '\0' ) {
00833             length++; /* since the trailing zero must be accounted */
00834             break;
00835         }
00836     }
00837 /*
00838         The string "\0!\0" is used as an image of the NULL.
00839 */
00840     if ( ( str[0] == '\0' ) /* empty string */
00841       && ( str[1] == '!' )  /* Special case? */
00842       && ( str[2] == '\0' ) /* Yes, pass 3 initial symbols */
00843             ) length += 2;  /* all 3 characters will be packed */
00844     length++;               /* Will be decremented in the following loop */
00845 /*
00846         The problem: packed size of byte may be not equal 1! So first, suppose
00847         it is 1, and if this is not the case decrease the length of the string
00848         until it fits the buffer:
00849 */
00850     do {
00851         if ( ( ret = MPI_Pack_size(--length,PF_BYTE,PF_COMM,&bytes) )
00852             != MPI_SUCCESS ) return(ret);
00853     } while ( bytes > buflength );
00854 /*
00855         Note, now if str[length-1] == '\0' then the string fits to the buffer
00856         (INCLUDING the trailing zero!);if not, the rest must be packed further!
00857
00858         Pack the length to PF_packbuf:
00859 */
00860     if ( ( ret = MPI_Pack(&length,1,PF_INT,PF_packbuf,PF_packsize,
00861             &PF_packpos,PF_COMM) ) != MPI_SUCCESS ) return(ret);
00862 /*
00863         Pack the string to PF_packbuf:
00864 */
00865     if ( ( ret = MPI_Pack((UBYTE *)str,length,PF_BYTE,PF_packbuf,PF_packsize,
00866             &PF_packpos,PF_COMM) ) != MPI_SUCCESS ) return(ret);
00867     return(length);
00868 }
00869
00870 /*
00871         #] PF_PackString :
00872         #[ PF_UnpackString :
00873 */
00874
00886 int PF_UnpackString(UBYTE *str)
00887 {
00888     int ret,length;
00889 /*
00890         Unpack the length:
00891 */
00892     if(  (ret = MPI_Unpack(PF_packbuf,PF_packsize,&PF_packpos,
00893             &length,1,PF_INT,PF_COMM))!= MPI_SUCCESS )
00894                 return(ret);
00895 /*
00896         Unpack the string:
00897 */
00898     if ( ( ret = MPI_Unpack(PF_packbuf,PF_packsize,&PF_packpos,
00899             str,length,PF_BYTE,PF_COMM) ) != MPI_SUCCESS ) return(ret);
00900 /*
00901         Now if str[length-1]=='\0' then the whole string
00902         (INCLUDING the trailing zero!) was unpacked ;if not, the rest
00903         must be unpacked to str+length.
```

```
00904 */
00905     return(length);
00906 }
00907 
00908 /*
00909         #] PF_UnpackString :
00910         #[ PF_Send :
00911 */
00912 
00933 int PF_Send(int to, int tag)
00934 {
00935     int err;
00936     err = MPI_Ssend(PF_packbuf, PF_packpos, MPI_PACKED, to, tag, PF_COMM);
00937     MPI_ERRCODE_CHECK(err);
00938     return 0;
00939 }
00940 
00941 /*
00942         #] PF_Send :
00943         #[ PF_Receive :
00944 */
00945 
00959 int PF_Receive(int src, int tag, int *psrc, int *ptag)
00960 {
00961     int err;
00962     MPI_Status status;
00963     PF_InitPackBuf();
00964     err = MPI_Recv(PF_packbuf, PF_packsize, MPI_PACKED, src, tag, PF_COMM, &status);
00965     MPI_ERRCODE_CHECK(err);
00966     if ( psrc ) *psrc = status.MPI_SOURCE;
00967     if ( ptag ) *ptag = status.MPI_TAG;
00968     return 0;
00969 }
00970 
00971 /*
00972         #] PF_Receive :
00973         #[ PF_Broadcast :
00974 */
00975 
00994 int PF_Broadcast(void)
00995 {
00996     int err;
00997 /*
00998  * If PF_SHORTBROADCAST is defined, then the broadcasting will be performed in
00999  * 2 steps. First, the size of the buffer will be broadcast, then the buffer of
01000  * exactly used size. This should be faster with slow connections, but slower on
01001  * SMP shmem MPI because of the latency.
01002  */
01003 #ifdef PF_SHORTBROADCAST
01004     int pos = PF_packpos;
01005 #endif
01006     if ( PF.me != MASTER ) {
01007         err = PF_InitPackBuf();
01008         if ( err ) return err;
01009     }
01010 #ifdef PF_SHORTBROADCAST
01011     err = MPI_Bcast(&pos, 1, MPI_INT, MASTER, PF_COMM);
01012     MPI_ERRCODE_CHECK(err);
01013     err = MPI_Bcast(PF_packbuf, pos, MPI_PACKED, MASTER, PF_COMM);
01014 #else
01015     err = MPI_Bcast(PF_packbuf, PF_packsize, MPI_PACKED, MASTER, PF_COMM);
01016 #endif
01017     MPI_ERRCODE_CHECK(err);
01018     return 0;
01019 }
01020 
01021 /*
01022         #] PF_Broadcast :
01023     #] The pack buffer :
01024     #[ Long pack stuff :
01025         #[ Explanations :
01026 
01027     The problems here are:
01028         1. We need to send/receive long dollar variables. For
01029     preprocessor-defined dollarvars we used multiply
01030     packing/broadcasting  (see parallel.c:PF_BroadcastPreDollar())
01031     since each variable must be broadcast immediately. For run-time
01032     the changed dollar variables, collecting and broadcasting are
01033     performed at the end of the module and all modified dollarvars
01034     are transferred "at once", that is why the size of packed and
01035     transferred buffers may be really very large.
01036         2. There is some strange feature of MPI_Bcast() on Altix MPI
01037     implementation, namely, sometimes it silently fails with big
01038     buffers. For better performance, it would be useful to send one
01039     big buffer instead of several small ones (since the latency is more
01040     important than the bandwidth). That is why we need two different
01041     sets of routines: for long point-to-point communication we collect
```

```
01042     big re-allocatable buffer, the corresponding routines have the
01043     prefix PF_longSingle, and for broadcasting we pack data into
01044     several smaller buffers, the corresponding routines have the
01045     prefix PF_longMulti.
01046         Note, from portability reasons we cannot split large packed
01047     buffer into small chunks, send them and collect back on the other
01048     side, see "Advice to users" on page 180 MPI--The Complete Reference
01049     Volume1, second edition.
01050         OPTIMIZING:
01051         We assume, for most communications, the single buffer of size
01052     PF_packsize is enough.
01053
01054         How does it work:
01055         For point-to-point, there is one big re-allocatable
01056     buffer PF_longPackBuf with two integer positions: PF_longPackPos
01057     and PF_longPackTop (due to re-allocatable character of the buffer,
01058     it is better to use integers rather than pointers).
01059         Each time of re-allocation, the size of the buffer
01060     PF_longPackBuf is incremented by the same size of a "standard" chunk
01061     PF_packsize.
01062         For broadcasting there is one linked list (PF_longMultiRoot),
01063     which contains either positions of a chunk of PF_longPackBuf, or
01064     it's own buffer. This is done for better memory utilisation:
01065     longSingle and longMulti are never used simultaneously.
01066         When a new cell is needed for LongMulti packing, we increment
01067     the counter PF_longPackN and just follow the list. If it is not
01068     possible, we allocate the cell's own buffer and link it to the end
01069     of the list PF_longMultiRoot.
01070         When PF_longPackPos is reallocated, we link new chunks into
01071     existing PF_longMultiRoot list before the first longMulti allocated
01072     cell's own buffer. The pointer PF_longMultiLastChunk points to the last
01073     cell of PF_longMultiRoot containing the pointer to the chunk of
01074     PF_longPackBuf.
01075         Initialization PF_longPackBuf is made by the function
01076     PF_longSingleReset(). In the begin of the PF_longPackBuf it packs
01077     the size of the last sent buffer. Upon sending, the program checks,
01078     whether there was at list one re-allocation (PF_longPackN>1) .
01079     If so, the sender first packs and sends small buffer
01080     (PF_longPackSmallBuf) containing one integer number -- the
01081     _negative_ new size of the send buffer. Getting the buffer, a
01082     receiver unpacks one integer and checks whether it is <0 . If so,
01083     the receiver will repeat receiving, but first it checks whether
01084     it has enough buffer and increase it, if necessary.
01085         Initialization PF_longMultiRoot is made by the function
01086     PF_longMultiReset(). In the begin of the first chunk it packs
01087     one integer -- the number 1. Upon sending, the program checks,
01088     how many cells were packed (PF_longPackN). If more than 1, the
01089     sender packs to the next cell the integer PF_longPackN, than
01090     packs PF_longPackN pairs of integers -- the information about how many
01091     times chunk on each cell was accessed by the packing procedure,
01092     this information is contained by the nPacks field of the cell
01093     structure, and how many non-complete items was at the end of this
01094     chunk the structure field lastLen. Then the sender sends first
01095     this auxiliary chunk.
01096     The receiver unpacks the integer from obtained chunk and, if this
01097     integer is more than 1, it gets more chunks, unpacking information
01098     from the first auxiliary chunk into the corresponding nPacks
01099     fields. Unpacking information from multiple chunks, the receiver
01100     knows, when the chunk is expired and it must switch to the next cell,
01101     successively decrementing corresponding nPacks field.
01102
01103     XXX: There are still some flaws:
01104     PF_LongSingleSend/PF_LongSingleReceive may fail, for example, for data
01105     transfers from the master to many slaves. Suppose that the master sends big
01106     data to slaves, which needs an increase of the buffer of the receivers. For
01107     the first data transfer, the master sends the new buffer size as the first
01108     message, and then sends the data as the second message, because
01109     PF_LongSinglePack records the increase of the buffer size on the master. For
01110     the next time, however, the master sends the data without sending the new
01111     buffer size, and then MPI_Recv fails due to the data overflow.
01112     In parallel.c, they are used for the communication from slaves to the
01113     master. In this case, this problem does not occur because the master always
01114     has enough buffer.
01115     The maximum size that PF_LongMultiBroadcast can broadcast is limited to
01116     around 320kB because the current implementation tries to pack all
01117     information of chained buffers into one buffer, whose size is PF_packsize
01118     = 1600B.
01119
01120         #] Explanations :
01121         #[ Variables :
01122 */
01123
01124 typedef struct longMultiStruct {
01125     UBYTE *buffer; /* NULL if */
01126     int bufpos;    /* if >=0, PF_longPackBuf+bufpos is the chunk start */
01127     int packpos;   /* the current position */
01128     int nPacks;    /* How many times PF_longPack operates on this cell */
```

```
01129     int lastLen;   /* if > 0, the last packing didn't fit completely to this
01130                       chunk, only lastLen items was packed, the rest is in
01131                       the next cell. */
01132     struct longMultiStruct *next;  /* next linked cell, or NULL */
01133 } PF_LONGMULTI;
01134
01135 static UBYTE *PF_longPackBuf = NULL;
01136 static void *PF_longPackSmallBuf = NULL;
01137 static int PF_longPackPos = 0;
01138 static int PF_longPackTop = 0;
01139 static PF_LONGMULTI *PF_longMultiRoot = NULL;
01140 static PF_LONGMULTI *PF_longMultiTop = NULL;
01141 static PF_LONGMULTI *PF_longMultiLastChunk = NULL;
01142 static int PF_longPackN = 0;
01143
01144 /*
01145         #] Variables :
01146         #[ Long pack private functions :
01147         #[ PF_longMultiNewCell :
01148 */
01149
01150 static inline int PF_longMultiNewCell(void)
01151 {
01152 /*
01153         Allocate a new cell:
01154 */
01155     PF_longMultiTop->next = (PF_LONGMULTI *)
01156             Malloc1(sizeof(PF_LONGMULTI),"PF_longMultiCell");
01157     if ( PF_longMultiTop->next == NULL ) return(-1);
01158 /*
01159         Allocate a private buffer:
01160 */
01161     PF_longMultiTop->next->buffer=(UBYTE*)
01162             Malloc1(sizeof(UBYTE)*PF_packsize,"PF_longMultiChunk");
01163     if ( PF_longMultiTop->next->buffer == NULL ) return(-1);
01164 /*
01165         For the private buffer position is -1:
01166 */
01167     PF_longMultiTop->next->bufpos = -1;
01168 /*
01169         This is the last cell in the chain:
01170 */
01171     PF_longMultiTop->next->next = NULL;
01172 /*
01173         packpos and nPacks are not initialized!
01174 */
01175     return(0);
01176 }
01177
01178 /*
01179         #] PF_longMultiNewCell :
01180         #[ PF_longMultiPack2NextCell :
01181 */
01182 static inline int PF_longMultiPack2NextCell(void)
01183 {
01184 /*
01185         Is there a free cell in the chain?
01186 */
01187     if ( PF_longMultiTop->next == NULL ) {
01188 /*
01189             No, allocate the new cell with a private buffer:
01190 */
01191         if ( PF_longMultiNewCell() ) return(-1);
01192     }
01193 /*
01194         Move to the next cell in the chain:
01195 */
01196     PF_longMultiTop = PF_longMultiTop->next;
01197 /*
01198         if >=0, the cell buffer is the chunk of PF_longPackBuf, initialize it:
01199 */
01200     if ( PF_longMultiTop->bufpos > -1 )
01201         PF_longMultiTop->buffer = PF_longPackBuf+PF_longMultiTop->bufpos;
01202 /*
01203     else -- the cell has it's own private buffer.
01204     Initialize the cell fields:
01205 */
01206     PF_longMultiTop->nPacks  = 0;
01207     PF_longMultiTop->lastLen = 0;
01208     PF_longMultiTop->packpos = 0;
01209     return(0);
01210 }
01211
01212 /*
01213         #] PF_longMultiPack2NextCell :
01214         #[ PF_longMultiNewChunkAdded :
01215 */
```

```
01216
01217 static inline int PF_longMultiNewChunkAdded(int n)
01218 {
01219 /*
01220         Store the list tail:
01221 */
01222     PF_LONGMULTI *MemCell = PF_longMultiLastChunk->next;
01223     int pos = PF_longPackTop;
01224
01225     while ( n-- > 0 ) {
01226 /*
01227             Allocate a new cell:
01228 */
01229         PF_longMultiLastChunk->next = (PF_LONGMULTI *)
01230                 Malloc1(sizeof(PF_LONGMULTI),"PF_longMultiCell");
01231         if ( PF_longMultiLastChunk->next == NULL ) return(-1);
01232 /*
01233             Update the Last Chunk Pointer:
01234 */
01235         PF_longMultiLastChunk = PF_longMultiLastChunk->next;
01236 /*
01237             Initialize the new cell:
01238 */
01239         PF_longMultiLastChunk->bufpos = pos;
01240         pos += PF_packsize;
01241         PF_longMultiLastChunk->buffer = NULL;
01242         PF_longMultiLastChunk->packpos = 0;
01243         PF_longMultiLastChunk->nPacks  = 0;
01244         PF_longMultiLastChunk->lastLen = 0;
01245     }
01246 /*
01247         Hitch the tail:
01248 */
01249     PF_longMultiLastChunk->next = MemCell;
01250     return(0);
01251 }
01252
01253 /*
01254         #] PF_longMultiNewChunkAdded :
01255         #[ PF_longCopyChunk :
01256 */
01257
01258 static inline void PF_longCopyChunk(int *to, int *from, int n)
01259 {
01260     NCOPYI(to,from,n)
01261 /*  for ( ; n > 0; n-- ) *to++ = *from++; */
01262 }
01263
01264 /*
01265         #] PF_longCopyChunk :
01266         #[ PF_longAddChunk :
01267
01268     The chunk must be increased by n*PF_packsize.
01269 */
01270
01271 static int PF_longAddChunk(int n, int mustRealloc)
01272 {
01273     UBYTE *newbuf;
01274     if ( ( newbuf = (UBYTE*)Malloc1(sizeof(UBYTE)*(PF_longPackTop+n*PF_packsize),
01275                 "PF_longPackBuf") ) == NULL ) return(-1);
01276 /*
01277         Allocate and chain a new cell for longMulti:
01278 */
01279     if ( PF_longMultiNewChunkAdded(n) ) return(-1);
01280 /*
01281         Copy the content to the new buffer:
01282 */
01283     if ( mustRealloc ) {
01284         PF_longCopyChunk((int*)newbuf,(int*)PF_longPackBuf,PF_longPackTop/sizeof(int));
01285     }
01286 /*
01287         Note, PF_packsize is multiple by sizeof(int) by construction!
01288 */
01289     PF_longPackTop += (n*PF_packsize);
01290 /*
01291         Free the old buffer and store the new one:
01292 */
01293     M_free(PF_longPackBuf,"PF_longPackBuf");
01294     PF_longPackBuf = newbuf;
01295 /*
01296         Count number of re-allocs:
01297 */
01298     PF_longPackN += n;
01299     return(0);
01300 }
01301
01302 /*
```

```
01303         #] PF_longAddChunk :
01304         #[ PF_longMultiHowSplit :
01305
01306     "count" of "type" elements in an input buffer occupy "bytes" bytes.
01307     We know from the algorithm, that it is too many. How to split
01308     the buffer so that the head fits to rest of a storage buffer?*/
01309 static inline int PF_longMultiHowSplit(int count, MPI_Datatype type, int bytes)
01310 {
01311     int ret, items, totalbytes;
01312
01313     if ( count < 2 ) return(0); /* Nothing to split */
01314 /*
01315         A rest of a storage buffer:
01316 */
01317     totalbytes = PF_packsize - PF_longMultiTop->packpos;
01318 /*
01319     Rough estimate:
01320 */
01321     items = (int)((double)totalbytes*count/bytes);
01322 /*
01323         Go to the up limit:
01324 */
01325     do {
01326         if ( ( ret = MPI_Pack_size(++items,type,PF_COMM,&bytes) )
01327             !=MPI_SUCCESS ) return(ret);
01328     } while ( bytes < totalbytes );
01329 /*
01330         Now the value of "items" is too large
01331         And now evaluate the exact value:
01332 */
01333     do {
01334         if ( ( ret = MPI_Pack_size(--items,type,PF_COMM,&bytes) )
01335             !=MPI_SUCCESS ) return(ret);
01336         if ( items == 0 ) /* Nothing about MPI_Pack_size(0) == 0 in standards! */
01337             return(0);
01338     } while ( bytes > totalbytes );
01339     return(items);
01340 }
01341 /*
01342         #] PF_longMultiHowSplit :
01343         #[ PF_longPackInit :
01344 */
01345
01346 static int PF_longPackInit(void)
01347 {
01348     int ret;
01349     PF_longPackBuf = (UBYTE*)Malloc1(sizeof(UBYTE)*PF_packsize,"PF_longPackBuf");
01350     if ( PF_longPackBuf == NULL ) return(-1);
01351 /*
01352         PF_longPackTop is not initialized yet, use in as a return value:
01353 */
01354     ret = MPI_Pack_size(1,MPI_INT,PF_COMM,&PF_longPackTop);
01355     if ( ret != MPI_SUCCESS ) return(ret);
01356
01357     PF_longPackSmallBuf =
01358             (void*)Malloc1(sizeof(UBYTE)*PF_longPackTop,"PF_longPackSmallBuf");
01359
01360     PF_longPackTop = PF_packsize;
01361     PF_longMultiRoot =
01362             (PF_LONGMULTI *)Malloc1(sizeof(PF_LONGMULTI),"PF_longMultiRoot");
01363     if ( PF_longMultiRoot == NULL ) return(-1);
01364     PF_longMultiRoot->bufpos = 0;
01365     PF_longMultiRoot->buffer = NULL;
01366     PF_longMultiRoot->next   = NULL;
01367     PF_longMultiLastChunk    = PF_longMultiRoot;
01368
01369     PF_longPackPos = 0;
01370     PF_longMultiRoot->packpos = 0;
01371     PF_longMultiTop = PF_longMultiRoot;
01372     PF_longPackN = 1;
01373     return(0);
01374 }
01375
01376 /*
01377         #] PF_longPackInit :
01378         #[ PF_longMultiPreparePrefix :
01379 */
01380
01381 static inline int PF_longMultiPreparePrefix(void)
01382 {
01383     int ret;
01384     PF_LONGMULTI *thePrefix;
01385     int i = PF_longPackN;
01386 /*
01387         Here we have PF_longPackN>1!
01388         New cell (at the list end) to create the auxiliary chunk:
01389 */
```

```
01390     if ( PF_longMultiPack2NextCell() ) return(-1);
01391 /*
01392         Store the pointer to the chunk we will proceed:
01393 */
01394     thePrefix = PF_longMultiTop;
01395 /*
01396         Pack PF_longPackN:
01397 */
01398     ret = MPI_Pack(&(PF_longPackN),
01399                    1,
01400                    MPI_INT,
01401                    thePrefix->buffer,
01402                    PF_packsize,
01403                    &(thePrefix->packpos),
01404                    PF_COMM);
01405     if ( ret != MPI_SUCCESS ) return(ret);
01406 /*
01407         And start from the beginning:
01408 */
01409     for ( PF_longMultiTop = PF_longMultiRoot; i > 0; i-- ) {
01410 /*
01411             Pack number of Pack hits:
01412 */
01413         ret = MPI_Pack(&(PF_longMultiTop->nPacks),
01414                        1,
01415                        MPI_INT,
01416                        thePrefix->buffer,
01417                        PF_packsize,
01418                        &(thePrefix->packpos),
01419                        PF_COMM);
01420 /*
01421                 Pack the length of the last fit portion:
01422 */
01423         ret |= MPI_Pack(&(PF_longMultiTop->lastLen),
01424                         1,
01425                         MPI_INT,
01426                         thePrefix->buffer,
01427                         PF_packsize,
01428                         &(thePrefix->packpos),
01429                         PF_COMM);
01430 /*
01431                 Check the size -- not necessary, MPI_Pack did it.
01432 */
01433         if (  ret != MPI_SUCCESS ) return(ret);
01434 /*
01435             Go to the next cell:
01436 */
01437         PF_longMultiTop = PF_longMultiTop->next;
01438     }
01439
01440     PF_longMultiTop = thePrefix;
01441 /*
01442         PF_longMultiTop is ready!
01443 */
01444     return(0);
01445 }
01446
01447 /*
01448         #] PF_longMultiPreparePrefix :
01449         #[ PF_longMultiProcessPrefix :
01450 */
01451
01452 static inline int PF_longMultiProcessPrefix(void)
01453 {
01454     int ret,i;
01455 /*
01456         We have PF_longPackN records packed in PF_longMultiRoot->buffer,
01457         pairs nPacks and lastLen. Loop through PF_longPackN cells,
01458         unpacking these integers into proper fields:
01459 */
01460     for ( PF_longMultiTop = PF_longMultiRoot, i = 0; i < PF_longPackN; i++ ) {
01461 /*
01462             Go to th next cell, allocating, when necessary:
01463 */
01464         if ( PF_longMultiPack2NextCell() ) return(-1);
01465 /*
01466             Unpack the number of Pack hits:
01467 */
01468         ret = MPI_Unpack(PF_longMultiRoot->buffer,
01469                          PF_packsize,
01470                          &(  PF_longMultiRoot->packpos),
01471                          &(PF_longMultiTop->nPacks),
01472                          1,
01473                          MPI_INT,
01474                          PF_COMM);
01475         if ( ret != MPI_SUCCESS ) return(ret);
01476 /*
```

```
01477             Unpack the length of the last fit portion:
01478 */
01479         ret = MPI_Unpack(PF_longMultiRoot->buffer,
01480                          PF_packsize,
01481                          &(  PF_longMultiRoot->packpos),
01482                          &(PF_longMultiTop->lastLen),
01483                          1,
01484                          MPI_INT,
01485                          PF_COMM);
01486         if ( ret != MPI_SUCCESS ) return(ret);
01487     }
01488     return(0);
01489 }
01490
01491 /*
01492         #] PF_longMultiProcessPrefix :
01493         #[ PF_longSingleReset :
01494 */
01495
01503 static inline int PF_longSingleReset(int is_sender)
01504 {
01505     int ret;
01506     PF_longPackPos=0;
01507     if ( is_sender ) {
01508         ret = MPI_Pack(&PF_longPackTop,1,MPI_INT,
01509             PF_longPackBuf,PF_longPackTop,&PF_longPackPos,PF_COMM);
01510         if ( ret != MPI_SUCCESS ) return(ret);
01511         PF_longPackN = 1;
01512     }
01513     else {
01514         PF_longPackN=0;
01515     }
01516     return(0);
01517 }
01518
01519 /*
01520         #] PF_longSingleReset :
01521         #[ PF_longMultiReset :
01522 */
01523
01531 static inline int PF_longMultiReset(int is_sender)
01532 {
01533     int ret = 0, theone = 1;
01534     PF_longMultiRoot->packpos = 0;
01535     if ( is_sender ) {
01536         ret = MPI_Pack(&theone,1,MPI_INT,
01537             PF_longPackBuf,PF_longPackTop,&(PF_longMultiRoot->packpos),PF_COMM);
01538         PF_longPackN = 1;
01539     }
01540     else {
01541         PF_longPackN = 0;
01542     }
01543     PF_longMultiRoot->nPacks = 0;   /* The auxiliary field is not counted */
01544     PF_longMultiRoot->lastLen = 0;
01545     PF_longMultiTop = PF_longMultiRoot;
01546     PF_longMultiRoot->buffer = PF_longPackBuf;
01547     return ret;
01548 }
01549
01550 /*
01551         #] PF_longMultiReset :
01552         #] Long pack private functions :
01553         #[ PF_PrepareLongSinglePack :
01554 */
01555
01561 int PF_PrepareLongSinglePack(void)
01562 {
01563     return PF_longSingleReset(1);
01564 }
01565
01566 /*
01567         #] PF_PrepareLongSinglePack :
01568         #[ PF_LongSinglePack :
01569 */
01570
01579 int PF_LongSinglePack(const void *buffer, size_t count, MPI_Datatype type)
01580 {
01581     int ret, bytes;
01582     /* XXX: Limited by int size. */
01583     if ( count > INT_MAX ) return -99;
01584     ret = MPI_Pack_size((int)count,type,PF_COMM,&bytes);
01585     if ( ret != MPI_SUCCESS ) return(ret);
01586
01587     while ( PF_longPackPos+bytes > PF_longPackTop ) {
01588         if ( PF_longAddChunk(1, 1) ) return(-1);
01589     }
01590 /*
```

```
01591         PF_longAddChunk(1, 1) means, the chunk must
01592         be increased by 1 and re-allocated
01593 */
01594     ret = MPI_Pack((void *)buffer,(int)count,type,
01595                    PF_longPackBuf,PF_longPackTop,&PF_longPackPos,PF_COMM);
01596     if ( ret != MPI_SUCCESS ) return(ret);
01597     return(0);
01598 }
01599
01600 /*
01601         #] PF_LongSinglePack :
01602         #[ PF_LongSingleUnpack :
01603 */
01604
01613 int PF_LongSingleUnpack(void *buffer, size_t count, MPI_Datatype type)
01614 {
01615     int ret;
01616     /* XXX: Limited by int size. */
01617     if ( count > INT_MAX ) return -99;
01618     ret = MPI_Unpack(PF_longPackBuf,PF_longPackTop,&PF_longPackPos,
01619                      buffer,(int)count,type,PF_COMM);
01620     if ( ret != MPI_SUCCESS ) return(ret);
01621     return(0);
01622 }
01623
01624 /*
01625         #] PF_LongSingleUnpack :
01626         #[ PF_LongSingleSend :
01627 */
01628
01650 int PF_LongSingleSend(int to, int tag)
01651 {
01652     int ret, pos = 0;
01653 /*
01654         Note, here we assume that this function couldn't be used
01655         with to == PF_ANY_SOURCE!
01656 */
01657     if ( PF_longPackN > 1 ) {
01658         /* The buffer was incremented, pack send the new size first: */
01659         int tmp = -PF_longPackTop;
01660 /*
01661             Negative value means there will be the second buffer
01662 */
01663         ret = MPI_Pack(&tmp, 1,PF_INT,
01664                        PF_longPackSmallBuf,PF_longPackTop,&pos,PF_COMM);
01665         if ( ret != MPI_SUCCESS ) return(ret);
01666         ret = MPI_Ssend(PF_longPackSmallBuf,pos,MPI_PACKED,to,tag,PF_COMM);
01667         if ( ret != MPI_SUCCESS ) return(ret);
01668     }
01669     ret = MPI_Ssend(PF_longPackBuf,PF_longPackPos,MPI_PACKED,to,tag,PF_COMM);
01670     if ( ret != MPI_SUCCESS ) return(ret);
01671     return(0);
01672 }
01673
01674 /*
01675         #] PF_LongSingleSend :
01676         #[ PF_LongSingleReceive :
01677 */
01678
01693 int PF_LongSingleReceive(int src, int tag, int *psrc, int *ptag)
01694 {
01695     int ret, missed, oncemore;
01696     MPI_Status status;
01697     PF_longSingleReset(0);
01698     do {
01699         ret = MPI_Recv(PF_longPackBuf,PF_longPackTop,MPI_PACKED,src,tag,
01700                        PF_COMM,&status);
01701         if ( ret != MPI_SUCCESS ) return(ret);
01702 /*
01703             The source and tag must be specified here for the case if
01704             MPI_Recv is performed more than once:
01705 */
01706         src = status.MPI_SOURCE;
01707         tag = status.MPI_TAG;
01708         if ( psrc ) *psrc = status.MPI_SOURCE;
01709         if ( ptag ) *ptag = status.MPI_TAG;
01710 /*
01711             Now we got either small buffer with the new PF_longPackTop,
01712             or just a regular chunk.
01713 */
01714         ret = MPI_Unpack(PF_longPackBuf,PF_longPackTop,&PF_longPackPos,
01715                          &missed,1,MPI_INT,PF_COMM);
01716         if ( ret != MPI_SUCCESS ) return(ret);
01717
01718         if ( missed < 0 ) { /* The small buffer was received. */
01719             oncemore = 1; /* repeat receiving afterwards */
01720                           /* Reallocate the buffer and get the data */
```

```
01721             missed = -missed;
01722 /*
01723                 restore after unpacking small from buffer:
01724 */
01725             PF_longPackPos = 0;
01726         }
01727         else {
01728             oncemore = 0;  /* That's all, no repetition */
01729         }
01730         if ( missed > PF_longPackTop ) {
01731             /*
01732              * The room must be increased. We need a re-allocation for the
01733              * case that there is no repetition.
01734              */
01735             if ( PF_longAddChunk( (missed-PF_longPackTop)/PF_packsize, !oncemore ) )
01736                 return(-1);
01737         }
01738     } while ( oncemore );
01739     return(0);
01740 }
01741
01742 /*
01743         #] PF_LongSingleReceive :
01744         #[ PF_PrepareLongMultiPack :
01745 */
01746
01752 int PF_PrepareLongMultiPack(void)
01753 {
01754     return PF_longMultiReset(1);
01755 }
01756
01757 /*
01758         #] PF_PrepareLongMultiPack :
01759         #[ PF_LongMultiPackImpl :
01760 */
01761
01771 int PF_LongMultiPackImpl(const void*buffer, size_t count, size_t eSize, MPI_Datatype type)
01772 {
01773     int ret, items;
01774
01775     /* XXX: Limited by int size. */
01776     if ( count > INT_MAX ) return -99;
01777
01778     ret = MPI_Pack_size((int)count,type,PF_COMM,&items);
01779     if ( ret != MPI_SUCCESS ) return(ret);
01780
01781     if ( PF_longMultiTop->packpos + items <= PF_packsize ) {
01782         ret = MPI_Pack((void *)buffer,(int)count,type,PF_longMultiTop->buffer,
01783                        PF_packsize,&(PF_longMultiTop->packpos),PF_COMM);
01784         if ( ret != MPI_SUCCESS ) return(ret);
01785         PF_longMultiTop->nPacks++;
01786         return(0);
01787     }
01788 /*
01789         The data do not fit to the rest of the buffer.
01790         There are two possibilities here: go to the next cell
01791         immediately, or first try to pack some portion. The function
01792         PF_longMultiHowSplit() returns the number of items could be
01793         packed in the end of the current cell:
01794 */
01795     if ( ( items = PF_longMultiHowSplit((int)count,type,items) ) < 0 ) return(items);
01796
01797     if ( items > 0 ) {   /* store the head */
01798         ret = MPI_Pack((void *)buffer,items,type,PF_longMultiTop->buffer,
01799                        PF_packsize,&(PF_longMultiTop->packpos),PF_COMM);
01800         if ( ret != MPI_SUCCESS ) return(ret);
01801         PF_longMultiTop->nPacks++;
01802         PF_longMultiTop->lastLen = items;
01803     }
01804 /*
01805         Now the rest should be packed to the new cell.
01806         Slide to the new cell:
01807 */
01808     if ( PF_longMultiPack2NextCell() ) return(-1);
01809     PF_longPackN++;
01810 /*
01811         Pack the rest to the next cell:
01812 */
01813     return(PF_LongMultiPackImpl((char *)buffer+items*eSize,count-items,eSize,type));
01814 }
01815
01816 /*
01817         #] PF_LongMultiPackImpl :
01818         #[ PF_LongMultiUnpackImpl :
01819 */
01820
01830 int PF_LongMultiUnpackImpl(void *buffer, size_t count, size_t eSize, MPI_Datatype type)
```

```
01831 {
01832     int ret;
01833
01834     /* XXX: Limited by int size. */
01835     if ( count > INT_MAX ) return -99;
01836
01837     if ( PF_longPackN < 2 ) { /* Just unpack the buffer from the single cell */
01838         ret = MPI_Unpack(
01839                     PF_longMultiTop->buffer,
01840                     PF_packsize,
01841                     &(PF_longMultiTop->packpos),
01842                     buffer,
01843                     count,type,PF_COMM);
01844         if ( ret != MPI_SUCCESS ) return(ret);
01845         return(0);
01846     }
01847 /*
01848         More than one cell is in use.
01849 */
01850     if ( ( PF_longMultiTop->nPacks > 1 )     /* the cell is not expired */
01851         ||          /* The last cell contains exactly required portion: */
01852         ( ( PF_longMultiTop->nPacks == 1 ) && ( PF_longMultiTop->lastLen == 0 ) )
01853     ) {    /* Just unpack the buffer from the current cell */
01854         ret = MPI_Unpack(
01855                     PF_longMultiTop->buffer,
01856                     PF_packsize,
01857                     &(PF_longMultiTop->packpos),
01858                     buffer,
01859                     count,type,PF_COMM);
01860         if ( ret != MPI_SUCCESS ) return(ret);
01861         (PF_longMultiTop->nPacks)--;
01862         return(0);
01863     }
01864     if ( ( PF_longMultiTop->nPacks == 1 ) && ( PF_longMultiTop->lastLen != 0 ) ) {
01865 /*
01866             Unpack the head:
01867 */
01868         ret = MPI_Unpack(
01869                     PF_longMultiTop->buffer,
01870                     PF_packsize,
01871                     &(PF_longMultiTop->packpos),
01872                     buffer,
01873                     PF_longMultiTop->lastLen,type,PF_COMM);
01874         if ( ret != MPI_SUCCESS ) return(ret);
01875 /*
01876             Decrement the counter by read items:
01877 */
01878         count -= PF_longMultiTop->lastLen;
01879         if ( count <= 0 ) return(-1);  /*Something is wrong! */
01880 /*
01881             Shift the output buffer position:
01882 */
01883         buffer = (char *)buffer + PF_longMultiTop->lastLen * eSize;
01884         (PF_longMultiTop->nPacks)--;
01885     }
01886 /*
01887         Here PF_longMultiTop->nPacks == 0
01888 */
01889     if ( ( PF_longMultiTop = PF_longMultiTop->next ) == NULL ) return(-1);
01890     return(PF_LongMultiUnpackImpl(buffer,count,eSize,type));
01891 }
01892
01893 /*
01894         #] PF_LongMultiUnpackImpl :
01895         #[ PF_LongMultiBroadcast :
01896 */
01897
01916 int PF_LongMultiBroadcast(void)
01917 {
01918     int ret, i;
01919
01920     if ( PF.me == MASTER ) {
01921 /*
01922             PF_longPackN is the number of packed chunks. If it is more
01923             than 1, we have to pack a new one and send it first
01924 */
01925         if ( PF_longPackN > 1 ) {
01926             if ( PF_longMultiPreparePrefix() ) return(-1);
01927             ret = MPI_Bcast((void*)PF_longMultiTop->buffer,
01928                             PF_packsize,MPI_PACKED,MASTER,PF_COMM);
01929             if ( ret != MPI_SUCCESS ) return(ret);
01930 /*
01931                 PF_longPackN was not incremented by PF_longMultiPreparePrefix()!
01932 */
01933         }
01934 /*
01935             Now we start from the beginning:
```

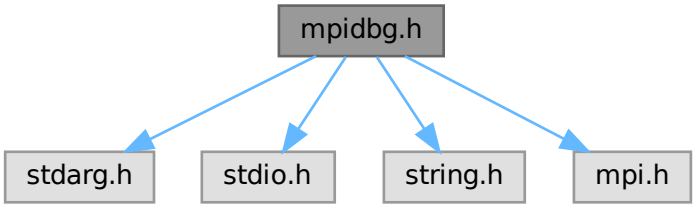
```
01936 */
01937         PF_longMultiTop = PF_longMultiRoot;
01938 /*
01939             Just broadcast all the chunks:
01940 */
01941         for ( i = 0; i < PF_longPackN; i++ ) {
01942             ret = MPI_Bcast((void*)PF_longMultiTop->buffer,
01943                             PF_packsize,MPI_PACKED,MASTER,PF_COMM);
01944             if ( ret != MPI_SUCCESS ) return(ret);
01945             PF_longMultiTop = PF_longMultiTop->next;
01946         }
01947         return(0);
01948     }
01949 /*
01950         else - the slave
01951 */
01952     PF_longMultiReset(0);
01953 /*
01954         Get the first chunk; it can be either the only data chunk, or
01955         an auxiliary chunk, if the data do not fit the single chunk:
01956 */
01957     ret = MPI_Bcast((void*)PF_longMultiRoot->buffer,
01958                     PF_packsize,MPI_PACKED,MASTER,PF_COMM);
01959     if ( ret != MPI_SUCCESS ) return(ret);
01960
01961     ret = MPI_Unpack((void*)PF_longMultiRoot->buffer,
01962                      PF_packsize,
01963                      &(PF_longMultiRoot->packpos),
01964                      &PF_longPackN,1,MPI_INT,PF_COMM);
01965     if ( ret != MPI_SUCCESS ) return(ret);
01966 /*
01967         Now in PF_longPackN we have the number of cells used
01968         for broadcasting. If it is >1, then we have to allocate
01969         enough cells, initialize them and receive all the chunks.
01970 */
01971     if ( PF_longPackN < 2 ) /* That's all, the single chunk is received. */
01972         return(0);
01973 /*
01974         Here we have to get PF_longPackN chunks. But, first,
01975         initialize cells by info from the received auxiliary chunk.
01976 */
01977     if ( PF_longMultiProcessPrefix() ) return(-1);
01978 /*
01979         Now we have free PF_longPackN cells, starting
01980         from PF_longMultiRoot->next,  with properly initialized
01981         nPacks and lastLen fields. Get chunks:
01982 */
01983     for ( PF_longMultiTop = PF_longMultiRoot->next, i = 0; i < PF_longPackN; i++ ) {
01984         ret = MPI_Bcast((void*)PF_longMultiTop->buffer,
01985                         PF_packsize,MPI_PACKED,MASTER,PF_COMM);
01986         if ( ret != MPI_SUCCESS ) return(ret);
01987         if ( i == 0 ) {   /* The first chunk, it contains extra "1". */
01988             int tmp;
01989 /*
01990                 Extract this 1 into tmp and forget about it.
01991 */
01992             ret = MPI_Unpack((void*)PF_longMultiTop->buffer,
01993                              PF_packsize,
01994                              &(PF_longMultiTop->packpos),
01995                              &tmp,1,MPI_INT,PF_COMM);
01996             if ( ret != MPI_SUCCESS ) return(ret);
01997         }
01998         PF_longMultiTop = PF_longMultiTop->next;
01999     }
02000 /*
02001         multiUnPack starts with PF_longMultiTop, skip auxiliary chunk in
02002         PF_longMultiRoot:
02003 */
02004     PF_longMultiTop = PF_longMultiRoot->next;
02005     return(0);
02006 }
02007
02008 /*
02009         #] PF_LongMultiBroadcast :
02010     #] Long pack stuff :
02011 */
```

4.83 mpidbg.h File Reference

```
#include <stdarg.h>
#include <stdio.h>
```

```
#include <string.h>
#include <mpi.h>
```

Include dependency graph for mpidbg.h:



Macros

- #define MPIDBG_LINESIZE 1024
- #define MPIDBG_BUFSIZE 128
- #define MPIDBG_RANK MPIDBG_Get_rank()
- #define MPIDBG_Out(...) MPIDBG_Out(file, line, func, __VA_ARGS__)
- #define MPIDBG_EXTARG const char *file, int line, const char *func
- #define MPI_Init(...) MPIDBG_Init(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Finalize() MPIDBG_Finalize(__FILE__, __LINE__, __func__)
- #define MPI_Send(...) MPIDBG_Send(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Recv(...) MPIDBG_Recv(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Bsend(...) MPIDBG_Bsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Ssend(...) MPIDBG_Ssend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Rsend(...) MPIDBG_Rsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Isend(...) MPIDBG_Isend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Ibsend(...) MPIDBG_Ibsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Issend(...) MPIDBG_Issend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Irsend(...) MPIDBG_Irsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Irecv(...) MPIDBG_Irecv(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Wait(...) MPIDBG_Wait(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Test(...) MPIDBG_Test(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Waitany(...) MPIDBG_Waitany(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Testany(...) MPIDBG_Testany(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Waitall(...) MPIDBG_Waitall(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Testall(...) MPIDBG_Testall(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Waitsome(...) MPIDBG_Waitsome(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Testsome(...) MPIDBG_Testsome(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Iprobe(...) MPIDBG_Iprobe(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Probe(...) MPIDBG_Probe(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Cancel(...) MPIDBG_Cancel(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Test_cancelled(...) MPIDBG_Test_cancelled(__VA_ARGS__, __FILE__, __LINE__, __func_↩ _)
- #define MPI_Barrier(...) MPIDBG_Barrier(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Bcast(...) MPIDBG_Bcast(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Reduce(...) MPIDBG_Reduce(__VA_ARGS__, __FILE__, __LINE__, __func__)

4.83.1 Detailed Description

MPI APIs with the logging feature. NOTE: This file needs C99.

Definition in file mpidbg.h.

4.83.2 Macro Definition Documentation

4.83.2.1 MPIDBG_LINESIZE

```
#define MPIDBG_LINESIZE 1024
```

Definition at line 49 of file mpidbg.h.

4.83.2.2 MPIDBG_BUFSIZE

```
#define MPIDBG_BUFSIZE 128
```

Definition at line 50 of file mpidbg.h.

4.83.2.3 MPIDBG_RANK

```
#define MPIDBG_RANK MPIDBG_Get_rank()
```

Definition at line 61 of file mpidbg.h.

4.83.2.4 MPIDBG_Out

```
#define MPIDBG_Out(
              ...  ) MPIDBG_Out(file, line, func, __VA_ARGS__)
```

Definition at line 78 of file mpidbg.h.

4.83.2.5 MPIDBG_EXTARG

```
#define MPIDBG_EXTARG const char *file, int line, const char *func
```

Definition at line 157 of file mpidbg.h.

4.83.2.6 MPI_Init

```
#define MPI_Init(
              ...  ) MPIDBG_Init(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 174 of file mpidbg.h.

4.83.2.7 MPI_Finalize

```
#define MPI_Finalize( ) MPIDBG_Finalize(__FILE__, __LINE__, __func__)
```

Definition at line 193 of file mpidbg.h.

4.83.2.8 MPI_Send

```
#define MPI_Send(
              ... ) MPIDBG_Send(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 212 of file mpidbg.h.

4.83.2.9 MPI_Recv

```
#define MPI_Recv(
              ... ) MPIDBG_Recv(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 235 of file mpidbg.h.

4.83.2.10 MPI_Bsend

```
#define MPI_Bsend(
              ... ) MPIDBG_Bsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 254 of file mpidbg.h.

4.83.2.11 MPI_Ssend

```
#define MPI_Ssend(
              ... ) MPIDBG_Ssend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 273 of file mpidbg.h.

4.83.2.12 MPI_Rsend

```
#define MPI_Rsend(
              ... ) MPIDBG_Rsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 292 of file mpidbg.h.

4.83.2.13 MPI_Isend

```
#define MPI_Isend(
              ... ) MPIDBG_Isend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 311 of file mpidbg.h.

4.83.2.14 MPI_Ibsend

```
#define MPI_Ibsend(
              ...  ) MPIDBG_Ibsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 330 of file mpidbg.h.

4.83.2.15 MPI_Issend

```
#define MPI_Issend(
              ...  ) MPIDBG_Issend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 349 of file mpidbg.h.

4.83.2.16 MPI_Irsend

```
#define MPI_Irsend(
              ...  ) MPIDBG_Irsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 368 of file mpidbg.h.

4.83.2.17 MPI_Irecv

```
#define MPI_Irecv(
              ...  ) MPIDBG_Irecv(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 387 of file mpidbg.h.

4.83.2.18 MPI_Wait

```
#define MPI_Wait(
              ...  ) MPIDBG_Wait(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 412 of file mpidbg.h.

4.83.2.19 MPI_Test

```
#define MPI_Test(
              ...  ) MPIDBG_Test(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 442 of file mpidbg.h.

4.83.2.20 MPI_Waitany

```
#define MPI_Waitany(
              ...  ) MPIDBG_Waitany(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 470 of file mpidbg.h.

4.83.2.21 MPI_Testany

```
#define MPI_Testany(
              ... ) MPIDBG_Testany(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 503 of file mpidbg.h.

4.83.2.22 MPI_Waitall

```
#define MPI_Waitall(
              ... ) MPIDBG_Waitall(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 529 of file mpidbg.h.

4.83.2.23 MPI_Testall

```
#define MPI_Testall(
              ... ) MPIDBG_Testall(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 560 of file mpidbg.h.

4.83.2.24 MPI_Waitsome

```
#define MPI_Waitsome(
              ... ) MPIDBG_Waitsome(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 586 of file mpidbg.h.

4.83.2.25 MPI_Testsome

```
#define MPI_Testsome(
              ... ) MPIDBG_Testsome(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 612 of file mpidbg.h.

4.83.2.26 MPI_Iprobe

```
#define MPI_Iprobe(
              ... ) MPIDBG_Iprobe(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 641 of file mpidbg.h.

4.83.2.27 MPI_Probe

```
#define MPI_Probe(
              ... ) MPIDBG_Probe(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 664 of file mpidbg.h.

4.83.2.28 MPI_Cancel

```
#define MPI_Cancel(
              ... ) MPIDBG_Cancel(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 683 of file mpidbg.h.

4.83.2.29 MPI_Test_cancelled

```
#define MPI_Test_cancelled(
              ... ) MPIDBG_Test_cancelled(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 707 of file mpidbg.h.

4.83.2.30 MPI_Barrier

```
#define MPI_Barrier(
              ... ) MPIDBG_Barrier(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 726 of file mpidbg.h.

4.83.2.31 MPI_Bcast

```
#define MPI_Bcast(
              ... ) MPIDBG_Bcast(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 745 of file mpidbg.h.

4.83.2.32 MPI_Reduce

```
#define MPI_Reduce(
              ... ) MPIDBG_Reduce(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 764 of file mpidbg.h.

4.84 mpidbg.h

Go to the documentation of this file.

```
00001 #ifndef MPIDBG_H
00002 #define MPIDBG_H
00003
00010 /* #[ License : */
00011 /*
00012  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00013  *   When using this file you are requested to refer to the publication
00014  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00015  *   This is considered a matter of courtesy as the development was paid
00016  *   for by FOM the Dutch physics granting agency and we would like to
00017  *   be able to track its scientific use to convince FOM of its value
00018  *   for the community.
00019  *
00020  *   This file is part of FORM.
00021  *
00022  *   FORM is free software: you can redistribute it and/or modify it under the
00023  *   terms of the GNU General Public License as published by the Free Software
00024  *   Foundation, either version 3 of the License, or (at your option) any later
00025  *   version.
00026  *
00027  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00028  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00029  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00030  *   details.
00031  *
00032  *   You should have received a copy of the GNU General Public License along
00033  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00034  */
00035 /* #] License : */
00036
00037 /*
00038     #[ Includes :
00039 */
00040
00041 #include <stdarg.h>
00042 #include <stdio.h>
00043 #include <string.h>
00044 #include <mpi.h>
00045 #if defined(MPIDEBUGGING_DELAY_US) && MPIDEBUGGING_DELAY_US > 0
00046 #include <unistd.h>  // for usleep()
00047 #endif
00048
00049 #define MPIDBG_LINESIZE 1024
00050 #define MPIDBG_BUFSIZE 128
00051
00052 /*
00053     #] Includes :
00054     #[ Utilities :
00055         #[ MPIDBG_RANK :
00056 */
00057
00058 static inline int MPIDBG_Get_rank(void) {
00059     return PF.me;  /* Assume we are working with ParFORM. */
00060 }
00061 #define MPIDBG_RANK MPIDBG_Get_rank()
00062
00063 /*
00064         #] MPIDBG_RANK :
00065         #[ MPIDBG_Out :
00066 */
00067
00068 static inline void MPIDBG_Out(const char *file, int line, const char *func, const char *fmt, ...) {
00069     char buf[MPIDBG_LINESIZE];
00070     va_list ap;
00071     va_start(ap, fmt);
00072     snprintf(buf,MPIDBG_LINESIZE, "*** [%d] %10s %4d @ %-16s: ", MPIDBG_RANK, file, line, func);
00073     vsnprintf(buf + strlen(buf),MPIDBG_LINESIZE-strlen(buf), fmt, ap);
00074     va_end(ap);
00075     /* Assume fprintf with a line will work well even in multi-processes. */
00076     fprintf(stderr, "%s\n", buf);
00077 }
00078 #define MPIDBG_Out(...) MPIDBG_Out(file, line, func, __VA_ARGS__)
00079
00080 /*
00081         #] MPIDBG_Out :
00082         #[ MPIDBG_insert_delay :
00083 */
00084
00085 static inline void MPIDBG_insert_delay(void)
00086 {
00087 #if defined(MPIDEBUGGING_DELAY_US) && MPIDEBUGGING_DELAY_US > 0
00088     usleep(MPIDEBUGGING_DELAY_US);
```

```
00089 #endif
00090 }
00091
00092 /*
00093         #] MPIDBG_insert_delay :
00094         #[ MPIDBG_sprint_requests :
00095 */
00096
00097 static inline void MPIDBG_sprint_requests(char *buf, int count, const MPI_Request *requests)
00098 {
00099     /* Assume sprintf never fail and returns >= 0... */
00100     buf += sprintf(buf, "(");
00101     int i, first = 1;
00102     for ( i = 0; i < count; i++ ) {
00103         if ( first ) {
00104             first = 0;
00105         }
00106         else {
00107             buf += sprintf(buf, ",");
00108         }
00109         if ( requests[i] != MPI_REQUEST_NULL ) {
00110             buf += sprintf(buf, "%d", i);
00111         }
00112         else {
00113             buf += sprintf(buf, "*");
00114         }
00115     }
00116     buf += sprintf(buf, ")");
00117 }
00118
00119 /*
00120         #] MPIDBG_sprint_requests :
00121         #[ MPIDBG_sprint_statuses :
00122 */
00123
00124 static inline void MPIDBG_sprint_statuses(char *buf, int count, const MPI_Request *old_requests, const
      MPI_Request *new_requests, const MPI_Status *statuses)
00125 {
00126     /* Assume sprintf never fail and returns >= 0... */
00127     buf += sprintf(buf, "(");
00128     int i, first = 1;
00129     for ( i = 0; i < count; i++ ) {
00130         if ( first ) {
00131             first = 0;
00132         }
00133         else {
00134             buf += sprintf(buf, ",");
00135         }
00136         if ( old_requests[i] != MPI_REQUEST_NULL && new_requests[i] == MPI_REQUEST_NULL ) {
00137             int ret_size = 0;
00138             MPI_Get_count((MPI_Status *)&statuses[i], MPI_BYTE, &ret_size);
00139             buf += sprintf(buf, "(source=%d,tag=%d,size=%d)", statuses[i].MPI_SOURCE,
      statuses[i].MPI_TAG, ret_size);
00140         } else {
00141             buf += sprintf(buf, "*");
00142         }
00143     }
00144     buf += sprintf(buf, ")");
00145 }
00146
00147 /*
00148         #] MPIDBG_sprint_statuses :
00149     #] Utilities :
00150     #[ MPI APIs :
00151 */
00152
00153 /*
00154  * The followings are the MPI APIs with the logging.
00155  */
00156
00157 #define MPIDBG_EXTARG const char *file, int line, const char *func
00158
00159 /*
00160         #[ MPI_Init :
00161 */
00162
00163 static inline int MPIDBG_Init(int* argc, char*** argv, MPIDBG_EXTARG)
00164 {
00165     int ret = MPI_Init(argc, argv);
00166     if ( ret == MPI_SUCCESS ) {
00167         MPIDBG_Out("MPI_Init: OK");
00168     }
00169     else {
00170         MPIDBG_Out("MPI_Init: Failed");
00171     }
00172     return ret;
00173 }
```

```
00174 #define MPI_Init(...) MPIDBG_Init(__VA_ARGS__, __FILE__, __LINE__, __func__)
00175
00176 /*
00177         #] MPI_Init :
00178         #[ MPI_Finalize :
00179 */
00180
00181 static inline int MPIDBG_Finalize(MPIDBG_EXTARG)
00182 {
00183     MPIDBG_Out("MPI_Finalize");
00184     int ret = MPI_Finalize();
00185     if ( ret == MPI_SUCCESS ) {
00186         MPIDBG_Out("MPI_Finalize: OK");
00187     }
00188     else {
00189         MPIDBG_Out("MPI_Finalize: Failed");
00190     }
00191     return ret;
00192 }
00193 #define MPI_Finalize() MPIDBG_Finalize(__FILE__, __LINE__, __func__)
00194
00195 /*
00196         #] MPI_Finalize :
00197         #[ MPI_Send :
00198 */
00199
00200 static inline int MPIDBG_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm
      comm, MPIDBG_EXTARG)
00201 {
00202     MPIDBG_Out("MPI_Send: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00203     int ret = MPI_Send(buf, count, datatype, dest, tag, comm);
00204     if ( ret == MPI_SUCCESS ) {
00205         MPIDBG_Out("MPI_Send: OK src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00206     }
00207     else {
00208         MPIDBG_Out("MPI_Send: Failed src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00209     }
00210     return ret;
00211 }
00212 #define MPI_Send(...) MPIDBG_Send(__VA_ARGS__, __FILE__, __LINE__, __func__)
00213
00214 /*
00215         #] MPI_Send :
00216         #[ MPI_Recv :
00217 */
00218
00219 static inline int MPIDBG_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag,
      MPI_Comm comm, MPI_Status *status, MPIDBG_EXTARG)
00220 {
00221     MPI_Status st;
00222     if ( status == MPI_STATUS_IGNORE ) status = &st;
00223     MPIDBG_Out("MPI_Recv: src=%d dest=%d tag=%d", source, MPIDBG_RANK, tag);
00224     int ret = MPI_Recv(buf, count, datatype, source, tag, comm, status);
00225     if ( ret == MPI_SUCCESS ) {
00226         int ret_count = 0;
00227         MPI_Get_count(status, datatype, &ret_count);
00228         MPIDBG_Out("MPI_Recv: OK src=%d dest=%d tag=%d count=%d", status->MPI_SOURCE, MPIDBG_RANK,
      status->MPI_TAG, ret_count);
00229     }
00230     else {
00231         MPIDBG_Out("MPI_Recv: Failed src=%d dest=%d tag=%d", source, MPIDBG_RANK, tag);
00232     }
00233     return ret;
00234 }
00235 #define MPI_Recv(...) MPIDBG_Recv(__VA_ARGS__, __FILE__, __LINE__, __func__)
00236
00237 /*
00238         #] MPI_Recv :
00239         #[ MPI_Bsend :
00240 */
00241
00242 static inline int MPIDBG_Bsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
      MPI_Comm comm, MPIDBG_EXTARG)
00243 {
00244     MPIDBG_Out("MPI_Bsend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00245     int ret = MPI_Bsend(buf, count, datatype, dest, tag, comm);
00246     if ( ret == MPI_SUCCESS ) {
00247         MPIDBG_Out("MPI_Bsend: OK src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00248     }
00249     else {
00250         MPIDBG_Out("MPI_Bsend: Failed src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00251     }
00252     return ret;
00253 }
00254 #define MPI_Bsend(...) MPIDBG_Bsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
00255
00256 /*
```

```
00257         #] MPI_Bsend :
00258         #[ MPI_Ssend :
00259 */
00260
00261 static inline int MPIDBG_Ssend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
      MPI_Comm comm, MPIDBG_EXTARG)
00262 {
00263     MPIDBG_Out("MPI_Ssend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00264     int ret = MPI_Ssend(buf, count, datatype, dest, tag, comm);
00265     if ( ret == MPI_SUCCESS ) {
00266         MPIDBG_Out("MPI_Ssend: OK src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00267     }
00268     else {
00269         MPIDBG_Out("MPI_Ssend: Failed src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00270     }
00271     return ret;
00272 }
00273 #define MPI_Ssend(...) MPIDBG_Ssend(__VA_ARGS__, __FILE__, __LINE__, __func__)
00274
00275 /*
00276         #] MPI_Ssend :
00277         #[ MPI_Rsend :
00278 */
00279
00280 static inline int MPIDBG_Rsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
      MPI_Comm comm, MPIDBG_EXTARG)
00281 {
00282     MPIDBG_Out("MPI_Rsend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00283     int ret = MPI_Rsend(buf, count, datatype, dest, tag, comm);
00284     if ( ret == MPI_SUCCESS ) {
00285         MPIDBG_Out("MPI_Rsend: OK src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00286     }
00287     else {
00288         MPIDBG_Out("MPI_Rsend: Failed src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00289     }
00290     return ret;
00291 }
00292 #define MPI_Rsend(...) MPIDBG_Rsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
00293
00294 /*
00295         #] MPI_Rsend :
00296         #[ MPI_Isend :
00297 */
00298
00299 static inline int MPIDBG_Isend(const void* buf, int count, MPI_Datatype datatype, int dest, int tag,
      MPI_Comm comm, MPI_Request *request, MPIDBG_EXTARG)
00300 {
00301     MPIDBG_Out("MPI_Isend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00302     int ret = MPI_Isend(buf, count, datatype, dest, tag, comm, request);
00303     if ( ret == MPI_SUCCESS ) {
00304         MPIDBG_Out("MPI_Isend: OK src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00305     }
00306     else {
00307         MPIDBG_Out("MPI_Isend: Failed src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00308     }
00309     return ret;
00310 }
00311 #define MPI_Isend(...) MPIDBG_Isend(__VA_ARGS__, __FILE__, __LINE__, __func__)
00312
00313 /*
00314         #] MPI_Isend :
00315         #[ MPI_Ibsend :
00316 */
00317
00318 static inline int MPIDBG_Ibsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
      MPI_Comm comm, MPI_Request *request, MPIDBG_EXTARG)
00319 {
00320     MPIDBG_Out("MPI_Ibsend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00321     int ret = MPI_Ibsend(buf, count, datatype, dest, tag, comm, request);
00322     if ( ret == MPI_SUCCESS ) {
00323         MPIDBG_Out("MPI_Ibsend: OK src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00324     }
00325     else {
00326         MPIDBG_Out("MPI_Ibsend: Failed src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag,
      count);
00327     }
00328     return ret;
00329 }
00330 #define MPI_Ibsend(...) MPIDBG_Ibsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
00331
00332 /*
00333         #] MPI_Ibsend :
00334         #[ MPI_Issend :
00335 */
00336
00337 static inline int MPIDBG_Issend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
      MPI_Comm comm, MPI_Request *request, MPIDBG_EXTARG)
```

```
00338 {
00339     MPIDBG_Out("MPI_Issend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00340     int ret = MPI_Issend(buf, count, datatype, dest, tag, comm, request);
00341     if ( ret == MPI_SUCCESS ) {
00342         MPIDBG_Out("MPI_Issend: OK src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00343     }
00344     else {
00345         MPIDBG_Out("MPI_Issend: Failed src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag,
    count);
00346     }
00347     return ret;
00348 }
00349 #define MPI_Issend(...) MPIDBG_Issend(__VA_ARGS__, __FILE__, __LINE__, __func__)
00350
00351 /*
00352         #] MPI_Issend :
00353         #[ MPI_Irsend :
00354 */
00355
00356 static inline int MPIDBG_Irsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm, MPI_Request *request, MPIDBG_EXTARG)
00357 {
00358     MPIDBG_Out("MPI_Irsend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00359     int ret = MPI_Irsend(buf, count, datatype, dest, tag, comm, request);
00360     if ( ret == MPI_SUCCESS ) {
00361         MPIDBG_Out("MPI_Irsend: OK src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00362     }
00363     else {
00364         MPIDBG_Out("MPI_Irsend: Failed src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag,
    count);
00365     }
00366     return ret;
00367 }
00368 #define MPI_Irsend(...) MPIDBG_Irsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
00369
00370 /*
00371         #] MPI_Irsend :
00372         #[ MPI_Irecv :
00373 */
00374
00375 static inline int MPIDBG_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag,
    MPI_Comm comm, MPI_Request *request, MPIDBG_EXTARG)
00376 {
00377     MPIDBG_Out("MPI_Irecv: src=%d dest=%d tag=%d", source, MPIDBG_RANK, tag);
00378     int ret = MPI_Irecv(buf, count, datatype, source, tag, comm, request);
00379     if ( ret == MPI_SUCCESS ) {
00380         MPIDBG_Out("MPI_Irecv: OK src=%d dest=%d tag=%d", source, MPIDBG_RANK, tag);
00381     }
00382     else {
00383         MPIDBG_Out("MPI_Irecv: Failed src=%d dest=%d tag=%d", source, MPIDBG_RANK, tag);
00384     }
00385     return ret;
00386 }
00387 #define MPI_Irecv(...) MPIDBG_Irecv(__VA_ARGS__, __FILE__, __LINE__, __func__)
00388
00389 /*
00390         #] MPI_Irecv :
00391         #[ MPI_Wait :
00392 */
00393
00394 static inline int MPIDBG_Wait(MPI_Request *request, MPI_Status *status, MPIDBG_EXTARG)
00395 {
00396     MPI_Status st;
00397     if ( status == MPI_STATUS_IGNORE ) status = &st;
00398     char buf1[MPIDBG_BUFSIZE * 1], buf2[MPIDBG_BUFSIZE * 1];
00399     MPI_Request old_request = *request;
00400     MPIDBG_sprint_requests(buf1, 1, request);
00401     MPIDBG_Out("MPI_Wait: rank=%d request=%s", MPIDBG_RANK, buf1);
00402     int ret = MPI_Wait(request, status);
00403     if ( ret == MPI_SUCCESS ) {
00404         MPIDBG_sprint_statuses(buf2, 1, request, &old_request, status);
00405         MPIDBG_Out("MPI_Wait: OK rank=%d request=%s result=%s", MPIDBG_RANK, buf1, buf2);
00406     }
00407     else {
00408         MPIDBG_Out("MPI_Wait: Failed rank=%d request=%s", MPIDBG_RANK, buf1);
00409     }
00410     return ret;
00411 }
00412 #define MPI_Wait(...) MPIDBG_Wait(__VA_ARGS__, __FILE__, __LINE__, __func__)
00413
00414 /*
00415         #] MPI_Wait :
00416         #[ MPI_Test :
00417 */
00418
00419 static inline int MPIDBG_Test(MPI_Request *request, int *flag, MPI_Status *status, MPIDBG_EXTARG)
00420 {
```

```
00421     MPI_Status st;
00422     if ( status == MPI_STATUS_IGNORE ) status = &st;
00423     char buf1[MPIDBG_BUFSIZE * 1], buf2[MPIDBG_BUFSIZE * 1];
00424     MPI_Request old_request = *request;
00425     MPIDBG_sprint_requests(buf1, 1, request);
00426     MPIDBG_Out("MPI_Test: rank=%d request=%s", MPIDBG_RANK, buf1);
00427     int ret = MPI_Test(request, flag, status);
00428     if ( ret == MPI_SUCCESS ) {
00429         if ( *flag ) {
00430             MPIDBG_sprint_statuses(buf2, 1, request, &old_request, status);
00431             MPIDBG_Out("MPI_Test: OK rank=%d request=%s result=%s", MPIDBG_RANK, buf1, buf2);
00432         }
00433         else {
00434             MPIDBG_Out("MPI_Test: OK rank=%d request=%s flag=false", MPIDBG_RANK, buf1);
00435         }
00436     }
00437     else {
00438         MPIDBG_Out("MPI_Test: Failed rank=%d request=%s", MPIDBG_RANK, buf1);
00439     }
00440     return ret;
00441 }
00442 #define MPI_Test(...) MPIDBG_Test(__VA_ARGS__, __FILE__, __LINE__, __func__)
00443
00444 /*
00445         #] MPI_Test :
00446         #[ MPI_Waitany :
00447 */
00448
00449 static inline int MPIDBG_Waitany(int count, MPI_Request *array_of_requests, int *index, MPI_Status
    *status, MPIDBG_EXTARG)
00450 {
00451     MPI_Status st;
00452     if ( status == MPI_STATUS_IGNORE ) status = &st;
00453     char buf1[MPIDBG_BUFSIZE * 1], buf2[MPIDBG_BUFSIZE * 1];
00454     MPI_Request old_requests[count];
00455     memcpy(old_requests, array_of_requests, sizeof(MPI_Request) * count);
00456     MPIDBG_sprint_requests(buf1, count, array_of_requests);
00457     MPIDBG_Out("MPI_Waitany: rank=%d request=%s", MPIDBG_RANK, buf1);
00458     int ret = MPI_Waitany(count, array_of_requests, index, status);
00459     if ( ret == MPI_SUCCESS ) {
00460         MPI_Status statuses[count];
00461         statuses[*index] = *status;
00462         MPIDBG_sprint_statuses(buf2, count, old_requests, array_of_requests, statuses);
00463         MPIDBG_Out("MPI_Waitany: OK rank=%d request=%s result=%s", MPIDBG_RANK, buf1, buf2);
00464     }
00465     else {
00466         MPIDBG_Out("MPI_Waitany: Failed rank=%d request=%s", MPIDBG_RANK, buf1);
00467     }
00468     return ret;
00469 }
00470 #define MPI_Waitany(...) MPIDBG_Waitany(__VA_ARGS__, __FILE__, __LINE__, __func__)
00471
00472 /*
00473         #] MPI_Waitany :
00474         #[ MPI_Testany :
00475 */
00476
00477 static inline int MPIDBG_Testany(int count, MPI_Request *array_of_requests, int *index, int *flag,
    MPI_Status *status, MPIDBG_EXTARG)
00478 {
00479     MPI_Status st;
00480     if ( status == MPI_STATUS_IGNORE ) status = &st;
00481     char buf1[MPIDBG_BUFSIZE * 1], buf2[MPIDBG_BUFSIZE * 1];
00482     MPI_Request old_requests[count];
00483     memcpy(old_requests, array_of_requests, sizeof(MPI_Request) * count);
00484     MPIDBG_sprint_requests(buf1, count, array_of_requests);
00485     MPIDBG_Out("MPI_Testany: rank=%d request=%s", MPIDBG_RANK, buf1);
00486     int ret = MPI_Testany(count, array_of_requests, index, flag, status);
00487     if ( ret == MPI_SUCCESS ) {
00488         if ( *flag ) {
00489             MPI_Status statuses[count];
00490             statuses[*index] = *status;
00491             MPIDBG_sprint_statuses(buf2, count, old_requests, array_of_requests, statuses);
00492             MPIDBG_Out("MPI_Testany: OK rank=%d request=%s result=%s", MPIDBG_RANK, buf1, buf2);
00493         }
00494         else {
00495             MPIDBG_Out("MPI_Testany: OK rank=%d request=%s flag=false", MPIDBG_RANK, buf1);
00496         }
00497     }
00498     else {
00499         MPIDBG_Out("MPI_Testany: Failed rank=%d request=%s", MPIDBG_RANK, buf1);
00500     }
00501     return ret;
00502 }
00503 #define MPI_Testany(...) MPIDBG_Testany(__VA_ARGS__, __FILE__, __LINE__, __func__)
00504
00505 /*
```

```
00506         #] MPI_Testany :
00507         #[ MPI_Waitall :
00508 */
00509
00510 static inline int MPIDBG_Waitall(int count, MPI_Request *array_of_requests, MPI_Status
      *array_of_statuses, MPIDBG_EXTARG)
00511 {
00512     MPI_Status st[count];
00513     if ( array_of_statuses == MPI_STATUSES_IGNORE ) array_of_statuses = st;
00514     char buf1[MPIDBG_BUFSIZE * count], buf2[MPIDBG_BUFSIZE * count];
00515     MPI_Request old_requests[count];
00516     memcpy(old_requests, array_of_requests, sizeof(MPI_Request) * count);
00517     MPIDBG_sprint_requests(buf1, count, array_of_requests);
00518     MPIDBG_Out("MPI_Waitall: rank=%d request=%s", MPIDBG_RANK, buf1);
00519     int ret = MPI_Waitall(count, array_of_requests, array_of_statuses);
00520     if ( ret == MPI_SUCCESS ) {
00521         MPIDBG_sprint_statuses(buf2, count, old_requests, array_of_requests, array_of_statuses);
00522         MPIDBG_Out("MPI_Waitall: OK rank=%d request=%s result=%s", MPIDBG_RANK, buf1, buf2);
00523     }
00524     else {
00525         MPIDBG_Out("MPI_Waitall: Failed rank=%d request=%s", MPIDBG_RANK, buf1);
00526     }
00527     return ret;
00528 }
00529 #define MPI_Waitall(...) MPIDBG_Waitall(__VA_ARGS__, __FILE__, __LINE__, __func__)
00530
00531 /*
00532         #] MPI_Waitall :
00533         #[ MPI_Testall :
00534 */
00535
00536 static inline int MPIDBG_Testall(int count, MPI_Request *array_of_requests, int *flag, MPI_Status
      *array_of_statuses, MPIDBG_EXTARG)
00537 {
00538     MPI_Status st[count];
00539     if ( array_of_statuses == MPI_STATUSES_IGNORE ) array_of_statuses = st;
00540     char buf1[MPIDBG_BUFSIZE * count], buf2[MPIDBG_BUFSIZE * count];
00541     MPI_Request old_requests[count];
00542     memcpy(old_requests, array_of_requests, sizeof(MPI_Request) * count);
00543     MPIDBG_sprint_requests(buf1, count, array_of_requests);
00544     MPIDBG_Out("MPI_Testall: rank=%d request=%s", MPIDBG_RANK, buf1);
00545     int ret = MPI_Testall(count, array_of_requests, flag, array_of_statuses);
00546     if ( ret == MPI_SUCCESS ) {
00547         if ( *flag ) {
00548             MPIDBG_sprint_statuses(buf2, count, old_requests, array_of_requests, array_of_statuses);
00549             MPIDBG_Out("MPI_Testall: OK rank=%d request=%s result=%s", MPIDBG_RANK, buf1, buf2);
00550         }
00551         else {
00552             MPIDBG_Out("MPI_Testall: OK rank=%d request=%s flag=false", MPIDBG_RANK, buf1);
00553         }
00554     }
00555     else {
00556         MPIDBG_Out("MPI_Testall: Failed rank=%d request=%s", MPIDBG_RANK, buf1);
00557     }
00558     return ret;
00559 }
00560 #define MPI_Testall(...) MPIDBG_Testall(__VA_ARGS__, __FILE__, __LINE__, __func__)
00561
00562 /*
00563         #] MPI_Testall :
00564         #[ MPI_Waitsome :
00565 */
00566
00567 static inline int MPIDBG_Waitsome(int incount, MPI_Request *array_of_requests, int *outcount, int
      *array_of_indices, MPI_Status *array_of_statuses, MPIDBG_EXTARG)
00568 {
00569     MPI_Status st[incount];
00570     if ( array_of_statuses == MPI_STATUSES_IGNORE ) array_of_statuses = st;
00571     char buf1[MPIDBG_BUFSIZE * incount], buf2[MPIDBG_BUFSIZE * incount];
00572     MPI_Request old_requests[incount];
00573     memcpy(old_requests, array_of_requests, sizeof(MPI_Request) * incount);
00574     MPIDBG_sprint_requests(buf1, incount, array_of_requests);
00575     MPIDBG_Out("MPI_Waitsome: rank=%d request=%s", MPIDBG_RANK, buf1);
00576     int ret = MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses);
00577     if ( ret == MPI_SUCCESS ) {
00578         MPIDBG_sprint_statuses(buf2, incount, old_requests, array_of_requests, array_of_statuses);
00579         MPIDBG_Out("MPI_Waitsome: OK rank=%d request=%s result=%s", MPIDBG_RANK, buf1, buf2);
00580     }
00581     else {
00582         MPIDBG_Out("MPI_Waitsome: Failed rank=%d request=%s", MPIDBG_RANK, buf1);
00583     }
00584     return ret;
00585 }
00586 #define MPI_Waitsome(...) MPIDBG_Waitsome(__VA_ARGS__, __FILE__, __LINE__, __func__)
00587
00588 /*
00589         #] MPI_Waitsome :
```

```
00590         #[ MPI_Testsome :
00591 */
00592
00593 static inline int MPIDBG_Testsome(int incount, MPI_Request *array_of_requests, int *outcount, int
      *array_of_indices, MPI_Status *array_of_statuses, MPIDBG_EXTARG)
00594 {
00595     MPI_Status st[incount];
00596     if ( array_of_statuses == MPI_STATUSES_IGNORE ) array_of_statuses = st;
00597     char buf1[MPIDBG_BUFSIZE * incount], buf2[MPIDBG_BUFSIZE * incount];
00598     MPI_Request old_requests[incount];
00599     memcpy(old_requests, array_of_requests, sizeof(MPI_Request) * incount);
00600     MPIDBG_sprint_requests(buf1, incount, array_of_requests);
00601     MPIDBG_Out("MPI_Testsome: rank=%d request=%s", MPIDBG_RANK, buf1);
00602     int ret = MPI_Testsome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses);
00603     if ( ret == MPI_SUCCESS ) {
00604         MPIDBG_sprint_statuses(buf2, incount, old_requests, array_of_requests, array_of_statuses);
00605         MPIDBG_Out("MPI_Testsome: OK rank=%d request=%s result=%s", MPIDBG_RANK, buf1, buf2);
00606     }
00607     else {
00608         MPIDBG_Out("MPI_Testsome: Failed rank=%d request=%s", MPIDBG_RANK, buf1);
00609     }
00610     return ret;
00611 }
00612 #define MPI_Testsome(...) MPIDBG_Testsome(__VA_ARGS__, __FILE__, __LINE__, __func__)
00613
00614 /*
00615         #] MPI_Testsome :
00616         #[ MPI_Iprobe :
00617 */
00618
00619 static inline int MPIDBG_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status,
      MPIDBG_EXTARG)
00620 {
00621     MPI_Status st;
00622     if ( status == MPI_STATUS_IGNORE ) status = &st;
00623     MPIDBG_Out("MPI_Iprobe: src=%d dest=%d tag=%d", source, MPIDBG_RANK, tag);
00624     MPIDBG_insert_delay();
00625     int ret = MPI_Iprobe(source, tag, comm, flag, status);
00626     if ( ret == MPI_SUCCESS ) {
00627         if ( *flag ) {
00628             int ret_size = 0;
00629             MPI_Get_count(status, MPI_BYTE, &ret_size);
00630             MPIDBG_Out("MPI_Iprobe: OK src=%d dest=%d tag=%d size=%d", status->MPI_SOURCE,
      MPIDBG_RANK, status->MPI_TAG, ret_size);
00631         }
00632         else {
00633             MPIDBG_Out("MPI_Iprobe: OK src=%d dest=%d tag=%d flag=false", source, MPIDBG_RANK, tag);
00634         }
00635     }
00636     else {
00637         MPIDBG_Out("MPI_Iprobe: Failed src=%d dest=%d tag=%d", source, MPIDBG_RANK, tag);
00638     }
00639     return ret;
00640 }
00641 #define MPI_Iprobe(...) MPIDBG_Iprobe(__VA_ARGS__, __FILE__, __LINE__, __func__)
00642
00643 /*
00644         #] MPI_Iprobe :
00645         #[ MPI_Probe :
00646 */
00647
00648 static inline int MPIDBG_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status, MPIDBG_EXTARG)
00649 {
00650     MPI_Status st;
00651     if ( status == MPI_STATUS_IGNORE ) status = &st;
00652     MPIDBG_Out("MPI_Probe: src=%d dest=%d tag=%d", source, MPIDBG_RANK, tag);
00653     int ret = MPI_Probe(source, tag, comm, status);
00654     if ( ret == MPI_SUCCESS ) {
00655         int ret_size = 0;
00656         MPI_Get_count(status, MPI_BYTE, &ret_size);
00657         MPIDBG_Out("MPI_Probe: OK src=%d dest=%d tag=%d size=%d", status->MPI_SOURCE, MPIDBG_RANK,
      status->MPI_TAG, ret_size);
00658     }
00659     else {
00660         MPIDBG_Out("MPI_Probe: Failed src=%d dest=%d tag=%d", source, MPIDBG_RANK, tag);
00661     }
00662     return ret;
00663 }
00664 #define MPI_Probe(...) MPIDBG_Probe(__VA_ARGS__, __FILE__, __LINE__, __func__)
00665
00666 /*
00667         #] MPI_Probe :
00668         #[ MPI_Cancel :
00669 */
00670
00671 static inline int MPIDBG_Cancel(MPI_Request *request, MPIDBG_EXTARG)
00672 {
```

```
00673     MPIDBG_Out("MPI_Cancel", MPIDBG_RANK);
00674     int ret = MPI_Cancel(request);
00675     if ( ret == MPI_SUCCESS ) {
00676         MPIDBG_Out("MPI_Cancel: OK");
00677     }
00678     else {
00679         MPIDBG_Out("MPI_Cancel: Failed");
00680     }
00681     return ret;
00682 }
00683 #define MPI_Cancel(...) MPIDBG_Cancel(__VA_ARGS__, __FILE__, __LINE__, __func__)
00684
00685 /*
00686         #] MPI_Cancel :
00687         #[ MPI_Test_cancelled :
00688 */
00689
00690 static inline int MPIDBG_Test_cancelled(MPI_Status *status, int *flag, MPIDBG_EXTARG)
00691 {
00692     MPIDBG_Out("MPI_Test_cancelled", MPIDBG_RANK);
00693     int ret = MPI_Test_cancelled(status, flag);
00694     if ( ret == MPI_SUCCESS ) {
00695         if ( *flag ) {
00696             MPIDBG_Out("MPI_Test_cancelled: OK flag=true");
00697         }
00698         else {
00699             MPIDBG_Out("MPI_Test_cancelled: OK flag=false");
00700         }
00701     }
00702     else {
00703         MPIDBG_Out("MPI_Test_cancelled: Failed");
00704     }
00705     return ret;
00706 }
00707 #define MPI_Test_cancelled(...) MPIDBG_Test_cancelled(__VA_ARGS__, __FILE__, __LINE__, __func__)
00708
00709 /*
00710         #] MPI_Test_cancelled :
00711         #[ MPI_Barrier :
00712 */
00713
00714 static inline int MPIDBG_Barrier(MPI_Comm comm, MPIDBG_EXTARG)
00715 {
00716     MPIDBG_Out("MPI_Barrier");
00717     int ret = MPI_Barrier(comm);
00718     if ( ret == MPI_SUCCESS ) {
00719         MPIDBG_Out("MPI_Barrier: OK");
00720     }
00721     else {
00722         MPIDBG_Out("MPI_Barrier: Failed");
00723     }
00724     return ret;
00725 }
00726 #define MPI_Barrier(...) MPIDBG_Barrier(__VA_ARGS__, __FILE__, __LINE__, __func__)
00727
00728 /*
00729         #] MPI_Barrier :
00730         #[ MPI_Bcast :
00731 */
00732
00733 static inline int MPIDBG_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm
      comm, MPIDBG_EXTARG)
00734 {
00735     MPIDBG_Out("MPI_Bcast: root=%d count=%d", root, count);
00736     int ret = MPI_Bcast(buffer, count, datatype, root, comm);
00737     if ( ret == MPI_SUCCESS ) {
00738         MPIDBG_Out("MPI_Bcast: OK root=%d count=%d", root, count);
00739     }
00740     else {
00741         MPIDBG_Out("MPI_Bcast: Failed root=%d count=%d", root, count);
00742     }
00743     return ret;
00744 }
00745 #define MPI_Bcast(...) MPIDBG_Bcast(__VA_ARGS__, __FILE__, __LINE__, __func__)
00746
00747 /*
00748         #] MPI_Bcast :
00749         #[ MPI_Reduce :
00750 */
00751
00752 static inline int MPIDBG_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
      MPI_Op op, int root, MPI_Comm comm, MPIDBG_EXTARG)
00753 {
00754     MPIDBG_Out("MPI_Reduce: root=%d count=%d", root, count);
00755     int ret = MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm);
00756     if ( ret == MPI_SUCCESS ) {
00757         MPIDBG_Out("MPI_Reduce: OK root=%d count=%d", root, count);
```

```
00758     }
00759     else {
00760         MPIDBG_Out("MPI_Reduce: Failed root=%d count=%d", root, count);
00761     }
00762     return ret;
00763 }
00764 #define MPI_Reduce(...) MPIDBG_Reduce(__VA_ARGS__, __FILE__, __LINE__, __func__)
00765
00766 /*
00767         #] MPI_Reduce :
00768     #] MPI APIs :
00769 */
00770
00771 #endif  /* MPIDBG_H */
```

4.85 mytime.cc

```
00001 /* #[ License : */
00002 /*
00003  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00004  *   When using this file you are requested to refer to the publication
00005  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00006  *   This is considered a matter of courtesy as the development was paid
00007  *   for by FOM the Dutch physics granting agency and we would like to
00008  *   be able to track its scientific use to convince FOM of its value
00009  *   for the community.
00010  *
00011  *   This file is part of FORM.
00012  *
00013  *   FORM is free software: you can redistribute it and/or modify it under the
00014  *   terms of the GNU General Public License as published by the Free Software
00015  *   Foundation, either version 3 of the License, or (at your option) any later
00016  *   version.
00017  *
00018  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00019  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00020  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00021  *   details.
00022  *
00023  *   You should have received a copy of the GNU General Public License along
00024  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00025  */
00026 /* #] License : */
00027
00028 #ifdef HAVE_CONFIG_H
00029 #include <config.h>
00030 #endif
00031
00032 // A timing routine for debugging. Only on Unix (where sys/time.h is available).
00033 #ifdef UNIX
00034
00035 #include <sys/time.h>
00036 #include <cstdlib>
00037 #include <cstdio>
00038 #include <string>
00039
00040 #ifndef timersub
00041 /* timersub is not in POSIX, but presents on most BSD derivatives.
00042    This implementation is borrowed from glibc. (TU 23 Oct 2011) */
00043 #define timersub(a, b, result) \
00044     do { \
00045         (result)->tv_sec = (a)->tv_sec - (b)->tv_sec; \
00046         (result)->tv_usec = (a)->tv_usec - (b)->tv_usec; \
00047         if ((result)->tv_usec < 0) { \
00048             --(result)->tv_sec; \
00049             (result)->tv_usec += 1000000; \
00050         } \
00051     } while (0)
00052 #endif
00053
00054 bool starttime_set = false;
00055 timeval starttime;
00056
00057 double thetime () {
00058     if (!starttime_set) {
00059         gettimeofday(&starttime,NULL);
00060         starttime_set=true;
00061     }
00062
00063   timeval now,diff;
00064   gettimeofday(&now,NULL);
00065   timersub(&now,&starttime,&diff);
00066   return diff.tv_sec+diff.tv_usec/1000000.0;
```

```
00067 }
00068
00069 std::string thetime_str() {
00070     char res[10];
00071     snprintf (res,10,"%.4lf", thetime());
00072     return res;
00073 }
00074
00075 #endif  // UNIX
```

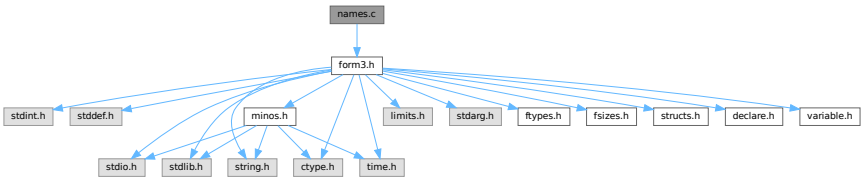
4.86 mytime.h

```
00001 #pragma once
00002 /* #[ License : */
00003 /*
00004  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00005  *   When using this file you are requested to refer to the publication
00006  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00007  *   This is considered a matter of courtesy as the development was paid
00008  *   for by FOM the Dutch physics granting agency and we would like to
00009  *   be able to track its scientific use to convince FOM of its value
00010  *   for the community.
00011  *
00012  *   This file is part of FORM.
00013  *
00014  *   FORM is free software: you can redistribute it and/or modify it under the
00015  *   terms of the GNU General Public License as published by the Free Software
00016  *   Foundation, either version 3 of the License, or (at your option) any later
00017  *   version.
00018  *
00019  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00020  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00021  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00022  *   details.
00023  *
00024  *   You should have received a copy of the GNU General Public License along
00025  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00026  */
00027 /* #] License : */
00028
00029 double thetime();
00030 std::string thetime_str();
```

4.87 names.c File Reference

`#include "form3.h"`

Include dependency graph for names.c:



Functions

- NAMENODE * GetNode (NAMETREE *nametree, UBYTE *name)
- int AddName (NAMETREE *nametree, UBYTE *name, WORD type, WORD number, int *nodenum)
- int GetName (NAMETREE *nametree, UBYTE *namein, WORD *number, int par)
- UBYTE * GetFunction (UBYTE *s, WORD *funnum)
- UBYTE * GetNumber (UBYTE *s, WORD *num)

- int GetLastExprName (UBYTE *name, WORD *number)
- int GetOName (NAMETREE *nametree, UBYTE *name, WORD *number, int par)
- int GetAutoName (UBYTE *name, WORD *number)
- int GetVar (UBYTE *name, WORD *type, WORD *number, int wantedtype, int par)
- int EntVar (WORD type, UBYTE *name, WORD x, WORD y, WORD z, WORD d)
- int GetDollar (UBYTE *name)
- void DumpTree (NAMETREE *nametree)
- void DumpNode (NAMETREE *nametree, WORD node, WORD depth)
- int CompactifyTree (NAMETREE *nametree, WORD par)
- void CopyTree (NAMETREE *newtree, NAMETREE *oldtree, WORD node, WORD par)
- void LinkTree (NAMETREE *tree, WORD offset, WORD numnodes)
- NAMETREE * MakeNameTree (void)
- void FreeNameTree (NAMETREE *n)
- void ClearWildcardNames (void)
- int AddWildcardName (UBYTE *name)
- int GetWildcardName (UBYTE *name)
- int AddSymbol (UBYTE *name, int minpow, int maxpow, int cplx, int dim)
- int CoSymbol (UBYTE *s)
- int AddIndex (UBYTE *name, int dim, int dim4)
- int CoIndex (UBYTE *s)
- UBYTE * DoDimension (UBYTE *s, int *dim, int *dim4)
- int CoDimension (UBYTE *s)
- int AddVector (UBYTE *name, int cplx, int dim)
- int CoVector (UBYTE *s)
- int AddFunction (UBYTE *name, int comm, int istensor, int cplx, int symprop, int dim, int argmax, int argmin)
- int CoCommuteInSet (UBYTE *s)
- int CoFunction (UBYTE *s, int comm, int istensor)
- int CoNFunction (UBYTE *s)
- int CoCFunction (UBYTE *s)
- int CoNTensor (UBYTE *s)
- int CoCTensor (UBYTE *s)
- int DoTable (UBYTE *s, int par)
- int CoTable (UBYTE *s)
- int CoNTable (UBYTE *s)
- int CoCTable (UBYTE *s)
- void EmptyTable (TABLES T)
- int AddSet (UBYTE *name, WORD dim)
- int DoElements (UBYTE *s, SETS set, UBYTE *name)
- int CoSet (UBYTE *s)
- int DoTempSet (UBYTE *from, UBYTE *to)
- int CoAuto (UBYTE *inp)
- int AddDollar (UBYTE *name, WORD type, WORD *start, LONG size)
- int ReplaceDollar (WORD number, WORD newtype, WORD *newstart, LONG newsize)
- int AddDubious (UBYTE *name)
- int MakeDubious (NAMETREE *nametree, UBYTE *name, WORD *number)
- int NameConflict (int type, UBYTE *name)
- int AddExpression (UBYTE *name, int x, int y)
- int GetLabel (UBYTE *name)
- void ResetVariables (int par)
- void RemoveDollars (void)
- void Globalize (int par)
- int TestName (UBYTE *name)

4.87.1 Detailed Description

The complete names administration. All variables with a name have to pass here to be properly registered, have structs of the proper type assigned to them etc. The file also contains the utility routines for maintaining the balanced trees that make searching for names rather fast.

Definition in file names.c.

4.87.2 Function Documentation

4.87.2.1 GetNode()

```
NAMENODE * GetNode (
              NAMETREE * nametree,
              UBYTE * name )
```

Definition at line 49 of file names.c.

4.87.2.2 AddName()

```
int AddName (
              NAMETREE * nametree,
              UBYTE * name,
              WORD type,
              WORD number,
              int * nodenum )
```

Definition at line 71 of file names.c.

4.87.2.3 GetName()

```
int GetName (
              NAMETREE * nametree,
              UBYTE * namein,
              WORD * number,
              int par )
```

Definition at line 254 of file names.c.

4.87.2.4 GetFunction()

```
UBYTE * GetFunction (
              UBYTE * s,
              WORD * funnum )
```

Definition at line 344 of file names.c.

4.87.2.5 GetNumber()

```
UBYTE * GetNumber (
             UBYTE * s,
             WORD * num )
```

Definition at line 390 of file names.c.

4.87.2.6 GetLastExprName()

```
int GetLastExprName (
             UBYTE * name,
             WORD * number )
```

Definition at line 440 of file names.c.

4.87.2.7 GetOName()

```
int GetOName (
             NAMETREE * nametree,
             UBYTE * name,
             WORD * number,
             int par )
```

Definition at line 462 of file names.c.

4.87.2.8 GetAutoName()

```
int GetAutoName (
             UBYTE * name,
             WORD * number )
```

Definition at line 481 of file names.c.

4.87.2.9 GetVar()

```
int GetVar (
             UBYTE * name,
             WORD * type,
             WORD * number,
             int wantedtype,
             int par )
```

Definition at line 526 of file names.c.

4.87.2.10 EntVar()

```
int EntVar (
            WORD type,
            UBYTE * name,
            WORD x,
            WORD y,
            WORD z,
            WORD d )
```

Definition at line 559 of file names.c.

4.87.2.11 GetDollar()

```
int GetDollar (
            UBYTE * name )
```

Definition at line 592 of file names.c.

4.87.2.12 DumpTree()

```
void DumpTree (
            NAMETREE * nametree )
```

Definition at line 604 of file names.c.

4.87.2.13 DumpNode()

```
void DumpNode (
            NAMETREE * nametree,
            WORD node,
            WORD depth )
```

Definition at line 617 of file names.c.

4.87.2.14 CompactifyTree()

```
int CompactifyTree (
            NAMETREE * nametree,
            WORD par )
```

Definition at line 636 of file names.c.

4.87.2.15 CopyTree()

```
void CopyTree (
            NAMETREE * newtree,
            NAMETREE * oldtree,
            WORD node,
            WORD par )
```

Definition at line 698 of file names.c.

4.87.2.16 LinkTree()

```
void LinkTree (
            NAMETREE * tree,
            WORD offset,
            WORD numnodes )
```

Definition at line 788 of file names.c.

4.87.2.17 MakeNameTree()

```
NAMETREE * MakeNameTree (
            void  )
```

Definition at line 819 of file names.c.

4.87.2.18 FreeNameTree()

```
void FreeNameTree (
            NAMETREE * n )
```

Definition at line 838 of file names.c.

4.87.2.19 ClearWildcardNames()

```
void ClearWildcardNames (
            void  )
```

Definition at line 853 of file names.c.

4.87.2.20 AddWildcardName()

```
int AddWildcardName (
            UBYTE * name )
```

Definition at line 858 of file names.c.

4.87.2.21 GetWildcardName()

```
int GetWildcardName (
            UBYTE * name )
```

Definition at line 902 of file names.c.

4.87.2.22 AddSymbol()

```
int AddSymbol (
             UBYTE * name,
             int minpow,
             int maxpow,
             int cplx,
             int dim )
```

Definition at line 924 of file names.c.

4.87.2.23 CoSymbol()

```
int CoSymbol (
             UBYTE * s )
```

Definition at line 950 of file names.c.

4.87.2.24 AddIndex()

```
int AddIndex (
             UBYTE * name,
             int dim,
             int dim4 )
```

Definition at line 1092 of file names.c.

4.87.2.25 CoIndex()

```
int CoIndex (
             UBYTE * s )
```

Definition at line 1116 of file names.c.

4.87.2.26 DoDimension()

```
UBYTE * DoDimension (
             UBYTE * s,
             int * dim,
             int * dim4 )
```

Definition at line 1164 of file names.c.

4.87.2.27 CoDimension()

```
int CoDimension (
             UBYTE * s )
```

Definition at line 1230 of file names.c.

4.87.2.28 AddVector()

```
int AddVector (
            UBYTE * name,
            int cplx,
            int dim )
```

Definition at line 1248 of file names.c.

4.87.2.29 CoVector()

```
int CoVector (
            UBYTE * s )
```

Definition at line 1271 of file names.c.

4.87.2.30 AddFunction()

```
int AddFunction (
            UBYTE * name,
            int comm,
            int istensor,
            int cplx,
            int symprop,
            int dim,
            int argmax,
            int argmin )
```

Definition at line 1330 of file names.c.

4.87.2.31 CoCommuteInSet()

```
int CoCommuteInSet (
            UBYTE * s )
```

Definition at line 1360 of file names.c.

4.87.2.32 CoFunction()

```
int CoFunction (
            UBYTE * s,
            int comm,
            int istensor )
```

Definition at line 1450 of file names.c.

4.87.2.33 CoNFunction()

```
int CoNFunction (
            UBYTE * s )
```

Definition at line 1630 of file names.c.

4.87.2.34 CoCFunction()

```
int CoCFunction (
            UBYTE * s )
```

Definition at line 1631 of file names.c.

4.87.2.35 CoNTensor()

```
int CoNTensor (
            UBYTE * s )
```

Definition at line 1632 of file names.c.

4.87.2.36 CoCTensor()

```
int CoCTensor (
            UBYTE * s )
```

Definition at line 1633 of file names.c.

4.87.2.37 DoTable()

```
int DoTable (
            UBYTE * s,
            int par )
```

Definition at line 1672 of file names.c.

4.87.2.38 CoTable()

```
int CoTable (
            UBYTE * s )
```

Definition at line 2054 of file names.c.

4.87.2.39 CoNTable()

```
int CoNTable (
            UBYTE * s )
```

Definition at line 2064 of file names.c.

4.87.2.40 CoCTable()

```
int CoCTable (
            UBYTE * s )
```

Definition at line 2074 of file names.c.

4.87.2.41 EmptyTable()

```
void EmptyTable (
            TABLES T )
```

Definition at line 2084 of file names.c.

4.87.2.42 AddSet()

```
int AddSet (
            UBYTE * name,
            WORD dim )
```

Definition at line 2128 of file names.c.

4.87.2.43 DoElements()

```
int DoElements (
            UBYTE * s,
            SETS set,
            UBYTE * name )
```

Definition at line 2161 of file names.c.

4.87.2.44 CoSet()

```
int CoSet (
            UBYTE * s )
```

Definition at line 2361 of file names.c.

4.87.2.45 DoTempSet()

```
int DoTempSet (
            UBYTE * from,
            UBYTE * to )
```

Definition at line 2488 of file names.c.

4.87.2.46 CoAuto()

```
int CoAuto (
            UBYTE * inp )
```

Definition at line 2580 of file names.c.

4.87.2.47 AddDollar()

```
int AddDollar (
            UBYTE * name,
            WORD type,
            WORD * start,
            LONG size )
```

Definition at line 2610 of file names.c.

4.87.2.48 ReplaceDollar()

```
int ReplaceDollar (
            WORD number,
            WORD newtype,
            WORD * newstart,
            LONG newsize )
```

Definition at line 2653 of file names.c.

4.87.2.49 AddDubious()

```
int AddDubious (
            UBYTE * name )
```

Definition at line 2686 of file names.c.

4.87.2.50 MakeDubious()

```
int MakeDubious (
            NAMETREE * nametree,
            UBYTE * name,
            WORD * number )
```

Definition at line 2700 of file names.c.

4.87.2.51 NameConflict()

```
int NameConflict (
            int type,
            UBYTE * name )
```

Definition at line 2735 of file names.c.

4.87.2.52 AddExpression()

```
int AddExpression (
            UBYTE * name,
            int x,
            int y )
```

Definition at line 2751 of file names.c.

4.87.2.53 GetLabel()

```
int GetLabel (
            UBYTE * name )
```

Definition at line 2794 of file names.c.

4.87.2.54 ResetVariables()

```
void ResetVariables (
            int par )
```

Definition at line 2839 of file names.c.

4.87.2.55 RemoveDollars()

```
void RemoveDollars (
            void  )
```

Definition at line 3135 of file names.c.

4.87.2.56 Globalize()

```
void Globalize (
            int par )
```

Definition at line 3162 of file names.c.

4.87.2.57 TestName()

```
int TestName (
            UBYTE * name )
```

Definition at line 3261 of file names.c.

4.88 names.c

Go to the documentation of this file.

```
00001
00009 /* #[ License : */
00010 /*
00011  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00012  *   When using this file you are requested to refer to the publication
00013  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00014  *   This is considered a matter of courtesy as the development was paid
00015  *   for by FOM the Dutch physics granting agency and we would like to
00016  *   be able to track its scientific use to convince FOM of its value
00017  *   for the community.
00018  *
00019  *   This file is part of FORM.
00020  *
00021  *   FORM is free software: you can redistribute it and/or modify it under the
00022  *   terms of the GNU General Public License as published by the Free Software
00023  *   Foundation, either version 3 of the License, or (at your option) any later
00024  *   version.
00025  *
00026  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00027  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00028  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00029  *   details.
00030  *
00031  *   You should have received a copy of the GNU General Public License along
00032  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00033  */
00034 /* #] License : */
00035 /*
00036     #[ Includes :
00037 */
00038
00039 #include "form3.h"
00040
00041 /* EXTERNLOCK(dummylock) */
00042
00043 /*
00044     #] Includes :
00045
00046     #[ GetNode :
00047 */
00048
00049 NAMENODE *GetNode(NAMETREE *nametree, UBYTE *name)
00050 {
00051     NAMENODE *n;
00052     int node, newnode, i;
00053     if ( nametree->namenode == 0 ) return(0);
00054     newnode = nametree->headnode;
00055     do {
00056         node = newnode;
00057         n = nametree->namenode+node;
00058         if ( ( i = StrCmp(name,nametree->namebuffer+n->name) ) < 0 )
00059             newnode = n->left;
00060         else if ( i > 0 ) newnode = n->right;
00061         else { return(n); }
00062     } while ( newnode >= 0 );
00063     return(0);
00064 }
00065
00066 /*
00067     #] GetNode :
00068     #[ AddName :
00069 */
00070
00071 int AddName(NAMETREE *nametree, UBYTE *name, WORD type, WORD number, int *nodenum)
00072 {
00073     NAMENODE *n, *nn, *nnn;
00074     UBYTE *s, *ss, *sss;
00075     LONG *c1,*c2, j, newsize;
00076     int node, newnode, node3, r, rr = 0, i, retval = 0;
00077     if ( nametree->namenode == 0 ) {
00078         s = name; i = 1; while ( *s ) { i++; s++; }
00079         j = INITNAMESIZE;
00080         if ( i > j ) j = i;
00081         nametree->namenode = (NAMENODE *)Malloc1(INITNODESIZE*sizeof(NAMENODE),
00082             "new nametree in AddName");
00083         nametree->namebuffer = (UBYTE *)Malloc1(j,
00084             "new namebuffer in AddName");
00085         nametree->nodesize = INITNODESIZE;
00086         nametree->namesize = j;
00087         nametree->namefill = i;
00088         nametree->nodefill = 1;
00089         nametree->headnode = 0;
```

```
00090         n = nametree->namenode;
00091         n->parent = n->left = n->right = -1;
00092         n->balance = 0;
00093         n->type = type;
00094         n->number = number;
00095         n->name = 0;
00096         s = name;
00097         ss = nametree->namebuffer;
00098         while ( *s ) *ss++ = *s++;
00099         *ss = 0;
00100         *nodenum = 0;
00101         return(retval);
00102     }
00103     newnode = nametree->headnode;
00104     do {
00105         node = newnode;
00106         n = nametree->namenode+node;
00107         if ( StrCmp(name,nametree->namebuffer+n->name) < 0 ) {
00108             newnode = n->left; r = -1;
00109         }
00110         else {
00111             newnode = n->right; r = 1;
00112         }
00113     } while ( newnode >= 0 );
00114 /*
00115     We are at the insertion point. Add the node.
00116 */
00117     if ( nametree->nodefill >= nametree->nodesize ) {   /* Double allocation */
00118         newsize = nametree->nodesize * 2;
00119         if ( newsize > MAXINNAMETREE ) newsize = MAXINNAMETREE;
00120         if ( nametree->nodefill >= MAXINNAMETREE ) {
00121             MesPrint("!!!More than %l names in one object",(LONG)MAXINNAMETREE);
00122             Terminate(-1);
00123         }
00124         nnn = (NAMENODE *)Malloc1(2*((LONG)newsize*sizeof(NAMENODE)),
00125             "extra names in AddName");
00126         c1 = (LONG *)nnn; c2 = (LONG *)nametree->namenode;
00127         i = (nametree->nodefill * sizeof(NAMENODE))/sizeof(LONG);
00128         while ( --i >= 0 ) *c1++ = *c2++;
00129         M_free(nametree->namenode,"nametree->namenode");
00130         nametree->namenode = nnn;
00131         nametree->nodesize = newsize;
00132         n = nametree->namenode+node;
00133     }
00134     *nodenum = newnode = nametree->nodefill++;
00135     nn = nametree->namenode+newnode;
00136     nn->parent = node;
00137     if ( r < 0 ) n->left = newnode; else n->right = newnode;
00138     nn->left = nn->right = -1;
00139     nn->type = type;
00140     nn->number = number;
00141     nn->balance = 0;
00142     i = 1; s = name; while ( *s ) { i++; s++; }
00143     while ( nametree->namefill + i >= nametree->namesize ) { /* Double alloc */
00144         sss = (UBYTE *)Malloc1(2*nametree->namesize,
00145             "extra names in AddName");
00146         s = sss; ss = nametree->namebuffer; j = nametree->namefill;
00147         while ( --j >= 0 ) *s++ = *ss++;
00148         M_free(nametree->namebuffer,"nametree->namebuffer");
00149         nametree->namebuffer = sss;
00150         nametree->namesize *= 2;
00151     }
00152     s = nametree->namebuffer+nametree->namefill;
00153     nn->name = nametree->namefill;
00154     retval = nametree->namefill;
00155     nametree->namefill += i;
00156     while ( *name ) *s++ = *name++;
00157     *s = 0;
00158 /*
00159     Adjust the balance factors
00160 */
00161     while ( node >= 0 ) {
00162         n = nametree->namenode + node;
00163         if ( newnode == n->left ) rr = -1;
00164         else rr = 1;
00165         if ( n->balance == -rr ) { n->balance = 0; return(retval); }
00166         else if ( n->balance == rr ) break;
00167         n->balance = rr;
00168         newnode = node;
00169         node = n->parent;
00170     }
00171     if ( node < 0 ) return(retval);
00172 /*
00173     We have to rebalance the tree. There are two basic operations.
00174     n/node is the unbalanced node. newnode is its child.
00175     rr is the old balance of n/node.
00176 */
```

```
00177     nn = nametree->namenode + newnode;
00178     if ( nn->balance == -rr ) {  /* The difficult case */
00179         if ( rr > 0 ) {
00180             node3 = nn->left;
00181             nnn = nametree->namenode + node3;
00182             nnn->parent = n->parent;
00183             n->parent = nn->parent = node3;
00184             if ( nnn->right >= 0 ) nametree->namenode[nnn->right].parent = newnode;
00185             if ( nnn->left >= 0 ) nametree->namenode[nnn->left].parent = node;
00186             n->right = nnn->left; nnn->left = node;
00187             nn->left = nnn->right; nnn->right = newnode;
00188             if ( nnn->balance > 0 ) { n->balance = -1; nn->balance = 0; }
00189             else if ( nnn->balance == 0 ) { n->balance = nn->balance = 0; }
00190             else                    { nn->balance = 1; n->balance = 0; }
00191         }
00192         else {
00193             node3 = nn->right;
00194             nnn = nametree->namenode + node3;
00195             nnn->parent = n->parent;
00196             n->parent = nn->parent = node3;
00197             if ( nnn->right >= 0 ) nametree->namenode[nnn->right].parent = node;
00198             if ( nnn->left >= 0 ) nametree->namenode[nnn->left].parent = newnode;
00199             n->left = nnn->right; nnn->right = node;
00200             nn->right = nnn->left; nnn->left = newnode;
00201             if ( nnn->balance < 0 ) { n->balance = 1; nn->balance = 0; }
00202             else if ( nnn->balance == 0 ) { n->balance = nn->balance = 0; }
00203             else                    { nn->balance = -1; n->balance = 0; }
00204         }
00205         nnn->balance = 0;
00206         if ( nnn->parent >= 0 ) {
00207             nn = nametree->namenode + nnn->parent;
00208             if ( node == nn->left ) nn->left = node3;
00209             else nn->right = node3;
00210         }
00211         if ( node == nametree->headnode ) nametree->headnode = node3;
00212     }
00213     else if ( nn->balance == rr ) {  /* The easy case */
00214         nn->parent = n->parent; n->parent = newnode;
00215         if ( rr > 0 ) {
00216             if ( nn->left >= 0 ) nametree->namenode[nn->left].parent = node;
00217             n->right = nn->left; nn->left = node;
00218         }
00219         else {
00220             if ( nn->right >= 0 ) nametree->namenode[nn->right].parent = node;
00221             n->left = nn->right; nn->right = node;
00222         }
00223         if ( nn->parent >= 0 ) {
00224             nnn = nametree->namenode + nn->parent;
00225             if ( node == nnn->left ) nnn->left = newnode;
00226             else nnn->right = newnode;
00227         }
00228         nn->balance = n->balance = 0;
00229         if ( node == nametree->headnode ) nametree->headnode = newnode;
00230     }
00231 #ifdef DEBUGON
00232     else {  /* Cannot be. Code here for debugging only */
00233 /* INTERNAL_ERROR_EXCL_START */
00234         MesPrint("!>We ran into an impossible case in AddName\n");
00235         DumpTree(nametree);
00236         Terminate(-1);
00237 /* INTERNAL_ERROR_EXCL_STOP */
00238     }
00239 #endif
00240     return(retval);
00241 }
00242
00243 /*
00244     #] AddName :
00245     #[ GetName :
00246
00247     When AutoDeclare is an active statement.
00248     If par == WITHAUTO and the variable is not found we have to check:
00249     1: that nametree != AC.exprnames && nametree != AC.dollarnames
00250     2: check that the variable is not in AC.exprnames after all.
00251     3: call GetAutoName and return its values.
00252 */
00253
00254 int GetName(NAMETREE *nametree, UBYTE *namein, WORD *number, int par)
00255 {
00256     NAMENODE *n;
00257     int node, newnode, i;
00258     UBYTE *s, *t, *u, *name;
00259 /*  name = ConstructName(namein,0); */
00260     name = namein;
00261     if ( nametree->namenode == 0 || nametree->namefill == 0 ) goto NotFound;
00262     newnode = nametree->headnode;
00263     do {
```

```
00264         node = newnode;
00265         n = nametree->namenode+node;
00266         if ( ( i = StrCmp(name,nametree->namebuffer+n->name) ) < 0 )
00267             newnode = n->left;
00268         else if ( i > 0 ) newnode = n->right;
00269         else {
00270             *number = n->number;
00271             return(n->type);
00272         }
00273     } while ( newnode >= 0 );
00274     s = name;
00275     while ( *s ) s++;
00276     if ( s > name && s[-1] == '_' && nametree == AC.varnames ) {
00277 /*
00278             The Kronecker delta d_ is very special. It is not really a function.
00279 */
00280         if ( s == name+2 && ( *name == 'd' || *name == 'D' ) ) {
00281             *number = DELTA-FUNCTION;
00282             return(CDELTA);
00283         }
00284 /*
00285             Test for N#_? type variables (summed indices)
00286 */
00287         if ( s > name+2 && *name == 'N' ) {
00288             t = name+1; i = 0;
00289             while ( FG.cTable[*t] == 1 ) i = 10*i + *t++ -'0';
00290             if ( s == t+1 ) {
00291                 *number = i + AM.IndDum - AM.OffsetIndex;
00292                 return(CINDEX);
00293             }
00294         }
00295 /*
00296             Now test for any built in object
00297 */
00298         newnode = nametree->headnode;
00299         do {
00300             node = newnode;
00301             n = nametree->namenode+node;
00302             if ( ( i = StrHICmp(name,nametree->namebuffer+n->name) ) < 0 )
00303                 newnode = n->left;
00304             else if ( i > 0 ) newnode = n->right;
00305             else {
00306                 *number = n->number; return(n->type);
00307             }
00308         } while ( newnode >= 0 );
00309 /*
00310             Now we test for the extra symbols of the type STR###_
00311             The string sits in AC.extrasym and is followed by digits.
00312             The name is only legal if the number is in the
00313             range 1,...,cbuf[AM.sbufnum].numrhs
00314 */
00315         t = name; u = AC.extrasym;
00316         while ( *t == *u ) { t++; u++; }
00317         if ( *u == 0 && *t != 0 ) { /* potential hit */
00318             WORD x = 0;
00319             while ( FG.cTable[*t] == 1 ) {
00320                 x = 10*x + (*t++ - '0');
00321             }
00322             if ( *t == '_' && x > 0 && x <= cbuf[AM.sbufnum].numrhs ) { /* Hit */
00323                 *number = MAXVARIABLES-x;
00324                 return(CSYMBOL);
00325             }
00326         }
00327     }
00328 NotFound:;
00329     if ( par != WITHAUTO || nametree == AC.autonames ) return(NAMENOTFOUND);
00330     return(GetAutoName(name,number));
00331 }
00332
00333 /*
00334     #] GetName :
00335     #[ GetFunction :
00336
00337     Gets either a function or a $ that should expand into a function
00338     during runtime. In the case of the $ the value in funnum is -dolnum-1.
00339     The return value is the position after the name of the function or the $.
00340 */
00341  /* UNFINISHED_FEATURE_EXCL_START */
00342 static WORD one = 1;
00343
00344 UBYTE *GetFunction(UBYTE *s,WORD *funnum)
00345 {
00346     int type;
00347     WORD numfun;
00348     UBYTE *t1, c;
00349     if ( *s == '$' ) {
00350         t1 = s+1; while ( FG.cTable[*t1] < 2 ) t1++;
```

```
00351         c = *t1; *t1 = 0;
00352         if ( ( type = GetName(AC.dollarnames,s+1,&numfun,NOAUTO) ) == CDOLLAR ) {
00353             *funnum = -numfun-2;
00354         }
00355         else {
00356             MesPrint("&%s is undefined",s);
00357             numfun = AddDollar(s+1,DOLINDEX,&one,1);
00358             *funnum = 0;
00359         }
00360     }
00361     else {
00362         t1 = SkipAName(s);
00363         c = *t1; *t1 = 0;
00364         if ( ( ( type = GetName(AC.varnames,s,&numfun,WITHAUTO) ) != CFUNCTION )
00365             || ( functions[numfun].spec > 0 ) ) {
00366             MesPrint("&%s should be a regular function",s);
00367             *funnum = 0;
00368             if ( type < 0 ) {
00369                 if ( GetName(AC.exprnames,s,&numfun,NOAUTO) == NAMENOTFOUND )
00370                 AddFunction(s,0,0,0,0,0,-1,-1);
00371             }
00372             *t1 = c;
00373             return(t1);
00374         }
00375         *funnum = numfun+FUNCTION;
00376     }
00377     *t1 = c;
00378     return(t1);
00379 }
00380 /* UNFINISHED_FEATURE_EXCL_STOP */
00381 /*
00382     #] GetFunction :
00383     #[ GetNumber :
00384
00385     Gets either a number or a $ that should expand into a number
00386     during runtime. In the case of the $ the value in num is -dolnum-2.
00387     The return value is the position after the number or the $.
00388 */
00389 /* UNFINISHED_FEATURE_EXCL_START */
00390 UBYTE *GetNumber(UBYTE *s,WORD *num)
00391 {
00392     int type;
00393     WORD numfun;
00394     UBYTE *t1, c;
00395     while ( *s == '+' ) s++;
00396     if ( *s == '$' ) {
00397         t1 = s+1; while ( FG.cTable[*t1] < 2 ) t1++;
00398         c = *t1; *t1 = 0;
00399         if ( ( type = GetName(AC.dollarnames,s+1,&numfun,NOAUTO) ) == CDOLLAR ) {
00400             *num = -numfun-2;
00401         }
00402         else {
00403             MesPrint("&%s is undefined",s);
00404             numfun = AddDollar(s+1,DOLINDEX,&one,1);
00405             *num = -1;
00406         }
00407     }
00408     else if ( *s >= '0' && *s <= '9' ) {
00409         ULONG x = *s++ - '0';
00410         while ( *s >= '0' && *s <= '9' ) { x = 10*x + (*s++-'0'); }
00411         t1 = s;
00412         if ( x >= MAXPOSITIVE ) goto illegal;
00413         *num = (WORD)x;
00414         return(t1);
00415     }
00416     else {
00417         if ( *s == '-' ) { s++; }
00418         if ( *s >= '0' && *s <= '9' ) { while ( *s >= '0' && *s <= '9' ) s++; t1 = s; }
00419         else { t1 = SkipAName(s); }
00420 illegal:
00421         *num = -1;
00422         MesPrint("&Illegal option in Canonicalize statement. Should be a nonnegative number or $
      variable.");
00423         return(t1);
00424     }
00425     *t1 = c;
00426     return(t1);
00427 }
00428 /* UNFINISHED_FEATURE_EXCL_STOP */
00429 /*
00430     #] GetNumber :
00431     #[ GetLastExprName :
00432
00433     When AutoDeclare is an active statement.
00434     If par == WITHAUTO and the variable is not found we have to check:
00435     1: that nametree != AC.exprnames && nametree != AC.dollarnames
00436     2: check that the variable is not in AC.exprnames after all.
```

```
00437     3: call GetAutoName and return its values.
00438 */
00439
00440 int GetLastExprName(UBYTE *name, WORD *number)
00441 {
00442     int i;
00443     EXPRESSIONS e;
00444     for ( i = NumExpressions; i > 0; i-- ) {
00445         e = Expressions+i-1;
00446         if ( StrCmp(AC.exprnames->namebuffer+e->name,name) == 0 ) {
00447             *number = i-1;
00448             return(1);
00449         }
00450     }
00451     return(0);
00452 }
00453
00454 /*
00455     #] GetLastExprName :
00456     #[ GetONname :
00457
00458     Adds the proper offsets, so we do not have to do that in the calling
00459     routine.
00460 */
00461
00462 int GetOName(NAMETREE *nametree, UBYTE *name, WORD *number, int par)
00463 {
00464     int retval = GetName(nametree,name,number,par);
00465     switch ( retval ) {
00466         case CVECTOR: *number += AM.OffsetVector; break;
00467         case CINDEX:  *number += AM.OffsetIndex;  break;
00468         case CFUNCTION: *number += FUNCTION; break;
00469         default: break;
00470     }
00471     return(retval);
00472 }
00473
00474 /*
00475     #] GetONname :
00476     #[ GetAutoName :
00477
00478     This routine gets the automatic declarations
00479 */
00480
00481 int GetAutoName(UBYTE *name, WORD *number)
00482 {
00483     UBYTE *s, c;
00484     int type;
00485     if ( GetName(AC.exprnames,name,number,NOAUTO) != NAMENOTFOUND )
00486                 return(NAMENOTFOUND);
00487     s = name;
00488     while ( *s ) { s++; }
00489     if ( s[-1] == '_' ) {
00490         return(NAMENOTFOUND);
00491     }
00492     while ( s > name ) {
00493         c = *s; *s = 0;
00494         type = GetName(AC.autonames,name,number,NOAUTO);
00495         *s = c;
00496         switch(type) {
00497             case CSYMBOL: {
00498                 SYMBOLS sym = ((SYMBOLS)(AC.AutoSymbolList.lijst)) + *number;
00499                 *number = AddSymbol(name,sym->minpower,sym->maxpower,sym->complex,sym->dimension);
00500                 return(type); }
00501             case CVECTOR: {
00502                 VECTORS vec = ((VECTORS)(AC.AutoVectorList.lijst)) + *number;
00503                 *number = AddVector(name,vec->complex,vec->dimension);
00504                 return(type); }
00505             case CINDEX: {
00506                 INDICES ind = ((INDICES)(AC.AutoIndexList.lijst)) + *number;
00507                 *number = AddIndex(name,ind->dimension,ind->nmin4);
00508                 return(type); }
00509             case CFUNCTION: {
00510                 FUNCTIONS fun = ((FUNCTIONS)(AC.AutoFunctionList.lijst)) + *number;
00511                 *number =
      AddFunction(name,fun->commute,fun->spec,fun->complex,fun->symmetric,fun->dimension,fun->maxnumargs,fun->minnumargs);
00512                 return(type); }
00513             default:
00514                 break;
00515         }
00516         s--;
00517     }
00518     return(NAMENOTFOUND);
00519 }
00520
00521 /*
00522     #] GetAutoName :
```

```
00523     #[ GetVar :
00524 */
00525
00526 int GetVar(UBYTE *name, WORD *type, WORD *number, int wantedtype, int par)
00527 {
00528     WORD funnum;
00529     int typ;
00530     if ( ( typ = GetName(AC.varnames,name,number,par) ) != wantedtype ) {
00531         if ( typ != NAMENOTFOUND ) {
00532             if ( wantedtype == -1 ) {
00533                 *type = typ;
00534                 return(1);
00535             }
00536             NameConflict(typ,name);
00537             MakeDubious(AC.varnames,name,&funnum);
00538             return(-1);
00539         }
00540         if ( ( typ = GetName(AC.exprnames,name,&funnum,par) ) != NAMENOTFOUND ) {
00541             if ( typ == wantedtype || wantedtype == -1 ) {
00542                 *number = funnum; *type = typ; return(1);
00543             }
00544             NameConflict(typ,name);
00545             return(-1);
00546         }
00547         return(NAMENOTFOUND);
00548     }
00549     if ( typ == -1 ) { return(0); }
00550     *type = typ;
00551     return(1);
00552 }
00553
00554 /*
00555     #] GetVar :
00556     #[ EntVar :
00557 */
00558
00559 int EntVar(WORD type, UBYTE *name, WORD x, WORD y, WORD z, WORD d)
00560 {
00561     switch ( type ) {
00562         case CSYMBOL:
00563             return(AddSymbol(name,y,z,x,d));
00564             break;
00565         case CINDEX:
00566             return(AddIndex(name,x,z));
00567             break;
00568         case CVECTOR:
00569             return(AddVector(name,x,d));
00570             break;
00571         case CFUNCTION:
00572             return(AddFunction(name,y,z,x,0,d,-1,-1));
00573             break;
00574         case CSET:
00575             AC.SetList.numtemp++;
00576             return(AddSet(name,d));
00577             break;
00578         case CEXPRESSION:
00579             return(AddExpression(name,x,y));
00580             break;
00581         default:
00582             break;
00583     }
00584     return(-1);
00585 }
00586
00587 /*
00588     #] EntVar :
00589     #[ GetDollar :
00590 */
00591
00592 int GetDollar(UBYTE *name)
00593 {
00594     WORD number;
00595     if ( GetName(AC.dollarnames,name,&number,NOAUTO) == NAMENOTFOUND ) return(-1);
00596     return((int)number);
00597 }
00598
00599 /*
00600     #] GetDollar :
00601     #[ DumpTree :
00602 */
00603 /* DEBUG_EXCL_START */
00604 void DumpTree(NAMETREE *nametree)
00605 {
00606     if ( nametree->headnode >= 0
00607         && nametree->namebuffer && nametree->namenode ) {
00608         DumpNode(nametree,nametree->headnode,0);
00609     }
```

```
00610 }
00611 /* DEBUG_EXCL_STOP */
00612 /*
00613     #] DumpTree :
00614     #[ DumpNode :
00615 */
00616 /* DEBUG_EXCL_START */
00617 void DumpNode(NAMETREE *nametree, WORD node, WORD depth)
00618 {
00619     NAMENODE *n;
00620     int i;
00621     char *name;
00622     n = nametree->namenode + node;
00623     if ( n->left >= 0 ) DumpNode(nametree,n->left,depth+1);
00624     for ( i = 0; i < depth; i++ ) printf(" ");
00625     name = (char *)(nametree->namebuffer+n->name);
00626     printf("%s(%d): {%d}(%d)(%d)[%d]\n",
00627         name,node,n->parent,n->left,n->right,n->balance);
00628     if ( n->right >= 0 ) DumpNode(nametree,n->right,depth+1);
00629 }
00630 /* DEBUG_EXCL_STOP */
00631 /*
00632     #] DumpNode :
00633     #[ CompactifyTree :
00634 */
00635
00636 int CompactifyTree(NAMETREE *nametree,WORD par)
00637 {
00638     NAMETREE newtree;
00639     NAMENODE *n;
00640     LONG i, j, ns, k;
00641     UBYTE *s;
00642
00643     for ( i = 0, j = 0, k = 0, n = nametree->namenode, ns = 0;
00644     i < nametree->nodefill; i++, n++ ) {
00645         if ( n->type != CDELETE ) {
00646             s = nametree->namebuffer+n->name;
00647             while ( *s ) { s++; ns++; }
00648             j++;
00649         }
00650         else k++;
00651     }
00652     if ( k == 0 ) return(0);
00653     if ( j == 0 ) {
00654         if ( nametree->namebuffer ) M_free(nametree->namebuffer,"nametree->namebuffer");
00655         if ( nametree->namenode ) M_free(nametree->namenode,"nametree->namenode");
00656         nametree->namebuffer = 0;
00657         nametree->namenode = 0;
00658         nametree->namesize = nametree->namefill =
00659         nametree->nodesize = nametree->nodefill =
00660         nametree->oldnamefill = nametree->oldnodefill = 0;
00661         nametree->globalnamefill = nametree->globalnodefill =
00662         nametree->clearnamefill = nametree->clearnodefill = 0;
00663         nametree->headnode = -1;
00664         return(0);
00665     }
00666     ns += j;
00667     if ( j < 10 ) j = 10;
00668     if ( ns < 100 ) ns = 100;
00669     newtree.namenode = (NAMENODE *)Malloc1(2*j*sizeof(NAMENODE),"compactify namestree");
00670     newtree.nodefill = 0; newtree.nodesize = 2*j;
00671     newtree.namebuffer = (UBYTE *)Malloc1(2*ns,"compactify namestree");
00672     newtree.namefill = 0; newtree.namesize = 2*ns;
00673     CopyTree(&newtree,nametree,nametree->headnode,par);
00674     newtree.namenode[newtree.nodefill>>1].parent = -1;
00675     LinkTree(&newtree,(WORD)0,newtree.nodefill);
00676     newtree.headnode = newtree.nodefill >> 1;
00677     M_free(nametree->namebuffer,"nametree->namebuffer");
00678     M_free(nametree->namenode,"nametree->namenode");
00679     nametree->namebuffer  = newtree.namebuffer;
00680     nametree->namenode    = newtree.namenode;
00681     nametree->namesize    = newtree.namesize;
00682     nametree->namefill    = newtree.namefill;
00683     nametree->nodesize    = newtree.nodesize;
00684     nametree->nodefill    = newtree.nodefill;
00685     nametree->oldnamefill = newtree.namefill;
00686     nametree->oldnodefill = newtree.nodefill;
00687     nametree->headnode    = newtree.headnode;
00688
00689 /*  DumpTree(nametree); */
00690     return(0);
00691 }
00692
00693 /*
00694     #] CompactifyTree :
00695     #[ CopyTree :
00696 */
```

```
00697
00698 void CopyTree(NAMETREE *newtree, NAMETREE *oldtree, WORD node, WORD par)
00699 {
00700     NAMENODE *n, *m;
00701     UBYTE *s, *t;
00702     n = oldtree->namenode+node;
00703     if ( n->left >= 0 ) CopyTree(newtree,oldtree,n->left,par);
00704     if ( n->type != CDELETE ) {
00705         m = newtree->namenode+newtree->nodefill;
00706         m->type = n->type;
00707         m->number = n->number;
00708         m->name = newtree->namefill;
00709         m->left = m->right = -1;
00710         m->balance = 0;
00711         switch ( n->type ) {
00712             case CSYMBOL:
00713                 if ( par == AUTONAMES ) {
00714                     autosymbols[n->number].name = newtree->namefill;
00715                     autosymbols[n->number].node = newtree->nodefill;
00716                 }
00717                 else {
00718                     symbols[n->number].name = newtree->namefill;
00719                     symbols[n->number].node = newtree->nodefill;
00720                 }
00721                 break;
00722             case CINDEX :
00723                 if ( par == AUTONAMES ) {
00724                     autoindices[n->number].name = newtree->namefill;
00725                     autoindices[n->number].node = newtree->nodefill;
00726                 }
00727                 else {
00728                     indices[n->number].name = newtree->namefill;
00729                     indices[n->number].node = newtree->nodefill;
00730                 }
00731                 break;
00732             case CVECTOR:
00733                 if ( par == AUTONAMES ) {
00734                     autovectors[n->number].name = newtree->namefill;
00735                     autovectors[n->number].node = newtree->nodefill;
00736                 }
00737                 else {
00738                     vectors[n->number].name = newtree->namefill;
00739                     vectors[n->number].node = newtree->nodefill;
00740                 }
00741                 break;
00742             case CFUNCTION:
00743                 if ( par == AUTONAMES ) {
00744                     autofunctions[n->number].name = newtree->namefill;
00745                     autofunctions[n->number].node = newtree->nodefill;
00746                 }
00747                 else {
00748                     functions[n->number].name = newtree->namefill;
00749                     functions[n->number].node = newtree->nodefill;
00750                 }
00751                 break;
00752             case CSET:
00753                 Sets[n->number].name = newtree->namefill;
00754                 Sets[n->number].node = newtree->nodefill;
00755                 break;
00756             case CEXPRESSION:
00757                 Expressions[n->number].name = newtree->namefill;
00758                 Expressions[n->number].node = newtree->nodefill;
00759                 break;
00760             case CDUBIOUS:
00761                 Dubious[n->number].name = newtree->namefill;
00762                 Dubious[n->number].node = newtree->nodefill;
00763                 break;
00764             case CDOLLAR:
00765                 Dollars[n->number].name = newtree->namefill;
00766                 Dollars[n->number].node = newtree->nodefill;
00767                 break;
00768             default:
00769 /* INTERNAL_ERROR_EXCL_START */
00770                 MesPrint("!>Illegal variable type in CopyTree: %d",n->type);
00771                 break;
00772 /* INTERNAL_ERROR_EXCL_STOP */
00773         }
00774         newtree->nodefill++;
00775         s = newtree->namebuffer + newtree->namefill;
00776         t = oldtree->namebuffer + n->name;
00777         while ( *t ) { *s++ = *t++; newtree->namefill++; }
00778         *s = 0; newtree->namefill++;
00779     }
00780     if ( n->right >= 0 ) CopyTree(newtree,oldtree,n->right,par);
00781 }
00782
00783 /*
```

```
00784     #] CopyTree :
00785     #[ LinkTree :
00786 */
00787
00788 void LinkTree(NAMETREE *tree, WORD offset, WORD numnodes)
00789 {
00790 /*
00791     Makes the tree into a binary tree
00792 */
00793     int med,numleft,numright,medleft,medright;
00794     med = numnodes » 1;
00795     numleft = med;
00796     numright = numnodes - med - 1;
00797     medleft = numleft » 1;
00798     medright = ( numright » 1 ) + med + 1;
00799     if ( numleft > 0 ) {
00800         tree->namenode[offset+med].left = offset+medleft;
00801         tree->namenode[offset+medleft].parent = offset+med;
00802     }
00803     if ( numright > 0 ) {
00804         tree->namenode[offset+med].right = offset+medright;
00805         tree->namenode[offset+medright].parent = offset+med;
00806     }
00807     if ( numleft  > 0 ) LinkTree(tree,offset,numleft);
00808     if ( numright > 0 ) LinkTree(tree,offset+med+1,numright);
00809     while ( numleft && numright ) { numleft »= 1; numright »= 1; }
00810     if ( numleft ) tree->namenode[offset+med].balance = -1;
00811     else if ( numright ) tree->namenode[offset+med].balance = 1;
00812 }
00813
00814 /*
00815     #] LinkTree :
00816     #[ MakeNameTree :
00817 */
00818
00819 NAMETREE *MakeNameTree(void)
00820 {
00821     NAMETREE *n;
00822     n = (NAMETREE *)Malloc1(sizeof(NAMETREE),"new nametree");
00823     n->namebuffer = 0;
00824     n->namenode = 0;
00825     n->namesize = n->namefill = n->nodesize = n->nodefill =
00826     n->oldnamefill = n->oldnodefill = 0;
00827     n->globalnamefill = n->globalnodefill =
00828     n->clearnamefill = n->clearnodefill = 0;
00829     n->headnode = -1;
00830     return(n);
00831 }
00832
00833 /*
00834     #] MakeNameTree :
00835     #[ FreeNameTree :
00836 */
00837 /* UNFINISHED_FEATURE_EXCL_START */
00838 void FreeNameTree(NAMETREE *n)
00839 {
00840     if ( n ) {
00841         if ( n->namebuffer ) M_free(n->namebuffer,"nametree->namebuffer");
00842         if ( n->namenode ) M_free(n->namenode,"nametree->namenode");
00843         M_free(n,"nametree");
00844     }
00845 }
00846 /* UNFINISHED_FEATURE_EXCL_STOP */
00847 /*
00848     #] FreeNameTree :
00849
00850     #[ WildcardNames :
00851 */
00852
00853 void ClearWildcardNames(void)
00854 {
00855     AC.NumWildcardNames = 0;
00856 }
00857
00858 int AddWildcardName(UBYTE *name)
00859 {
00860     GETIDENTITY
00861     int size = 0, tocopy, i;
00862     UBYTE *s = name, *t, *newbuffer;
00863     while ( *s ) { s++; size++; }
00864     for ( i = 0, t = AC.WildcardNames; i < AC.NumWildcardNames; i++ ) {
00865         s = name;
00866         while ( ( *s == *t ) && *s ) { s++; t++; }
00867         if ( *s == 0 && *t == 0 ) return(i+1);
00868         while ( *t ) t++;
00869         t++;
00870     }
```

```
00871     tocopy = t - AC.WildcardNames;
00872     if ( tocopy + size + 1 > AC.WildcardBufferSize ) {
00873         if ( AC.WildcardBufferSize == 0 ) {
00874             AC.WildcardBufferSize = size+1;
00875             if ( AC.WildcardBufferSize < 100 ) AC.WildcardBufferSize = 100;
00876         }
00877         else if ( size+1 >= AC.WildcardBufferSize ) {
00878             AC.WildcardBufferSize += size+1;
00879         }
00880         else {
00881             AC.WildcardBufferSize *= 2;
00882         }
00883         newbuffer = (UBYTE *)Malloc1((LONG)AC.WildcardBufferSize,"argument list names");
00884         t = newbuffer;
00885         if ( AC.WildcardNames ) {
00886             s = AC.WildcardNames;
00887             while ( tocopy > 0 ) { *t++ = *s++; tocopy--; }
00888             M_free(AC.WildcardNames,"AC.WildcardNames");
00889         }
00890         AC.WildcardNames = newbuffer;
00891         M_free(AT.WildArgTaken,"AT.WildArgTaken");
00892         AT.WildArgTaken = (WORD *)Malloc1((LONG)AC.WildcardBufferSize*sizeof(WORD)/2
00893                 ,"argument list names");
00894     }
00895     s = name;
00896     while ( *s ) *t++ = *s++;
00897     *t = 0;
00898     AC.NumWildcardNames++;
00899     return(AC.NumWildcardNames);
00900 }
00901
00902 int GetWildcardName(UBYTE *name)
00903 {
00904     UBYTE *s, *t;
00905     int i;
00906     for ( i = 0, t = AC.WildcardNames; i < AC.NumWildcardNames; i++ ) {
00907         s = name;
00908         while ( ( *s == *t ) && *s ) { s++; t++; }
00909         if ( *s == 0 && *t == 0 ) return(i+1);
00910         while ( *t ) t++;
00911         t++;
00912     }
00913     return(0);
00914 }
00915
00916 /*
00917     #] WildcardNames :
00918
00919     #[ AddSymbol :
00920
00921     The actual addition. Special routine for additions 'on the fly'
00922 */
00923
00924 int AddSymbol(UBYTE *name, int minpow, int maxpow, int cplx, int dim)
00925 {
00926     int nodenum, numsymbol = AC.Symbols->num;
00927     UBYTE *s = name;
00928     SYMBOLS sym = (SYMBOLS)FromVarList(AC.Symbols);
00929     bzero(sym,sizeof(struct SyMbOl));
00930     sym->name = AddName(*AC.activenames,name,CSYMBOL,numsymbol,&nodenum);
00931     sym->minpower = minpow;
00932     sym->maxpower = maxpow;
00933     sym->complex  = cplx;
00934     sym->flags    = 0;
00935     sym->node     = nodenum;
00936     sym->dimension= dim;
00937     while ( *s ) s++;
00938     sym->namesize = (s-name)+1;
00939     return(numsymbol);
00940 }
00941
00942 /*
00943     #] AddSymbol :
00944     #[ CoSymbol :
00945
00946     Symbol declarations.   name[#{R|I|C}][([min]:[max])]
00947     Note that we know already that the parentheses match properly
00948 */
00949
00950 int CoSymbol(UBYTE *s)
00951 {
00952     int type, error = 0, minpow, maxpow, cplx, sgn, dim;
00953     WORD numsymbol;
00954     UBYTE *name, *oldc, c, cc;
00955     do {
00956         minpow = -MAXPOWER;
00957         maxpow =  MAXPOWER;
```

```
00958         cplx = 0;
00959         dim = 0;
00960         name = s;
00961         if ( ( s = SkipAName(s) ) == 0 ) {
00962 IllForm:    MesPrint("&Illegally formed name in symbol statement");
00963             error = 1;
00964             s = SkipField(name,0);
00965             goto eol;
00966         }
00967         oldc = s; cc = c = *s; *s = 0;
00968         if ( TestName(name) ) { *s = c; goto IllForm; }
00969         if ( cc == '#' ) {
00970             s++;
00971                 if ( tolower(*s) == 'r' ) cplx = VARTYPENONE;
00972             else if ( tolower(*s) == 'c' ) cplx = VARTYPECOMPLEX;
00973             else if ( tolower(*s) == 'i' ) cplx = VARTYPEIMAGINARY;
00974             else if ( ( ( *s == '-' || *s == '+' || *s == '=' )
00975                  && ( s[1] >= '0' && s[1] <= '9' ) )
00976                  || ( *s >= '0' && *s <= '9' ) ) {
00977                 LONG x;
00978                 sgn = 0;
00979                 if ( *s == '-' ) { sgn = VARTYPEMINUS; s++; }
00980                 else if ( *s == '+' || *s == '=' ) { sgn = 0; s++; }
00981                 x = *s -'0';
00982                 while ( s[1] >= '0' && s[1] <= '9' ) {
00983                     x = 10*x + (s[1] - '0'); s++;
00984                 }
00985                 if ( x >= MAXPOWER || x <= 1 ) {
00986                     MesPrint("&Illegal value for root of unity %s",name);
00987                     error = 1;
00988                 }
00989                 else {
00990                     maxpow = x;
00991                 }
00992                 cplx = VARTYPEROOTOFUNITY | sgn;
00993             }
00994             else {
00995                 MesPrint("&Illegal specification for complexity of symbol %s",name);
00996                 *oldc = c;
00997                 error = 1;
00998                 s = SkipField(s,0);
00999                 goto eol;
01000             }
01001             s++; cc = *s;
01002         }
01003         if ( cc == '{' ) {
01004             s++;
01005             if ( ( *s == 'd' || *s == 'D' ) && s[1] == '=' ) {
01006                 s += 2;
01007                 if ( *s == '-' || *s == '+' || FG.cTable[*s] == 1 ) {
01008                     ParseSignedNumber(dim,s)
01009                     if ( dim < -HALFMAX || dim > HALFMAX ) {
01010                         MesPrint("&Warning: dimension of %s (%d) out of range"
01011                         ,name,dim);
01012                     }
01013                 }
01014                 if ( *s != '}' ) goto IllDim;
01015                 else s++;
01016             }
01017             else {
01018 IllDim:         MesPrint("&Error: Illegal dimension field for variable %s",name);
01019                 error = 1;
01020                 s = SkipField(s,0);
01021                 goto eol;
01022             }
01023             cc = *s;
01024         }
01025         if ( cc == '(' ) {
01026             if ( ( cplx & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY ) {
01027                 MesPrint("&Root of unity property for %s cannot be combined with power
      restrictions",name);
01028                 error = 1;
01029             }
01030             s++;
01031             if ( *s == '-' || *s == '+' || FG.cTable[*s] == 1 ) {
01032                 ParseSignedNumber(minpow,s)
01033                 if ( minpow < -MAXPOWER ) {
01034                     minpow = -MAXPOWER;
01035                     if ( AC.WarnFlag )
01036                     MesPrint("&Warning: minimum power of %s corrected to %d"
01037                     ,name,-MAXPOWER);
01038                 }
01039             }
01040             if ( *s != ':' ) {
01041 skippar:        error = 1;
01042                 s = SkipField(s,1);
01043                 goto eol;
```

```
01044             }
01045             else s++;
01046             if ( *s == '-' || *s == '+' || FG.cTable[*s] == 1 ) {
01047                 ParseSignedNumber(maxpow,s)
01048                 if ( maxpow > MAXPOWER ) {
01049                     maxpow = MAXPOWER;
01050                     if ( AC.WarnFlag )
01051                     MesPrint("&Warning: maximum power of %s corrected to %d"
01052                     ,name,MAXPOWER);
01053                 }
01054             }
01055             if ( *s != ')' ) goto skippar;
01056             s++;
01057         }
01058         if ( ( AC.AutoDeclareFlag == 0 &&
01059          ( ( type = GetName(AC.exprnames,name,&numsymbol,NOAUTO) )
01060                   != NAMENOTFOUND ) )
01061         || ( ( type = GetName(*(AC.activenames),name,&numsymbol,NOAUTO) ) != NAMENOTFOUND ) ) {
01062             if ( type != CSYMBOL ) error = NameConflict(type,name);
01063             else {
01064                 SYMBOLS sym = (SYMBOLS)(AC.Symbols->lijst) + numsymbol;
01065                 if ( ( numsymbol == AC.lPolyFunVar ) && ( AC.lPolyFunType > 0 )
01066                      && ( AC.lPolyFun != 0 ) && ( minpow > -MAXPOWER || maxpow < MAXPOWER ) ) {
01067                     MesPrint("&The symbol %s is used by power expansions in the PolyRatFun!",name);
01068                     error = 1;
01069                 }
01070                 sym->complex  = cplx;
01071                 sym->minpower = minpow;
01072                 sym->maxpower = maxpow;
01073                 sym->dimension= dim;
01074             }
01075         }
01076         else {
01077             AddSymbol(name,minpow,maxpow,cplx,dim);
01078         }
01079         *oldc = c;
01080 eol:    while ( *s == ',' ) s++;
01081     } while ( *s );
01082     return(error);
01083 }
01084
01085 /*
01086     #] CoSymbol :
01087     #[ AddIndex :
01088
01089     The actual addition. Special routine for additions 'on the fly'
01090 */
01091
01092 int AddIndex(UBYTE *name, int dim, int dim4)
01093 {
01094     int nodenum, numindex = AC.Indices->num;
01095     INDICES ind = (INDICES)FromVarList(AC.Indices);
01096     UBYTE *s = name;
01097     bzero(ind,sizeof(struct InDeX));
01098     ind->name      = AddName(*AC.activenames,name,CINDEX,numindex,&nodenum);
01099     ind->type      = 0;
01100     ind->dimension = dim;
01101     ind->flags     = 0;
01102     ind->nmin4     = dim4;
01103     ind->node      = nodenum;
01104     while ( *s ) s++;
01105     ind->namesize = (s-name)+1;
01106     return(numindex);
01107 }
01108
01109 /*
01110     #] AddIndex :
01111     #[ CoIndex :
01112
01113     Index declarations. name[={number|symbol[:othersymbol]}]
01114 */
01115
01116 int CoIndex(UBYTE *s)
01117 {
01118     int type, error = 0, dim, dim4;
01119     WORD numindex;
01120     UBYTE *name, *oldc, c;
01121     do {
01122         dim = AC.lDefDim;
01123         dim4 = AC.lDefDim4;
01124         name = s;
01125         if ( ( s = SkipAName(s) ) == 0 ) {
01126 IllForm:    MesPrint("&Illegally formed name in index statement");
01127             error = 1;
01128             s = SkipField(name,0);
01129             goto eol;
01130         }
```

```
01131         oldc = s; c = *s; *s = 0;
01132         if ( TestName(name) ) { *s = c; goto IllForm; }
01133         if ( c == '=' ) {
01134             s++;
01135             if ( ( s = DoDimension(s,&dim,&dim4) ) == 0 ) {
01136                 *oldc = c;
01137                 error = 1;
01138                 s = SkipField(name,0);
01139                 goto eol;
01140             }
01141         }
01142         if ( ( AC.AutoDeclareFlag == 0 &&
01143         ( ( type = GetName(AC.exprnames,name,&numindex,NOAUTO) )
01144          != NAMENOTFOUND ) )
01145         || ( ( type = GetName(*(AC.activenames),name,&numindex,NOAUTO) ) != NAMENOTFOUND ) ) {
01146             if ( type != CINDEX ) error = NameConflict(type,name);
01147             else { /* reset the dimensions */
01148                 indices[numindex].dimension = dim;
01149                 indices[numindex].nmin4     = dim4;
01150             }
01151         }
01152         else AddIndex(name,dim,dim4);
01153         *oldc = c;
01154 eol:    while ( *s == ',' ) s++;
01155     } while ( *s );
01156     return(error);
01157 }
01158
01159 /*
01160     #] CoIndex :
01161     #[ DoDimension :
01162 */
01163
01164 UBYTE *DoDimension(UBYTE *s, int *dim, int *dim4)
01165 {
01166     UBYTE c, *t = s;
01167     int type, error = 0;
01168     WORD numsymbol;
01169     NAMETREE **oldtree = AC.activenames;
01170     LIST* oldsymbols = AC.Symbols;
01171     *dim4 = -NMIN4SHIFT;
01172     if ( FG.cTable[*s] == 1 ) {
01173 retry:
01174         ParseNumber(*dim,s)
01175 #if ( BITSINWORD/8 < 4 )
01176         if ( *dim >= (1 << (BITSINWORD-1)) ) goto illeg;
01177 #endif
01178         *dim4 = *dim - 4;
01179         return(s);
01180     }
01181     else if ( ( (FG.cTable[*s] == 0 ) || ( *s == '[' ) )
01182         && ( s = SkipAName(s) ) != 0 ) {
01183         AC.activenames = &(AC.varnames);
01184         AC.Symbols = &(AC.SymbolList);
01185         c = *s; *s = 0;
01186         if ( ( ( type = GetName(AC.exprnames,t,&numsymbol,NOAUTO) ) != NAMENOTFOUND )
01187         || ( ( type = GetName(AC.varnames,t,&numsymbol,WITHAUTO) ) != NAMENOTFOUND ) ) {
01188             if ( type != CSYMBOL ) error = NameConflict(type,t);
01189         }
01190         else {
01191             numsymbol = AddSymbol(t,-MAXPOWER,MAXPOWER,0,0);
01192             if ( AC.WarnFlag )
01193             MesPrint("&Warning: Implicit declaration of %s as a symbol",t);
01194         }
01195         *dim = -numsymbol;
01196         if ( ( *s = c ) == ':' ) {
01197             s++;
01198             t = s;
01199             if ( ( s = SkipAName(s) ) == 0 ) goto illeg;
01200             if ( ( ( type = GetName(AC.exprnames,t,&numsymbol,NOAUTO) ) != NAMENOTFOUND )
01201             || ( ( type = GetName(AC.varnames,t,&numsymbol,WITHAUTO) ) != NAMENOTFOUND ) ) {
01202                 if ( type != CSYMBOL ) error = NameConflict(type,t);
01203             }
01204             else {
01205                 numsymbol = AddSymbol(t,-MAXPOWER,MAXPOWER,0,0);
01206                 if ( AC.WarnFlag )
01207                 MesPrint("&Warning: Implicit declaration of %s as a symbol",t);
01208             }
01209             *dim4 = -numsymbol-NMIN4SHIFT;
01210         }
01211     }
01212     else if ( *s == '+' && FG.cTable[s[1]] == 1 ) {
01213         s++; goto retry;
01214     }
01215     else {
01216 illeg:  MesPrint("&Illegal dimension specification. Should be number >= 0, symbol or symbol:symbol");
01217         return(0);
```

```
01218     }
01219     AC.Symbols = oldsymbols;
01220     AC.activenames = oldtree;
01221     if ( error ) return(0);
01222     return(s);
01223 }
01224
01225 /*
01226     #] DoDimension :
01227     #[ CoDimension :
01228 */
01229
01230 int CoDimension(UBYTE *s)
01231 {
01232     s = DoDimension(s,&AC.lDefDim,&AC.lDefDim4);
01233     if ( s == 0 ) return(1);
01234     if ( *s != 0 ) {
01235         MesPrint("&Argument of dimension statement should be number >= 0, symbol or symbol:symbol");
01236         return(1);
01237     }
01238     return(0);
01239 }
01240
01241 /*
01242     #] CoDimension :
01243     #[ AddVector :
01244
01245     The actual addition. Special routine for additions 'on the fly'
01246 */
01247
01248 int AddVector(UBYTE *name, int cplx, int dim)
01249 {
01250     int nodenum, numvector = AC.Vectors->num;
01251     VECTORS v = (VECTORS)FromVarList(AC.Vectors);
01252     UBYTE *s = name;
01253     bzero(v,sizeof(struct VeCtOr));
01254     v->name = AddName(*AC.activenames,name,CVECTOR,numvector,&nodenum);
01255     v->complex = cplx;
01256     v->node    = nodenum;
01257     v->dimension = dim;
01258     v->flags = 0;
01259     while ( *s ) s++;
01260     v->namesize = (s-name)+1;
01261     return(numvector);
01262 }
01263
01264 /*
01265     #] AddVector :
01266     #[ CoVector :
01267
01268     Vector declarations. The descriptor string is "(,%n)"
01269 */
01270
01271 int CoVector(UBYTE *s)
01272 {
01273     int type, error = 0, dim;
01274     WORD numvector;
01275     UBYTE *name, c, *endname;
01276     do {
01277         name = s;
01278         dim = 0;
01279         if ( ( s = SkipAName(s) ) == 0 ) {
01280 IllForm:    MesPrint("&Illegally formed name in vector statement");
01281             error = 1;
01282             s = SkipField(s,0);
01283         }
01284         else {
01285             c = *s; *s = 0, endname = s;
01286             if ( TestName(name) ) { *s = c; goto IllForm; }
01287             if ( c == '{' ) {
01288                 s++;
01289                 if ( ( *s == 'd' || *s == 'D' ) && s[1] == '=' ) {
01290                     s += 2;
01291                     if ( *s == '-' || *s == '+' || FG.cTable[*s] == 1 ) {
01292                         ParseSignedNumber(dim,s)
01293                         if ( dim < -HALFMAX || dim > HALFMAX ) {
01294                             MesPrint("&Warning: dimension of %s (%d) out of range"
01295                             ,name,dim);
01296                         }
01297                     }
01298                     if ( *s != '}' ) goto IllDim;
01299                     else s++;
01300                 }
01301                 else {
01302 IllDim:             MesPrint("&Error: Illegal dimension field for variable %s",name);
01303                     error = 1;
01304                     s = SkipField(s,0);
```

```
01305                       while ( *s == ',' ) s++;
01306                       continue;
01307                   }
01308               }
01309               if ( ( AC.AutoDeclareFlag == 0 &&
01310                 ( ( type = GetName(AC.exprnames,name,&numvector,NOAUTO) )
01311                  != NAMENOTFOUND ) )
01312               || ( ( type = GetName(*(AC.activenames),name,&numvector,NOAUTO) ) != NAMENOTFOUND ) ) {
01313                   if ( type != CVECTOR ) error = NameConflict(type,name);
01314               }
01315               else AddVector(name,0,dim);
01316               *endname = c;
01317           }
01318           while ( *s == ',' ) s++;
01319       } while ( *s );
01320       return(error);
01321   }
01322
01323   /*
01324       #] CoVector :
01325       #[ AddFunction :
01326
01327       The actual addition. Special routine for additions 'on the fly'
01328   */
01329
01330   int AddFunction(UBYTE *name, int comm, int istensor, int cplx, int symprop, int dim, int argmax, int
      argmin)
01331   {
01332       int nodenum, numfunction = AC.Functions->num;
01333       FUNCTIONS fun = (FUNCTIONS)FromVarList(AC.Functions);
01334       UBYTE *s = name;
01335       bzero(fun,sizeof(struct FuNcTiOn));
01336       fun->name = AddName(*AC.activenames,name,CFUNCTION,numfunction,&nodenum);
01337       fun->commute = comm;
01338       fun->spec = istensor;
01339       fun->complex  = cplx;
01340       fun->tabl = 0;
01341       fun->flags = 0;
01342       fun->node = nodenum;
01343       fun->symminfo = 0;
01344       fun->symmetric = symprop;
01345       fun->dimension = dim;
01346       fun->maxnumargs = argmax;
01347       fun->minnumargs = argmin;
01348       while ( *s ) s++;
01349       fun->namesize = (s-name)+1;
01350       return(numfunction);
01351   }
01352
01353   /*
01354       #] AddFunction :
01355       #[ CoCommuteInSet :
01356
01357       Commuting,f1,...,fn;
01358   */
01359
01360   int CoCommuteInSet(UBYTE *s)
01361   {
01362       UBYTE *name, *ss, c, *start = s;
01363       WORD number, type, *g, *gg;
01364       int error = 0, i, len = StrLen(s), len2 = 0;
01365       if ( AC.CommuteInSet != 0 ) {
01366           g = AC.CommuteInSet;
01367           while ( *g ) g += *g;
01368           len2 = g - AC.CommuteInSet;
01369           if ( len2+len+3 > AC.SizeCommuteInSet ) {
01370               gg = (WORD *)Malloc1((len2+len+3)*sizeof(WORD),"CommuteInSet");
01371               for ( i = 0; i < len2; i++ ) gg[i] = AC.CommuteInSet[i];
01372               gg[len2] = 0;
01373               M_free(AC.CommuteInSet,"CommuteInSet");
01374               AC.CommuteInSet = gg;
01375               AC.SizeCommuteInSet = len+len2+3;
01376               g = AC.CommuteInSet+len2;
01377           }
01378       }
01379       else {
01380           AC.SizeCommuteInSet = len+2;
01381           g = AC.CommuteInSet = (WORD *)Malloc1((len+3)*sizeof(WORD),"CommuteInSet");
01382           *g = 0;
01383       }
01384       gg = g++;
01385       ss = s-1;
01386       for(;;) {
01387           while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
01388           if ( *s == 0 ) {
01389               if ( s - start >= len ) break;
01390               *s = '}'; s++;
```

```
01391             *g = 0;
01392             *gg = g-gg;
01393             if ( *gg < 2 ) {
01394                 MesPrint("&There should be at least two noncommuting functions or tensors in a
    commuting statement.");
01395                 error = 1;
01396             }
01397             else if ( *gg == 2 ) {
01398                 gg[2] = gg[1]; gg[3] = 0; gg[0] = 3;
01399             }
01400             gg = g++;
01401             continue;
01402         }
01403         if ( s > ss ) {
01404             if ( *s != '{' ) {
01405                 MesPrint("&The CommuteInSet statement should have sets enclosed in {}.");
01406                 error = 1;
01407                 break;
01408             }
01409             ss = s;
01410             SKIPBRA2(ss) /* Note that parentheses were tested before */
01411             *ss = 0;
01412             s++;
01413         }
01414         name = s;
01415         s = SkipAName(s);
01416         c = *s; *s = 0;
01417         if ( ( type = GetName(AC.varnames,name,&number,NOAUTO) ) != CFUNCTION ) {
01418             MesPrint("&%s is not a function or tensor",name);
01419             error = 1;
01420         }
01421         else if ( functions[number].commute == 0 ){
01422             MesPrint("&%s is not a noncommuting function or tensor",name);
01423             error = 1;
01424         }
01425         else {
01426             *g++ = number+FUNCTION;
01427             functions[number].flags |= COULDCOMMUTE;
01428             if ( number+FUNCTION >= GAMMA && number+FUNCTION <= GAMMASEVEN ) {
01429                 functions[GAMMA-FUNCTION].flags |= COULDCOMMUTE;
01430                 functions[GAMMAI-FUNCTION].flags |= COULDCOMMUTE;
01431                 functions[GAMMAFIVE-FUNCTION].flags |= COULDCOMMUTE;
01432                 functions[GAMMASIX-FUNCTION].flags |= COULDCOMMUTE;
01433                 functions[GAMMASEVEN-FUNCTION].flags |= COULDCOMMUTE;
01434             }
01435         }
01436         *s = c;
01437     }
01438     return(error);
01439 }
01440
01441 /*
01442     #] CoCommuteInSet :
01443     #[ CoFunction + ...:
01444
01445     Function declarations.
01446     The second parameter indicates commutation properties.
01447     The third parameter tells whether we have a tensor.
01448 */
01449
01450 int CoFunction(UBYTE *s, int comm, int istensor)
01451 {
01452     int type, error = 0, cplx, symtype, dim, argmax, argmin;
01453     WORD numfunction, reverseorder = 0, addone;
01454     UBYTE *name, *oldc, *par, c, cc;
01455     do {
01456         symtype = cplx = 0, argmin = argmax = -1;
01457         dim = 0;
01458         name = s;
01459         if ( ( s = SkipAName(s) ) == 0 ) {
01460 IllForm:    MesPrint("&Illegally formed function/tensor name");
01461             error = 1;
01462             s = SkipField(name,0);
01463             goto eol;
01464         }
01465         oldc = s; cc = c = *s; *s = 0;
01466         if ( TestName(name) ) { *s = c; goto IllForm; }
01467         if ( c == '#' ) {
01468             s++;
01469                 if ( tolower(*s) == 'r' ) cplx = VARTYPENONE;
01470             else if ( tolower(*s) == 'c' ) cplx = VARTYPECOMPLEX;
01471             else if ( tolower(*s) == 'i' ) cplx = VARTYPEIMAGINARY;
01472             else {
01473                 MesPrint("&Illegal specification for complexity of %s",name);
01474                 *oldc = c;
01475                 error = 1;
01476                 s = SkipField(s,0);
```

```
01477                 goto eol;
01478             }
01479             s++; cc = *s;
01480         }
01481         if ( cc == '{' ) {
01482             s++;
01483             if ( ( *s == 'd' || *s == 'D' ) && s[1] == '=' ) {
01484                 s += 2;
01485                 if ( *s == '-' || *s == '+' || FG.cTable[*s] == 1 ) {
01486                     ParseSignedNumber(dim,s)
01487                     if ( dim < -HALFMAX || dim > HALFMAX ) {
01488                         MesPrint("&Warning: dimension of %s (%d) out of range"
01489                         ,name,dim);
01490                     }
01491                 }
01492                 if ( *s != '}' ) goto IllDim;
01493                 else s++;
01494             }
01495             else {
01496 IllDim:         MesPrint("&Error: Illegal dimension field for variable %s",name);
01497                 error = 1;
01498                 s = SkipField(s,0);
01499                 goto eol;
01500             }
01501             cc = *s;
01502         }
01503         if ( cc == '(' ) {
01504             s++;
01505             if ( *s == '-' ) {
01506                 reverseorder = REVERSEORDER;
01507                 s++;
01508             }
01509             else {
01510                 reverseorder = 0;
01511             }
01512             par = s;
01513             while ( FG.cTable[*s] == 0 ) s++;
01514             cc = *s; *s = 0;
01515             if ( s <= par ) {
01516 illegsym:       *s = cc;
01517                 MesPrint("&Illegal specification for symmetry of %s",name);
01518                 *oldc = c;
01519                 error = 1;
01520                 s = SkipField(s,1);
01521                 goto eol;
01522             }
01523             if ( StrICont(par,(UBYTE *)"symmetric") == 0 ) symtype = SYMMETRIC;
01524             else if ( StrICont(par,(UBYTE *)"antisymmetric") == 0 ) symtype = ANTISYMMETRIC;
01525             else if ( ( StrICont(par,(UBYTE *)"cyclesymmetric") == 0 )
01526             || ( StrICont(par,(UBYTE *)"cyclic") == 0 ) ) symtype = CYCLESYMMETRIC;
01527             else if ( ( StrICont(par,(UBYTE *)"rcyclesymmetric") == 0 )
01528             || ( StrICont(par,(UBYTE *)"rcyclic") == 0 )
01529             || ( StrICont(par,(UBYTE *)"reversecyclic") == 0 ) ) symtype = RCYCLESYMMETRIC;
01530             else goto illegsym;
01531             *s = cc;
01532             if ( *s != ')' || ( s[1] && s[1] != ',' && s[1] != '<' ) ) {
01533                 Warning("Excess information in symmetric properties currently ignored");
01534                 s = SkipField(s,1);
01535             }
01536             else s++;
01537             symtype |= reverseorder;
01538             cc = *s;
01539         }
01540 retry:;
01541         if ( cc == '<' ) {
01542             s++; addone = 0;
01543             if ( *s == '=' ) { addone++; s++; }
01544             argmax = 0;
01545             while ( FG.cTable[*s] == 1 ) { argmax = 10*argmax + *s++ - '0'; }
01546             argmax += addone;
01547             par = s;
01548             while ( FG.cTable[*s] == 0 ) s++;
01549             if ( s > par ) {
01550                 cc = *s; *s = 0;
01551                 if ( ( StrICont(par,(UBYTE *)"arguments") == 0 )
01552                 || ( StrICont(par,(UBYTE *)"args") == 0 ) ) {}
01553                 else {
01554                     Warning("Illegal information in number of arguments properties currently
      ignored");
01555                 }
01556                 *s = cc;
01557             }
01558             if ( argmax <= 0 ) {
01559                 MesPrint("&Error: Cannot have fewer than 0 arguments for variable %s",name);
01560                 error = 1;
01561             }
01562             cc = *s;
```

```
01563         }
01564         if ( cc == '>' ) {
01565             s++; addone = 1;
01566             if ( *s == '=' ) { addone = 0; s++; }
01567             argmin = 0;
01568             while ( FG.cTable[*s] == 1 ) { argmin = 10*argmin + *s++ - '0'; }
01569             argmin += addone;
01570             par = s;
01571             while ( FG.cTable[*s] == 0 ) s++;
01572             if ( s > par ) {
01573                 cc = *s; *s = 0;
01574                 if ( ( StrICont(par,(UBYTE *)"arguments") == 0 )
01575                 || ( StrICont(par,(UBYTE *)"args") == 0 ) ) {}
01576                 else {
01577                     Warning("Illegal information in number of arguments properties currently
      ignored");
01578                 }
01579                 *s = cc;
01580             }
01581             cc = *s;
01582         }
01583         if ( cc == '<' ) goto retry;
01584         if ( ( AC.AutoDeclareFlag == 0 &&
01585          ( ( type = GetName(AC.exprnames,name,&numfunction,NOAUTO) )
01586          != NAMENOTFOUND ) )
01587         || ( ( type = GetName(*(AC.activenames),name,&numfunction,NOAUTO) ) != NAMENOTFOUND ) ) {
01588             if ( type != CFUNCTION ) error = NameConflict(type,name);
01589             else {
01590 /*              FUNCTIONS fun = (FUNCTIONS)(AC.Functions->lijst) + numfunction-FUNCTION; */
01591                 FUNCTIONS fun = (FUNCTIONS)(AC.Functions->lijst) + numfunction;
01592
01593                 if ( fun->tabl != 0 ) {
01594                     MesPrint("&Illegal attempt to change table into function");
01595                     error = 1;
01596                 }
01597
01598                 fun->complex = cplx;
01599                 fun->commute = comm;
01600                 if ( istensor && fun->spec == 0 ) {
01601                     MesPrint("&Function %s changed to tensor",name);
01602                     error = 1;
01603                 }
01604                 else if ( istensor == 0 && fun->spec > 0 ) {
01605                     MesPrint("&Tensor %s changed to function",name);
01606                     error = 1;
01607                 }
01608                 else if ( fun->spec == VERTEXFUNCTION ) {
01609                     MesPrint("&Function or Tensor %s already declared as a Particle",name);
01610                     error = 1;
01611                 }
01612                 fun->spec = istensor;
01613                 if ( fun->symmetric != symtype ) {
01614                     fun->symmetric = symtype;
01615                     AC.SymChangeFlag = 1;
01616                 }
01617                 fun->maxnumargs = argmax;
01618                 fun->minnumargs = argmin;
01619             }
01620         }
01621         else {
01622             AddFunction(name,comm,istensor,cplx,symtype,dim,argmax,argmin);
01623         }
01624         *oldc = c;
01625 eol:    while ( *s == ',' ) s++;
01626     } while ( *s );
01627     return(error);
01628 }
01629
01630 int CoNFunction(UBYTE *s) { return(CoFunction(s,1,0)); }
01631 int CoCFunction(UBYTE *s) { return(CoFunction(s,0,0)); }
01632 int CoNTensor(UBYTE *s) { return(CoFunction(s,1,2)); }
01633 int CoCTensor(UBYTE *s) { return(CoFunction(s,0,2)); }
01634
01635 /*
01636     #] CoFunction + ...:
01637     #[ DoTable :
01638
01639         Syntax:
01640         Table [check] [strict|relax] [zerofill] name(:1:2,...,regular arguments);
01641         name must be the name of a regular function.
01642         the table indices must be the first arguments.
01643         The parenthesis indicates 'name' as opposed to the options.
01644
01645         We leave behind:
01646         a struct tabl in the FUNCTION struct
01647         Regular table:
01648             an array tablepointers for the pointers to elements of rhs
```

```
01649             in the compiler struct cbuf[T->bufnum]
01650             an array MINMAX T->mm with the minima and maxima
01651             a prototype array
01652             an offset in the compiler buffer for the pattern to be matched
01653         Sparse table:
01654             Just the number of dimensions
01655             We will keep track of the number of defined elements in totind
01656             and in tablepointers we will have numind+1 positions for each
01657             element. The first numind elements for the indices and the
01658             last one for the element in cbuf[T->bufnum].rhs
01659 
01660             If the number of dimensions is *<number>, there is not a fixed
01661             number of dimensions. Just a maximum. In that case the first
01662             index should be the number of other dimensions. This first index
01663             does not count in <number>.
01664 
01665         Complication: to preserve speed we need a prototype and a pattern
01666         for each thread when we use WITHPTHREADS. This is because we write
01667         into those when looking for the pattern.
01668 */
01669 
01670 static int nwarntab = 1;
01671 
01672 int DoTable(UBYTE *s, int par)
01673 {
01674     GETIDENTITY
01675     UBYTE *name, *p, *inp, c;
01676     int i, j, k, sparseflag = 0, rflag = 0, checkflag = 0;
01677     int error = 0, ret, oldcbufnum, oldEside;
01678     WORD funnum, type, *OldWork, *w, *ww, *t, *tt, *flags1, oldnumrhs,oldnumlhs;
01679     LONG oldcpointer;
01680     MINMAX *mm, *mm1;
01681     LONG x, y;
01682     TABLES T;
01683     CBUF *C;
01684 
01685     while ( *s == ',' ) s++;
01686     do {
01687         name = s;
01688         if ( ( s = SkipAName(s) ) == 0 ) {
01689 IllForm:    MesPrint("&Illegal name or option in table declaration");
01690             return(1);
01691         }
01692         c = *s; *s = 0;
01693         if ( TestName(name) ) { *s = c; goto IllForm; }
01694         *s = c;
01695         if ( *s == '(' ) break;
01696         if ( *s != ',' ) {
01697             MesPrint("&Illegal definition of table");
01698             return(1);
01699         }
01700         *s = 0;
01701 /*
01702         Secondary options
01703 */
01704         if      ( StrICmp(name,(UBYTE *)("check" )) == 0 ) checkflag = 1;
01705         else if ( StrICmp(name,(UBYTE *)("zero"  )) == 0 ) checkflag = 2;
01706         else if ( StrICmp(name,(UBYTE *)("one"   )) == 0 ) checkflag = 3;
01707         else if ( StrICmp(name,(UBYTE *)("strict")) == 0 ) rflag = 1;
01708         else if ( StrICmp(name,(UBYTE *)("relax" )) == 0 ) rflag = -1;
01709         else if ( StrICmp(name,(UBYTE *)("zerofill" )) == 0 ) { rflag = -2; checkflag = 2; }
01710         else if ( StrICmp(name,(UBYTE *)("onefill" )) == 0 ) { rflag = -3; checkflag = 3; }
01711         else if ( StrICmp(name,(UBYTE *)("sparse")) == 0 ) sparseflag |= 1;
01712         else if ( StrICmp(name,(UBYTE *)("base")) == 0 ) sparseflag |= 3;
01713         else if ( StrICmp(name,(UBYTE *)("tablebase")) == 0 ) sparseflag |= 3;
01714         else {
01715             MesPrint("&Illegal option in table definition: '%s'",name);
01716             error = 1;
01717         }
01718         *s++ = ',';
01719         while ( *s == ',' ) s++;
01720     } while ( *s );
01721     if ( name == s || *s == 0 ) {
01722         MesPrint("&Illegal name or option in table declaration");
01723         return(1);
01724     }
01725     *s = 0;     /* *s could only have been a parenthesis */
01726     if ( sparseflag ) {
01727         if ( checkflag == 1 ) rflag = 0;
01728         else if ( checkflag == 2 ) rflag = -2;
01729         else if ( checkflag == 3 ) rflag = -3;
01730         else rflag = -1;
01731     }
01732     if ( ( ret = GetVar(name,&type,&funnum,CFUNCTION,NOAUTO) ) ==
01733                     NAMENOTFOUND ) {
01734         if ( par == 0 ) {
01735             funnum = EntVar(CFUNCTION,name,0,1,0,0);
```

```
01736         }
01737         else if ( par == 1 || par == 2 ) {
01738             funnum = EntVar(CFUNCTION,name,0,0,0,0);
01739         }
01740     }
01741     else if ( ret <= 0 ) {
01742         funnum = EntVar(CFUNCTION,name,0,0,0,0);
01743         error = 1;
01744     }
01745     else {
01746         if ( par == 2 ) {
01747             if ( nwarntab ) {
01748                 Warning("Table now declares its (commuting) function.");
01749                 Warning("Earlier definition in Function statement obsolete. Please remove.");
01750                 nwarntab = 0;
01751             }
01752         }
01753         else {
01754             error = 1;
01755             MesPrint("&(N)(C)Tables should not be declared previously");
01756         }
01757     }
01758     if ( functions[funnum].spec > 0 ) {
01759         MesPrint("&Tensors cannot become tables");
01760         return(1);
01761     }
01762     if ( functions[funnum].symmetric > 0 ) {
01763         MesPrint("&Functions with nontrivial symmetrization properties cannot become tables");
01764         return(1);
01765     }
01766     if ( functions[funnum].tabl ) {
01767         MesPrint("&Redefinition of an existing table is not allowed.");
01768         return(1);
01769     }
01770     functions[funnum].tabl = T = (TABLES)Malloc1(sizeof(struct TaBlEs),"table");
01771 /*
01772     Next we find the size of the table (if it is not sparse)
01773 */
01774     T->defined = T->mdefined = 0; T->sparse = sparseflag; T->mm = 0; T->flags = 0;
01775     T->numtree = 0; T->rootnum = 0; T->MaxTreeSize = 0;
01776     T->boomlijst = 0;
01777     T->strict = rflag;
01778     T->bounds = checkflag;
01779     T->bufnum = inicbufs();
01780     T->argtail = 0;
01781     T->spare = 0;
01782     T->bufferssize = 8;
01783     T->buffers = (WORD *)Malloc1(sizeof(WORD)*T->bufferssize,"Table buffers");
01784     T->buffersfill = 0;
01785     T->buffers[T->buffersfill++] = T->bufnum;
01786     T->mode = 0;
01787     T->numdummies = 0;
01788     mm = T->mm;
01789     T->numind = 0;
01790     if ( rflag > 0 ) AC.MustTestTable++;
01791     T->totind = 0;      /* Table hasn't been checked */
01792
01793     p = s; *s = '(';
01794     if ( sparseflag ) {
01795 /*
01796         First copy the tail, just in case we will construct a tablebase
01797         Note that we keep the ( to indicate a tail
01798         The actual arguments can be found after the comma. Before we have
01799         the dimension which the tablebase will need for consistency checking.
01800 */
01801         inp = p+1;
01802         SKIPBRA3(inp)
01803         c = *inp; *inp = 0;
01804         T->argtail = strDup1(p,"argtail");
01805         *inp = c;
01806 /*
01807         Now the regular compilation
01808 */
01809         inp = p++;
01810         if ( *p == '<' ) {
01811             WORD inc = 1;
01812             p++;
01813             if ( *p == '=' ) { inc++; p++; }
01814 /*
01815             We will use one extra number for telling how many there really are.
01816 */
01817             x = 0;
01818             while ( *p <= '9' && *p >= '0' ) x = 10*x + (*p++-'0');
01819             x = -x-inc;
01820             if ( x == -1 ) {
01821                 MesPrint("&Maximum number of dimensions in *-table should be at least one.");
01822                 error = 1;
```

```
01823                   goto FinishUp2;
01824               }
01825           }
01826           else {
01827               ParseNumber(x,p)
01828               if ( FG.cTable[p[-1]] != 1 || ( *p != ',' && *p != ')' ) ) {
01829                   p = inp;
01830                   MesPrint("&First argument in a sparse table must be a number of dimensions");
01831                   error = 1;
01832                   x = 1;
01833               }
01834           }
01835           T->numind = x;
01836           T->mm = (MINMAX *)Malloc1(ABS(x)*sizeof(MINMAX),"table dimensions");
01837           T->flags = (WORD *)Malloc1(ABS(x)*sizeof(WORD),"table flags");
01838           mm = T->mm;
01839           inp = p;
01840           if ( *inp != ')' ) inp++;
01841           T->totind = 0;            /* At the moment there are this many */
01842           T->tablepointers = 0;
01843           T->reserved = 0;
01844       }
01845       else {
01846           T->numind = 0;
01847           T->totind = 1;
01848           for(;;) {   /* Read the dimensions as far as they can be recognized */
01849               inp = ++p;
01850               if ( FG.cTable[*p] != 1 && *p != '+' && *p != '-' ) break;
01851               ParseSignedNumber(x,p)
01852               if ( FG.cTable[p[-1]] != 1 || *p != ':' ) break;
01853               p++;
01854               ParseSignedNumber(y,p)
01855               if ( FG.cTable[p[-1]] != 1 || ( *p != ',' && *p != ')' ) ) {
01856                   MesPrint("&Illegal dimension field in table declaration");
01857                   return(1);
01858               }
01859               mm1 = (MINMAX *)Malloc1((T->numind+1)*sizeof(MINMAX),"table dimensions");
01860               flags1 = (WORD *)Malloc1((T->numind+1)*sizeof(WORD),"table flags");
01861               for ( i = 0; i < T->numind; i++ ) { mm1[i] = T->mm[i]; flags1[i] = T->flags[i]; }
01862               if ( T->mm ) M_free(T->mm,"table dimensions");
01863               if ( T->flags ) M_free(T->flags,"table flags");
01864               T->mm = mm1;
01865               T->flags = flags1;
01866               mm = T->mm + T->numind;
01867               mm->mini = x; mm->maxi = y;
01868               T->totind *= mm->maxi-mm->mini+1;
01869               T->numind++;
01870               if ( *p == ')' ) { inp = p; break; }
01871           }
01872           w = T->tablepointers
01873             = (WORD *)Malloc1(TABLEEXTENSION*sizeof(WORD)*(T->totind),"table pointers");
01874           i = T->totind;
01875           for ( i = TABLEEXTENSION*T->totind; i > 0; i-- ) *w++ = -1; /* means: undefined */
01876           for ( i = T->numind-1, x = 1; i >= 0; i-- ) {
01877               T->mm[i].size = x;    /* Defines increment in this dimension */
01878               x *= T->mm[i].maxi - T->mm[i].mini + 1;
01879           }
01880       }
01881   /*
01882       Now we redo the 'function part' and send it to the compiler.
01883       The prototype has to be picked up properly.
01884   */
01885       AT.WorkPointer++; /* We need one extra word later */
01886       OldWork = AT.WorkPointer;
01887       oldcbufnum = AC.cbufnum;
01888       AC.cbufnum = T->bufnum;
01889       C = cbuf+AC.cbufnum;
01890       oldcpointer = C->Pointer - C->Buffer;
01891       oldnumlhs = C->numlhs;
01892       oldnumrhs = C->numrhs;
01893       AddLHS(AC.cbufnum);
01894       while ( s >= name ) *--inp = *s--;
01895       w = AT.WorkPointer;
01896       AC.ProtoType = w;
01897       *w++ = SUBEXPRESSION;
01898       *w++ = SUBEXPSIZE;
01899       *w++ = 0;
01900       *w++ = 1;
01901       *w++ = AC.cbufnum;
01902       FILLSUB(w)
01903       AC.WildC = w;
01904       AC.NwildC = 0;
01905       AT.WorkPointer = w + 4*AM.MaxWildcards;
01906       if ( ( ret = CompileAlgebra(inp,LHSIDE,AC.ProtoType) ) < 0 ) {
01907           error = 1; goto FinishUp;
01908       }
01909       if ( AC.NwildC && SortWild(w,AC.NwildC) ) error = 1;
```

```
01910     w += AC.NwildC;
01911     i = w-OldWork;
01912     OldWork[1] = i;
01913 /*
01914     Basically we have to pull this pattern through Generator in case
01915     there are functions inside functions, or parentheses.
01916     We have to temporarily disable the .tabl to avoid problems with
01917     TestSub.
01918     Essential: we need to start NewSort twice to avoid the PutOut routines.
01919     The ground pattern is sitting in C->numrhs, but it could be that it
01920     has subexpressions in it. Hence it has to be worked out as the lhs in
01921     id statements (in comexpr.c).
01922 */
01923     OldWork[2] = C->numrhs;
01924     *w++ = 1; *w++ = 1; *w++ = 3;
01925     OldWork[-1] = w-OldWork+1;
01926     AT.WorkPointer = w;
01927     ww = C->rhs[C->numrhs];
01928     for ( j = 0; j < *ww; j++ ) w[j] = ww[j];
01929     AT.WorkPointer = w+*w;
01930     if ( *ww == 0 || ww[*ww] != 0 ) {
01931         MesPrint("&Illegal table pattern definition");
01932         AC.lhdollarflag = 0;
01933         error = 1;
01934     }
01935     if ( error ) goto FinishUp;
01936
01937     if ( NewSort(BHEAD0) || NewSort(BHEAD0) ) { error = 1; goto FinishUp; }
01938     AN.RepPoint = AT.RepCount + 1;
01939     AC.lhdollarflag = 0; oldEside = AR.Eside; AR.Eside = LHSIDE;
01940     AR.Cnumlhs = C->numlhs;
01941     functions[funnum].tabl = 0;
01942     if ( Generator(BHEAD w,C->numlhs) ) {
01943         functions[funnum].tabl = T;
01944         AR.Eside = oldEside;
01945         LowerSortLevel(); LowerSortLevel(); goto FinishUp;
01946     }
01947     functions[funnum].tabl = T;
01948     AR.Eside = oldEside;
01949     AT.WorkPointer = w;
01950     if ( EndSort(BHEAD w,0) < 0 ) { LowerSortLevel(); goto FinishUp; }
01951     if ( *w == 0 || *(w+*w) != 0 ) {
01952         MesPrint("&Irregular pattern in table definition");
01953         error = 1;
01954         goto FinishUp;
01955     }
01956     LowerSortLevel();
01957     if ( AC.lhdollarflag ) {
01958         MesPrint("&Unexpanded dollar variables are not allowed in table definition");
01959         error = 1;
01960         goto FinishUp;
01961     }
01962     AT.WorkPointer = ww = w + *w;
01963     if ( ww[-1] != 3 || ww[-2] != 1 || ww[-3] != 1 ) {
01964         MesPrint("&Coefficient of pattern in table definition should be 1.");
01965         error = 1;
01966         goto FinishUp;
01967     }
01968     AC.DumNum = 0;
01969 /*
01970     Now we have to allocate space for prototype+pattern
01971     In the case of TFORM we need extra pointers, because each worker has its own
01972 */
01973     j = *w + ABS(T->numind)*2-3;
01974 #ifdef WITHPTHREADS
01975     { int n;
01976     T->prototypeSize = ((i+j)*sizeof(WORD)+2*sizeof(WORD *)) * AM.totalnumberofthreads;
01977     T->prototype = (WORD **)Malloc1(T->prototypeSize,"table prototype");
01978     T->pattern = T->prototype + AM.totalnumberofthreads;
01979     t = (WORD *)(T->pattern + AM.totalnumberofthreads);
01980     for ( n = 0; n < AM.totalnumberofthreads; n++ ) {
01981         T->prototype[n] = t;
01982         for ( k = 0; k < i; k++ ) *t++ = OldWork[k];
01983     }
01984     T->pattern[0] = t;
01985     j--; w++;
01986     w[1] += ABS(T->numind)*2;
01987     for ( k = 0; k < FUNHEAD; k++ ) *t++ = *w++;
01988     j -= FUNHEAD;
01989     for ( k = 0; k < ABS(T->numind); k++ ) { *t++ = -SNUMBER; *t++ = 0; j -= 2; }
01990     for ( k = 0; k < j; k++ ) *t++ = *w++;
01991     if ( sparseflag ) T->pattern[0][1] = t - T->pattern[0];
01992     k = t - T->pattern[0];
01993     for ( n = 1; n < AM.totalnumberofthreads; n++ ) {
01994         T->pattern[n] = t; tt = T->pattern[0];
01995         for ( i = 0; i < k; i++ ) *t++ = *tt++;
01996     }
```

```
01997     }
01998 #else
01999     T->prototypeSize = (i+j)*sizeof(WORD);
02000     T->prototype = (WORD *)Malloc1(T->prototypeSize, "table prototype");
02001     T->pattern = T->prototype + i;
02002     for ( k = 0; k < i; k++ ) T->prototype[k] = OldWork[k];
02003     t = T->pattern;
02004     j--; w++;
02005     w[1] += ABS(T->numind)*2;
02006     for ( k = 0; k < FUNHEAD; k++ ) *t++ = *w++;
02007     j -= FUNHEAD;
02008     for ( k = 0; k < ABS(T->numind); k++ ) { *t++ = -SNUMBER; *t++ = 0; j -= 2; }
02009     for ( k = 0; k < j; k++ ) *t++ = *w++;
02010     if ( sparseflag ) T->pattern[1] = t - T->pattern;
02011 #endif
02012 /*
02013     At this point we can pop the compilerbuffer.
02014 */
02015     C->Pointer = C->Buffer + oldcpointer;
02016     C->numrhs = oldnumrhs;
02017     C->numlhs = oldnumlhs;
02018 /*
02019     Now check whether wildcards get converted to dollars (for PARALLEL)
02020     We give a warning!
02021 */
02022 #ifdef WITHPTHREADS
02023     t = T->prototype[0];
02024 #else
02025     t = T->prototype;
02026 #endif
02027     tt = t + t[1]; t += SUBEXPSIZE;
02028     while ( t < tt ) {
02029         if ( *t == LOADDOLLAR ) {
02030             Warning("The use of $-variable assignments in tables disables parallel\
02031  execution for the whole program.");
02032             AM.hparallelflag |= NOPARALLEL_TBLDOLLAR;
02033             AC.mparallelflag |= NOPARALLEL_TBLDOLLAR;
02034             AddPotModdollar(t[2]);
02035         }
02036         t += t[1];
02037     }
02038 FinishUp:;
02039     AT.WorkPointer = OldWork - 1;
02040     AC.cbufnum = oldcbufnum;
02041 FinishUp2:;
02042     if ( T->sparse ) ClearTableTree(T);
02043     if ( ( sparseflag & 2 ) != 0 ) {
02044         if ( T->spare == 0 ) { SpareTable(T); }
02045     }
02046     return(error);
02047 }
02048
02049 /*
02050     #] DoTable :
02051     #[ CoTable :
02052 */
02053
02054 int CoTable(UBYTE *s)
02055 {
02056     return(DoTable(s,2));
02057 }
02058
02059 /*
02060     #] CoTable :
02061     #[ CoNTable :
02062 */
02063
02064 int CoNTable(UBYTE *s)
02065 {
02066     return(DoTable(s,0));
02067 }
02068
02069 /*
02070     #] CoNTable :
02071     #[ CoCTable :
02072 */
02073
02074 int CoCTable(UBYTE *s)
02075 {
02076     return(DoTable(s,1));
02077 }
02078
02079 /*
02080     #] CoCTable :
02081     #[ EmptyTable :
02082 */
02083
```

```
02084 void EmptyTable(TABLES T)
02085 {
02086     int j;
02087     if ( T->sparse ) ClearTableTree(T);
02088     if ( T->boomlijst ) M_free(T->boomlijst,"TableTree");
02089     T->boomlijst = 0;
02090     for (j = 0; j < T->buffersfill; j++ ) { /* was <= */
02091         finishcbuf(T->buffers[j]);
02092     }
02093     if ( T->buffers ) M_free(T->buffers,"Table buffers");
02094     finishcbuf(T->bufnum);
02095     T->bufnum = inicbufs();
02096     T->bufferssize = 8;
02097     T->buffers = (WORD *)Malloc1(sizeof(WORD)*T->bufferssize,"Table buffers");
02098     T->buffersfill = 0;
02099     T->buffers[T->buffersfill++] = T->bufnum;
02100     T->defined = T->mdefined = 0; T->flags = 0;
02101     T->numtree = 0; T->rootnum = 0; T->MaxTreeSize = 0;
02102     T->spare = 0; T->reserved = 0;
02103     if ( T->spare ) {
02104         TABLES TT = T->spare;
02105         if ( TT->mm ) M_free(TT->mm,"tableminmax");
02106         if ( TT->flags ) M_free(TT->flags,"tableflags");
02107         if ( TT->tablepointers ) M_free(TT->tablepointers,"tablepointers");
02108         for (j = 0; j < TT->buffersfill; j++ ) {
02109             finishcbuf(TT->buffers[j]);
02110         }
02111         if ( TT->boomlijst ) M_free(TT->boomlijst,"TableTree");
02112         if ( TT->buffers ) M_free(TT->buffers,"Table buffers");
02113         M_free(TT,"table");
02114         SpareTable(T);
02115     }
02116     else {
02117         WORD *w = T->tablepointers;
02118         j = T->totind;
02119         for ( j = TABLEEXTENSION*T->totind; j > 0; j-- ) *w++ = -1; /* means: undefined */
02120     }
02121 }
02122
02123 /*
02124     #] EmptyTable :
02125     #[ AddSet :
02126 */
02127
02128 int AddSet(UBYTE *name, WORD dim)
02129 {
02130     int nodenum, numset = AC.SetList.num;
02131     SETS set = (SETS)FromVarList(&AC.SetList);
02132     UBYTE *s;
02133     if ( name ) {
02134         set->name = AddName(AC.varnames,name,CSET,numset,&nodenum);
02135         s = name;
02136         while ( *s ) s++;
02137         set->namesize = (s-name)+1;
02138         set->node  = nodenum;
02139     }
02140     else {
02141         set->name = 0;
02142         set->namesize = 0;
02143         set->node  = -1;
02144     }
02145     set->first =
02146     set->last  = AC.SetElementList.num; /* set has no elements yet */
02147     set->type  = -1;                    /* undefined as of yet */
02148     set->dimension = dim;
02149     set->flags = 0;
02150     return(numset);
02151 }
02152
02153 /*
02154     #] AddSet :
02155     #[ DoElements :
02156
02157     Remark (25-mar-2011): If the dimension has been set (dim != MAXPOSITIVE)
02158     we want to test dimensions. Numbers count as dimension zero?
02159 */
02160
02161 int DoElements(UBYTE *s, SETS set, UBYTE *name)
02162 {
02163     int type, error = 0, x, sgn, i;
02164     WORD numset, *e;
02165     UBYTE c, *cname;
02166     while ( *s ) {
02167         if ( *s == ',' ) { s++; continue; }
02168         sgn = 0;
02169         while ( *s == '-' || *s == '+' ) {
02170             if ( *s == '-' ) sgn ^= 1;
```

```
02171             s++;
02172         }
02173         cname = s;
02174         if ( FG.cTable[*s] == 0 || *s == '_' || *s == '[' ) {
02175             if ( ( s = SkipAName(s) ) == 0 ) {
02176                 MesPrint("&Illegal name in set definition");
02177                 return(1);
02178             }
02179             c = *s; *s = 0;
02180             if ( ( ( type = GetName(AC.exprnames,cname,&numset,NOAUTO) ) == NAMENOTFOUND )
02181             && ( ( type = GetOName(AC.varnames,cname,&numset,WITHAUTO) ) == NAMENOTFOUND ) ) {
02182                 DUBIOUSV dv;
02183                 int nodenum;
02184                 MesPrint("&%s has not been declared",cname);
02185 /*
02186                 We enter a 'dubious' declaration to cut down on errors
02187 */
02188                 numset = AC.DubiousList.num;
02189                 dv = (DUBIOUSV)FromVarList(&AC.DubiousList);
02190                 dv->name = AddName(AC.varnames,cname,CDUBIOUS,numset,&nodenum);
02191                 dv->node = nodenum;
02192                 set->type = type = CDUBIOUS;
02193                 set->dimension = 0;
02194                 error = 1;
02195             }
02196             if ( set->type == -1 ) {
02197                 if ( type == CSYMBOL ) {
02198                     for ( i = set->first; i < set->last; i++ ) {
02199                         SetElements[i] += 2*MAXPOWER;
02200                     }
02201                 }
02202                 set->type = type;
02203             }
02204             if ( set->type != type && set->type != CDUBIOUS
02205             && type != CDUBIOUS ) {
02206                 if ( set->type != CNUMBER || ( type != CSYMBOL
02207                 && type != CINDEX ) ) {
02208                     MesPrint(
02209                     "&%s has not the same type as the other members of the set"
02210                     ,cname);
02211                     error = 1;
02212                     set->type = CDUBIOUS;
02213                 }
02214                 else {
02215                     if ( type == CSYMBOL ) {
02216                         for ( i = set->first; i < set->last; i++ ) {
02217                             SetElements[i] += 2*MAXPOWER;
02218                         }
02219                     }
02220                     set->type = type;
02221                 }
02222             }
02223             if ( set->dimension != MAXPOSITIVE ) { /* Dimension check */
02224                 switch ( set->type ) {
02225                     case CSYMBOL:
02226                         if ( symbols[numset].dimension != set->dimension ) {
02227                             MesPrint("&Dimension check failed in set %s, symbol %s",
02228                                 VARNAME(Sets,(set-Sets)),
02229                                 VARNAME(symbols,numset));
02230                             error = 1;
02231                             set->dimension = MAXPOSITIVE;
02232                         }
02233                         break;
02234                     case CVECTOR:
02235                         if ( vectors[numset-AM.OffsetVector].dimension != set->dimension ) {
02236                             MesPrint("&Dimension check failed in set %s, vector %s",
02237                                 VARNAME(Sets,(set-Sets)),
02238                                 VARNAME(vectors,(numset-AM.OffsetVector)));
02239                             error = 1;
02240                             set->dimension = MAXPOSITIVE;
02241                         }
02242                         break;
02243                     case CFUNCTION:
02244                         if ( functions[numset-FUNCTION].dimension != set->dimension ) {
02245                             MesPrint("&Dimension check failed in set %s, function %s",
02246                                 VARNAME(Sets,(set-Sets)),
02247                                 VARNAME(functions,(numset-FUNCTION)));
02248                             error = 1;
02249                         }
02250                         break;
02251                             set->dimension = MAXPOSITIVE;
02252                 }
02253             }
02254             if ( sgn ) {
02255                 if ( type != CVECTOR ) {
02256                     MesPrint("&Illegal use of - sign in set. Can use only with vector or number");
02257                     error = 1;
```

```
02258                 }
02259 /*
02260                 numset = AM.OffsetVector - numset;
02261                 numset |= SPECMASK;
02262                 numset = AM.OffsetVector - numset;
02263 */
02264                 numset -= WILDMASK;
02265             }
02266             *s = c;
02267             if ( name == 0 && *s == '?' ) {
02268                 s++;
02269                 switch ( set->type ) {
02270                     case CSYMBOL:
02271                         numset = -numset; break;
02272                     case CVECTOR:
02273                         numset += WILDOFFSET; break;
02274                     case CINDEX:
02275                         numset |= WILDMASK; break;
02276                     case CFUNCTION:
02277                         numset |= WILDMASK; break;
02278                 }
02279                 AC.wildflag = 1;
02280             }
02281 /*
02282             Now add the element to the set.
02283 */
02284             e = (WORD *)FromVarList(&AC.SetElementList);
02285             *e = numset;
02286             (set->last)++;
02287         }
02288         else if ( FG.cTable[*s] == 1 ) {
02289             ParseNumber(x,s)
02290             if ( sgn ) x = -x;
02291             if ( x >= MAXPOWER || x <= -MAXPOWER ||
02292             ( set->type == CINDEX && ( x < 0 || x >= AM.OffsetIndex ) ) ) {
02293                 MesPrint("&Illegal value for set element: %d",x);
02294                 if ( AC.firstconstindex ) {
02295                 MesPrint("&0 <= Fixed indices < ConstIndex(which is %d)",
02296                     AM.OffsetIndex-1);
02297                 MesPrint("&For setting ConstIndex, read the chapter on the setup file");
02298                 AC.firstconstindex = 0;
02299                 }
02300                 error = 1;
02301                 x = 0;
02302             }
02303 /*
02304             Check what is allowed with the type.
02305 */
02306             if ( set->type == -1 ) {
02307                 if ( x < 0 || x >= AM.OffsetIndex ) {
02308                     for ( i = set->first; i < set->last; i++ ) {
02309                         SetElements[i] += 2*MAXPOWER;
02310                     }
02311                     set->type = CSYMBOL;
02312                 }
02313                 else set->type = CNUMBER;
02314             }
02315             else if ( set->type == CDUBIOUS ) {}
02316             else if ( set->type == CNUMBER && x < 0 ) {
02317                 for ( i = set->first; i < set->last; i++ ) {
02318                     SetElements[i] += 2*MAXPOWER;
02319                 }
02320                 set->type = CSYMBOL;
02321             }
02322             else if ( set->type != CSYMBOL && ( x < 0 ||
02323             ( set->type != CINDEX && set->type != CNUMBER ) ) ) {
02324                 MesPrint("&Illegal mixture of element types in set");
02325                 error = 1;
02326                 set->type = CDUBIOUS;
02327             }
02328 /*
02329             Allocate an element
02330 */
02331             e = (WORD *)FromVarList(&AC.SetElementList);
02332             (set->last)++;
02333             if ( set->type == CSYMBOL ) *e = x + 2*MAXPOWER;
02334 /*          else if ( set->type == CINDEX ) *e = x; */
02335             else *e = x;
02336         }
02337         else {
02338             MesPrint("&Illegal object in list of set elements");
02339             return(1);
02340         }
02341     }
02342     if ( error == 0 && ( ( set->flags & ORDEREDSET ) == ORDEREDSET ) ) {
02343 /*
02344         The set->last-set->first list of numbers must be sorted.
```

```
02345         Because we plan here potentially thousands of elements we use
02346         a simple version of splitmerge. In ordered sets we can search
02347         later with a binary search.
02348 */
02349         SimpleSplitMerge(SetElements+set->first,set->last-set->first);
02350     }
02351     return(error);
02352 }
02353
02354 /*
02355     #] DoElements :
02356     #[ CoSet :
02357
02358     Set declarations.
02359 */
02360
02361 int CoSet(UBYTE *s)
02362 {
02363     int type, error = 0, ordered = 0;
02364     UBYTE *name = s, c, *ss;
02365     SETS set;
02366     WORD numberofset, dim = MAXPOSITIVE;
02367 #ifdef WITHFLOAT
02368 /* UNFINISHED_FEATURE_EXCL_START */
02369 /*--------------------------------------------------------------------*/
02370     {
02371         WORD numeq = 0;
02372         LONG x;
02373         ss = s;
02374         while ( *ss && *ss != ':' ) { if ( *ss == '=' ) numeq++; ss++; }
02375         if ( *ss == 0 && numeq == 1 ) { /* We have the Set var = value; variety */
02376             while ( FG.cTable[*s] == 0 ) s++;
02377             ss = s; c = *s; *s = 0;
02378             if ( c != '=' ) {
02379 Proper:
02380                 MesPrint("&Proper syntax for value-set is `Set name = value'");
02381                 return(1);
02382             }
02383             x = 0; s++;
02384             while ( *s >= '0' && *s <= '9' ) x = 10*x + (*s++-'0');
02385             if ( *s ) goto Proper;
02386             if ( StrICmp(name,(UBYTE *)"maxweight") == 0 ) {
02387                 AC.tMaxWeight = x;   /* Temporary. Made permanent later */
02388             }
02389             else if ( StrICmp(name,(UBYTE *)"defaultprecision") == 0 ) {
02390                 AC.tDefaultPrecision = x;   /* Temporary. Made permanent later */
02391             }
02392             else {
02393                 MesPrint("&Illegal subkey in value set: %s",name);
02394                 return(1);
02395             }
02396         }
02397     }
02398 /*--------------------------------------------------------------------*/
02399 /* UNFINISHED_FEATURE_EXCL_STOP */
02400 #endif
02401     if ( ( s = SkipAName(s) ) == 0 ) {
02402 IllForm:MesPrint("&Illegal name for set");
02403         return(1);
02404     }
02405     c = *s; *s = 0;
02406     if ( TestName(name) ) goto IllForm;
02407     if ( ( ( type = GetName(AC.exprnames,name,&numberofset,NOAUTO) ) != NAMENOTFOUND )
02408     || ( ( type = GetName(AC.varnames,name,&numberofset,NOAUTO) ) != NAMENOTFOUND ) ) {
02409         if ( type != CSET ) NameConflict(type,name);
02410         else {
02411             MesPrint("&There is already a set with the name %s",name);
02412         }
02413         return(1);
02414     }
02415     if ( c == 0 ) {
02416         numberofset = AddSet(name,0);
02417         set = Sets + numberofset;
02418         return(0);      /* empty set */
02419     }
02420     *s = c; ss = s;     /* ss marks the end of the name */
02421     if ( *s == '(' ) {
02422         UBYTE *sss, cc;
02423         s++; sss = s;   /* Beginning of option */
02424         while ( *s != ',' && *s != ')' && *s ) s++;
02425         cc = *s; *s = 0;
02426         if ( StrICont(sss,(UBYTE *)"ordered") == 0 ) {
02427             ordered = ORDEREDSET;
02428         }
02429         else {
02430             MesPrint("&Error: Illegal option in set definition: %s",sss);
02431             error = 1;
```

```
02432         }
02433         *s = cc;
02434         if ( *s != ')' ) {
02435             MesPrint("&Error: Currently only one option allowed in set definition.");
02436             error = 1;
02437             while ( *s && *s != ')' ) s++;
02438         }
02439         s++;
02440     }
02441     if ( *s == '{' ) {
02442         s++;
02443         if ( ( *s == 'd' || *s == 'D' ) && s[1] == '=' ) {
02444             s += 2;
02445             if ( *s == '-' || *s == '+' || FG.cTable[*s] == 1 ) {
02446                 ParseSignedNumber(dim,s)
02447                 if ( dim < -HALFMAX || dim > HALFMAX ) {
02448                     MesPrint("&Warning: dimension of %s (%d) out of range"
02449                     ,name,dim);
02450                 }
02451             }
02452             if ( *s != '}' ) goto IllDim;
02453             else s++;
02454         }
02455         else {
02456 IllDim:     MesPrint("&Error: Illegal dimension field for set %s",name);
02457             error = 1;
02458             s = SkipField(s,0);
02459         }
02460         while ( *s == ',' ) s++;
02461     }
02462     c = *ss; *ss = 0;
02463     numberofset = AddSet(name,dim);
02464     *ss = c;
02465     set = Sets + numberofset;
02466     set->flags |= ordered;
02467     if ( *s != ':' ) {
02468         MesPrint("&Proper syntax is `Set name:elements'");
02469         return(1);
02470     }
02471     s++;
02472     error = DoElements(s,set,name);
02473     AC.SetList.numtemp = AC.SetList.num;
02474     AC.SetElementList.numtemp = AC.SetElementList.num;
02475     return(error);
02476 }
02477
02478 /*
02479     #] CoSet :
02480     #[ DoTempSet :
02481
02482         Gets a {} set definition and returns a set number if the set is
02483         properly structured. This number refers either to an already
02484         existing set, or to a set that is defined here.
02485         From and to refer to the contents. They exclude the {}.
02486 */
02487
02488 int DoTempSet(UBYTE *from, UBYTE *to)
02489 {
02490     int i, num, j, sgn;
02491     WORD *e, *ep;
02492     UBYTE c;
02493     int setnum = AddSet(0,MAXPOSITIVE);
02494     SETS set = Sets + setnum, setp;
02495     set->name = -1;
02496     set->type = -1;
02497     c = *to; *to = 0;
02498     AC.wildflag = 0;
02499     while ( *from == ',' ) from++;
02500     if ( *from == '<' || *from == '>' ) {
02501         set->type = CRANGE;
02502         set->first = 3*MAXPOWER;
02503         set->last = -3*MAXPOWER;
02504         while ( *from == '<' || *from == '>' ) {
02505             if ( *from == '<' ) {
02506                 j = 1; from++;
02507                 if ( *from == '=' ) { from++; j++; }
02508             }
02509             else {
02510                 j = -1; from++;
02511                 if ( *from == '=' ) { from++; j--; }
02512             }
02513             sgn = 1;
02514             while ( *from == '-' || *from == '+' ) {
02515                 if ( *from == '-' ) sgn = -sgn;
02516                 from++;
02517             }
02518             ParseNumber(num,from)
```

```
02519             if ( *from && *from != ',' ) {
02520                 MesPrint("&Illegal number in ranged set definition");
02521                 return(-1);
02522             }
02523             if ( sgn < 0 ) num = -num;
02524             if ( num >= MAXPOWER || num <= -MAXPOWER ) {
02525                 Warning("Value in ranged set too big. Adjusted to infinity.");
02526                 if ( num > 0 ) num =  3*MAXPOWER;
02527                 else           num = -3*MAXPOWER;
02528             }
02529             else if ( j ==  2 ) num += 2*MAXPOWER;
02530             else if ( j == -2 ) num -= 2*MAXPOWER;
02531             if ( j > 0 ) set->first = num;
02532             else set->last = num;
02533             while ( *from == ',' ) from++;
02534         }
02535         if ( *from ) {
02536             MesPrint("&Definition of ranged set contains illegal objects");
02537             return(-1);
02538         }
02539     }
02540     else if ( DoElements(from,set,(UBYTE *)0) != 0 ) {
02541         AC.SetElementList.num = set->first;
02542         AC.SetList.num--; *to = c;
02543         return(-1);
02544     }
02545     *to = c;
02546 /*
02547     Now we have to test whether this set exists already.
02548 */
02549     num = set->last - set->first;
02550     for ( setp = Sets, i = 0; i < AC.SetList.num-1; i++, setp++ ) {
02551         if ( num != setp->last - setp->first ) continue;
02552         if ( set->type != setp->type ) continue;
02553         if ( set->type == CRANGE ) {
02554             if ( set->first == setp->first ) return(setp-Sets);
02555         }
02556         else {
02557             e = SetElements + set->first;
02558             ep = SetElements + setp->first;
02559             j = num;
02560             while ( --j >= 0 ) if ( *e++ != *ep++ ) break;
02561             if ( j < 0 ) {
02562                 AC.SetElementList.num = set->first;
02563                 AC.SetList.num--;
02564                 return(setp - Sets);
02565             }
02566         }
02567     }
02568     return(setnum);
02569 }
02570
02571 /*
02572     #] DoTempSet :
02573     #[ CoAuto :
02574
02575     To prepare first:
02576         Use of the proper pointers in the various declaration routines
02577         Proper action in .store and .clear
02578 */
02579
02580 int CoAuto(UBYTE *inp)
02581 {
02582     int retval;
02583
02584     AC.Symbols = &(AC.AutoSymbolList);
02585     AC.Vectors = &(AC.AutoVectorList);
02586     AC.Indices = &(AC.AutoIndexList);
02587     AC.Functions = &(AC.AutoFunctionList);
02588     AC.activenames = &(AC.autonames);
02589     AC.AutoDeclareFlag = WITHAUTO;
02590
02591     while ( *inp == ',' ) inp++;
02592     retval = CompileStatement(inp);
02593
02594     AC.AutoDeclareFlag = 0;
02595     AC.Symbols = &(AC.SymbolList);
02596     AC.Vectors = &(AC.VectorList);
02597     AC.Indices = &(AC.IndexList);
02598     AC.Functions = &(AC.FunctionList);
02599     AC.activenames = &(AC.varnames);
02600     return(retval);
02601 }
02602
02603 /*
02604     #] CoAuto :
02605     #[ AddDollar :
```

```
02606
02607     The actual addition. Special routine for additions 'on the fly'
02608 */
02609
02610 int AddDollar(UBYTE *name, WORD type, WORD *start, LONG size)
02611 {
02612     int nodenum, numdollar = AP.DollarList.num;
02613     WORD *s, *t;
02614     DOLLARS dol = (DOLLARS)FromVarList(&AP.DollarList);
02615     dol->name = AddName(AC.dollarnames,name,CDOLLAR,numdollar,&nodenum);
02616     dol->type = type;
02617     dol->node = nodenum;
02618     dol->zero = 0;
02619     dol->numdummies = 0;
02620 #ifdef WITHPTHREADS
02621     INIRECLOCK(dol->pthreadslock);
02622 #endif
02623     dol->nfactors = 0;
02624     dol->factors = 0;
02625     AddRHS(AM.dbufnum,1);
02626     AddLHS(AM.dbufnum);
02627     if ( start && size > 0 ) {
02628         dol->size = size;
02629         dol->where =
02630         s = (WORD *)Malloc1((size+1)*sizeof(WORD),"$-variable contents");
02631         t = start;
02632         while ( --size >= 0 ) *s++ = *t++;
02633         *s = 0;
02634     }
02635     else { dol->where = &(AM.dollarzero); dol->size = 0; }
02636     cbuf[AM.dbufnum].rhs[numdollar] = dol->where;
02637     cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
02638     cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
02639
02640     return(numdollar);
02641 }
02642
02643 /*
02644     #] AddDollar :
02645     #[ ReplaceDollar :
02646
02647     Replacements of dollar variables can happen at any time.
02648     For debugging purposes we should have a tracing facility.
02649
02650     Not in use????
02651 */
02652 /* UNFINISHED_FEATURE_EXCL_START */
02653 int ReplaceDollar(WORD number, WORD newtype, WORD *newstart, LONG newsize)
02654 {
02655     int error = 0;
02656     DOLLARS dol = Dollars + number;
02657     WORD *s, *t;
02658     LONG i;
02659     dol->type = newtype;
02660     if ( dol->size == newsize && newsize > 0 && newstart ) {
02661         s = dol->where; t = newstart; i = newsize;
02662         while ( --i >= 0 ) { if ( *s++ != *t++ ) break; }
02663         if ( i < 0 ) return(0);
02664     }
02665     if ( dol->where && dol->where != &(dol->zero) ) {
02666         M_free(dol->where,"dollar->where"); dol->where = &(dol->zero); dol->size = 0;
02667     }
02668     if ( newstart && newsize > 0 ) {
02669         dol->size = newsize;
02670         dol->where =
02671         s = (WORD *)Malloc1((newsize+1)*sizeof(WORD),"$-variable contents");
02672         t = newstart; i = newsize;
02673         while ( --i >= 0 ) *s++ = *t++;
02674         *s = 0;
02675     }
02676     return(error);
02677 }
02678 /* UNFINISHED_FEATURE_EXCL_STOP */
02679 /*
02680     #] ReplaceDollar :
02681     #[ AddDubious :
02682
02683     This adds a variable of which we do not know the proper type.
02684 */
02685
02686 int AddDubious(UBYTE *name)
02687 {
02688     int nodenum, numdubious = AC.DubiousList.num;
02689     DUBIOUSV dub = (DUBIOUSV)FromVarList(&AC.DubiousList);
02690     dub->name = AddName(AC.varnames,name,CDUBIOUS,numdubious,&nodenum);
02691     dub->node = nodenum;
02692     return(numdubious);
```

```
02693 }
02694
02695 /*
02696     #] AddDubious :
02697     #[ MakeDubious :
02698 */
02699
02700 int MakeDubious(NAMETREE *nametree, UBYTE *name, WORD *number)
02701 {
02702     NAMENODE *n;
02703     int node, newnode, i;
02704     if ( nametree->namenode == 0 ) return(-1);
02705     newnode = nametree->headnode;
02706     do {
02707         node = newnode;
02708         n = nametree->namenode+node;
02709         if ( ( i = StrCmp(name,nametree->namebuffer+n->name) ) < 0 )
02710             newnode = n->left;
02711         else if ( i > 0 ) newnode = n->right;
02712         else {
02713             if ( n->type != CDUBIOUS ) {
02714                 int numdubious = AC.DubiousList.num;
02715                 FUNCTIONS dub = (FUNCTIONS)FromVarList(&AC.DubiousList);
02716                 dub->name = n->name;
02717                 n->number = numdubious;
02718             }
02719             *number = n->number;
02720             return(CDUBIOUS);
02721         }
02722     } while ( newnode >= 0 );
02723     return(-1);
02724 }
02725
02726 /*
02727     #] MakeDubious :
02728     #[ NameConflict :
02729 */
02730
02731 static char *nametype[] = { "symbol", "index", "vector", "function",
02732         "set", "expression" };
02733 static char *plural[] = { "","n","","","","n" };
02734
02735 int NameConflict(int type, UBYTE *name)
02736 {
02737     if ( type == NAMENOTFOUND ) {
02738         MesPrint("&%s has not been declared",name);
02739     }
02740     else if ( type != CDUBIOUS )
02741         MesPrint("&%s has been declared as a%s %s already"
02742             ,name,plural[type],nametype[type]);
02743     return(1);
02744 }
02745
02746 /*
02747     #] NameConflict :
02748     #[ AddExpression :
02749 */
02750
02751 int AddExpression(UBYTE *name, int x, int y)
02752 {
02753     int nodenum, numexpr = AC.ExpressionList.num;
02754     EXPRESSIONS expr = (EXPRESSIONS)FromVarList(&AC.ExpressionList);
02755     UBYTE *s;
02756     expr->status = x;
02757     expr->printflag = y;
02758     PUTZERO(expr->onfile);
02759     PUTZERO(expr->size);
02760     expr->renum = 0;
02761     expr->renumlists = 0;
02762     expr->hidelevel = 0;
02763     expr->inmem = 0;
02764     expr->bracketinfo = expr->newbracketinfo = 0;
02765     if ( name ) {
02766         expr->name = AddName(AC.exprnames,name,CEXPRESSION,numexpr,&nodenum);
02767         expr->node = nodenum;
02768         expr->replace = NEWLYDEFINEDEXPRESSION ;
02769         s = name;
02770         while ( *s ) s++;
02771         expr->namesize = (s-name)+1;
02772     }
02773     else {
02774         expr->replace = REDEFINEDEXPRESSION;
02775         expr->name = AC.TransEname;
02776         expr->node = -1;
02777         expr->namesize = 0;
02778     }
02779     expr->vflags = 0;
```

```
02780     expr->numdummies = 0;
02781     expr->numfactors = 0;
02782 #ifdef PARALLELCODE
02783     expr->partodo = 0;
02784 #endif
02785     expr->uflags = 0;
02786     return(numexpr);
02787 }
02788
02789 /*
02790     #] AddExpression :
02791     #[ GetLabel :
02792 */
02793
02794 int GetLabel(UBYTE *name)
02795 {
02796     int i;
02797     LONG newnum;
02798     UBYTE **NewLabelNames;
02799     int *NewLabel;
02800     for ( i = 0; i < AC.NumLabels; i++ ) {
02801         if ( StrCmp(name,AC.LabelNames[i]) == 0 ) return(i);
02802     }
02803     if ( AC.NumLabels >= AC.MaxLabels ) {
02804         newnum = 2*AC.MaxLabels;
02805         if ( newnum == 0 ) newnum = 10;
02806         if ( newnum > 32765 ) newnum = 32765;
02807         if ( newnum == AC.MaxLabels ) {
02808             MesPrint("&More than 32765 labels in one module. Please simplify.");
02809             Terminate(-1);
02810         }
02811         NewLabelNames = (UBYTE **)Malloc1((sizeof(UBYTE *)+sizeof(int))
02812                 *newnum,"Labels");
02813         NewLabel = (int *)(NewLabelNames+newnum);
02814         for ( i = 0; i< AC.MaxLabels; i++ ) {
02815             NewLabelNames[i] = AC.LabelNames[i];
02816             NewLabel[i] = AC.Labels[i];
02817         }
02818         if ( AC.LabelNames ) M_free(AC.LabelNames,"Labels");
02819         AC.LabelNames = NewLabelNames;
02820         AC.Labels = NewLabel;
02821         AC.MaxLabels = newnum;
02822     }
02823     i = AC.NumLabels++;
02824     AC.LabelNames[i] = strDup1(name,"Labels");
02825     AC.Labels[i] = -1;
02826     return(i);
02827 }
02828
02829 /*
02830     #] GetLabel :
02831     #[ ResetVariables :
02832
02833     Resets the variables.
02834     par = 0  The list of temporary sets (after each .sort)
02835     par = 1  The list of local variables (after each .store)
02836     par = 2  All variables (after each .clear)
02837 */
02838
02839 void ResetVariables(int par)
02840 {
02841     int i, j;
02842     TABLES T;
02843     switch ( par ) {
02844     case 0 : /* Only the sets without a name */
02845         AC.SetList.num        = AC.SetList.numtemp;
02846         AC.SetElementList.num = AC.SetElementList.numtemp;
02847         break;
02848     case 2 :
02849         for ( i = AC.SymbolList.numclear; i < AC.SymbolList.num; i++ )
02850             AC.varnames->namenode[symbols[i].node].type = CDELETE;
02851         AC.SymbolList.num = AC.SymbolList.numglobal = AC.SymbolList.numclear;
02852         for ( i = AC.VectorList.numclear; i < AC.VectorList.num; i++ )
02853             AC.varnames->namenode[vectors[i].node].type = CDELETE;
02854         AC.VectorList.num = AC.VectorList.numglobal = AC.VectorList.numclear;
02855         for ( i = AC.IndexList.numclear; i < AC.IndexList.num; i++ )
02856             AC.varnames->namenode[indices[i].node].type = CDELETE;
02857         AC.IndexList.num = AC.IndexList.numglobal = AC.IndexList.numclear;
02858         for ( i = AC.FunctionList.numclear; i < AC.FunctionList.num; i++ ) {
02859             AC.varnames->namenode[functions[i].node].type = CDELETE;
02860             if ( ( T = functions[i].tabl ) != 0 ) {
02861                 if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
02862                 if ( T->prototype ) M_free(T->prototype,"tableprototype");
02863                 if ( T->mm ) M_free(T->mm,"tableminmax");
02864                 if ( T->flags ) M_free(T->flags,"tableflags");
02865                 if ( T->argtail ) M_free(T->argtail,"table arguments");
02866                 if ( T->boomlijst ) M_free(T->boomlijst,"TableTree");
```

```
02867                 for (j = 0; j < T->buffersfill; j++ ) { /* was <= */
02868                     finishcbuf(T->buffers[j]);
02869                 }
02870                 /*[07apr2004 mt]:*/ /*memory leak*/
02871                 if ( T->buffers ) M_free(T->buffers,"Table buffers");
02872                 /*:[07apr2004 mt]*/
02873                 finishcbuf(T->bufnum);
02874                 if ( T->spare ) {
02875                     TABLES TT = T->spare;
02876                     if ( TT->mm ) M_free(TT->mm,"tableminmax");
02877                     if ( TT->flags ) M_free(TT->flags,"tableflags");
02878                     if ( TT->tablepointers ) M_free(TT->tablepointers,"tablepointers");
02879                     for (j = 0; j < TT->buffersfill; j++ ) { /* was <= */
02880                         finishcbuf(TT->buffers[j]);
02881                     }
02882                     if ( TT->boomlijst ) M_free(TT->boomlijst,"TableTree");
02883                     /*[07apr2004 mt]:*/ /*memory leak*/
02884                     if ( TT->buffers )M_free(TT->buffers,"Table buffers");
02885                     /*:[07apr2004 mt]*/
02886                     M_free(TT,"table");
02887                 }
02888                 M_free(T,"table");
02889             }
02890         }
02891         AC.FunctionList.num = AC.FunctionList.numglobal = AC.FunctionList.numclear;
02892         for ( i = AC.SetList.numclear; i < AC.SetList.num; i++ ) {
02893             if ( Sets[i].node >= 0 )
02894                 AC.varnames->namenode[Sets[i].node].type = CDELETE;
02895         }
02896         AC.SetList.numtemp = AC.SetList.num = AC.SetList.numglobal = AC.SetList.numclear;
02897         for ( i = AC.DubiousList.numclear; i < AC.DubiousList.num; i++ )
02898             AC.varnames->namenode[Dubious[i].node].type = CDELETE;
02899         AC.DubiousList.num = AC.DubiousList.numglobal = AC.DubiousList.numclear;
02900         AC.SetElementList.numtemp = AC.SetElementList.num =
02901             AC.SetElementList.numglobal = AC.SetElementList.numclear;
02902         CompactifyTree(AC.varnames,VARNAMES);
02903         AC.varnames->namefill = AC.varnames->globalnamefill = AC.varnames->clearnamefill;
02904         AC.varnames->nodefill = AC.varnames->globalnodefill = AC.varnames->clearnodefill;
02905
02906         for ( i = AC.AutoSymbolList.numclear; i < AC.AutoSymbolList.num; i++ )
02907             AC.autonames->namenode[
02908                     ((SYMBOLS)(AC.AutoSymbolList.lijst))[i].node].type = CDELETE;
02909         AC.AutoSymbolList.num = AC.AutoSymbolList.numglobal
02910                             = AC.AutoSymbolList.numclear;
02911         for ( i = AC.AutoVectorList.numclear; i < AC.AutoVectorList.num; i++ )
02912             AC.autonames->namenode[
02913                     ((VECTORS)(AC.AutoVectorList.lijst))[i].node].type = CDELETE;
02914         AC.AutoVectorList.num = AC.AutoVectorList.numglobal
02915                             = AC.AutoVectorList.numclear;
02916         for ( i = AC.AutoIndexList.numclear; i < AC.AutoIndexList.num; i++ )
02917             AC.autonames->namenode[
02918                     ((INDICES)(AC.AutoIndexList.lijst))[i].node].type = CDELETE;
02919         AC.AutoIndexList.num = AC.AutoIndexList.numglobal
02920                            = AC.AutoIndexList.numclear;
02921         for ( i = AC.AutoFunctionList.numclear; i < AC.AutoFunctionList.num; i++ ) {
02922             AC.autonames->namenode[
02923                     ((FUNCTIONS)(AC.AutoFunctionList.lijst))[i].node].type = CDELETE;
02924             if ( ( T = ((FUNCTIONS)(AC.AutoFunctionList.lijst))[i].tabl ) != 0 ) {
02925                 if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
02926                 if ( T->prototype ) M_free(T->prototype,"tableprototype");
02927                 if ( T->mm ) M_free(T->mm,"tableminmax");
02928                 if ( T->flags ) M_free(T->flags,"tableflags");
02929                 if ( T->argtail ) M_free(T->argtail,"table arguments");
02930                 if ( T->boomlijst ) M_free(T->boomlijst,"TableTree");
02931                 for (j = 0; j < T->buffersfill; j++ ) { /* was <= */
02932                     finishcbuf(T->buffers[j]);
02933                 }
02934                 if ( T->spare ) {
02935                     TABLES TT = T->spare;
02936                     if ( TT->mm ) M_free(TT->mm,"tableminmax");
02937                     if ( TT->flags ) M_free(TT->flags,"tableflags");
02938                     if ( TT->tablepointers ) M_free(TT->tablepointers,"tablepointers");
02939                     for (j = 0; j < TT->buffersfill; j++ ) { /* was <= */
02940                         finishcbuf(TT->buffers[j]);
02941                     }
02942                     if ( TT->boomlijst ) M_free(TT->boomlijst,"TableTree");
02943                     M_free(TT,"table");
02944                 }
02945                 M_free(T,"table");
02946             }
02947         }
02948         AC.AutoFunctionList.num = AC.AutoFunctionList.numglobal
02949                                 = AC.AutoFunctionList.numclear;
02950         CompactifyTree(AC.autonames,AUTONAMES);
02951         AC.autonames->namefill = AC.autonames->globalnamefill
02952                                = AC.autonames->clearnamefill;
02953         AC.autonames->nodefill = AC.autonames->globalnodefill
```

```
02954                                = AC.autonames->clearnodefill;
02955         ReleaseTB();
02956         break;
02957     case 1 :
02958         for ( i = AC.SymbolList.numglobal; i < AC.SymbolList.num; i++ )
02959             AC.varnames->namenode[symbols[i].node].type = CDELETE;
02960         AC.SymbolList.num = AC.SymbolList.numglobal;
02961         for ( i = AC.VectorList.numglobal; i < AC.VectorList.num; i++ )
02962             AC.varnames->namenode[vectors[i].node].type = CDELETE;
02963         AC.VectorList.num = AC.VectorList.numglobal;
02964         for ( i = AC.IndexList.numglobal; i < AC.IndexList.num; i++ )
02965             AC.varnames->namenode[indices[i].node].type = CDELETE;
02966         AC.IndexList.num = AC.IndexList.numglobal;
02967         for ( i = AC.FunctionList.numglobal; i < AC.FunctionList.num; i++ ) {
02968             AC.varnames->namenode[functions[i].node].type = CDELETE;
02969             if ( ( T = functions[i].tabl ) != 0 ) {
02970                 if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
02971                 if ( T->prototype ) M_free(T->prototype,"tableprototype");
02972                 if ( T->mm ) M_free(T->mm,"tableminmax");
02973                 if ( T->flags ) M_free(T->flags,"tableflags");
02974                 if ( T->argtail ) M_free(T->argtail,"table arguments");
02975                 if ( T->boomlijst ) M_free(T->boomlijst,"TableTree");
02976                 for (j = 0; j < T->buffersfill; j++ ) { /* was <= */
02977                     finishcbuf(T->buffers[j]);
02978                 }
02979                 /*[07apr2004 mt]:*/  /*memory leak*/
02980                 if ( T->buffers ) M_free(T->buffers,"Table buffers");
02981                 /*:[07apr2004 mt]*/
02982                 finishcbuf(T->bufnum);
02983                 if ( T->spare ) {
02984                     TABLES TT = T->spare;
02985                     if ( TT->mm ) M_free(TT->mm,"tableminmax");
02986                     if ( TT->flags ) M_free(TT->flags,"tableflags");
02987                     if ( TT->tablepointers ) M_free(TT->tablepointers,"tablepointers");
02988                     for (j = 0; j < TT->buffersfill; j++ ) { /* was <= */
02989                         finishcbuf(TT->buffers[j]);
02990                     }
02991                     if ( TT->boomlijst ) M_free(TT->boomlijst,"TableTree");
02992                     /*[07apr2004 mt]:*/ /*memory leak*/
02993                     if ( TT->buffers ) M_free(TT->buffers,"Table buffers");
02994                     /*:[07apr2004 mt]*/
02995                     M_free(TT,"table");
02996                 }
02997                 M_free(T,"table");
02998             }
02999         }
03000 #ifdef TABLECLEANUP
03001         {
03002         int j;
03003         WORD *tp;
03004         for ( i = 0; i < AC.FunctionList.numglobal; i++ ) {
03005 /*
03006             Now, if the table definition is from after the .global
03007             while the function is from before, there is a problem.
03008             This could be resolved by defining CTable (=Table), Ntable
03009             and do away with the previous function definition.
03010 */
03011             if ( ( T = functions[i].tabl ) != 0 ) {
03012 /*
03013                 First restore overwritten definitions.
03014 */
03015                 if ( T->sparse ) {
03016                     T->totind = T->mdefined;
03017                     for ( j = 0, tp = T->tablepointers; j < T->totind; j++ ) {
03018                         tp += ABS(T->numind);
03019 #if TABLEEXTENSION == 2
03020                         tp[0] = tp[1];
03021 #else
03022                         tp[0] = tp[2];
03023                         tp[1] = tp[3];
03024                         tp[4] = tp[5];
03025 #endif
03026                         tp += TABLEEXTENSION;
03027                     }
03028                     RedoTableTree(T,T->totind);
03029                     if ( T->spare ) {
03030                         TABLES TT = T->spare;
03031                         TT->totind = TT->mdefined;
03032                         for ( j = 0, tp = TT->tablepointers; j < TT->totind; j++ ) {
03033                             tp += ABS(TT->numind);
03034 #if TABLEEXTENSION == 2
03035                             tp[0] = tp[1];
03036 #else
03037                             tp[0] = tp[2];
03038                             tp[1] = tp[3];
03039                             tp[4] = tp[5];
03040 #endif
```

```
03041                               tp += TABLEEXTENSION;
03042                           }
03043                           RedoTableTree(TT,TT->totind);
03044                           cbuf[TT->bufnum].numlhs = cbuf[TT->bufnum].mnumlhs;
03045                           cbuf[TT->bufnum].numrhs = cbuf[TT->bufnum].mnumrhs;
03046                       }
03047                   }
03048                   else {
03049                       for ( j = 0, tp = T->tablepointers; j < T->totind; j++ ) {
03050 #if TABLEEXTENSION == 2
03051                           tp[0] = tp[1];
03052 #else
03053                           tp[0] = tp[2];
03054                           tp[1] = tp[3];
03055                           tp[4] = tp[5];
03056 #endif
03057                       }
03058                       T->defined = T->mdefined;
03059                   }
03060                   cbuf[T->bufnum].numlhs = cbuf[T->bufnum].mnumlhs;
03061                   cbuf[T->bufnum].numrhs = cbuf[T->bufnum].mnumrhs;
03062               }
03063           }
03064           }
03065 #endif
03066           AC.FunctionList.num = AC.FunctionList.numglobal;
03067           for ( i = AC.SetList.numglobal; i < AC.SetList.num; i++ ) {
03068               if ( Sets[i].node >= 0 )
03069                   AC.varnames->namenode[Sets[i].node].type = CDELETE;
03070           }
03071           AC.SetList.numtemp = AC.SetList.num = AC.SetList.numglobal;
03072           for ( i = AC.DubiousList.numglobal; i < AC.DubiousList.num; i++ )
03073               AC.varnames->namenode[Dubious[i].node].type = CDELETE;
03074           AC.DubiousList.num = AC.DubiousList.numglobal;
03075           AC.SetElementList.numtemp = AC.SetElementList.num =
03076               AC.SetElementList.numglobal;
03077           CompactifyTree(AC.varnames,VARNAMES);
03078           AC.varnames->namefill = AC.varnames->globalnamefill;
03079           AC.varnames->nodefill = AC.varnames->globalnodefill;
03080
03081           for ( i = AC.AutoSymbolList.numglobal; i < AC.AutoSymbolList.num; i++ )
03082               AC.autonames->namenode[
03083                       ((SYMBOLS)(AC.AutoSymbolList.lijst))[i].node].type = CDELETE;
03084           AC.AutoSymbolList.num = AC.AutoSymbolList.numglobal;
03085           for ( i = AC.AutoVectorList.numglobal; i < AC.AutoVectorList.num; i++ )
03086               AC.autonames->namenode[
03087                       ((VECTORS)(AC.AutoVectorList.lijst))[i].node].type = CDELETE;
03088           AC.AutoVectorList.num = AC.AutoVectorList.numglobal;
03089           for ( i = AC.AutoIndexList.numglobal; i < AC.AutoIndexList.num; i++ )
03090               AC.autonames->namenode[
03091                       ((INDICES)(AC.AutoIndexList.lijst))[i].node].type = CDELETE;
03092           AC.AutoIndexList.num = AC.AutoIndexList.numglobal;
03093           for ( i = AC.AutoFunctionList.numglobal; i < AC.AutoFunctionList.num; i++ ) {
03094               AC.autonames->namenode[
03095                       ((FUNCTIONS)(AC.AutoFunctionList.lijst))[i].node].type = CDELETE;
03096               if ( ( T = ((FUNCTIONS)(AC.AutoFunctionList.lijst))[i].tabl ) != 0 ) {
03097                   if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
03098                   if ( T->prototype ) M_free(T->prototype,"tableprototype");
03099                   if ( T->mm ) M_free(T->mm,"tableminmax");
03100                   if ( T->flags ) M_free(T->flags,"tableflags");
03101                   if ( T->argtail ) M_free(T->argtail,"table arguments");
03102                   if ( T->boomlijst ) M_free(T->boomlijst,"TableTree");
03103                   for (j = 0; j < T->buffersfill; j++ ) { /* was <= */
03104                       finishcbuf(T->buffers[j]);
03105                   }
03106                   if ( T->spare ) {
03107                       TABLES TT = T->spare;
03108                       if ( TT->mm ) M_free(TT->mm,"tableminmax");
03109                       if ( TT->flags ) M_free(TT->flags,"tableflags");
03110                       if ( TT->tablepointers ) M_free(TT->tablepointers,"tablepointers");
03111                       for (j = 0; j < TT->buffersfill; j++ ) { /* was <= */
03112                           finishcbuf(TT->buffers[j]);
03113                       }
03114                       if ( TT->boomlijst ) M_free(TT->boomlijst,"TableTree");
03115                       M_free(TT,"table");
03116                   }
03117                   M_free(T,"table");
03118               }
03119           }
03120           AC.AutoFunctionList.num = AC.AutoFunctionList.numglobal;
03121
03122           CompactifyTree(AC.autonames,AUTONAMES);
03123
03124           AC.autonames->namefill = AC.autonames->globalnamefill;
03125           AC.autonames->nodefill = AC.autonames->globalnodefill;
03126           break;
03127       }
```

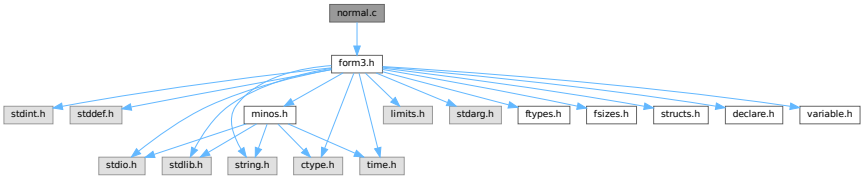
```
03128 }
03129
03130 /*
03131     #] ResetVariables :
03132     #[ RemoveDollars :
03133 */
03134
03135 void RemoveDollars(void)
03136 {
03137     DOLLARS d;
03138     CBUF *C = cbuf + AM.dbufnum;
03139     int numdollar = AP.DollarList.num;
03140     if ( numdollar > 0 ) {
03141       while ( numdollar > AM.gcNumDollars ) {
03142         numdollar--;
03143         d = Dollars + numdollar;
03144         if ( d->where && d->where != &(d->zero) && d->where != &(AM.dollarzero) ) {
03145             M_free(d->where,"dollar->where"); d->where = &(d->zero); d->size = 0;
03146         }
03147         AC.dollarnames->namenode[d->node].type = CDELETE;
03148       }
03149       AP.DollarList.num = AM.gcNumDollars;
03150       CompactifyTree(AC.dollarnames,DOLLARNAMES);
03151
03152       C->numrhs = C->mnumrhs;
03153       C->numlhs = C->mnumlhs;
03154     }
03155 }
03156
03157 /*
03158     #] RemoveDollars :
03159     #[ Globalize :
03160 */
03161
03162 void Globalize(int par)
03163 {
03164     int i, j;
03165     WORD *tp;
03166     if ( par == 1 ) {
03167         AC.SymbolList.numclear     = AC.SymbolList.num;
03168         AC.VectorList.numclear     = AC.VectorList.num;
03169         AC.IndexList.numclear      = AC.IndexList.num;
03170         AC.FunctionList.numclear   = AC.FunctionList.num;
03171         AC.SetList.numclear        = AC.SetList.num;
03172         AC.DubiousList.numclear    = AC.DubiousList.num;
03173         AC.SetElementList.numclear = AC.SetElementList.num;
03174         AC.varnames->clearnamefill = AC.varnames->namefill;
03175         AC.varnames->clearnodefill = AC.varnames->nodefill;
03176
03177         AC.AutoSymbolList.numclear   = AC.AutoSymbolList.num;
03178         AC.AutoVectorList.numclear   = AC.AutoVectorList.num;
03179         AC.AutoIndexList.numclear    = AC.AutoIndexList.num;
03180         AC.AutoFunctionList.numclear = AC.AutoFunctionList.num;
03181         AC.autonames->clearnamefill  = AC.autonames->namefill;
03182         AC.autonames->clearnodefill  = AC.autonames->nodefill;
03183     }
03184 /*  for ( i = AC.FunctionList.numglobal; i < AC.FunctionList.num; i++ ) { */
03185     for ( i = MAXBUILTINFUNCTION-FUNCTION; i < AC.FunctionList.num; i++ ) {
03186 /*
03187         We need here not only the not-yet-global functions. The already
03188         global ones may have obtained extra elements.
03189 */
03190         if ( functions[i].tabl ) {
03191             TABLES T = functions[i].tabl;
03192             if ( T->sparse ) {
03193                 T->mdefined = T->totind;
03194                 for ( j = 0, tp = T->tablepointers; j < T->totind; j++ ) {
03195                     tp += ABS(T->numind);
03196 #if TABLEEXTENSION == 2
03197                     tp[1] = tp[0];
03198 #else
03199                     tp[2] = tp[0]; tp[3] = tp[1]; tp[5] = tp[4] & (~ELEMENTUSED);
03200 #endif
03201                     tp += TABLEEXTENSION;
03202                 }
03203                 if ( T->spare ) {
03204                     TABLES TT = T->spare;
03205                     TT->mdefined = TT->totind;
03206                     for ( j = 0, tp = TT->tablepointers; j < TT->totind; j++ ) {
03207                         tp += ABS(TT->numind);
03208 #if TABLEEXTENSION == 2
03209                         tp[1] = tp[0];
03210 #else
03211                         tp[2] = tp[0]; tp[3] = tp[1]; tp[5] = tp[4] & (~ELEMENTUSED);
03212 #endif
03213                         tp += TABLEEXTENSION;
03214                     }
```

```
03215                     cbuf[TT->bufnum].mnumlhs = cbuf[TT->bufnum].numlhs;
03216                     cbuf[TT->bufnum].mnumrhs = cbuf[TT->bufnum].numrhs;
03217                 }
03218             }
03219             else {
03220                 T->mdefined = T->defined;
03221                 for ( j = 0, tp = T->tablepointers; j < T->totind; j++ ) {
03222 #if TABLEEXTENSION == 2
03223                     tp[1] = tp[0];
03224 #else
03225                     tp[2] = tp[0]; tp[3] = tp[1]; tp[5] = tp[4] & (~ELEMENTUSED);
03226 #endif
03227                 }
03228             }
03229             cbuf[T->bufnum].mnumlhs = cbuf[T->bufnum].numlhs;
03230             cbuf[T->bufnum].mnumrhs = cbuf[T->bufnum].numrhs;
03231         }
03232     }
03233     for ( i = AC.AutoFunctionList.numglobal; i < AC.AutoFunctionList.num; i++ ) {
03234         if ( ((FUNCTIONS)(AC.AutoFunctionList.lijst))[i].tabl )
03235             ((FUNCTIONS)(AC.AutoFunctionList.lijst))[i].tabl->mdefined =
03236                 ((FUNCTIONS)(AC.AutoFunctionList.lijst))[i].tabl->defined;
03237     }
03238     AC.SymbolList.numglobal     = AC.SymbolList.num;
03239     AC.VectorList.numglobal     = AC.VectorList.num;
03240     AC.IndexList.numglobal      = AC.IndexList.num;
03241     AC.FunctionList.numglobal   = AC.FunctionList.num;
03242     AC.SetList.numglobal        = AC.SetList.num;
03243     AC.DubiousList.numglobal    = AC.DubiousList.num;
03244     AC.SetElementList.numglobal = AC.SetElementList.num;
03245     AC.varnames->globalnamefill = AC.varnames->namefill;
03246     AC.varnames->globalnodefill = AC.varnames->nodefill;
03247 
03248     AC.AutoSymbolList.numglobal   = AC.AutoSymbolList.num;
03249     AC.AutoVectorList.numglobal   = AC.AutoVectorList.num;
03250     AC.AutoIndexList.numglobal    = AC.AutoIndexList.num;
03251     AC.AutoFunctionList.numglobal = AC.AutoFunctionList.num;
03252     AC.autonames->globalnamefill  = AC.autonames->namefill;
03253     AC.autonames->globalnodefill  = AC.autonames->nodefill;
03254 }
03255 
03256 /*
03257     #] Globalize :
03258     #[ TestName :
03259 */
03260 
03261 int TestName(UBYTE *name)
03262 {
03263     if ( *name == '[' ) {
03264         while ( *name ) name++;
03265         if ( name[-1] == ']' ) return(0);
03266         return(-1);
03267     }
03268     while ( *name ) {
03269         if ( *name == '_' ) return(-1);
03270         if ( *name == '$' ) return(-1);
03271         name++;
03272     }
03273     return(0);
03274 }
03275 
03276 /*
03277     #] TestName :
03278 */
```

4.89 normal.c File Reference

```
#include "form3.h"
```

Include dependency graph for normal.c:



Macros

- #define MAXNUMBEROFNONCOMTERMS 2

Functions

- int CompareFunctions (WORD *fleft, WORD *fright)
- int Commute (WORD *fleft, WORD *fright)
- int Normalize (PHEAD WORD *term)
- int ExtraSymbol (WORD sym, WORD pow, WORD nsym, WORD *ppsym, WORD *ncoef)
- int DoTheta (PHEAD WORD *t)
- int DoDelta (WORD *t)
- void DoRevert (WORD *fun, WORD *tmp)
- WORD DetCommu (WORD *terms)
- WORD DoesCommu (WORD *term)
- int TreatPolyRatFun (PHEAD WORD *prf)
- void DropCoefficient (PHEAD WORD *term)
- void DropSymbols (PHEAD WORD *term)
- int SymbolNormalize (WORD *term)
- int TestFunFlag (PHEAD WORD *tfun)
- int BracketNormalize (PHEAD WORD *term)

4.89.1 Detailed Description

Mainly the routine Normalize. This routine brings terms to standard form. Its main buffers are in AT, but they have a fixed size controlled by NORMSIZE, which limit the maximum complexity of terms which can be normalized.

Normalize is called recursively, currently via: Normalize -> ExpandRat -> Normalize.

Definition in file normal.c.

4.89.2 Macro Definition Documentation

4.89.2.1 MAXNUMBEROFNONCOMTERMS

```
#define MAXNUMBEROFNONCOMTERMS 2
```

Definition at line 4574 of file normal.c.

4.89.3 Function Documentation

4.89.3.1 CompareFunctions()

```
int CompareFunctions (
            WORD * fleft,
            WORD * fright )
```

Definition at line 55 of file normal.c.

4.89.3.2 Commute()

```
int Commute (
            WORD * fleft,
            WORD * fright )
```

Definition at line 119 of file normal.c.

4.89.3.3 Normalize()

```
int Normalize (
            PHEAD WORD * term )
```

Definition at line 193 of file normal.c.

4.89.3.4 ExtraSymbol()

```
int ExtraSymbol (
            WORD sym,
            WORD pow,
            WORD nsym,
            WORD * ppsym,
            WORD * ncoef )
```

Definition at line 4262 of file normal.c.

4.89.3.5 DoTheta()

```
int DoTheta (
            PHEAD WORD * t )
```

Definition at line 4324 of file normal.c.

4.89.3.6 DoDelta()

```
int DoDelta (
            WORD * t )
```

Definition at line 4421 of file normal.c.

4.89.3.7 DoRevert()

```
void DoRevert (
             WORD * fun,
             WORD * tmp )
```

Definition at line 4488 of file normal.c.

4.89.3.8 DetCommu()

```
WORD DetCommu (
             WORD * terms )
```

Definition at line 4576 of file normal.c.

4.89.3.9 DoesCommu()

```
WORD DoesCommu (
             WORD * term )
```

Definition at line 4635 of file normal.c.

4.89.3.10 TreatPolyRatFun()

```
int TreatPolyRatFun (
             PHEAD WORD * prf )
```

Definition at line 5026 of file normal.c.

4.89.3.11 DropCoefficient()

```
void DropCoefficient (
             PHEAD WORD * term )
```

Definition at line 5162 of file normal.c.

4.89.3.12 DropSymbols()

```
void DropSymbols (
             PHEAD WORD * term )
```

Definition at line 5180 of file normal.c.

4.89.3.13 SymbolNormalize()

```
int SymbolNormalize (
            WORD * term )
```

Routine normalizes terms that contain only symbols. Regular minimum and maximum properties are ignored.

We check whether there are negative powers in the output. This is not allowed.

Definition at line 5210 of file normal.c.

Referenced by InFunction(), and poly_sort().

4.89.3.14 TestFunFlag()

```
int TestFunFlag (
            PHEAD WORD * tfun )
```

Definition at line 5311 of file normal.c.

4.89.3.15 BracketNormalize()

```
int BracketNormalize (
            PHEAD WORD * term )
```

Definition at line 5342 of file normal.c.

4.90 normal.c

Go to the documentation of this file.

```
00001
00011 /* #[ License : */
00012 /*
00013  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00014  *   When using this file you are requested to refer to the publication
00015  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00016  *   This is considered a matter of courtesy as the development was paid
00017  *   for by FOM the Dutch physics granting agency and we would like to
00018  *   be able to track its scientific use to convince FOM of its value
00019  *   for the community.
00020  *
00021  *   This file is part of FORM.
00022  *
00023  *   FORM is free software: you can redistribute it and/or modify it under the
00024  *   terms of the GNU General Public License as published by the Free Software
00025  *   Foundation, either version 3 of the License, or (at your option) any later
00026  *   version.
00027  *
00028  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00029  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00030  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00031  *   details.
00032  *
00033  *   You should have received a copy of the GNU General Public License along
00034  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00035  */
00036 /* #] License : */
00037 /*
00038     #[ Includes : normal.c
00039 */
00040
00041 #include "form3.h"
00042 #ifdef WITHFLOAT
```

```
00043 #include <gmp.h>
00044
00045 int PackFloat(WORD *,mpf_t);
00046 int UnpackFloat(mpf_t, WORD *);
00047 void RatToFloat(mpf_t result, UWORD *formrat, int ratsize);
00048 #endif
00049 /*
00050     #] Includes :
00051     #[ Normalize :
00052         #[ CompareFunctions :
00053 */
00054
00055 int CompareFunctions(WORD *fleft,WORD *fright)
00056 {
00057     WORD k, kk;
00058     if ( AC.properorderflag ) {
00059         if ( ( *fleft >= (FUNCTION+WILDOFFSET)
00060         && functions[*fleft-FUNCTION-WILDOFFSET].spec >= TENSORFUNCTION )
00061         || ( *fleft >= FUNCTION && *fleft < (FUNCTION + WILDOFFSET)
00062         && functions[*fleft-FUNCTION].spec >= TENSORFUNCTION ) ) {}
00063         else {
00064             WORD *s1, *s2, *ss1, *ss2;
00065             s1 = fleft+FUNHEAD; s2 = fright+FUNHEAD;
00066             ss1 = fleft + fleft[1]; ss2 = fright + fright[1];
00067             while ( s1 < ss1 && s2 < ss2 ) {
00068                 k = CompArg(s1,s2);
00069                 if ( k > 0 ) return(1);
00070                 if ( k < 0 ) return(0);
00071                 NEXTARG(s1)
00072                 NEXTARG(s2)
00073             }
00074             if ( s1 < ss1 ) return(1);
00075             return(0);
00076         }
00077         k = fleft[1] - FUNHEAD;
00078         kk = fright[1] - FUNHEAD;
00079         fleft += FUNHEAD;
00080         fright += FUNHEAD;
00081         while ( k > 0 && kk > 0 ) {
00082             if ( *fleft < *fright ) return(0);
00083             else if ( *fleft++ > *fright++ ) return(1);
00084             k--; kk--;
00085         }
00086         if ( k > 0 ) return(1);
00087         return(0);
00088     }
00089     else {
00090         k = fleft[1] - FUNHEAD;
00091         kk = fright[1] - FUNHEAD;
00092         fleft += FUNHEAD;
00093         fright += FUNHEAD;
00094         while ( k > 0 && kk > 0 ) {
00095             if ( *fleft < *fright ) return(0);
00096             else if ( *fleft++ > *fright++ ) return(1);
00097             k--; kk--;
00098         }
00099         if ( k > 0 ) return(1);
00100         return(0);
00101     }
00102 }
00103
00104 /*
00105         #] CompareFunctions :
00106         #[ Commute :
00107
00108     This function gets two adjacent function pointers and decides
00109     whether these two functions should be exchanged to obtain a
00110     natural ordering.
00111
00112     Currently there is only an ordering of gamma matrices belonging
00113     to different spin lines.
00114
00115     Note that we skip for now the cases of (F)^(3/2) or 1/F and a few more
00116     of such funny functions.
00117 */
00118
00119 int Commute(WORD *fleft, WORD *fright)
00120 {
00121     WORD fun1, fun2;
00122     if ( *fleft == DOLLAREXPRESSION || *fright == DOLLAREXPRESSION ) return(0);
00123     fun1 = ABS(*fleft); fun2 = ABS(*fright);
00124     if ( *fleft >= GAMMA && *fleft <= GAMMASEVEN
00125         && *fright >= GAMMA && *fright <= GAMMASEVEN ) {
00126         if ( fleft[FUNHEAD] < AM.OffsetIndex && fleft[FUNHEAD] > fright[FUNHEAD] )
00127             return(1);
00128         return(0);
00129     }
```

```
00130     if ( fun1 >= WILDOFFSET ) fun1 -= WILDOFFSET;
00131     if ( fun2 >= WILDOFFSET ) fun2 -= WILDOFFSET;
00132     if ( ( ( functions[fun1-FUNCTION].flags & COULDCOMMUTE ) == 0 )
00133       || ( ( functions[fun2-FUNCTION].flags & COULDCOMMUTE ) == 0 ) ) return(0);
00134 /*
00135     if other conditions will come here, keep in mind that if *fleft < 0
00136     or *fright < 0 they are arguments in the exponent function as in f^(3/2)
00137 */
00138     if ( AC.CommuteInSet == 0 ) return(0);
00139 /*
00140     The code for CompareFunctions can be stolen from the commuting case.
00141
00142     We need the syntax:
00143         Commute Fun1,Fun2,...,Fun`n';
00144     For this Fun1,...,Fun`n' need to be noncommuting functions.
00145     These functions will commute with all members of the group.
00146     In the AC.paircommute buffer the representation is
00147         `n'+1,element1,...,element`n',`m'+1,element1,...,element`m',0
00148     A function can belong to more than one group.
00149     If a function commutes with itself, it is most efficient to make a separate
00150     group of two elements for it as in
00151         Commute T,T;    -> 3,T,T
00152 */
00153     if ( fun1 >= fun2 ) {
00154         WORD *group = AC.CommuteInSet, *g1, *g2, *g3;
00155         while ( *group > 0 ) {
00156             g3 = group + *group;
00157             g1 = group+1;
00158             while ( g1 < g3 ) {
00159                 if ( *g1 == fun1 || ( fun1 <= GAMMASEVEN && fun1 >= GAMMA
00160                  && *g1 <= GAMMASEVEN && *g1 >= GAMMA ) ) {
00161                     g2 = group+1;
00162                     while ( g2 < g3 ) {
00163                         if ( g1 != g2 && ( *g2 == fun2 ||
00164                          ( fun2 <= GAMMASEVEN && fun2 >= GAMMA
00165                          && *g2 <= GAMMASEVEN && *g2 >= GAMMA ) ) ) {
00166                             if ( fun1 != fun2 ) return(1);
00167                             if ( *fleft < 0 ) return(0);
00168                             if ( *fright < 0 ) return(1);
00169                             return(CompareFunctions(fleft,fright));
00170                         }
00171                         g2++;
00172                     }
00173                     break;
00174                 }
00175                 g1++;
00176             }
00177             group = g3;
00178         }
00179     }
00180     return(0);
00181 }
00182
00183 /*
00184         #] Commute :
00185         #[ Normalize :
00186
00187     This is the big normalization routine. It has a great need
00188     to be economical.
00189     The limit on the number of objects coming in is given by NORMSIZE.
00190
00191 */
00192
00193 int Normalize(PHEAD WORD *term)
00194 {
00195 /*
00196     #[ Declarations :
00197 */
00198     GETBIDENTITY
00199     WORD *t, *m, *r, i, j, k, l, nsym, *ss, *tt, *u;
00200     WORD shortnum, stype;
00201     WORD *stop, *to = 0, *from = 0;
00202
00203     WORD *ppsym, *ppvec, *ppdot, *ppdel;
00204     WORD nvec, ndot, ndel, nind, neps, nden, ncom, nnco, ncon;
00205
00206     AT.NormDepth++;
00207 #ifdef DEBUGGING
00208     if ( AT.NormDepth > 2 ) {
00209         // We don't expect this to happen in the current codebase.
00210         Terminate(-1);
00211     }
00212 #endif
00213     if ( AT.NormDepth > AT.NormDataSize ) {
00214         NORMDATA **top = AT.NormData + AT.NormDataSize;
00215         DoubleBuffer((void **)&(AT.NormData), (void **)&(top),
00216             sizeof(*AT.NormData), "double NormData pointers");
```

```
00217         AT.NormDataSize *= 2;
00218         for ( LONG i = AT.NormDepth-1; i < AT.NormDataSize; i++ ) {
00219             AT.NormData[i] = NULL;
00220         }
00221     }
00222     if ( AT.NormData[AT.NormDepth-1] == NULL ) {
00223         AT.NormData[AT.NormDepth-1] = AllocNormData();
00224     }
00225
00226     WORD  *psym = AT.NormData[AT.NormDepth-1]->psym;
00227     WORD  *pvec = AT.NormData[AT.NormDepth-1]->pvec;
00228     WORD  *pdot = AT.NormData[AT.NormDepth-1]->pdot;
00229     WORD  *pdel = AT.NormData[AT.NormDepth-1]->pdel;
00230     WORD  *pind = AT.NormData[AT.NormDepth-1]->pind;
00231     WORD **peps = AT.NormData[AT.NormDepth-1]->peps;
00232     WORD **pden = AT.NormData[AT.NormDepth-1]->pden;
00233     WORD **pcom = AT.NormData[AT.NormDepth-1]->pcom;
00234     WORD **pnco = AT.NormData[AT.NormDepth-1]->pnco;
00235     WORD **pcon = AT.NormData[AT.NormDepth-1]->pcon;
00236
00237     WORD *n_coef, ncoef;                /* Accumulator for the coefficient */
00238     WORD *n_llnum, *lnum, nnum;
00239     WORD *termout, oldtoprhs = 0, subtype;
00240     WORD ReplaceType, ReplaceVeto = 0, didcontr;
00241     int regval = 0;
00242     WORD *ReplaceSub;
00243     WORD *fillsetexp;
00244     CBUF *C = cbuf+AT.ebufnum;
00245     WORD *ANsc = 0, *ANsm = 0, *ANsr = 0, PolyFunMode;
00246 #ifdef WITHFLOAT
00247     WORD withfloat = 0;
00248     WORD *firstfloat = 0;
00249 #endif
00250     LONG oldcpointer = 0, x;
00251     n_coef = TermMalloc("NormCoef");
00252     n_llnum = TermMalloc("n_llnum");
00253     lnum = n_llnum+1;
00254 /*
00255     int termflag;
00256 */
00257 /*
00258     #] Declarations :
00259     #[ Setup :
00260 PrintTerm(term,"Normalize");
00261 */
00262
00263     if ( AT.SS == AT.S0 ) {
00264         if ( *term > AT.SS->verbMaxTermSize ) {
00265             AT.SS->verbMaxTermSize = *term;
00266         }
00267     }
00268
00269 Restart:
00270     didcontr = 0;
00271     ReplaceType = -1;
00272     t = term;
00273     if ( !*t ) {
00274         AT.NormDepth--;
00275         TermFree(n_coef,"NormCoef");
00276         TermFree(n_llnum,"n_llnum");
00277         return(regval);
00278     }
00279     r = t + *t;
00280     ncoef = r[-1];
00281     i = ABS(ncoef);
00282     r -= i;
00283     m = r;
00284     t = n_coef;
00285     NCOPY(t,r,i);
00286     termout = AT.WorkPointer;
00287     AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
00288     fillsetexp = termout+1;
00289     AN.PolyNormFlag = 0; PolyFunMode = AN.PolyFunTodo;
00290 /*
00291     termflag = 0;
00292 */
00293 /*
00294     #] Setup :
00295     #[ First scan :
00296 */
00297     nsym = nvec = ndot = ndel = neps = nden =
00298     nind = ncom = nnco = ncon = 0;
00299     ppsym = psym;
00300     ppvec = pvec;
00301     ppdot = pdot;
00302     ppdel = pdel;
00303     t = term + 1;
```

```
00304 conscan:;
00305     if ( t < m ) do {
00306         r = t + t[1];
00307         switch ( *t ) {
00308             case SYMBOL :
00309                 t += 2;
00310                 from = m;
00311                 do {
00312                     if ( t[1] == 0 ) {
00313 /*                      if ( *t == 0 || *t == MAXPOWER ) goto NormZZ; */
00314                         t += 2;
00315                         goto NextSymbol;
00316                     }
00317                     if ( *t <= DENOMINATORSYMBOL && *t >= COEFFSYMBOL ) {
00318 /*
00319                         if ( AN.NoScrat2 == 0 ) {
00320                             AN.NoScrat2 = (UWORD *)Malloc1((AM.MaxTal+2)*sizeof(UWORD),"Normalize");
00321                         }
00322 */
00323                         if ( AN.cTerm ) m = AN.cTerm;
00324                         else m = term;
00325                         m += *m;
00326                         ncoef = REDLENG(ncoef);
00327                         if ( *t == COEFFSYMBOL ) {
00328                           i = t[1];
00329                           nnum = REDLENG(m[-1]);
00330                           m -= ABS(m[-1]);
00331                           if ( i > 0 ) {
00332                             while ( i > 0 ) {
00333                                 if ( MulRat(BHEAD (UWORD *)n_coef,ncoef,(UWORD *)m,nnum,
00334                                 (UWORD *)n_coef,&ncoef) ) goto FromNorm;
00335                                 i--;
00336                             }
00337                           }
00338                           else if ( i < 0 ) {
00339                             while ( i < 0 ) {
00340                                 if ( DivRat(BHEAD (UWORD *)n_coef,ncoef,(UWORD *)m,nnum,
00341                                 (UWORD *)n_coef,&ncoef) ) goto FromNorm;
00342                                 i++;
00343                             }
00344                           }
00345                         }
00346                         else {
00347                           i = m[-1];
00348                           nnum = (ABS(i)-1)/2;
00349                           if ( *t == NUMERATORSYMBOL ) { m -= nnum + 1; }
00350                           else { m--; }
00351                           while ( *m == 0 && nnum > 1 ) { m--; nnum--; }
00352                           m -= nnum;
00353                           if ( i < 0 && *t == NUMERATORSYMBOL ) nnum = -nnum;
00354                           i = t[1];
00355                           if ( i > 0 ) {
00356                             while ( i > 0 ) {
00357                                 if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)m,nnum) )
00358                                         goto FromNorm;
00359                                 i--;
00360                             }
00361                           }
00362                           else if ( i < 0 ) {
00363                             while ( i < 0 ) {
00364                                 if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)m,nnum) )
00365                                         goto FromNorm;
00366                                 i++;
00367                             }
00368                           }
00369                         }
00370                         ncoef = INCLENG(ncoef);
00371                         t += 2;
00372                         goto NextSymbol;
00373                     }
00374                     else if ( *t == DIMENSIONSYMBOL ) {
00375                         if ( AN.cTerm ) m = AN.cTerm;
00376                         else m = term;
00377                         k = DimensionTerm(m);
00378                         if ( k == 0 ) goto NormZero;
00379                         if ( k == MAXPOSITIVE ) {
00380                             MLOCK(ErrorMessageLock);
00381                             MesPrint("Dimension_ is undefined in term %t");
00382                             MUNLOCK(ErrorMessageLock);
00383                             goto NormMin;
00384                         }
00385                         if ( k == -MAXPOSITIVE ) {
00386                             MLOCK(ErrorMessageLock);
00387                             MesPrint("Dimension_ out of range in term %t");
00388                             MUNLOCK(ErrorMessageLock);
00389                             goto NormMin;
00390                         }
```

```
00391                       if ( k > 0 ) { *((UWORD *)lnum) = k; nnum = 1; }
00392                       else { *((UWORD *)lnum) = -k; nnum = -1; }
00393                       ncoef = REDLENG(ncoef);
00394                       if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) ) goto FromNorm;
00395                       ncoef = INCLENG(ncoef);
00396                       t += 2;
00397                       goto NextSymbol;
00398                   }
00399                   if ( ( *t >= MAXPOWER && *t < 2*MAXPOWER )
00400                       || ( *t < -MAXPOWER && *t > -2*MAXPOWER ) ) {
00401 /*
00402             #[ TO SNUMBER :
00403 */
00404               if ( *t < 0 ) {
00405                   *t += MAXPOWER;
00406                   *t = -*t;
00407                   if ( t[1] & 1 ) ncoef = -ncoef;
00408               }
00409               else if ( *t == MAXPOWER ) {
00410                   if ( t[1] > 0 ) goto NormZero;
00411                   goto NormInf;
00412               }
00413               else {
00414                   *t -= MAXPOWER;
00415               }
00416               lnum[0] = *t;
00417               nnum = 1;
00418               if ( t[1] && RaisPow(BHEAD (UWORD *)lnum,&nnum,(UWORD)(ABS(t[1]))) )
00419                   goto FromNorm;
00420               ncoef = REDLENG(ncoef);
00421               if ( t[1] < 0 ) {
00422                   if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) )
00423                       goto FromNorm;
00424               }
00425               else if ( t[1] > 0 ) {
00426                   if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) )
00427                       goto FromNorm;
00428               }
00429               ncoef = INCLENG(ncoef);
00430 /*
00431             #] TO SNUMBER :
00432 */
00433                       t += 2;
00434                       goto NextSymbol;
00435                   }
00436                   if ( ( *t <= NumSymbols && *t > -MAXPOWER )
00437                    && ( symbols[*t].complex & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY ) {
00438                       if ( t[1] <= 2*MAXPOWER && t[1] >= -2*MAXPOWER ) {
00439                       t[1] %= symbols[*t].maxpower;
00440                       if ( t[1] < 0 ) t[1] += symbols[*t].maxpower;
00441                       if ( ( symbols[*t].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
00442                           if ( ( ( symbols[*t].maxpower & 1 ) == 0 ) &&
00443                           ( t[1] >= symbols[*t].maxpower/2 ) ) {
00444                               t[1] -= symbols[*t].maxpower/2; ncoef = -ncoef;
00445                           }
00446                       }
00447                       if ( t[1] == 0 ) { t += 2; goto NextSymbol; }
00448                       }
00449                   }
00450                   i = nsym;
00451                   m = ppsym;
00452                   if ( i > 0 ) do {
00453                       m -= 2;
00454                       if  ( *t == *m ) {
00455                           t++; m++;
00456                           if  ( *t > 2*MAXPOWER || *t < -2*MAXPOWER
00457                           ||  *m > 2*MAXPOWER || *m < -2*MAXPOWER ) {
00458                               MLOCK(ErrorMessageLock);
00459                               MesPrint("Illegal wildcard power combination.");
00460                               MUNLOCK(ErrorMessageLock);
00461                               goto NormMin;
00462                           }
00463                           *m += *t;
00464                           if ( ( t[-1] <= NumSymbols && t[-1] > -MAXPOWER )
00465                            && ( symbols[t[-1]].complex & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY
    ) {
00466                               *m %= symbols[t[-1]].maxpower;
00467                               if ( *m < 0 ) *m += symbols[t[-1]].maxpower;
00468                               if ( ( symbols[t[-1]].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
00469                                   if ( ( ( symbols[t[-1]].maxpower & 1 ) == 0 ) &&
00470                                   ( *m >= symbols[t[-1]].maxpower/2 ) ) {
00471                                       *m -= symbols[t[-1]].maxpower/2; ncoef = -ncoef;
00472                                   }
00473                               }
00474                           }
00475                           if  ( *m >= 2*MAXPOWER || *m <= -2*MAXPOWER ) {
00476                               MLOCK(ErrorMessageLock);
```

```
00477                                   MesPrint("Power overflow during normalization");
00478                                   MUNLOCK(ErrorMessageLock);
00479                                   goto NormMin;
00480                               }
00481                               if ( !*m ) {
00482                                   m--;
00483                                   while ( i < nsym )
00484                                       { *m = m[2]; m++; *m = m[2]; m++; i++; }
00485                                   ppsym -= 2;
00486                                   nsym--;
00487                               }
00488                               t++;
00489                               goto NextSymbol;
00490                           }
00491                       } while ( *t < *m && --i > 0 );
00492                       m = ppsym;
00493                       while ( i < nsym )
00494                           { m--; m[2] = *m; m--; m[2] = *m; i++; }
00495                       *m++ = *t++;
00496                       *m = *t++;
00497                       ppsym += 2;
00498                       nsym++;
00499 NextSymbol:;
00500                   } while ( t < r );
00501                   m = from;
00502                   break;
00503               case VECTOR :
00504                   t += 2;
00505                   do {
00506                       if ( t[0] == AM.vectorzero ) goto NormZero;
00507                       if ( t[1] == FUNNYVEC ) {
00508                           pind[nind++] = *t;
00509                           t += 2;
00510                       }
00511                       else if ( t[1] < 0 ) {
00512                           if ( *t == NOINDEX && t[1] == NOINDEX ) t += 2;
00513                           else {
00514                               if ( t[1] == AM.vectorzero ) goto NormZero;
00515                               *ppdot++ = *t++; *ppdot++ = *t++; *ppdot++ = 1; ndot++;
00516                           }
00517                       }
00518                       else { *ppvec++ = *t++; *ppvec++ = *t++; nvec += 2; }
00519                   } while ( t < r );
00520                   break;
00521               case DOTPRODUCT :
00522                   t += 2;
00523                   do {
00524                       if ( t[2] == 0 ) t += 3;
00525                       else if ( ndot > 0 && t[0] == ppdot[-3]
00526                           && t[1] == ppdot[-2] ) {
00527                           ppdot[-1] += t[2];
00528                           t += 3;
00529                           if ( ppdot[-1] == 0 ) { ppdot -= 3; ndot--; }
00530                       }
00531                       else if ( t[0] == AM.vectorzero || t[1] == AM.vectorzero ) {
00532                           if ( t[2] > 0 ) goto NormZero;
00533                           goto NormInf;
00534                       }
00535                       else {
00536                           *ppdot++ = *t++; *ppdot++ = *t++;
00537                           *ppdot++ = *t++; ndot++;
00538                       }
00539                   } while ( t < r );
00540                   break;
00541               case HAAKJE :
00542                   break;
00543               case SETSET:
00544                   if ( WildFill(BHEAD termout,term,AT.dummysubexp) < 0 ) goto FromNorm;
00545                   i = *termout;
00546                   t = termout; m = term;
00547                   NCOPY(m,t,i);
00548                   goto Restart;
00549               case DOLLAREXPRESSION :
00550 /*
00551                   We have DOLLAREXPRESSION,4,number,power
00552                   Replace by SUBEXPRESSION and exit elegantly to let
00553                   TestSub pick it up. Of course look for special cases first.
00554                   Note that we have a special compiler buffer for the values.
00555 */
00556                   if ( AR.Eside != LHSIDE ) {
00557                   DOLLARS d = Dollars + t[2];
00558 #ifdef WITHPTHREADS
00559                   int nummodopt, ptype = -1;
00560                   if ( AS.MultiThreaded ) {
00561                       for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
00562                           if ( t[2] == ModOptdollars[nummodopt].number ) break;
00563                       }
```

```
00564                     if ( nummodopt < NumModOptdollars ) {
00565                         ptype = ModOptdollars[nummodopt].type;
00566                         if ( DollarLocalCopy(ptype) ) {
00567                             d = ModOptdollars[nummodopt].dstruct+AT.identity;
00568                         }
00569                         else {
00570                             LOCK(d->pthreadslock);
00571                         }
00572                     }
00573                 }
00574 #endif
00575                 if ( d->type == DOLZERO ) {
00576 #ifdef WITHPTHREADS
00577                     if ( ptype > 0 && ! DollarLocalCopy(ptype) ) { UNLOCK(d->pthreadslock); }
00578 #endif
00579                     if ( t[3] == 0 ) goto NormZZ;
00580                     if ( t[3] < 0 ) goto NormInf;
00581                     goto NormZero;
00582                 }
00583                 else if ( d->type == DOLNUMBER ) {
00584                     nnum = d->where[0];
00585                     if ( nnum > 0 ) {
00586                         nnum = d->where[nnum-1];
00587                         if ( nnum < 0 ) { ncoef = -ncoef; nnum = -nnum; }
00588                         nnum = (nnum-1)/2;
00589                         for ( i = 1; i <= nnum; i++ ) lnum[i-1] = d->where[i];
00590                     }
00591                     if ( nnum == 0 || ( nnum == 1 && lnum[0] == 0 ) ) {
00592 #ifdef WITHPTHREADS
00593                         if ( ptype > 0 && ! DollarLocalCopy(ptype) ) { UNLOCK(d->pthreadslock); }
00594 #endif
00595                         if ( t[3] < 0 ) goto NormInf;
00596                         else if ( t[3] == 0 ) goto NormZZ;
00597                         goto NormZero;
00598                     }
00599                     if ( t[3] && RaisPow(BHEAD (UWORD *)lnum,&nnum,(UWORD)(ABS(t[3]))) ) goto
      FromNorm;
00600                     ncoef = REDLENG(ncoef);
00601                     if ( t[3] < 0 ) {
00602                         if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) ) {
00603 #ifdef WITHPTHREADS
00604                             if ( ptype > 0 && ! DollarLocalCopy(ptype) ) { UNLOCK(d->pthreadslock); }
00605 #endif
00606                             goto FromNorm;
00607                         }
00608                     }
00609                     else if ( t[3] > 0 ) {
00610                         if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) ) {
00611 #ifdef WITHPTHREADS
00612                             if ( ptype > 0 && ! DollarLocalCopy(ptype) ) { UNLOCK(d->pthreadslock); }
00613 #endif
00614                             goto FromNorm;
00615                         }
00616                     }
00617                     ncoef = INCLENG(ncoef);
00618                 }
00619                 else if ( d->type == DOLINDEX ) {
00620                     if ( d->index == 0 ) {
00621 #ifdef WITHPTHREADS
00622                         if ( ptype > 0 && ! DollarLocalCopy(ptype) ) { UNLOCK(d->pthreadslock); }
00623 #endif
00624                         goto NormZero;
00625                     }
00626                     if ( d->index != NOINDEX ) pind[nind++] = d->index;
00627                 }
00628                 else if ( d->type == DOLTERMS ) {
00629                     if ( t[3] >= MAXPOWER || t[3] <= -MAXPOWER ) {
00630                         if ( d->where[0] == 0 ) goto NormZero;
00631                         if ( d->where[d->where[0]] != 0 ) {
00632 IllDollarExp:
00633                             MLOCK(ErrorMessageLock);
00634                             MesPrint("!!!Illegal $ expansion with wildcard power!!!");
00635                             MUNLOCK(ErrorMessageLock);
00636                             goto FromNorm;
00637                         }
00638 /*
00639                         At this point we should only admit symbols and dotproducts
00640                         We expand the dollar directly and do not send it back.
00641 */
00642                         {   WORD *td, *tdstop, dj;
00643                             td = d->where+1;
00644                             tdstop = d->where+d->where[0];
00645                             if ( tdstop[-1] != 3 || tdstop[-2] != 1
00646                                 || tdstop[-3] != 1 ) goto IllDollarExp;
00647                             tdstop -= 3;
00648                             if ( td >= tdstop ) goto IllDollarExp;
00649                             while ( td < tdstop ) {
```

```
00650                                   if ( *td == SYMBOL ) {
00651                                       for ( dj = 2; dj < td[1]; dj += 2 ) {
00652                                           if ( td[dj+1] == 1 ) {
00653                                               *ppsym++ = td[dj];
00654                                               *ppsym++ = t[3];
00655                                               nsym++;
00656                                           }
00657                                           else if ( td[dj+1] == -1 ) {
00658                                               *ppsym++ = td[dj];
00659                                               *ppsym++ = -t[3];
00660                                               nsym++;
00661                                           }
00662                                           else goto IllDollarExp;
00663                                       }
00664                                   }
00665                                   else if ( *td == DOTPRODUCT ) {
00666                                       for ( dj = 2; dj < td[1]; dj += 3 ) {
00667                                           if ( td[dj+2] == 1 ) {
00668                                               *ppdot++ = td[dj];
00669                                               *ppdot++ = td[dj+1];
00670                                               *ppdot++ = t[3];
00671                                               ndot++;
00672                                           }
00673                                           else if ( td[dj+2] == -1 ) {
00674                                               *ppdot++ = td[dj];
00675                                               *ppdot++ = td[dj+1];
00676                                               *ppdot++ = -t[3];
00677                                               ndot++;
00678                                           }
00679                                           else goto IllDollarExp;
00680                                       }
00681                                   }
00682                                   else goto IllDollarExp;
00683                                   td += td[1];
00684                               }
00685                               regval = 2;
00686                               break;
00687                           }
00688                       }
00689                       t[0] = SUBEXPRESSION;
00690                       t[4] = AM.dbufnum;
00691                       if ( t[3] == 0 ) {
00692 #ifdef WITHPTHREADS
00693                           if ( ptype > 0 && ! DollarLocalCopy(ptype) ) { UNLOCK(d->pthreadslock); }
00694 #endif
00695                           break;
00696                       }
00697                       regval = 2;
00698                       t = r;
00699                       while ( t < m ) {
00700                           if ( *t == DOLLAREXPRESSION ) {
00701 #ifdef WITHPTHREADS
00702                               if ( ptype > 0 && ! DollarLocalCopy(ptype) ) { UNLOCK(d->pthreadslock); }
00703 #endif
00704                               d = Dollars + t[2];
00705 #ifdef WITHPTHREADS
00706                               if ( AS.MultiThreaded ) {
00707                                   for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
00708                                       if ( t[2] == ModOptdollars[nummodopt].number ) break;
00709                                   }
00710                                   if ( nummodopt < NumModOptdollars ) {
00711                                       ptype = ModOptdollars[nummodopt].type;
00712                                       if ( DollarLocalCopy(ptype) ) {
00713                                           d = ModOptdollars[nummodopt].dstruct+AT.identity;
00714                                       }
00715                                       else {
00716                                           LOCK(d->pthreadslock);
00717                                       }
00718                                   }
00719                               }
00720 #endif
00721                               if ( d->type == DOLTERMS ) {
00722                                   t[0] = SUBEXPRESSION;
00723                                   t[4] = AM.dbufnum;
00724                               }
00725                           }
00726                           t += t[1];
00727                       }
00728 #ifdef WITHPTHREADS
00729                       if ( ptype > 0 && ! DollarLocalCopy(ptype) ) { UNLOCK(d->pthreadslock); }
00730 #endif
00731                       goto RegEnd;
00732                   }
00733                   else {
00734 #ifdef WITHPTHREADS
00735                       if ( ptype > 0 && ! DollarLocalCopy(ptype) ) { UNLOCK(d->pthreadslock); }
00736 #endif
```

```
00737                     MLOCK(ErrorMessageLock);
00738                     MesPrint("!!!This $ variation has not been implemented yet!!!");
00739                     MUNLOCK(ErrorMessageLock);
00740                     goto NormMin;
00741                 }
00742 #ifdef WITHPTHREADS
00743                 if ( ptype > 0 && ! DollarLocalCopy(ptype) ) { UNLOCK(d->pthreadslock); }
00744 #endif
00745                 }
00746                 else {
00747                     pnco[nnco++] = t;
00748 /*
00749                     The next statement should be safe as the value is used
00750                     only by the compiler (ie the master).
00751 */
00752                     AC.lhdollarflag = 1;
00753                 }
00754                 break;
00755             case DELTA :
00756                 t += 2;
00757                 do {
00758                     if ( *t < 0 ) {
00759                         if ( *t == SUMMEDIND ) {
00760                             if ( t[1] < -NMIN4SHIFT ) {
00761                                 k = -t[1]-NMIN4SHIFT;
00762                                 k = ExtraSymbol(k,1,nsym,ppsym,&ncoef);
00763                                 nsym += k;
00764                                 ppsym += (k * 2);
00765                             }
00766                             else if ( t[1] == 0 ) goto NormZero;
00767                             else {
00768                                 if ( t[1] < 0 ) {
00769                                     lnum[0] = -t[1];
00770                                     nnum = -1;
00771                                 }
00772                                 else {
00773                                     lnum[0] = t[1];
00774                                     nnum = 1;
00775                                 }
00776                                 ncoef = REDLENG(ncoef);
00777                                 if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) )
00778                                     goto FromNorm;
00779                                 ncoef = INCLENG(ncoef);
00780                             }
00781                             t += 2;
00782                         }
00783                         else if ( *t == NOINDEX && t[1] == NOINDEX ) t += 2;
00784                         else if ( *t == EMPTYINDEX && t[1] == EMPTYINDEX ) {
00785                             *ppdel++ = *t++; *ppdel++ = *t++; ndel += 2;
00786                         }
00787                         else
00788                         if ( t[1] < 0 ) {
00789                             *ppdot++ = *t++; *ppdot++ = *t++; *ppdot++ = 1; ndot++;
00790                         }
00791                         else {
00792                             *ppvec++ = *t++; *ppvec++ = *t++; nvec += 2;
00793                         }
00794                     }
00795                     else {
00796                         if ( t[1] < 0 ) {
00797                             *ppvec++ = t[1]; *ppvec++ = *t; t+=2; nvec += 2;
00798                         }
00799                         else { *ppdel++ = *t++; *ppdel++ = *t++; ndel += 2; }
00800                     }
00801                 } while ( t < r );
00802                 break;
00803             case FACTORIAL :
00804 /*
00805                 (FACTORIAL,FUNHEAD+2,..,-SNUMBER,number)
00806 */
00807                 if ( t[FUNHEAD] == -SNUMBER && t[1] == FUNHEAD+2
00808                                             && t[FUNHEAD+1] >= 0 ) {
00809                     if ( Factorial(BHEAD t[FUNHEAD+1],(UWORD *)lnum,&nnum) )
00810                         goto FromNorm;
00811 MulIn:
00812                     ncoef = REDLENG(ncoef);
00813                     if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) ) goto FromNorm;
00814                     ncoef = INCLENG(ncoef);
00815                 }
00816                 else pcom[ncom++] = t;
00817                 break;
00818             case BERNOULLIFUNCTION :
00819 /*
00820                 (BERNOULLIFUNCTION,FUNHEAD+2,..,-SNUMBER,number)
00821 */
00822                 if ( ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] >= 0 )
00823                 && ( t[1] == FUNHEAD+2 || ( t[1] == FUNHEAD+4 &&
```

```
00824                 t[FUNHEAD+2] == -SNUMBER && ABS(t[FUNHEAD+3]) == 1 ) ) ) {
00825                     WORD inum, mnum;
00826                     if ( Bernoulli(t[FUNHEAD+1],(UWORD *)lnum,&nnum) )
00827                         goto FromNorm;
00828                     if ( nnum == 0 ) goto NormZero;
00829                     inum = nnum; if ( inum < 0 ) inum = -inum;
00830                     inum--; inum /= 2;
00831                     mnum = inum;
00832                     while ( lnum[mnum-1] == 0 ) mnum--;
00833                     if ( nnum < 0 ) mnum = -mnum;
00834                     ncoef = REDLENG(ncoef);
00835                     if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,mnum) ) goto FromNorm;
00836                     mnum = inum;
00837                     while ( lnum[inum+mnum-1] == 0 ) mnum--;
00838                     if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)(lnum+inum),mnum) ) goto
      FromNorm;
00839                     ncoef = INCLENG(ncoef);
00840                     if ( t[1] == FUNHEAD+4 && t[FUNHEAD+1] == 1
00841                      && t[FUNHEAD+3] == -1 ) ncoef = -ncoef;
00842                 }
00843                 else pcom[ncom++] = t;
00844                 break;
00845             case NUMARGSFUN:
00846 /*
00847                 Numerical function giving the number of arguments.
00848 */
00849                 k = 0;
00850                 t += FUNHEAD;
00851                 while ( t < r ) {
00852                     k++;
00853                     NEXTARG(t);
00854                 }
00855                 if ( k == 0 ) goto NormZero;
00856                 *((UWORD *)lnum) = k;
00857                 nnum = 1;
00858                 goto MulIn;
00859             case NUMFACTORS:
00860 /*
00861                 Numerical function giving the number of factors in an expression.
00862 */
00863                 t += FUNHEAD;
00864                 if ( *t == -EXPRESSION ) {
00865                     k = AS.OldNumFactors[t[1]];
00866                 }
00867                 else if ( *t == -DOLLAREXPRESSION ) {
00868                     k = Dollars[t[1]].nfactors;
00869                 }
00870                 else {
00871                     pcom[ncom++] = t;
00872                     break;
00873                 }
00874                 if ( k == 0 ) goto NormZero;
00875                 *((UWORD *)lnum) = k;
00876                 nnum = 1;
00877                 goto MulIn;
00878             case NUMTERMSFUN:
00879 /*
00880                 Numerical function giving the number of terms in the single argument.
00881 */
00882                 if ( t[FUNHEAD] < 0 ) {
00883                     if ( t[FUNHEAD] <= -FUNCTION && t[1] == FUNHEAD+1 ) break;
00884                     if ( t[FUNHEAD] > -FUNCTION && t[1] == FUNHEAD+2 ) {
00885                         if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] == 0 ) goto NormZero;
00886                         break;
00887                     }
00888                     pcom[ncom++] = t;
00889                     break;
00890                 }
00891                 if ( t[FUNHEAD] > 0 && t[FUNHEAD] == t[1]-FUNHEAD ) {
00892                     k = 0;
00893                     t += FUNHEAD+ARGHEAD;
00894                     while ( t < r ) {
00895                         k++;
00896                         t += *t;
00897                     }
00898                     if ( k == 0 ) goto NormZero;
00899                     *((UWORD *)lnum) = k;
00900                     nnum = 1;
00901                     goto MulIn;
00902                 }
00903                 else pcom[ncom++] = t;
00904                 break;
00905             case COUNTFUNCTION:
00906                 if ( AN.cTerm ) {
00907                     k = CountFun(AN.cTerm,t);
00908                 }
00909                 else {
```

```
00910                   k = CountFun(term,t);
00911               }
00912               if ( k == 0 ) goto NormZero;
00913               if ( k > 0 ) { *((UWORD *)lnum) = k; nnum = 1; }
00914               else { *((UWORD *)lnum) = -k; nnum = -1; }
00915               goto MulIn;
00916               break;
00917           case MAKERATIONAL:
00918               if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+2] == -SNUMBER
00919                   && t[1] == FUNHEAD+4 && t[FUNHEAD+3] > 1 ) {
00920                   WORD x1[2], sgn;
00921                   if ( t[FUNHEAD+1] == 0 ) goto NormZero;
00922                   if ( t[FUNHEAD+1] < 0 ) { t[FUNHEAD+1] = -t[FUNHEAD+1]; sgn = -1; }
00923                   else sgn = 1;
00924                   if ( MakeRational(t[FUNHEAD+1],t[FUNHEAD+3],x1,x1+1) ) {
00925                       static int warnflag = 1;
00926                       if ( warnflag ) {
00927                           MesPrint("%w Warning: fraction could not be reconstructed in
      MakeRational_");
00928                           warnflag = 0;
00929                       }
00930                       x1[0] = t[FUNHEAD+1]; x1[1] = 1;
00931                   }
00932                   if ( sgn < 0 ) { t[FUNHEAD+1] = -t[FUNHEAD+1]; x1[0] = -x1[0]; }
00933                   if ( x1[0] < 0 ) { sgn = -1; x1[0] = -x1[0]; }
00934                   else sgn = 1;
00935                   ncoef = REDLENG(ncoef);
00936                   if ( MulRat(BHEAD (UWORD *)n_coef,ncoef,(UWORD *)x1,sgn,
00937                   (UWORD *)n_coef,&ncoef) ) goto FromNorm;
00938                   ncoef = INCLENG(ncoef);
00939               }
00940               else {
00941                   WORD narg = 0, *tt, *ttstop, *arg1 = 0, *arg2 = 0;
00942                   UWORD *x1, *x2, *xx;
00943                   WORD nx1,nx2,nxx;
00944                   ttstop = t + t[1]; tt = t+FUNHEAD;
00945                   while ( tt < ttstop ) {
00946                       narg++;
00947                       if ( narg == 1 ) arg1 = tt;
00948                       else arg2 = tt;
00949                       NEXTARG(tt);
00950                   }
00951                   if ( narg != 2 ) goto defaultcase;
00952                   if ( *arg2 == -SNUMBER && arg2[1] <= 1 ) goto defaultcase;
00953                   else if ( *arg2 > 0 && ttstop[-1] < 0 ) goto defaultcase;
00954                   x1 = NumberMalloc("Norm-MakeRational");
00955                   if ( *arg1 == -SNUMBER ) {
00956                       if ( arg1[1] == 0 ) {
00957                           NumberFree(x1,"Norm-MakeRational");
00958                           goto NormZero;
00959                       }
00960                       if ( arg1[1] < 0 ) { x1[0] = -arg1[1]; nx1 = -1; }
00961                       else { x1[0] = arg1[1]; nx1 = 1; }
00962                   }
00963                   else if ( *arg1 > 0 ) {
00964                       WORD *tc;
00965                       nx1 = (ABS(arg2[-1])-1)/2;
00966                       tc = arg1+ARGHEAD+1+nx1;
00967                       if ( tc[0] != 1 ) {
00968                           NumberFree(x1,"Norm-MakeRational");
00969                           goto defaultcase;
00970                       }
00971                       for ( i = 1; i < nx1; i++ ) if ( tc[i] != 0 ) {
00972                           NumberFree(x1,"Norm-MakeRational");
00973                           goto defaultcase;
00974                       }
00975                       tc = arg1+ARGHEAD+1;
00976                       for ( i = 0; i < nx1; i++ ) x1[i] = tc[i];
00977                       if ( arg2[-1] < 0 ) nx1 = -nx1;
00978                   }
00979                   else {
00980                       NumberFree(x1,"Norm-MakeRational");
00981                       goto defaultcase;
00982                   }
00983                   x2 = NumberMalloc("Norm-MakeRational");
00984                   if ( *arg2 == -SNUMBER ) {
00985                       if ( arg2[1] <= 1 ) {
00986                           NumberFree(x2,"Norm-MakeRational");
00987                           NumberFree(x1,"Norm-MakeRational");
00988                           goto defaultcase;
00989                       }
00990                       else { x2[0] = arg2[1]; nx2 = 1; }
00991                   }
00992                   else if ( *arg2 > 0 ) {
00993                       WORD *tc;
00994                       nx2 = (ttstop[-1]-1)/2;
00995                       tc = arg2+ARGHEAD+1+nx2;
```

```
00996                       if ( tc[0] != 1 ) {
00997                           NumberFree(x2,"Norm-MakeRational");
00998                           NumberFree(x1,"Norm-MakeRational");
00999                           goto defaultcase;
01000                       }
01001                       for ( i = 1; i < nx2; i++ ) if ( tc[i] != 0 ) {
01002                           NumberFree(x2,"Norm-MakeRational");
01003                           NumberFree(x1,"Norm-MakeRational");
01004                           goto defaultcase;
01005                       }
01006                       tc = arg2+ARGHEAD+1;
01007                       for ( i = 0; i < nx2; i++ ) x2[i] = tc[i];
01008                   }
01009                   else {
01010                       NumberFree(x2,"Norm-MakeRational");
01011                       NumberFree(x1,"Norm-MakeRational");
01012                       goto defaultcase;
01013                   }
01014                   if ( BigLong(x1,ABS(nx1),x2,nx2) >= 0 ) {
01015                       UWORD *x3 = NumberMalloc("Norm-MakeRational");
01016                       UWORD *x4 = NumberMalloc("Norm-MakeRational");
01017                       WORD nx3, nx4;
01018                       DivLong(x1,nx1,x2,nx2,x3,&nx3,x4,&nx4);
01019                       for ( i = 0; i < ABS(nx4); i++ ) x1[i] = x4[i];
01020                       nx1 = nx4;
01021                       NumberFree(x4,"Norm-MakeRational");
01022                       NumberFree(x3,"Norm-MakeRational");
01023                   }
01024                   xx = (UWORD *)(TermMalloc("Norm-MakeRational"));
01025                   if ( MakeLongRational(BHEAD x1,nx1,x2,nx2,xx,&nxx) ) {
01026                       static int warnflag = 1;
01027                       if ( warnflag ) {
01028                           MesPrint("%w Warning: fraction could not be reconstructed in
    MakeRational_");
01029                           warnflag = 0;
01030                       }
01031                       ncoef = REDLENG(ncoef);
01032                       if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,x1,nx1) )
01033                           goto FromNorm;
01034                   }
01035                   else {
01036                       ncoef = REDLENG(ncoef);
01037                       if ( MulRat(BHEAD (UWORD *)n_coef,ncoef,xx,nxx,
01038                       (UWORD *)n_coef,&ncoef) ) goto FromNorm;
01039                   }
01040                   ncoef = INCLENG(ncoef);
01041                   TermFree(xx,"Norm-MakeRational");
01042                   NumberFree(x2,"Norm-MakeRational");
01043                   NumberFree(x1,"Norm-MakeRational");
01044               }
01045               break;
01046           case TERMFUNCTION:
01047               if ( t[1] == FUNHEAD && AN.cTerm ) {
01048                   ANsr = r; ANsm = m; ANsc = AN.cTerm;
01049                   AN.cTerm = 0;
01050                   t = ANsc + 1;
01051                   m = ANsc + *ANsc;
01052                   ncoef = REDLENG(ncoef);
01053                   nnum = REDLENG(m[-1]);
01054                   m -= ABS(m[-1]);
01055                   if ( MulRat(BHEAD (UWORD *)n_coef,ncoef,(UWORD *)m,nnum,
01056                   (UWORD *)n_coef,&ncoef) ) goto FromNorm;
01057                   ncoef = INCLENG(ncoef);
01058                   r = t;
01059               }
01060               break;
01061           case FIRSTBRACKET:
01062               if ( ( t[1] == FUNHEAD+2 ) && t[FUNHEAD] == -EXPRESSION ) {
01063                   if ( GetFirstBracket(termout,t[FUNHEAD+1]) < 0 ) goto FromNorm;
01064                   if ( *termout == 0 ) goto NormZero;
01065                   if ( *termout > 4 ) {
01066                       WORD *r1, *r2, *r3;
01067                       while ( r < m ) *t++ = *r++;
01068                       r1 = term + *term;
01069                       r2 = termout + *termout; r2 -= ABS(r2[-1]);
01070                       while ( r < r1 ) *r2++ = *r++;
01071                       r3 = termout + 1;
01072                       while ( r3 < r2 ) *t++ = *r3++;
01073                       *term = t - term;
01074                       if ( AT.WorkPointer > term && AT.WorkPointer < t )
01075                           AT.WorkPointer = t;
01076                       goto Restart;
01077                   }
01078               }
01079               break;
01080           case FIRSTTERM:
01081           case CONTENTTERM:
```

```
01082                 if ( ( t[1] == FUNHEAD+2 ) && t[FUNHEAD] == -EXPRESSION ) {
01083                     {
01084                         EXPRESSIONS e = Expressions+t[FUNHEAD+1];
01085                         POSITION oldondisk = AS.OldOnFile[t[FUNHEAD+1]];
01086                         if ( e->replace == NEWLYDEFINEDEXPRESSION ) {
01087                             AS.OldOnFile[t[FUNHEAD+1]] = e->onfile;
01088                         }
01089                         if ( *t == FIRSTTERM ) {
01090                             if ( GetFirstTerm(termout,t[FUNHEAD+1],0) < 0 ) goto FromNorm;
01091                         }
01092                         else if ( *t == CONTENTTERM ) {
01093                             if ( GetContent(termout,t[FUNHEAD+1]) < 0 ) goto FromNorm;
01094                         }
01095                         AS.OldOnFile[t[FUNHEAD+1]] = oldondisk;
01096                         if ( *termout == 0 ) goto NormZero;
01097                     }
01098 PasteIn:;
01099                     {
01100                         WORD *r1, *r2, *r3, *r4, *r5, nr1, *rterm;
01101                         r2 = termout + *termout; lnum = r2 - ABS(r2[-1]);
01102                         nnum = REDLENG(r2[-1]);
01103 
01104                         r1 = term + *term; r3 = r1 - ABS(r1[-1]);
01105                         nr1 = REDLENG(r1[-1]);
01106                         if ( Mully(BHEAD (UWORD *)lnum,&nnum,(UWORD *)r3,nr1) ) goto FromNorm;
01107                         nnum = INCLENG(nnum); nr1 = ABS(nnum); lnum[nr1-1] = nnum;
01108                         rterm = TermMalloc("FirstTerm/ContentTerm");
01109                         r4 = rterm+1; r5 = term+1; while ( r5 < t ) *r4++ = *r5++;
01110                         r5 = termout+1; while ( r5 < lnum ) *r4++ = *r5++;
01111                         r5 = r; while ( r5 < r3 ) *r4++ = *r5++;
01112                         r5 = lnum; NCOPY(r4,r5,nr1);
01113                         *rterm = r4-rterm;
01114                         nr1 = *rterm; r1 = term; r2 = rterm; NCOPY(r1,r2,nr1);
01115                         TermFree(rterm,"FirstTerm/ContentTerm");
01116                         if ( AT.WorkPointer > term && AT.WorkPointer < r1 )
01117                             AT.WorkPointer = r1;
01118                         goto Restart;
01119                     }
01120                 }
01121                 else if ( ( t[1] == FUNHEAD+2 ) && t[FUNHEAD] == -DOLLAREXPRESSION ) {
01122                     DOLLARS d = Dollars + t[FUNHEAD+1], newd = 0;
01123                     int idol, ido;
01124 #ifdef WITHPTHREADS
01125                     int nummodopt, dtype = -1;
01126                     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01127                         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01128                             if ( t[FUNHEAD+1] == ModOptdollars[nummodopt].number ) break;
01129                         }
01130                         if ( nummodopt < NumModOptdollars ) {
01131                             dtype = ModOptdollars[nummodopt].type;
01132                             if ( DollarLocalCopy(dtype) ) {
01133                                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
01134                             }
01135                         }
01136                     }
01137 #endif
01138                     if ( d->where && ( d->type == DOLTERMS || d->type == DOLNUMBER ) ) {
01139                         newd = d;
01140                     }
01141                     else {
01142                         if ( ( newd = DolToTerms(BHEAD t[FUNHEAD+1]) ) == 0 )
01143                             goto NormZero;
01144                     }
01145                     if ( newd->where[0] == 0 ) {
01146                         M_free(newd,"Copy of dollar variable");
01147                         goto NormZero;
01148                     }
01149                     if ( *t == FIRSTTERM ) {
01150                         idol = newd->where[0];
01151                         for ( ido = 0; ido < idol; ido++ ) termout[ido] = newd->where[ido];
01152                     }
01153                     else if ( *t == CONTENTTERM ) {
01154                         WORD *tterm;
01155                         tterm = newd->where;
01156                         idol = tterm[0];
01157                         for ( ido = 0; ido < idol; ido++ ) termout[ido] = tterm[ido];
01158                         tterm += *tterm;
01159                         while ( *tterm ) {
01160                             if ( ContentMerge(BHEAD termout,tterm) < 0 ) goto FromNorm;
01161                             tterm += *tterm;
01162                         }
01163                     }
01164                     if ( newd != d ) {
01165                         if ( newd->factors ) M_free(newd->factors,"Dollar factors");
01166                         M_free(newd,"Copy of dollar variable");
01167                         newd = 0;
01168                     }
```

```
01169                   goto PasteIn;
01170               }
01171               break;
01172           case TERMSINEXPR:
01173               {
01174               if ( ( t[1] == FUNHEAD+2 ) && t[FUNHEAD] == -EXPRESSION ) {
01175                   x = TermsInExpression(t[FUNHEAD+1]);
01176 multermnum:       if ( x == 0 ) goto NormZero;
01177                   if ( x < 0 ) {
01178                       x = -x;
01179                       if ( x > (LONG)WORDMASK ) { lnum[0] = x & WORDMASK;
01180                           lnum[1] = x » BITSINWORD; nnum = -2;
01181                       }
01182                       else { lnum[0] = x; nnum = -1; }
01183                   }
01184                   else if ( x > (LONG)WORDMASK ) {
01185                       lnum[0] = x & WORDMASK;
01186                       lnum[1] = x » BITSINWORD;
01187                       nnum = 2;
01188                   }
01189                   else { lnum[0] = x; nnum = 1; }
01190                   ncoef = REDLENG(ncoef);
01191                   if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) )
01192                       goto FromNorm;
01193                   ncoef = INCLENG(ncoef);
01194               }
01195               else if ( ( t[1] == FUNHEAD+2 ) && t[FUNHEAD] == -DOLLAREXPRESSION ) {
01196                   x = TermsInDollar(t[FUNHEAD+1]);
01197                   goto multermnum;
01198               }
01199               else { pcom[ncom++] = t; }
01200               }
01201               break;
01202           case SIZEOFFUNCTION:
01203               {
01204               if ( ( t[1] == FUNHEAD+2 ) && t[FUNHEAD] == -EXPRESSION ) {
01205                   x = SizeOfExpression(t[FUNHEAD+1]);
01206                   goto multermnum;
01207               }
01208               else if ( ( t[1] == FUNHEAD+2 ) && t[FUNHEAD] == -DOLLAREXPRESSION ) {
01209                   x = SizeOfDollar(t[FUNHEAD+1]);
01210                   goto multermnum;
01211               }
01212               else { pcom[ncom++] = t; }
01213               }
01214               break;
01215           case MATCHFUNCTION:
01216           case PATTERNFUNCTION:
01217               break;
01218           case BINOMIAL:
01219 /*
01220               Binomial function for internal use for the moment.
01221               The routine in reken.c should be more efficient.
01222 */
01223               if ( t[1] == FUNHEAD+4 && t[FUNHEAD] == -SNUMBER
01224               && t[FUNHEAD+1] >= 0 && t[FUNHEAD+2] == -SNUMBER
01225               && t[FUNHEAD+3] >= 0 && t[FUNHEAD+1] >= t[FUNHEAD+3] ) {
01226                   if ( t[FUNHEAD+1] > t[FUNHEAD+3] ) {
01227                       if ( GetBinom((UWORD *)lnum,&nnum,
01228                       t[FUNHEAD+1],t[FUNHEAD+3]) ) goto FromNorm;
01229                       if ( nnum == 0 ) goto NormZero;
01230                       goto MulIn;
01231                   }
01232               }
01233               else pcom[ncom++] = t;
01234               break;
01235           case SIGNFUN:
01236 /*
01237               Numerical function giving (-1)^arg
01238 */
01239               if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER ) {
01240                   if ( ( t[FUNHEAD+1] & 1 ) != 0 ) ncoef = -ncoef;
01241               }
01242               else if ( ( t[FUNHEAD] > 0 ) && ( t[1] == FUNHEAD+t[FUNHEAD] )
01243               && ( t[FUNHEAD] == ARGHEAD+1+abs(t[t[1]-1]) ) ) {
01244                   UWORD *numer1,*denom1;
01245                   WORD nsize = abs(t[t[1]-1]), nnsize, isize;
01246                   nnsize = (nsize-1)/2;
01247                   numer1 = (UWORD *)(t + FUNHEAD+ARGHEAD+1);
01248                   denom1 = numer1 + nnsize;
01249                   for ( isize = 1; isize < nnsize; isize++ ) {
01250                       if ( denom1[isize] ) break;
01251                   }
01252                   if ( ( denom1[0] != 1 ) || isize < nnsize ) {
01253                       pcom[ncom++] = t;
01254                   }
01255                   else {
```

```
01256                       if ( ( numer1[0] & 1 ) != 0 ) ncoef = -ncoef;
01257                   }
01258               }
01259               else {
01260                   goto doflags;
01261 /*                pcom[ncom++] = t; */
01262               }
01263               break;
01264           case SIGFUNCTION:
01265 /*
01266               Numerical function giving the sign of the numerical argument
01267               The sign of zero is 1.
01268               If there are roots of unity they are part of the sign.
01269 */
01270               if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER ) {
01271                   if ( t[FUNHEAD+1] < 0 ) ncoef = -ncoef;
01272               }
01273               else if ( ( t[1] == FUNHEAD+2 ) && ( t[FUNHEAD] == -SYMBOL )
01274                   && ( ( t[FUNHEAD+1] <= NumSymbols && t[FUNHEAD+1] > -MAXPOWER )
01275                   && ( symbols[t[FUNHEAD+1]].complex & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY )
     ) {
01276                   k = t[FUNHEAD+1];
01277                   from = m;
01278                   i = nsym;
01279                   m = ppsym;
01280                   if ( i > 0 ) do {
01281                       m -= 2;
01282                       if  ( k == *m ) {
01283                           m++;
01284                           *m = *m + 1;
01285                           *m %= symbols[k].maxpower;
01286                           if ( ( symbols[k].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
01287                               if ( ( ( symbols[k].maxpower & 1 ) == 0 ) &&
01288                               ( *m >= symbols[k].maxpower/2 ) ) {
01289                                   *m -= symbols[k].maxpower/2; ncoef = -ncoef;
01290                               }
01291                           }
01292                           if ( !*m ) {
01293                               m--;
01294                               while ( i < nsym )
01295                                   { *m = m[2]; m++; *m = m[2]; m++; i++; }
01296                               ppsym -= 2;
01297                               nsym--;
01298                           }
01299                           goto sigDoneSymbol;
01300                       }
01301                   } while ( k < *m && --i > 0 );
01302                   m = ppsym;
01303                   while ( i < nsym )
01304                       { m--; m[2] = *m; m--; m[2] = *m; i++; }
01305                   *m++ = k;
01306                   *m = 1;
01307                   ppsym += 2;
01308                   nsym++;
01309 sigDoneSymbol:;
01310                   m = from;
01311               }
01312               else if ( ( t[FUNHEAD] > 0 ) && ( t[1] == FUNHEAD+t[FUNHEAD] ) ) {
01313                   if ( t[FUNHEAD] == ARGHEAD+1+abs(t[t[1]-1]) ) {
01314                       if ( t[t[1]-1] < 0 ) ncoef = -ncoef;
01315                   }
01316 /*
01317                       Now we should fish out the roots of unity
01318 */
01319                   else if ( ( t[FUNHEAD+ARGHEAD]+FUNHEAD+ARGHEAD == t[1] )
01320                   && ( t[FUNHEAD+ARGHEAD+1] == SYMBOL ) ) {
01321                       WORD *ts = t + FUNHEAD+ARGHEAD+3;
01322                       WORD its = ts[-1]-2;
01323                       while ( its > 0 ) {
01324                           if ( ( *ts != 0 ) && ( ( *ts > NumSymbols || *ts <= -MAXPOWER )
01325                            || ( symbols[*ts].complex & VARTYPEROOTOFUNITY ) != VARTYPEROOTOFUNITY )
     ) {
01326                               goto signogood;
01327                           }
01328                           ts += 2; its -= 2;
01329                       }
01330 /*
01331                       Now we have only roots of unity which should be
01332                       registered in the list of symbols.
01333 */
01334                       if ( t[t[1]-1] < 0 ) ncoef = -ncoef;
01335                       ts = t + FUNHEAD+ARGHEAD+3;
01336                       its = ts[-1]-2;
01337                       from = m;
01338                       while ( its > 0 ) {
01339                           i = nsym;
01340                           m = ppsym;
```

```
01341                               if ( i > 0 ) do {
01342                                   m -= 2;
01343                                   if  ( *ts == *m ) {
01344                                       ts++; m++;
01345                                       *m += *ts;
01346                                       if ( ( ts[-1] <= NumSymbols && ts[-1] > -MAXPOWER ) &&
01347                                        ( symbols[ts[-1]].complex & VARTYPEROOTOFUNITY ) ==
      VARTYPEROOTOFUNITY ) {
01348                                           *m %= symbols[ts[-1]].maxpower;
01349                                           if ( *m < 0 ) *m += symbols[ts[-1]].maxpower;
01350                                           if ( ( symbols[ts[-1]].complex & VARTYPEMINUS ) ==
      VARTYPEMINUS ) {
01351                                               if ( ( ( symbols[ts[-1]].maxpower & 1 ) == 0 ) &&
01352                                               ( *m >= symbols[ts[-1]].maxpower/2 ) ) {
01353                                                   *m -= symbols[ts[-1]].maxpower/2; ncoef = -ncoef;
01354                                               }
01355                                           }
01356                                       }
01357                                       if ( !*m ) {
01358                                           m--;
01359                                           while ( i < nsym )
01360                                               { *m = m[2]; m++; *m = m[2]; m++; i++; }
01361                                           ppsym -= 2;
01362                                           nsym--;
01363                                       }
01364                                       ts++; its -= 2;
01365                                       goto sigNextSymbol;
01366                                   }
01367                               } while ( *ts < *m && --i > 0 );
01368                               m = ppsym;
01369                               while ( i < nsym )
01370                                   { m--; m[2] = *m; m--; m[2] = *m; i++; }
01371                               *m++ = *ts++;
01372                               *m = *ts++;
01373                               ppsym += 2;
01374                               nsym++; its -= 2;
01375 sigNextSymbol:;
01376                           }
01377                           m = from;
01378                       }
01379                       else {
01380 signogood:                pcom[ncom++] = t;
01381                       }
01382                   }
01383                   else pcom[ncom++] = t;
01384                   break;
01385               case ABSFUNCTION:
01386 /*
01387                   Numerical function giving the absolute value of the
01388                   numerical argument. Or roots of unity.
01389 */
01390                   if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER ) {
01391                       k = t[FUNHEAD+1];
01392                       if ( k < 0 ) k = -k;
01393                       if ( k == 0 ) goto NormZero;
01394                       *((UWORD *)lnum) = k; nnum = 1;
01395                       goto MulIn;
01396
01397                   }
01398                   else if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SYMBOL ) {
01399                       k = t[FUNHEAD+1];
01400                       if ( ( k > NumSymbols || k <= -MAXPOWER )
01401                        || ( symbols[k].complex & VARTYPEROOTOFUNITY ) != VARTYPEROOTOFUNITY )
01402                           goto absnogood;
01403                   }
01404                   else if ( ( t[FUNHEAD] > 0 ) && ( t[1] == FUNHEAD+t[FUNHEAD] )
01405                   && ( t[1] == FUNHEAD+ARGHEAD+t[FUNHEAD+ARGHEAD] ) ) {
01406                       if ( t[FUNHEAD] == ARGHEAD+1+abs(t[t[1]-1]) ) {
01407                           WORD *ts;
01408 absnosymbols:             ts = t + t[1] -1;
01409                           ncoef = REDLENG(ncoef);
01410                           nnum = REDLENG(*ts);
01411                           if ( nnum < 0 ) nnum = -nnum;
01412                           if ( MulRat(BHEAD (UWORD *)n_coef,ncoef,
01413                           (UWORD *)(ts-ABS(*ts)+1),nnum,
01414                           (UWORD *)n_coef,&ncoef) ) goto FromNorm;
01415                           ncoef = INCLENG(ncoef);
01416                       }
01417 /*
01418                       Now get rid of the roots of unity. This includes i_
01419 */
01420                       else if ( t[FUNHEAD+ARGHEAD+1] == SYMBOL ) {
01421                           WORD *ts = t+FUNHEAD+ARGHEAD+1;
01422                           WORD its = ts[1] - 2;
01423                           ts += 2;
01424                           while ( its > 0 ) {
01425                               if ( *ts == 0 ) { }
```

```
01426                           else if ( ( *ts > NumSymbols || *ts <= -MAXPOWER )
01427                            || ( symbols[*ts].complex & VARTYPEROOTOFUNITY )
01428                               != VARTYPEROOTOFUNITY ) goto absnogood;
01429                           its -= 2; ts += 2;
01430                       }
01431                       goto absnosymbols;
01432                   }
01433                   else {
01434 absnogood:                pcom[ncom++] = t;
01435                   }
01436               }
01437               else pcom[ncom++] = t;
01438               break;
01439           case MODFUNCTION:
01440           case MOD2FUNCTION:
01441 /*
01442               Mod function. Does work if two arguments and the
01443               second argument is a positive short number
01444 */
01445               if ( t[1] == FUNHEAD+4 && t[FUNHEAD] == -SNUMBER
01446                   && t[FUNHEAD+2] == -SNUMBER && t[FUNHEAD+3] > 1 ) {
01447                   WORD tmod;
01448                   tmod = (t[FUNHEAD+1]%t[FUNHEAD+3]);
01449                   if ( tmod < 0 ) tmod += t[FUNHEAD+3];
01450                   if ( *t == MOD2FUNCTION && tmod > t[FUNHEAD+3]/2 )
01451                       tmod -= t[FUNHEAD+3];
01452                   if ( tmod < 0 ) {
01453                       *((UWORD *)lnum) = -tmod;
01454                       nnum = -1;
01455                   }
01456                   else if ( tmod > 0 ) {
01457                       *((UWORD *)lnum) = tmod;
01458                       nnum = 1;
01459                   }
01460                   else goto NormZero;
01461                   goto MulIn;
01462               }
01463               else if ( t[1] > t[FUNHEAD+2] && t[FUNHEAD] > 0
01464               && t[FUNHEAD+t[FUNHEAD]] == -SNUMBER
01465               && t[FUNHEAD+t[FUNHEAD]+1] > 1
01466               && t[1] == FUNHEAD+2+t[FUNHEAD] ) {
01467                   WORD *ttt = t+FUNHEAD, iii;
01468                   iii = ttt[*ttt-1];
01469                   if ( *ttt == ttt[ARGHEAD]+ARGHEAD &&
01470                       ttt[ARGHEAD] == ABS(iii)+1 ) {
01471                       WORD ncmod = 1;
01472                       WORD cmod = ttt[*ttt+1];
01473                       iii = REDLENG(iii);
01474                       if ( *t == MODFUNCTION ) {
01475                           if ( TakeModulus((UWORD *)(ttt+ARGHEAD+1)
01476                           ,&iii,(UWORD *)(&cmod),ncmod,UNPACK|NOINVERSES) )
01477                               goto FromNorm;
01478                       }
01479                       else {
01480                           if ( TakeModulus((UWORD *)(ttt+ARGHEAD+1)
01481                           ,&iii,(UWORD *)(&cmod),ncmod,UNPACK|POSNEG|NOINVERSES) )
01482                               goto FromNorm;
01483                       }
01484                       if ( *t == MOD2FUNCTION && ttt[ARGHEAD+1] > cmod/2 && iii > 0 ) {
01485                           ttt[ARGHEAD+1] -= cmod;
01486                       }
01487                       if ( ttt[ARGHEAD+1] < 0 ) {
01488                           *((UWORD *)lnum) = -ttt[ARGHEAD+1];
01489                           nnum = -1;
01490                       }
01491                       else if ( ttt[ARGHEAD+1] > 0 ) {
01492                           *((UWORD *)lnum) = ttt[ARGHEAD+1];
01493                           nnum = 1;
01494                       }
01495                       else goto NormZero;
01496                       goto MulIn;
01497                   }
01498               }
01499               else if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER ) {
01500                   *((UWORD *)lnum) = t[FUNHEAD+1];
01501                   if ( *lnum == 0 ) goto NormZero;
01502                   nnum = 1;
01503                   goto MulIn;
01504               }
01505               else if ( ( ( t[FUNHEAD] < 0 ) && ( t[FUNHEAD] == -SNUMBER )
01506               && ( t[1] >= ( FUNHEAD+6+ARGHEAD ) )
01507               && ( t[FUNHEAD+2] >= 4+ARGHEAD )
01508               && ( t[t[1]-1] == t[FUNHEAD+2+ARGHEAD]-1 ) ) ||
01509               ( ( t[FUNHEAD] > 0 )
01510               && ( t[FUNHEAD]-ARGHEAD-1 == ABS(t[FUNHEAD+t[FUNHEAD]-1]) )
01511               && ( t[FUNHEAD+t[FUNHEAD]]-ARGHEAD-1 == t[t[1]-1] ) ) ) {
01512 /*
```

```
01513                     Check that the last (long) number is integer
01514 */
01515                     WORD *ttt = t + t[1], iii, iii1;
01516                     UWORD coefbuf[2], *coef2, ncoef2;
01517                     iii = (ttt[-1]-1)/2;
01518                     ttt -= iii;
01519                     if ( ttt[-1] != 1 ) {
01520 exitfromhere:
01521                         pcom[ncom++] = t;
01522                         break;
01523                     }
01524                     iii--;
01525                     for ( iii1 = 0; iii1 < iii; iii1++ ) {
01526                         if ( ttt[iii1] != 0 ) goto exitfromhere;
01527                     }
01528 /*
01529                     Now we have a hit!
01530                     The first argument will be put in lnum.
01531                     It will be a rational.
01532                     The second argument will be a long integer in coef2.
01533 */
01534                     ttt = t + FUNHEAD;
01535                     if ( *ttt < 0 ) {
01536                         if ( ttt[1] < 0 ) {
01537                             nnum = -1; lnum[0] = -ttt[1]; lnum[1] = 1;
01538                         }
01539                         else {
01540                             nnum = 1; lnum[0] = ttt[1]; lnum[1] = 1;
01541                         }
01542                     }
01543                     else {
01544                         nnum = ABS(ttt[ttt[0]-1] - 1);
01545                         for ( iii = 0; iii < nnum; iii++ ) {
01546                             lnum[iii] = ttt[ARGHEAD+1+iii];
01547                         }
01548                         nnum = nnum/2;
01549                         if ( ttt[ttt[0]-1] < 0 ) nnum = -nnum;
01550                     }
01551                     NEXTARG(ttt);
01552                     if ( *ttt < 0 ) {
01553                         coef2 = coefbuf;
01554                         ncoef2 = 3; *coef2 = (UWORD)(ttt[1]);
01555                         coef2[1] = 1;
01556                     }
01557                     else {
01558                         coef2 = (UWORD *)(ttt+ARGHEAD+1);
01559                         ncoef2 = (ttt[ttt[0]-1]-1)/2;
01560                     }
01561                     if ( TakeModulus((UWORD *)lnum,&nnum,(UWORD *)coef2,ncoef2,
01562                                 UNPACK|NOINVERSES|FROMFUNCTION) ) {
01563                         goto FromNorm;
01564                     }
01565                     if ( *t == MOD2FUNCTION && nnum > 0 ) {
01566                         UWORD *coef3 = NumberMalloc("Mod2Function"), two = 2;
01567                         WORD ncoef3;
01568                         if ( MulLong((UWORD *)lnum,nnum,&two,1,coef3,&ncoef3) )
01569                             goto FromNorm;
01570                         if ( BigLong(coef3,ncoef3,(UWORD *)coef2,ncoef2) > 0 ) {
01571                             nnum = -nnum;
01572                             AddLong((UWORD *)lnum,nnum,(UWORD *)coef2,ncoef2
01573                                     ,(UWORD *)lnum,&nnum);
01574                             nnum = -nnum;
01575                         }
01576                         NumberFree(coef3,"Mod2Function");
01577                     }
01578 /*
01579                     Do we have to pack? No, because the answer is not a fraction
01580 */
01581                     ncoef = REDLENG(ncoef);
01582                     if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) )
01583                             goto FromNorm;
01584                     ncoef = INCLENG(ncoef);
01585                 }
01586                 else pcom[ncom++] = t;
01587                 break;
01588             case EXTEUCLIDEAN:
01589                 {
01590                     WORD argcount = 0, *tc, *ts, xc, xs, *tcc;
01591                     UWORD *Num1, *Num2, *Num3, *Num4;
01592                     WORD size1, size2, size3, size4, space;
01593                     tc = t+FUNHEAD; ts = t + t[1];
01594                     while ( argcount < 3 && tc < ts ) { NEXTARG(tc); argcount++; }
01595                     if ( argcount != 2 ) goto defaultcase;
01596                     if ( t[FUNHEAD] == -SNUMBER ) {
01597                         if ( t[FUNHEAD+1] <= 1 ) goto defaultcase;
01598                         if ( t[FUNHEAD+2] == -SNUMBER ) {
01599                             if ( t[FUNHEAD+3] <= 1 ) goto defaultcase;
```

```
01600                           Num2 = NumberMalloc("modinverses");
01601                           *Num2 = t[FUNHEAD+3]; size2 = 1;
01602                       }
01603                       else {
01604                           if ( ts[-1] < 0 ) goto defaultcase;
01605                           if ( ts[-1] != t[FUNHEAD+2]-ARGHEAD-1 ) goto defaultcase;
01606                           xs = (ts[-1]-1)/2;
01607                           tcc = ts-xs-1;
01608                           if ( *tcc != 1 ) goto defaultcase;
01609                           for ( i = 1; i < xs; i++ ) {
01610                               if ( tcc[i] != 0 ) goto defaultcase;
01611                           }
01612                           Num2 = NumberMalloc("modinverses");
01613                           size2 = xs;
01614                           for ( i = 0; i < xs; i++ ) Num2[i] = t[FUNHEAD+ARGHEAD+3+i];
01615                       }
01616                       Num1 = NumberMalloc("modinverses");
01617                       *Num1 = t[FUNHEAD+1]; size1 = 1;
01618                   }
01619                   else {
01620                       tc = t + FUNHEAD + t[FUNHEAD];
01621                       if ( tc[-1] < 0 ) goto defaultcase;
01622                       if ( tc[-1] != t[FUNHEAD]-ARGHEAD-1 ) goto defaultcase;
01623                       xc = (tc[-1]-1)/2;
01624                       tcc = tc-xc-1;
01625                       if ( *tcc != 1 ) goto defaultcase;
01626                       for ( i = 1; i < xc; i++ ) {
01627                           if ( tcc[i] != 0 ) goto defaultcase;
01628                       }
01629                       if ( *tc == -SNUMBER ) {
01630                           if ( tc[1] <= 1 ) goto defaultcase;
01631                           Num2 = NumberMalloc("modinverses");
01632                           *Num2 = tc[1]; size2 = 1;
01633                       }
01634                       else {
01635                           if ( ts[-1] < 0 ) goto defaultcase;
01636                           if ( ts[-1] != t[FUNHEAD+2]-ARGHEAD-1 ) goto defaultcase;
01637                           xs = (ts[-1]-1)/2;
01638                           tcc = ts-xs-1;
01639                           if ( *tcc != 1 ) goto defaultcase;
01640                           for ( i = 1; i < xs; i++ ) {
01641                               if ( tcc[i] != 0 ) goto defaultcase;
01642                           }
01643                           Num2 = NumberMalloc("modinverses");
01644                           size2 = xs;
01645                           for ( i = 0; i < xs; i++ ) Num2[i] = tc[ARGHEAD+1+i];
01646                       }
01647                       Num1 = NumberMalloc("modinverses");
01648                       size1 = xc;
01649                       for ( i = 0; i < xc; i++ ) Num1[i] = t[FUNHEAD+ARGHEAD+1+i];
01650                   }
01651                   Num3 = NumberMalloc("modinverses");
01652                   Num4 = NumberMalloc("modinverses");
01653                   GetLongModInverses(BHEAD Num1,size1,Num2,size2
01654                               ,Num3,&size3,Num4,&size4);
01655 /*
01656                   Now we have to compose the answer. This needs more space
01657                   and hence we have to put this inside the term.
01658                   Compute first how much extra space we need.
01659                   Then move the trailing part of the term upwards.
01660                   Do not forget relevant pointers!!! (r, m, termout, AT.WorkPointer)
01661 */
01662                   space = 0;
01663                   if ( ( size3 == 1 || size3 == -1 ) && (*Num3&TOPBITONLY) == 0 ) space += 2;
01664                   else space += ARGHEAD + 2*ABS(size3) + 2;
01665                   if ( ( size4 == 1 || size4 == -1 ) && (*Num4&TOPBITONLY) == 0 ) space += 2;
01666                   else space += ARGHEAD + 2*ABS(size4) + 2;
01667                   tt = term + *term; u = tt + space;
01668                   while ( tt >= ts ) *--u = *--tt;
01669                   m += space; r += space;
01670                   *term += space;
01671                   t[1] += space;
01672                   if ( ( size3 == 1 || size3 == -1 ) && (*Num3&TOPBITONLY) == 0 ) {
01673                       *ts++ = -SNUMBER; *ts = (WORD)(*Num3);
01674                       if ( size3 < 0 ) *ts = -*ts;
01675                       ts++;
01676                   }
01677                   else {
01678                       *ts++ = 2*ABS(size3)+ARGHEAD+2;
01679                       *ts++ = 0; FILLARG(ts)
01680                       *ts++ = 2*ABS(size3)+1;
01681                       for ( i = 0; i < ABS(size3); i++ ) *ts++ = Num3[i];
01682                       *ts++ = 1;
01683                       for ( i = 1; i < ABS(size3); i++ ) *ts++ = 0;
01684                       if ( size3 < 0 ) *ts++ = 2*size3-1;
01685                       else             *ts++ = 2*size3+1;
01686                   }
```

```
01687                     if ( ( size4 == 1 || size4 == -1 ) && (*Num4&TOPBITONLY) == 0 ) {
01688                         *ts++ = -SNUMBER; *ts = *Num4;
01689                         if ( size4 < 0 ) *ts = -*ts;
01690                         ts++;
01691                     }
01692                     else {
01693                         *ts++ = 2*ABS(size4)+ARGHEAD+2;
01694                         *ts++ = 0; FILLARG(ts)
01695                         *ts++ = 2*ABS(size4)+2;
01696                         for ( i = 0; i < ABS(size4); i++ ) *ts++ = Num4[i];
01697                         *ts++ = 1;
01698                         for ( i = 1; i < ABS(size4); i++ ) *ts++ = 0;
01699                         if ( size4 < 0 ) *ts++ = 2*size4-1;
01700                         else             *ts++ = 2*size4+1;
01701                     }
01702                     NumberFree(Num4,"modinverses");
01703                     NumberFree(Num3,"modinverses");
01704                     NumberFree(Num1,"modinverses");
01705                     NumberFree(Num2,"modinverses");
01706                     t[2] = 0; /* mark function as clean. */
01707                     goto Restart;
01708                 }
01709                 break;
01710             case GCDFUNCTION:
01711 #ifdef EVALUATEGCD
01712 #ifdef NEWGCDFUNCTION
01713                 {
01714 /*
01715                 Has two integer arguments
01716                 Four cases: S,S, S,L, L,S, L,L
01717 */
01718                 WORD *num1, *num2, size1, size2, stor1, stor2, *ttt, ti;
01719                 if ( t[1] == FUNHEAD+4 && t[FUNHEAD] == -SNUMBER
01720                 && t[FUNHEAD+2] == -SNUMBER && t[FUNHEAD+1] != 0
01721                 && t[FUNHEAD+3] != 0 ) {    /* Short,Short */
01722                     stor1 = t[FUNHEAD+1];
01723                     stor2 = t[FUNHEAD+3];
01724                     if ( stor1 < 0 ) stor1 = -stor1;
01725                     if ( stor2 < 0 ) stor2 = -stor2;
01726                     num1 = &stor1; num2 = &stor2;
01727                     size1 = size2 = 1;
01728                     goto gcdcalc;
01729                 }
01730                 else if ( t[1] > FUNHEAD+4 ) {
01731                     if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] != 0
01732                     && t[FUNHEAD+2] == t[1]-FUNHEAD-2 &&
01733                     ABS(t[t[1]-1]) == t[FUNHEAD+2]-1-ARGHEAD ) { /* Short,Long */
01734                         num2 = t + t[1];
01735                         size2 = ABS(num2[-1]);
01736                         ttt = num2-1;
01737                         num2 -= size2;
01738                         size2 = (size2-1)/2;
01739                         ti = size2;
01740                         while ( ti > 1 && ttt[-1] == 0 ) { ttt--; ti--; }
01741                         if ( ti == 1 && ttt[-1] == 1 ) {
01742                             stor1 = t[FUNHEAD+1];
01743                             if ( stor1 < 0 ) stor1 = -stor1;
01744                             num1 = &stor1;
01745                             size1 = 1;
01746                             goto gcdcalc;
01747                         }
01748                         else pcom[ncom++] = t;
01749                     }
01750                     else if ( t[FUNHEAD] > 0 &&
01751                     t[FUNHEAD]-1-ARGHEAD == ABS(t[t[FUNHEAD]+FUNHEAD-1]) ) {
01752                         num1 = t + FUNHEAD + t[FUNHEAD];
01753                         size1 = ABS(num1[-1]);
01754                         ttt = num1-1;
01755                         num1 -= size1;
01756                         size1 = (size1-1)/2;
01757                         ti = size1;
01758                         while ( ti > 1 && ttt[-1] == 0 ) { ttt--; ti--; }
01759                         if ( ti == 1 && ttt[-1] == 1 ) {
01760                          if ( t[1]-FUNHEAD == t[FUNHEAD]+2 && t[t[1]-2] == -SNUMBER
01761                          && t[t[1]-1] != 0 ) { /* Long,Short */
01762                             stor2 = t[t[1]-1];
01763                             if ( stor2 < 0 ) stor2 = -stor2;
01764                             num2 = &stor2;
01765                             size2 = 1;
01766                             goto gcdcalc;
01767                          }
01768                          else if ( t[1]-FUNHEAD == t[FUNHEAD]+t[FUNHEAD+t[FUNHEAD]]
01769                          && ABS(t[t[1]-1]) == t[FUNHEAD+t[FUNHEAD]] - ARGHEAD-1 ) {
01770                           num2 = t + t[1];
01771                           size2 = ABS(num2[-1]);
01772                           ttt = num2-1;
01773                           num2 -= size2;
```

```
01774                           size2 = (size2-1)/2;
01775                           ti = size2;
01776                           while ( ti > 1 && ttt[-1] == 0 ) { ttt--; ti--; }
01777                           if ( ti == 1 && ttt[-1] == 1 ) {
01778 gcdcalc:                    if ( GcdLong(BHEAD (UWORD *)num1,size1,(UWORD *)num2,size2
01779                                 ,(UWORD *)lnum,&nnum) ) goto FromNorm;
01780                             goto MulIn;
01781                           }
01782                           else pcom[ncom++] = t;
01783                         }
01784                         else pcom[ncom++] = t;
01785                       }
01786                       else pcom[ncom++] = t;
01787                   }
01788                   else pcom[ncom++] = t;
01789               }
01790               else pcom[ncom++] = t;
01791               }
01792 #else
01793               {
01794                   WORD *gcd = AT.WorkPointer;
01795                   if ( ( gcd = EvaluateGcd(BHEAD t) ) == 0 ) goto FromNorm;
01796                   if ( *gcd == 4 && gcd[1] == 1 && gcd[2] == 1 && gcd[4] == 0 ) {
01797                       AT.WorkPointer = gcd;
01798                   }
01799                   else if ( gcd[*gcd] == 0 ) {
01800                       WORD *t1, iii, change, *num, *den, numsize, densize;
01801                       if ( gcd[*gcd-1] < *gcd-1 ) {
01802                           t1 = gcd+1;
01803                           for ( iii = 2; iii < t1[1]; iii += 2 ) {
01804                               change = ExtraSymbol(t1[iii],t1[iii+1],nsym,ppsym,&ncoef);
01805                               nsym += change;
01806                               ppsym += change * 2;
01807                           }
01808                       }
01809                       t1 = gcd + *gcd;
01810                       iii = t1[-1]; num = t1-iii; numsize = (iii-1)/2;
01811                       den = num + numsize; densize = numsize;
01812                       while ( numsize > 1 && num[numsize-1] == 0 ) numsize--;
01813                       while ( densize > 1 && den[densize-1] == 0 ) densize--;
01814                       if ( numsize > 1 || num[0] != 1 ) {
01815                           ncoef = REDLENG(ncoef);
01816                           if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)num,numsize) ) goto
      FromNorm;
01817                           ncoef = INCLENG(ncoef);
01818                       }
01819                       if ( densize > 1 || den[0] != 1 ) {
01820                           ncoef = REDLENG(ncoef);
01821                           if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)den,densize) ) goto
      FromNorm;
01822                           ncoef = INCLENG(ncoef);
01823                       }
01824                       AT.WorkPointer = gcd;
01825                   }
01826                   else {  /* a whole expression */
01827 /*
01828                       Action: Put the expression in a compiler buffer.
01829                       Insert a SUBEXPRESSION subterm
01830                       Set the return value of the routine such that in
01831                       Generator the term gets sent again to TestSub.
01832
01833                       1: put in C (ebufnum)
01834                       2: after that the WorkSpace is free again.
01835                       3: insert the SUBEXPRESSION
01836                       4: copy the top part of the term down
01837 */
01838                       LONG size = AT.WorkPointer - gcd;
01839
01840                       ss = AddRHS(AT.ebufnum,1);
01841                       while ( (ss + size + 10) > C->Top ) ss = DoubleCbuffer(AT.ebufnum,ss,13);
01842                       tt = gcd;
01843                       NCOPY(ss,tt,size);
01844                       C->rhs[C->numrhs+1] = ss;
01845                       C->Pointer = ss;
01846
01847                       t[0] = SUBEXPRESSION;
01848                       t[1] = SUBEXPSIZE;
01849                       t[2] = C->numrhs;
01850                       t[3] = 1;
01851                       t[4] = AT.ebufnum;
01852                       t += 5;
01853                       tt = term + *term;
01854                       while ( r < tt ) *t++ = *r++;
01855                       *term = t - term;
01856
01857                       regval = 1;
01858                       goto RegEnd;
```

```
01859                     }
01860                 }
01861 #endif
01862 #else
01863                 MesPrint(" Unexpected call to EvaluateGCD");
01864                 Terminate(-1);
01865 #endif
01866                 break;
01867             case MINFUNCTION:
01868             case MAXFUNCTION:
01869                 if ( t[1] == FUNHEAD ) break;
01870                 {
01871                     WORD *ttt = t + FUNHEAD;
01872                     WORD *tttstop = t + t[1];
01873                     WORD tterm[4], iii;
01874                     while ( ttt < tttstop ) {
01875                         if ( *ttt > 0 ) {
01876                             if ( ttt[ARGHEAD]-1 > ABS(ttt[*ttt-1]) ) goto nospec;
01877                             ttt += *ttt;
01878                         }
01879                         else {
01880                             if ( *ttt != -SNUMBER ) goto nospec;
01881                             ttt += 2;
01882                         }
01883                     }
01884 /*
01885                     Function has only numerical arguments
01886                     Pick up the first argument.
01887 */
01888                     ttt = t + FUNHEAD;
01889                     if ( *ttt > 0 ) {
01890 loadnew1:
01891                         for ( iii = 0; iii < ttt[ARGHEAD]; iii++ )
01892                             n_llnum[iii] = ttt[ARGHEAD+iii];
01893                         ttt += *ttt;
01894                     }
01895                     else {
01896 loadnew2:
01897                         if ( ttt[1] == 0 ) {
01898                             n_llnum[0] = n_llnum[1] = n_llnum[2] = n_llnum[3] = 0;
01899                         }
01900                         else {
01901                             n_llnum[0] = 4;
01902                             if ( ttt[1] > 0 ) { n_llnum[1] = ttt[1]; n_llnum[3] = 3; }
01903                             else { n_llnum[1] = -ttt[1]; n_llnum[3] = -3; }
01904                             n_llnum[2] = 1;
01905                         }
01906                         ttt += 2;
01907                     }
01908 /*
01909                     Now loop over the other arguments
01910 */
01911                     while ( ttt < tttstop ) {
01912                         if ( *ttt > 0 ) {
01913                             if ( n_llnum[0] == 0 ) {
01914                                 if ( ( *t == MINFUNCTION && ttt[*ttt-1] < 0 )
01915                                 || ( *t == MAXFUNCTION && ttt[*ttt-1] > 0 ) )
01916                                     goto loadnew1;
01917                             }
01918                             else {
01919                                 ttt += ARGHEAD;
01920                                 iii = CompCoef(n_llnum,ttt);
01921                                 if ( ( iii > 0 && *t == MINFUNCTION )
01922                                 || ( iii < 0 && *t == MAXFUNCTION ) ) {
01923                                     for ( iii = 0; iii < ttt[0]; iii++ )
01924                                         n_llnum[iii] = ttt[iii];
01925                                 }
01926                             }
01927                             ttt += *ttt;
01928                         }
01929                         else {
01930                             if ( n_llnum[0] == 0 ) {
01931                                 if ( ( *t == MINFUNCTION && ttt[1] < 0 )
01932                                 || ( *t == MAXFUNCTION && ttt[1] > 0 ) )
01933                                     goto loadnew2;
01934                             }
01935                             else if ( ttt[1] == 0 ) {
01936                                 if ( ( *t == MINFUNCTION && n_llnum[*n_llnum-1] > 0 )
01937                                 || ( *t == MAXFUNCTION && n_llnum[*n_llnum-1] < 0 ) ) {
01938                                     n_llnum[0] = 0;
01939                                 }
01940                             }
01941                             else {
01942                                 tterm[0] = 4; tterm[2] = 1;
01943                                 if ( ttt[1] < 0 ) { tterm[1] = -ttt[1]; tterm[3] = -3; }
01944                                 else { tterm[1] = ttt[1]; tterm[3] = 3; }
01945                                 iii = CompCoef(n_llnum,tterm);
```

```
01946                                 if ( ( iii > 0 && *t == MINFUNCTION )
01947                                 || ( iii < 0 && *t == MAXFUNCTION ) ) {
01948                                     for ( iii = 0; iii < 4; iii++ )
01949                                         n_llnum[iii] = tterm[iii];
01950                                 }
01951                             }
01952                             ttt += 2;
01953                         }
01954                     }
01955                     if ( n_llnum[0] == 0 ) goto NormZero;
01956                     ncoef = REDLENG(ncoef);
01957                     nnum = REDLENG(n_llnum[*n_llnum-1]);
01958                     if ( MulRat(BHEAD (UWORD *)n_coef,ncoef,(UWORD *)lnum,nnum,
01959                     (UWORD *)n_coef,&ncoef) ) goto FromNorm;
01960                     ncoef = INCLENG(ncoef);
01961                 }
01962                 break;
01963             case INVERSEFACTORIAL:
01964                 if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] >= 0 ) {
01965                     if ( Factorial(BHEAD t[FUNHEAD+1],(UWORD *)lnum,&nnum) )
01966                         goto FromNorm;
01967                     ncoef = REDLENG(ncoef);
01968                     if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) ) goto FromNorm;
01969                     ncoef = INCLENG(ncoef);
01970                 }
01971                 else {
01972 nospec:             pcom[ncom++] = t;
01973                 }
01974                 break;
01975             case MAXPOWEROF:
01976                 if ( ( t[FUNHEAD] == -SYMBOL )
01977                      && ( t[FUNHEAD+1] > 0 ) && ( t[1] == FUNHEAD+2 ) ) {
01978                     *((UWORD *)lnum) = symbols[t[FUNHEAD+1]].maxpower;
01979                     nnum = 1;
01980                     goto MulIn;
01981                 }
01982                 else { pcom[ncom++] = t; }
01983                 break;
01984             case MINPOWEROF:
01985                 if ( ( t[FUNHEAD] == -SYMBOL )
01986                      && ( t[FUNHEAD] > 0 ) && ( t[1] == FUNHEAD+2 ) ) {
01987                     *((UWORD *)lnum) = symbols[t[FUNHEAD+1]].minpower;
01988                     nnum = 1;
01989                     goto MulIn;
01990                 }
01991                 else { pcom[ncom++] = t; }
01992                 break;
01993             case PRIMENUMBER :
01994                 if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER
01995                      && t[FUNHEAD+1] > 0 ) {
01996                     UWORD xp = (UWORD)(NextPrime(BHEAD t[FUNHEAD+1]));
01997                     ncoef = REDLENG(ncoef);
01998                     if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,&xp,1) ) goto FromNorm;
01999                     ncoef = INCLENG(ncoef);
02000                 }
02001                 else goto defaultcase;
02002                 break;
02003             case MOEBIUS:
02004 /*
02005                 Numerical function giving -1,0,1 or no evaluation
02006 */
02007                 if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER
02008                     && t[FUNHEAD+1] > 0 ) {
02009                     WORD val = Moebius(BHEAD t[FUNHEAD+1]);
02010                     if ( val == 0 ) goto NormZero;
02011                     if ( val < 0 ) ncoef = -ncoef;
02012                 }
02013                 else {
02014                     pcom[ncom++] = t;
02015                 }
02016                 break;
02017             case LNUMBER :
02018                 ncoef = REDLENG(ncoef);
02019                 if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)(t+3),t[2]) ) goto FromNorm;
02020                 ncoef = INCLENG(ncoef);
02021                 break;
02022             case SNUMBER :
02023                 if ( t[2] < 0 ) {
02024                     t[2] = -t[2];
02025                     if ( t[3] & 1 ) ncoef = -ncoef;
02026                 }
02027                 else if ( t[2] == 0 ) {
02028                     if ( t[3] < 0 ) goto NormInf;
02029                     goto NormZero;
02030                 }
02031                 lnum[0] = t[2];
02032                 nnum = 1;
```

```
02033                 if ( t[3] && RaisPow(BHEAD (UWORD *)lnum,&nnum,(UWORD)(ABS(t[3]))) ) goto FromNorm;
02034                 ncoef = REDLENG(ncoef);
02035                 if ( t[3] < 0 ) {
02036                     if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) )
02037                         goto FromNorm;
02038                 }
02039                 else if ( t[3] > 0 ) {
02040                     if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) )
02041                         goto FromNorm;
02042                 }
02043                 ncoef = INCLENG(ncoef);
02044                 break;
02045             case GAMMA :
02046             case GAMMAI :
02047             case GAMMAFIVE :
02048             case GAMMASIX :
02049             case GAMMASEVEN :
02050                 if ( t[1] == FUNHEAD ) {
02051                     MLOCK(ErrorMessageLock);
02052                     MesPrint("Gamma matrix without spin line encountered.");
02053                     MUNLOCK(ErrorMessageLock);
02054                     goto NormMin;
02055                 }
02056                 pnco[nnco++] = t;
02057                 t += FUNHEAD+1;
02058                 goto ScanCont;
02059             case LEVICIVITA :
02060                 peps[neps++] = t;
02061                 if ( ( t[2] & DIRTYFLAG ) == DIRTYFLAG ) {
02062                     t[2] &= ~DIRTYFLAG;
02063                     t[2] |= DIRTYSYMFLAG;
02064                 }
02065                 t += FUNHEAD;
02066 ScanCont:       while ( t < r ) {
02067                     if ( *t >= AM.OffsetIndex &&
02068                         ( *t >= AM.DumInd || ( *t < AM.WilInd &&
02069                         indices[*t-AM.OffsetIndex].dimension ) ) )
02070                         pcon[ncon++] = t;
02071                     t++;
02072                 }
02073                 break;
02074             case EXPONENT :
02075                 {
02076                     WORD *rr;
02077                     k = 1;
02078                     rr = t + FUNHEAD;
02079                     if ( *rr == ARGHEAD || ( *rr == -SNUMBER && rr[1] == 0 ) )
02080                         k = 0;
02081                     if ( *rr == -SNUMBER && rr[1] == 1 ) break;
02082                     if ( *rr <= -FUNCTION ) k = *rr;
02083                     NEXTARG(rr)
02084                     if ( *rr == ARGHEAD || ( *rr == -SNUMBER && rr[1] == 0 ) ) {
02085                         if ( k == 0 ) goto NormZZ;
02086                         break;
02087                     }
02088                     if ( *rr == -SNUMBER && rr[1] > 0 && rr[1] < MAXPOWER && k < 0 ) {
02089                         k = -k;
02090                         if ( functions[k-FUNCTION].commute ) {
02091                             for ( i = 0; i < rr[1]; i++ ) pnco[nnco++] = rr-1;
02092                         }
02093                         else {
02094                             for ( i = 0; i < rr[1]; i++ ) pcom[ncom++] = rr-1;
02095                         }
02096                         break;
02097                     }
02098                     if ( k == 0 ) goto NormZero;
02099                     if ( t[FUNHEAD] == -SYMBOL && *rr == -SNUMBER && t[1] == FUNHEAD+4 ) {
02100                         if ( rr[1] < MAXPOWER ) {
02101                             t[FUNHEAD+2] = t[FUNHEAD+1]; t += FUNHEAD+2;
02102                             from = m;
02103                             goto NextSymbol;
02104                         }
02105                     }
02106 /*
02107                     if ( ( t[FUNHEAD] > 0 && t[FUNHEAD+1] != 0 )
02108                     || ( *rr > 0 && rr[1] != 0 ) ) {}
02109                     else
02110 */
02111                     t[2] &= ~DIRTYSYMFLAG;
02112
02113                     pnco[nnco++] = t;
02114                 }
02115                 break;
02116             case DENOMINATOR :
02117                 t[2] &= ~DIRTYSYMFLAG;
02118                 pden[nden++] = t;
02119                 pnco[nnco++] = t;
```

```
02120                 break;
02121             case INDEX :
02122                 t += 2;
02123                 do {
02124                     if ( *t == 0 || *t == AM.vectorzero ) goto NormZero;
02125                     if ( *t > 0 && *t < AM.OffsetIndex ) {
02126                         lnum[0] = *t++;
02127                         nnum = 1;
02128                         ncoef = REDLENG(ncoef);
02129                         if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) )
02130                             goto FromNorm;
02131                         ncoef = INCLENG(ncoef);
02132                     }
02133                     else if ( *t == NOINDEX ) t++;
02134                     else pind[nind++] = *t++;
02135                 } while ( t < r );
02136                 break;
02137             case SUBEXPRESSION :
02138                 if ( t[3] == 0 ) break;
02139             case EXPRESSION :
02140                 goto RegEnd;
02141             case ROOTFUNCTION :
02142 /*
02143                 Tries to take the n-th root inside the rationals
02144                 If this is not possible, it clears all flags and
02145                 hence tries no more.
02146                 Notation:
02147                 root_(power(=integer),(rational)number)
02148 */
02149                 { WORD nc;
02150                 if ( t[2] == 0 ) goto defaultcase;
02151                 if ( t[FUNHEAD] != -SNUMBER || t[FUNHEAD+1] < 0 ) goto defaultcase;
02152                 if ( t[FUNHEAD+2] == -SNUMBER ) {
02153                     if ( t[FUNHEAD+1] == 0 && t[FUNHEAD+3] == 0 ) goto NormZZ;
02154                     if ( t[FUNHEAD+1] == 0 ) break;
02155                     if ( t[FUNHEAD+3] < 0 ) {
02156                         AT.WorkPointer[0] = -t[FUNHEAD+3];
02157                         nc = -1;
02158                     }
02159                     else {
02160                         AT.WorkPointer[0] = t[FUNHEAD+3];
02161                         nc = 1;
02162                     }
02163                     AT.WorkPointer[1] = 1;
02164                 }
02165                 else if ( t[FUNHEAD+2] == t[1]-FUNHEAD-2
02166                 && t[FUNHEAD+2] == t[FUNHEAD+2+ARGHEAD]+ARGHEAD
02167                 && ABS(t[t[1]-1]) == t[FUNHEAD+2+ARGHEAD] - 1 ) {
02168                     WORD *r1, *r2;
02169                     if ( t[FUNHEAD+1] == 0 ) break;
02170                     i = t[t[1]-1]; r1 = t + FUNHEAD+ARGHEAD+3;
02171                     nc = REDLENG(i);
02172                     i = ABS(i) - 1;
02173                     r2 = AT.WorkPointer;
02174                     while ( --i >= 0 ) *r2++ = *r1++;
02175                 }
02176                 else goto defaultcase;
02177                 if ( TakeRatRoot((UWORD *)AT.WorkPointer,&nc,t[FUNHEAD+1]) ) {
02178                     t[2] = 0;
02179                     goto defaultcase;
02180                 }
02181                 ncoef = REDLENG(ncoef);
02182                 if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)AT.WorkPointer,nc) )
02183                         goto FromNorm;
02184                 if ( nc < 0 ) nc = -nc;
02185                 if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)(AT.WorkPointer+nc),nc) )
02186                         goto FromNorm;
02187                 ncoef = INCLENG(ncoef);
02188                 }
02189                 break;
02190             case RANDOMFUNCTION :
02191                 {
02192                 WORD nnc, nc, nca, nr;
02193                 UWORD xx;
02194 /*
02195                 Needs one positive integer argument.
02196                 returns (wranf()%argument)+1.
02197                 We may call wranf several times to paste UWORDS together
02198                 when we need long numbers.
02199                 We make little errors when taking the % operator
02200                 (not 100% uniform). We correct for that by redoing the
02201                 calculation in the (unlikely) case that we are in leftover area
02202 */
02203                 if ( t[1] == FUNHEAD ) goto defaultcase;
02204                 if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER &&
02205                     t[FUNHEAD+1] > 0 ) {
02206                     if ( t[FUNHEAD+1] == 1 ) break;
```

```
02207 redoshort:
02208                     ((UWORD *)AT.WorkPointer)[0] = wranf(BHEAD0);
02209                     ((UWORD *)AT.WorkPointer)[1] = wranf(BHEAD0);
02210                     nr = 2;
02211                     if ( ((UWORD *)AT.WorkPointer)[1] == 0 ) {
02212                         nr = 1;
02213                         if ( ((UWORD *)AT.WorkPointer)[0] == 0 ) {
02214                             nr = 0;
02215                         }
02216                     }
02217                     xx = (UWORD)(t[FUNHEAD+1]);
02218                     if ( nr ) {
02219                     DivLong((UWORD *)AT.WorkPointer,nr
02220                             ,&xx,1
02221                             ,((UWORD *)AT.WorkPointer)+4,&nnc
02222                             ,((UWORD *)AT.WorkPointer)+2,&nc);
02223                     ((UWORD *)AT.WorkPointer)[4] = 0;
02224                     ((UWORD *)AT.WorkPointer)[5] = 0;
02225                     ((UWORD *)AT.WorkPointer)[6] = 1;
02226                     DivLong((UWORD *)AT.WorkPointer+4,3
02227                             ,&xx,1
02228                             ,((UWORD *)AT.WorkPointer)+9,&nnc
02229                             ,((UWORD *)AT.WorkPointer)+7,&nca);
02230                     AddLong((UWORD *)AT.WorkPointer+4,3
02231                             ,((UWORD *)AT.WorkPointer)+7,-nca
02232                             ,((UWORD *)AT.WorkPointer)+9,&nnc);
02233                     if ( BigLong((UWORD *)AT.WorkPointer,nr
02234                             ,((UWORD *)AT.WorkPointer)+9,nnc) >= 0 ) goto redoshort;
02235                     }
02236                     else nc = 0;
02237                     if ( nc == 0 ) {
02238                         AT.WorkPointer[2] = (WORD)xx;
02239                         nc = 1;
02240                     }
02241                     ncoef = REDLENG(ncoef);
02242                     if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,((UWORD *)(AT.WorkPointer))+2,nc) )
02243                             goto FromNorm;
02244                     ncoef = INCLENG(ncoef);
02245                 }
02246                 else if ( t[FUNHEAD] > 0 && t[1] == t[FUNHEAD]+FUNHEAD
02247                 && ABS(t[t[1]-1]) == t[FUNHEAD]-1-ARGHEAD && t[t[1]-1] > 0 ) {
02248                     WORD nna, nnb, nni, nnb2, nnb2a;
02249                     UWORD *nnt;
02250                     nna = t[t[1]-1];
02251                     nnb2 = nna-1;
02252                     nnb = nnb2/2;
02253                     nnt = (UWORD *)(t+t[1]-1-nnb);  /* start of denominator */
02254                     if ( *nnt != 1 ) goto defaultcase;
02255                     for ( nni = 1; nni < nnb; nni++ ) {
02256                         if ( nnt[nni] != 0 ) goto defaultcase;
02257                     }
02258                     nnt = (UWORD *)(t + FUNHEAD + ARGHEAD + 1);
02259
02260                     for ( nni = 0; nni < nnb2; nni++ ) {
02261                         ((UWORD *)AT.WorkPointer)[nni] = wranf(BHEAD0);
02262                     }
02263                     nnb2a = nnb2;
02264                     while ( nnb2a > 0 && ((UWORD *)AT.WorkPointer)[nnb2a-1] == 0 ) nnb2a--;
02265                     if ( nnb2a > 0 ) {
02266                     DivLong((UWORD *)AT.WorkPointer,nnb2a
02267                             ,nnt,nnb
02268                             ,((UWORD *)AT.WorkPointer)+2*nnb2,&nnc
02269                             ,((UWORD *)AT.WorkPointer)+nnb2,&nc);
02270                     for ( nni = 0; nni < nnb2; nni++ ) {
02271                         ((UWORD *)AT.WorkPointer)[nni+2*nnb2] = 0;
02272                     }
02273                     ((UWORD *)AT.WorkPointer)[3*nnb2] = 1;
02274                     DivLong((UWORD *)AT.WorkPointer+2*nnb2,nnb2+1
02275                             ,nnt,nnb
02276                             ,((UWORD *)AT.WorkPointer)+4*nnb2+1,&nnc
02277                             ,((UWORD *)AT.WorkPointer)+3*nnb2+1,&nca);
02278                     AddLong((UWORD *)AT.WorkPointer+2*nnb2,nnb2+1
02279                             ,((UWORD *)AT.WorkPointer)+3*nnb2+1,-nca
02280                             ,((UWORD *)AT.WorkPointer)+4*nnb2+1,&nnc);
02281                     if ( BigLong((UWORD *)AT.WorkPointer,nnb2a
02282                             ,((UWORD *)AT.WorkPointer)+4*nnb2+1,nnc) >= 0 ) goto redoshort;
02283                     }
02284                     else nc = 0;
02285                     if ( nc == 0 ) {
02286                         for ( nni = 0; nni < nnb; nni++ ) {
02287                             ((UWORD *)AT.WorkPointer)[nnb2+nni] = nnt[nni];
02288                         }
02289                         nc = nnb;
02290                     }
02291                     ncoef = REDLENG(ncoef);
02292                     if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,((UWORD *)(AT.WorkPointer))+nnb2,nc) )
02293                             goto FromNorm;
```

```
02294                   ncoef = INCLENG(ncoef);
02295               }
02296               else goto defaultcase;
02297               }
02298               break;
02299           case RANPERM :
02300               if ( *t == RANPERM && t[1] > FUNHEAD && t[FUNHEAD] <= -FUNCTION ) {
02301                   WORD **pwork;
02302                   WORD *mm, *ww, *ow = AT.WorkPointer;
02303                   WORD *Array, *targ, *argstop, narg = 0, itot;
02304                   int ie;
02305                   argstop = t+t[1];
02306                   targ = t+FUNHEAD+1;
02307                   while ( targ < argstop ) {
02308                       narg++; NEXTARG(targ);
02309                   }
02310                   WantAddPointers(narg);
02311                   pwork = AT.pWorkSpace + AT.pWorkPointer;
02312                   targ = t+FUNHEAD+1; narg = 0;
02313                   while ( targ < argstop ) {
02314                       pwork[narg++] = targ;
02315                       NEXTARG(targ);
02316                   }
02317 /*
02318                   Make a random permutation of the numbers 0,...,narg-1
02319                   The following code works also for narg == 0 and narg == 1
02320 */
02321                   ow = AT.WorkPointer;
02322                   Array = AT.WorkPointer;
02323                   AT.WorkPointer += narg;
02324                   for ( i = 0; i < narg; i++ ) Array[i] = i;
02325                   for ( i = 2; i <= narg; i++ ) {
02326                       itot = (WORD)(iranf(BHEAD i));
02327                       for ( j = 0; j < itot; j++ ) CYCLE1(WORD,Array,i)
02328                   }
02329                   mm = AT.WorkPointer;
02330                   *mm++ = -t[FUNHEAD];
02331                   *mm++ = t[1] - 1;
02332                   for ( ie = 2; ie < FUNHEAD; ie++ ) *mm++ = t[ie];
02333                   for ( i = 0; i < narg; i++ ) {
02334                       ww = pwork[Array[i]];
02335                       CopyArg(mm,ww);
02336                   }
02337                   mm = AT.WorkPointer; t++; ww = t;
02338                   i = mm[1]; NCOPY(ww,mm,i)
02339                   AT.WorkPointer = ow;
02340                   goto TryAgain;
02341               }
02342               pnco[nnco++] = t;
02343               break;
02344           case PUTFIRST : /* First argument should be a function, second a number */
02345               if ( ( t[2] & DIRTYFLAG ) != 0 && t[FUNHEAD] <= -FUNCTION
02346               && t[FUNHEAD+1] == -SNUMBER && t[FUNHEAD+2] > 0 ) {
02347                   WORD *rr = t+t[1], *mm = t+FUNHEAD+3, *tt, *tt1, *tt2, num = 0;
02348 /*
02349                   now count the arguments. If not enough: no action.
02350 */
02351                   while ( mm < rr ) { num++; NEXTARG(mm); }
02352                   if ( num < t[FUNHEAD+2] ) { pnco[nnco++] = t; break; }
02353                   /* Replace "putfirst_" with the resulting function code. mm goes to arg start. */
02354                   *t = -t[FUNHEAD]; mm = t+FUNHEAD+3;
02355                   /* Set i to the arg number we are putting first, then move mm to its start. */
02356                   i = t[FUNHEAD+2];
02357                   while ( --i > 0 ) { NEXTARG(mm); }
02358                   /* Keep a pointer to the arguments trailing the selected: */
02359                   WORD *argTail = mm;
02360                   NEXTARG(argTail);
02361                   tt = TermMalloc("Select_"); /* Move selected out of the way into tmp space */
02362                   tt1 = tt;
02363                   /* Normal argument: */
02364                   if ( *mm > 0 ) {
02365                       for ( i = 0; i < *mm; i++ ) *tt1++ = mm[i];
02366                   }
02367                   /* Fast-notation function: single word */
02368                   else if ( *mm <= -FUNCTION ) { *tt1++ = *mm; }
02369                   /* Fast-notation symbol, number etc: two words */
02370                   else { *tt1++ = mm[0]; *tt1++ = mm[1]; }
02371                   /* Put tt2 at the start of the original arguments */
02372                   tt2 = t+FUNHEAD+3;
02373                   /* Copy leading original arguments after the new first, in the tmp space. */
02374                   while ( tt2 < mm ) *tt1++ = *tt2++;
02375                   /* i contains the size so far. tt1 goes to the start of the tmp space.
02376                       tt2 to the final argument location (we overwrite "putfirst_") */
02377                   i = tt1-tt; tt1 = tt; tt2 = t+FUNHEAD;
02378                   /* Copy everything so far to its final place */
02379                   NCOPY(tt2,tt1,i);
02380                   /* We are finished with the tmp space */
```

```
02381                   TermFree(tt,"Select_");
02382                   /* Now copy the trailing args. Use the stored pointer, *mm has been edited
02383                       during the copy to tt2 above! NEXTARG(mm) would produce nonsense. */
02384                   while ( argTail < rr ) *tt2++ = *argTail++;
02385                   /* Set the size of all function args */
02386                   t[1] = tt2 - t;
02387                   /* Now copy the rest of the term, and set the final term size */
02388                   rr = term + *term;
02389                   while ( argTail < rr ) *tt2++ = *argTail++;
02390                   *term = tt2-term;
02391                   if ( functions[*t-FUNCTION].spec == TENSORFUNCTION ) {
02392                       // If the output function is a tensor, we have one more job to do.
02393                       // The args are formatted as single words, representing indices or vectors.
02394                       // There is no ARGHEAD.
02395                       WORD *dst = t + FUNHEAD;
02396                       WORD *src = dst + 1;
02397                       // Strip the type information from the args:
02398                       while ( src < t + t[1] ) {
02399                           *dst = *src;
02400                           dst++; src++; src++;
02401                       }
02402                       t[1] = dst - t;
02403                       // Now copy the rest of the term. We've advanced src one too many above.
02404                       src--;
02405                       while ( src < term + *term ) {
02406                           *dst++ = *src++;
02407                       }
02408                       *term = dst - term;
02409                   }
02410                   goto Restart;
02411               }
02412               else pnco[nnco++] = t;
02413               break;
02414 #ifdef WITHFLOAT
02415           case FLOATFUN :
02416 /*
02417               If it is a proper float_ we give it special treatment.
02418               If it is not proper, we treat it as a regular commuting function.
02419 */
02420               if ( withfloat == 0 ) {
02421                   if ( TestFloat(t) == 0 ) goto defaultcase;
02422                   firstfloat = t;
02423                   withfloat = 1;
02424               }
02425               else { /* There are now at least two float_'s. Time for action. */
02426                   if ( TestFloat(t) == 0 ) goto defaultcase;
02427                   if ( withfloat == 1 ) UnpackFloat(aux4,firstfloat);
02428                   withfloat++;
02429                   UnpackFloat(aux5,t);
02430                   mpf_mul(aux4,aux4,aux5);
02431               }
02432               break;
02433 #endif
02434           case INTFUNCTION :
02435 /*
02436               Can be resolved if the first argument is a number
02437               and the second argument either doesn't exist or has
02438               the value +1, 0, -1
02439               +1 : rounding up
02440               0  : rounding towards zero
02441               -1 : rounding down (same as no argument)
02442 */
02443               if ( t[1] <= FUNHEAD ) break;
02444               {
02445                   WORD *rr, den, num;
02446                   to = t + FUNHEAD;
02447                   if ( *to > 0 ) {
02448                       if ( *to == ARGHEAD ) goto NormZero;
02449                       rr = to + *to;
02450                       i = rr[-1];
02451                       j = ABS(i);
02452                       if ( to[ARGHEAD] != j+1 ) goto NoInteg;
02453                       if ( rr >= r ) k = -1;
02454                       else if ( *rr == ARGHEAD ) { k = 0; rr += ARGHEAD; }
02455                       else if ( *rr == -SNUMBER ) { k = rr[1]; rr += 2; }
02456                       else goto NoInteg;
02457                       if ( rr != r ) goto NoInteg;
02458                       if ( k > 1 || k < -1 ) goto NoInteg;
02459                       to += ARGHEAD+1;
02460                       j = (j-1) >> 1;
02461                       i = ( i < 0 ) ? -j: j;
02462                       UnPack((UWORD *)to,i,&den,&num);
02463 /*
02464                       Potentially the use of NoScrat2 is unsafe.
02465                       It makes the routine not reentrant, but because it is
02466                       used only locally and because we only call the
02467                       low level routines DivLong and AddLong which never
```

```
02468                         make calls involving Normalize, things are OK after all
02469 */
02470                         if ( AN.NoScrat2 == 0 ) {
02471                             AN.NoScrat2 = (UWORD *)Malloc1((AM.MaxTal+2)*sizeof(UWORD),"Normalize");
02472                         }
02473                         if ( DivLong((UWORD *)to,num,(UWORD *)(to+j),den
02474                         ,(UWORD *)AT.WorkPointer,&num,AN.NoScrat2,&den) ) goto FromNorm;
02475                         if ( k < 0 && den < 0 ) {
02476                             *AN.NoScrat2 = 1;
02477                             den = -1;
02478                             if ( AddLong((UWORD *)AT.WorkPointer,num
02479                             ,AN.NoScrat2,den,(UWORD *)AT.WorkPointer,&num) )
02480                                 goto FromNorm;
02481                         }
02482                         else if ( k > 0 && den > 0 ) {
02483                             *AN.NoScrat2 = 1;
02484                             den = 1;
02485                             if ( AddLong((UWORD *)AT.WorkPointer,num,
02486                             AN.NoScrat2,den,(UWORD *)AT.WorkPointer,&num) )
02487                                 goto FromNorm;
02488                         }
02489
02490                     }
02491                     else if ( *to == -SNUMBER ) {   /* No rounding needed */
02492                         if ( to[1] < 0 ) { *AT.WorkPointer = -to[1]; num = -1; }
02493                         else if ( to[1] == 0 ) goto NormZero;
02494                         else { *AT.WorkPointer = to[1]; num = 1; }
02495                     }
02496                     else goto NoInteg;
02497                     if ( num == 0 ) goto NormZero;
02498                     ncoef = REDLENG(ncoef);
02499                     if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)AT.WorkPointer,num) )
02500                         goto FromNorm;
02501                     ncoef = INCLENG(ncoef);
02502                     break;
02503                 }
02504 NoInteg:;
02505 /*
02506                 Fall through if it cannot be resolved
02507 */
02508             default :
02509 defaultcase:;
02510                 if ( *t < FUNCTION ) {
02511 /* INTERNAL_ERROR_EXCL_START */
02512                     MLOCK(ErrorMessageLock);
02513                     MesPrint("!>Illegal code in Norm");
02514 #ifdef DEBUGON
02515                     {
02516                         UBYTE OutBuf[140];
02517                         AO.OutFill = AO.OutputLine = OutBuf;
02518                         t = term;
02519                         AO.OutSkip = 3;
02520                         FiniLine();
02521                         i = *t;
02522                         while ( --i >= 0 ) {
02523                             TalToLine((UWORD)(*t++));
02524                             TokenToLine((UBYTE *)"  ");
02525                         }
02526                         AO.OutSkip = 0;
02527                         FiniLine();
02528                     }
02529 #endif
02530                     MUNLOCK(ErrorMessageLock);
02531                     goto NormMin;
02532 /* INTERNAL_ERROR_EXCL_STOP */
02533                 }
02534                 if ( *t == REPLACEMENT ) {
02535                     if ( AR.Eside != LHSIDE ) ReplaceVeto--;
02536                     pcom[ncom++] = t;
02537                     break;
02538                 }
02539 /*
02540                 if ( *t == AM.termfunnum && t[1] == FUNHEAD+2
02541                 && t[FUNHEAD] == -DOLLAREXPRESSION ) termflag++;
02542 */
02543                 if ( *t == DUMMYFUN || *t == DUMMYTEN ) {}
02544                 else {
02545                     if ( *t < (FUNCTION + WILDOFFSET) ) {
02546                         if ( ( ( functions[*t-FUNCTION].maxnumargs > 0 )
02547                         || ( functions[*t-FUNCTION].minnumargs > 0 ) )
02548                          && ( ( t[2] & DIRTYFLAG ) != 0 ) ) {
02549 /*
02550                             Number of arguments is bounded. And we have not checked.
02551 */
02552                             WORD *ta = t + FUNHEAD, *tb = t + t[1];
02553                             int numarg = 0;
02554                             while ( ta < tb ) { numarg++; NEXTARG(ta) }
```

```
02555                               if ( ( functions[*t-FUNCTION].maxnumargs > 0 )
02556                               && ( numarg >= functions[*t-FUNCTION].maxnumargs ) )
02557                                   goto NormZero;
02558                               if ( ( functions[*t-FUNCTION].minnumargs > 0 )
02559                               && ( numarg < functions[*t-FUNCTION].minnumargs ) )
02560                                   goto NormZero;
02561                           }
02562 doflags:
02563                           if ( ( ( t[2] & DIRTYFLAG ) != 0 ) && ( functions[*t-FUNCTION].tabl == 0 ) ) {
02564                               t[2] &= ~DIRTYFLAG;
02565                               t[2] |= DIRTYSYMFLAG;
02566                           }
02567                           if ( functions[*t-FUNCTION].commute ) { pnco[nnco++] = t; }
02568                           else { pcom[ncom++] = t; }
02569                       }
02570                       else {
02571                           if ( ( ( t[2] & DIRTYFLAG ) != 0 ) && ( functions[*t-FUNCTION-WILDOFFSET].tabl
      == 0 ) ) {
02572                               t[2] &= ~DIRTYFLAG;
02573                               t[2] |= DIRTYSYMFLAG;
02574                           }
02575                           if ( functions[*t-FUNCTION-WILDOFFSET].commute ) {
02576                               pnco[nnco++] = t;
02577                           }
02578                           else { pcom[ncom++] = t; }
02579                       }
02580                   }
02581
02582                   /* Now hunt for contractible indices */
02583
02584                   if ( ( *t < (FUNCTION + WILDOFFSET)
02585                    && functions[*t-FUNCTION].spec >= TENSORFUNCTION ) || (
02586                    *t >= (FUNCTION + WILDOFFSET)
02587                    && functions[*t-FUNCTION-WILDOFFSET].spec >= TENSORFUNCTION ) ) {
02588                       if ( *t >= GAMMA && *t <= GAMMASEVEN ) t++;
02589                       t += FUNHEAD;
02590                       while ( t < r ) {
02591                           if ( *t == AM.vectorzero ) goto NormZero;
02592                           if ( *t >= AM.OffsetIndex && ( *t >= AM.DumInd
02593                           || ( *t < AM.WilInd && indices[*t-AM.OffsetIndex].dimension ) ) ) {
02594                               pcon[ncon++] = t;
02595                           }
02596                           else if ( *t == FUNNYWILD ) { t++; }
02597                           t++;
02598                       }
02599                   }
02600                   else {
02601                       t += FUNHEAD;
02602                       while ( t < r ) {
02603                           if ( *t > 0 ) {
02604 /*
02605                               Here we should worry about a recursion
02606                               A problem is the possibility of a construct
02607                               like  f(mu+nu)
02608 */
02609                               t += *t;
02610                           }
02611                           else if ( *t <= -FUNCTION ) t++;
02612                           else if ( *t == -INDEX ) {
02613                               if ( t[1] >= AM.OffsetIndex &&
02614                                   ( t[1] >= AM.DumInd || ( t[1] < AM.WilInd
02615                                   && indices[t[1]-AM.OffsetIndex].dimension ) ) )
02616                                   pcon[ncon++] = t+1;
02617                               t += 2;
02618                           }
02619                           else if ( *t == -SYMBOL ) {
02620                               if ( t[1] >= MAXPOWER && t[1] < 2*MAXPOWER ) {
02621                                   *t = -SNUMBER;
02622                                   t[1] -= MAXPOWER;
02623                               }
02624                               else if ( t[1] < -MAXPOWER && t[1] > -2*MAXPOWER  ) {
02625                                   *t = -SNUMBER;
02626                                   t[1] += MAXPOWER;
02627                               }
02628                               else t += 2;
02629                           }
02630                           else t += 2;
02631                       }
02632                   }
02633                   break;
02634           }
02635           t = r;
02636 TryAgain:;
02637       } while ( t < m );
02638       if ( ANsc ) {
02639           AN.cTerm = ANsc;
02640           r = t = ANsr; m = ANsm;
```

```
02641         ANsc = ANsm = ANsr = 0;
02642         goto conscan;
02643     }
02644 /*
02645     #] First scan :
02646     #[ Easy denominators :
02647
02648     Easy denominators are denominators that can be replaced by
02649     negative powers of individual subterms. This may add to all
02650     our sublists.
02651
02652 */
02653     if ( nden ) {
02654         for ( k = 0, i = 0; i < nden; i++ ) {
02655             t = pden[i];
02656             if ( ( t[2] & DIRTYFLAG ) == 0 ) continue;
02657             r = t + t[1]; m = t + FUNHEAD;
02658             if ( m >= r ) {
02659                 for ( j = i+1; j < nden; j++ ) pden[j-1] = pden[j];
02660                 nden--;
02661                 for ( j = 0; j < nnco; j++ ) if ( pnco[j] == t ) break;
02662                 for ( j++; j < nnco; j++ ) pnco[j-1] = pnco[j];
02663                 nnco--;
02664                 i--;
02665             }
02666             else {
02667                 NEXTARG(m);
02668                 if ( m >= r ) continue;
02669 /*
02670                 We have more than one argument. Split the function.
02671 */
02672                 if ( k == 0 ) {
02673                     k = 1; to = termout; from = term;
02674                 }
02675                 while ( from < t ) *to++ = *from++;
02676                 m = t + FUNHEAD;
02677                 while ( m < r ) {
02678                     stop = to;
02679                     *to++ = DENOMINATOR;
02680                     for ( j = 1; j < FUNHEAD; j++ ) *to++ = 0;
02681                     if ( *m < -FUNCTION ) *to++ = *m++;
02682                     else if ( *m < 0 ) { *to++ = *m++; *to++ = *m++; }
02683                     else {
02684                         j = *m; while ( --j >= 0 ) *to++ = *m++;
02685                     }
02686                     stop[1] = WORDDIF(to,stop);
02687                 }
02688                 from = r;
02689                 if ( i == nden - 1 ) {
02690                     stop = term + *term;
02691                     while ( from < stop ) *to++ = *from++;
02692                     i = *termout = WORDDIF(to,termout);
02693                     to = term; from = termout;
02694                     while ( --i >= 0 ) *to++ = *from++;
02695                     goto Restart;
02696                 }
02697             }
02698         }
02699         for ( i = 0; i < nden; i++ ) {
02700             t = pden[i];
02701             if ( ( t[2] & DIRTYFLAG ) == 0 ) continue;
02702             t[2] = 0;
02703             if ( t[FUNHEAD] == -SYMBOL ) {
02704                 WORD change;
02705                 t += FUNHEAD+1;
02706                 change = ExtraSymbol(*t,-1,nsym,ppsym,&ncoef);
02707                 nsym += change;
02708                 ppsym += change * 2;
02709                 goto DropDen;
02710             }
02711             else if ( t[FUNHEAD] == -SNUMBER ) {
02712                 t += FUNHEAD+1;
02713                 if ( *t == 0 ) goto NormInf;
02714                 if ( *t < 0 ) { *AT.WorkPointer = -*t; j = -1; }
02715                 else { *AT.WorkPointer = *t; j = 1; }
02716                 ncoef = REDLENG(ncoef);
02717                 if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)AT.WorkPointer,j) )
02718                     goto FromNorm;
02719                 ncoef = INCLENG(ncoef);
02720                 goto DropDen;
02721             }
02722             else if ( t[FUNHEAD] == ARGHEAD ) goto NormInf;
02723             else if ( t[FUNHEAD] > 0 && t[FUNHEAD+ARGHEAD] ==
02724             t[FUNHEAD]-ARGHEAD ) {
02725                 /* Only one term */
02726                 r = t + t[1] - 1;
02727                 t += FUNHEAD + ARGHEAD + 1;
```

```
02728                 j = *r;
02729                 m = r - ABS(*r) + 1;
02730                 if ( j != 3 || ( ( *m != 1 ) || ( m[1] != 1 ) ) ) {
02731                     ncoef = REDLENG(ncoef);
02732                     if ( DivRat(BHEAD (UWORD *)n_coef,ncoef,(UWORD *)m,REDLENG(j),(UWORD
      *)n_coef,&ncoef) ) goto FromNorm;
02733                     ncoef = INCLENG(ncoef);
02734                     j = ABS(j) - 3;
02735                     t[-FUNHEAD-ARGHEAD] -= j;
02736                     t[-ARGHEAD-1] -= j;
02737                     t[-1] -= j;
02738                     m[0] = m[1] = 1;
02739                     m[2] = 3;
02740                 }
02741                 while ( t < m ) {
02742                     r = t + t[1];
02743                     if ( *t == SYMBOL || *t == DOTPRODUCT ) {
02744                         k = t[1];
02745                         pden[i][1] -= k;
02746                         pden[i][FUNHEAD] -= k;
02747                         pden[i][FUNHEAD+ARGHEAD] -= k;
02748                         m -= k;
02749                         stop = m + 3;
02750                         tt = to = t;
02751                         from = r;
02752                         if ( *t == SYMBOL ) {
02753                             t += 2;
02754                             while ( t < r ) {
02755                                 WORD change;
02756                                 change = ExtraSymbol(*t,-t[1],nsym,ppsym,&ncoef);
02757                                 nsym += change;
02758                                 ppsym += change * 2;
02759                                 t += 2;
02760                             }
02761                         }
02762                         else {
02763                             t += 2;
02764                             while ( t < r ) {
02765                                 *ppdot++ = *t++;
02766                                 *ppdot++ = *t++;
02767                                 *ppdot++ = -*t++;
02768                                 ndot++;
02769                             }
02770                         }
02771                         while ( to < stop ) *to++ = *from++;
02772                         r = tt;
02773                     }
02774 #ifdef WITHFLOAT
02775                     else if ( *t == FLOATFUN && TestFloat(t) ) {
02776                         k = t[1];
02777                         pden[i][1] -= k;
02778                         pden[i][FUNHEAD] -= k;
02779                         pden[i][FUNHEAD+ARGHEAD] -= k;
02780                         UnpackFloat(aux5,t);
02781                         if ( withfloat == 0 ) {
02782                             mpf_ui_div(aux4,1,aux5);
02783                             withfloat = 2;
02784                         }
02785                         else {
02786                             if ( withfloat == 1 ) UnpackFloat(aux4,firstfloat);
02787                             mpf_div(aux4,aux4,aux5);
02788                             withfloat++;
02789                         }
02790                         tt = to = t;
02791                         while ( r < m ) *to++ = *r++;
02792                         *to++ = 1; *to++ = 1; *to++ = 3;
02793                         if ( ncoef < 0 ) t[-1] = -t[-1];
02794                         m -= k;
02795                         stop = m + 3;
02796                         r = tt;
02797                     }
02798 #endif
02799                     t = r;
02800                 }
02801                 if ( pden[i][1] == 4+FUNHEAD+ARGHEAD ) {
02802 DropDen:
02803                     for ( j = 0; j < nnco; j++ ) {
02804                         if ( pden[i] == pnco[j] ) {
02805                             --nnco;
02806                             while ( j < nnco ) {
02807                                 pnco[j] = pnco[j+1];
02808                                 j++;
02809                             }
02810                             break;
02811                         }
02812                     }
02813                     pden[i--] = pden[--nden];
```

```
02814                 }
02815             }
02816         }
02817     }
02818 /*
02819     #] Easy denominators :
02820     #[ Index Contractions :
02821 */
02822     if ( ndel ) {
02823         t = pdel;
02824         for ( i = 0; i < ndel; i += 2 ) {
02825             if ( t[0] == t[1] ) {
02826                 if ( t[0] == EMPTYINDEX ) {}
02827                 else if ( *t < AM.OffsetIndex ) {
02828                     k = AC.FixIndices[*t];
02829                     if ( k < 0 ) { j = -1; k = -k; }
02830                     else if ( k > 0 ) j = 1;
02831                     else goto NormZero;
02832                     goto WithFix;
02833                 }
02834                 else if ( *t >= AM.DumInd ) {
02835                     k = AC.lDefDim;
02836                     if ( k ) goto docontract;
02837                 }
02838                 else if ( *t >= AM.WilInd ) {
02839                     k = indices[*t-AM.OffsetIndex-WILDOFFSET].dimension;
02840                     if ( k ) goto docontract;
02841                 }
02842                 else if ( ( k = indices[*t-AM.OffsetIndex].dimension ) != 0 ) {
02843 docontract:
02844                     if ( k > 0 ) {
02845                         j = 1;
02846 WithFix:                shortnum = k;
02847                         ncoef = REDLENG(ncoef);
02848                         if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)(&shortnum),j) )
02849                             goto FromNorm;
02850                         ncoef = INCLENG(ncoef);
02851                     }
02852                     else {
02853                         WORD change;
02854                         change = ExtraSymbol((WORD)(-k),(WORD)1,nsym,ppsym,&ncoef);
02855                         nsym += change;
02856                         ppsym += change * 2;
02857                     }
02858                     t[1] = pdel[ndel-1];
02859                     t[0] = pdel[ndel-2];
02860 HaveCon:
02861                     ndel -= 2;
02862                     i -= 2;
02863                 }
02864             }
02865             else {
02866                 if ( *t < AM.OffsetIndex && t[1] < AM.OffsetIndex ) goto NormZero;
02867                 j = *t - AM.OffsetIndex;
02868                 if ( j >= 0 && ( ( *t >= AM.DumInd && AC.lDefDim )
02869                 || ( *t < AM.WilInd && indices[j].dimension ) ) ) {
02870                     for ( j = i + 2, m = pdel+j; j < ndel; j += 2, m += 2 ) {
02871                         if ( *t == *m ) {
02872                             *t = m[1];
02873                             *m++ = pdel[ndel-2];
02874                             *m = pdel[ndel-1];
02875                             goto HaveCon;
02876                         }
02877                         else if ( *t == m[1] ) {
02878                             *t = *m;
02879                             *m++ = pdel[ndel-2];
02880                             *m   = pdel[ndel-1];
02881                             goto HaveCon;
02882                         }
02883                     }
02884                 }
02885                 j = t[1]-AM.OffsetIndex;
02886                 if ( j >= 0 && ( ( t[1] >= AM.DumInd && AC.lDefDim )
02887                 || ( t[1] < AM.WilInd && indices[j].dimension ) ) ) {
02888                     for ( j = i + 2, m = pdel+j; j < ndel; j += 2, m += 2 ) {
02889                         if ( t[1] == *m ) {
02890                             t[1] = m[1];
02891                             *m++ = pdel[ndel-2];
02892                             *m   = pdel[ndel-1];
02893                             goto HaveCon;
02894                         }
02895                         else if ( t[1] == m[1] ) {
02896                             t[1] = *m;
02897                             *m++ = pdel[ndel-2];
02898                             *m   = pdel[ndel-1];
02899                             goto HaveCon;
02900                         }
```

```
02901                     }
02902                 }
02903                 t += 2;
02904             }
02905         }
02906         if ( ndel > 0 ) {
02907             if ( nvec ) {
02908                 t = pdel;
02909                 for ( i = 0; i < ndel; i++ ) {
02910                     if ( *t >= AM.OffsetIndex && ( ( *t >= AM.DumInd && AC.lDefDim ) ||
02911                     ( *t < AM.WilInd && indices[*t-AM.OffsetIndex].dimension ) ) ) {
02912                         r = pvec + 1;
02913                         for ( j = 1; j < nvec; j += 2 ) {
02914                             if ( *r == *t ) {
02915                                 if ( i & 1 ) {
02916                                     *r = t[-1];
02917                                     *t-- = pdel[--ndel];
02918                                     i -= 2;
02919                                 }
02920                                 else {
02921                                     *r = t[1];
02922                                     t[1] = pdel[--ndel];
02923                                     i--;
02924                                 }
02925                                 *t-- = pdel[--ndel];
02926                                 break;
02927                             }
02928                             r += 2;
02929                         }
02930                     }
02931                     t++;
02932                 }
02933             }
02934             if ( ndel > 0 && ncon ) {
02935                 t = pdel;
02936                 for ( i = 0; i < ndel; i++ ) {
02937                     if ( *t >= AM.OffsetIndex && ( ( *t >= AM.DumInd && AC.lDefDim ) ||
02938                     ( *t < AM.WilInd && indices[*t-AM.OffsetIndex].dimension ) ) ) {
02939                         for ( j = 0; j < ncon; j++ ) {
02940                             if ( *pcon[j] == *t ) {
02941                                 if ( i & 1 ) {
02942                                     *pcon[j] = t[-1];
02943                                     *t-- = pdel[--ndel];
02944                                     i -= 2;
02945                                 }
02946                                 else {
02947                                     *pcon[j] = t[1];
02948                                     t[1] = pdel[--ndel];
02949                                     i--;
02950                                 }
02951                                 *t-- = pdel[--ndel];
02952                                 didcontr++;
02953                                 r = pcon[j];
02954                                 for ( j = 0; j < nnco; j++ ) {
02955                                     m = pnco[j];
02956                                     if ( r > m && r < m+m[1] ) {
02957                                         m[2] |= DIRTYSYMFLAG;
02958                                         break;
02959                                     }
02960                                 }
02961                                 for ( j = 0; j < ncom; j++ ) {
02962                                     m = pcom[j];
02963                                     if ( r > m && r < m+m[1] ) {
02964                                         m[2] |= DIRTYSYMFLAG;
02965                                         break;
02966                                     }
02967                                 }
02968                                 for ( j = 0; j < neps; j++ ) {
02969                                     m = peps[j];
02970                                     if ( r > m && r < m+m[1] ) {
02971                                         m[2] |= DIRTYSYMFLAG;
02972                                         break;
02973                                     }
02974                                 }
02975                                 break;
02976                             }
02977                         }
02978                     }
02979                     t++;
02980                 }
02981             }
02982         }
02983     }
02984     if ( nvec ) {
02985         t = pvec + 1;
02986         for ( i = 3; i < nvec; i += 2 ) {
02987             k = *t - AM.OffsetIndex;
```

```
02988             if ( k >= 0 && ( ( *t > AM.DumInd && AC.lDefDim )
02989             || ( *t < AM.WilInd && indices[k].dimension ) ) ) {
02990                 r = t + 2;
02991                 for ( j = i; j < nvec; j += 2 ) {
02992                     if ( *r == *t ) {   /* Another dotproduct */
02993                         *ppdot++ = t[-1];
02994                         *ppdot++ = r[-1];
02995                         *ppdot++ = 1;
02996                         ndot++;
02997                         *r-- = pvec[--nvec];
02998                         *r   = pvec[--nvec];
02999                         *t-- = pvec[--nvec];
03000                         *t-- = pvec[--nvec];
03001                         i -= 2;
03002                         break;
03003                     }
03004                     r += 2;
03005                 }
03006             }
03007             t += 2;
03008         }
03009         if ( nvec > 0 && ncon ) {
03010             t = pvec + 1;
03011             for ( i = 1; i < nvec; i += 2 ) {
03012                 k = *t - AM.OffsetIndex;
03013                 if ( k >= 0 && ( ( *t >= AM.DumInd && AC.lDefDim )
03014                 || ( *t < AM.WilInd && indices[k].dimension ) ) ) {
03015                     for ( j = 0; j < ncon; j++ ) {
03016                         if ( *pcon[j] == *t ) {
03017                             *pcon[j] = t[-1];
03018                             *t-- = pvec[--nvec];
03019                             *t-- = pvec[--nvec];
03020                             r = pcon[j];
03021                             pcon[j] = pcon[--ncon];
03022                             i -= 2;
03023                             for ( j = 0; j < nnco; j++ ) {
03024                                 m = pnco[j];
03025                                 if ( r > m && r < m+m[1] ) {
03026                                     m[2] |= DIRTYSYMFLAG;
03027                                     break;
03028                                 }
03029                             }
03030                             for ( j = 0; j < ncom; j++ ) {
03031                                 m = pcom[j];
03032                                 if ( r > m && r < m+m[1] ) {
03033                                     m[2] |= DIRTYSYMFLAG;
03034                                     break;
03035                                 }
03036                             }
03037                             for ( j = 0; j < neps; j++ ) {
03038                                 m = peps[j];
03039                                 if ( r > m && r < m+m[1] ) {
03040                                     m[2] |= DIRTYSYMFLAG;
03041                                     break;
03042                                 }
03043                             }
03044                             break;
03045                         }
03046                     }
03047                 }
03048                 t += 2;
03049             }
03050         }
03051     }
03052 /*
03053     #] Index Contractions :
03054     #[ NonCommuting Functions :
03055 */
03056     m = fillsetexp;
03057     if ( nnco ) {
03058         for ( i = 0; i < nnco; i++ ) {
03059             t = pnco[i];
03060             if ( ( *t >= (FUNCTION+WILDOFFSET)
03061             && functions[*t-FUNCTION-WILDOFFSET].spec <= 0 )
03062             || ( *t >= FUNCTION && *t < (FUNCTION + WILDOFFSET)
03063             && functions[*t-FUNCTION].spec <= 0 ) ) {
03064                 DoRevert(t,m);
03065                 if ( didcontr ) {
03066                     r = t + FUNHEAD;
03067                     t += t[1];
03068                     while ( r < t ) {
03069                         if ( *r == -INDEX && r[1] >= 0 && r[1] < AM.OffsetIndex ) {
03070                             *r = -SNUMBER;
03071                             didcontr--;
03072                             pnco[i][2] |= DIRTYSYMFLAG;
03073                         }
03074                         NEXTARG(r)
```

```
03075                       }
03076                   }
03077               }
03078           }
03079
03080           /* First should come the code for function properties. */
03081
03082           /* First we test for symmetric properties and the DIRTYSYMFLAG */
03083
03084           for ( i = 0; i < nnco; i++ ) {
03085               t = pnco[i];
03086               if ( *t > 0 && ( t[2] & DIRTYSYMFLAG ) && *t != DOLLAREXPRESSION ) {
03087                   l = 0; /* to make the compiler happy */
03088                   if ( ( *t >= (FUNCTION+WILDOFFSET)
03089                   && ( l = functions[*t-FUNCTION-WILDOFFSET].symmetric ) > 0 )
03090                   || ( *t >= FUNCTION && *t < (FUNCTION + WILDOFFSET)
03091                   && ( l = functions[*t-FUNCTION].symmetric ) > 0 ) ) {
03092                       if ( *t >= (FUNCTION+WILDOFFSET) ) {
03093                           *t -= WILDOFFSET;
03094                           j = FullSymmetrize(BHEAD t,l);
03095                           *t += WILDOFFSET;
03096                       }
03097                       else j = FullSymmetrize(BHEAD t,l);
03098                       if ( (l & ~REVERSEORDER) == ANTISYMMETRIC ) {
03099                           if ( ( j & 2 ) != 0 ) goto NormZero;
03100                           if ( ( j & 1 ) != 0 ) ncoef = -ncoef;
03101                       }
03102                   }
03103                   else t[2] &= ~DIRTYSYMFLAG;
03104               }
03105           }
03106
03107           /* Non commuting functions are then tested for commutation
03108              rules. If needed their order is exchanged. */
03109
03110           k = nnco - 1;
03111           for ( i = 0; i < k; i++ ) {
03112               j = i;
03113               while ( Commute(pnco[j],pnco[j+1]) ) {
03114                   t = pnco[j]; pnco[j] = pnco[j+1]; pnco[j+1] = t;
03115                   l = j-1;
03116                   while ( l >= 0 && Commute(pnco[l],pnco[l+1]) ) {
03117                       t = pnco[l]; pnco[l] = pnco[l+1]; pnco[l+1] = t;
03118                       l--;
03119                   }
03120                   if ( ++j >= k ) break;
03121               }
03122           }
03123
03124           /* Finally they are written to output. gamma matrices
03125              are bundled if possible */
03126
03127           for ( i = 0; i < nnco; i++ ) {
03128               t = pnco[i];
03129               if ( *t == IDFUNCTION ) AN.idfunctionflag = 1;
03130               if ( *t >= GAMMA && *t <= GAMMASEVEN ) {
03131                   WORD gtype;
03132                   to = m;
03133                   *m++ = GAMMA;
03134                   m++;
03135                   FILLFUN(m)
03136                   *m++ = stype = t[FUNHEAD];      /* type of string */
03137                   j = 0;
03138                   nnum = 0;
03139                   do {
03140                       r = t + t[1];
03141                       if ( *t == GAMMAFIVE ) {
03142                           gtype = GAMMA5; t += FUNHEAD; goto onegammamatrix; }
03143                       else if ( *t == GAMMASIX ) {
03144                           gtype = GAMMA6; t += FUNHEAD; goto onegammamatrix; }
03145                       else if ( *t == GAMMASEVEN ) {
03146                           gtype = GAMMA7; t += FUNHEAD; goto onegammamatrix; }
03147                       t += FUNHEAD+1;
03148                       while ( t < r ) {
03149                           gtype = *t;
03150 onegammamatrix:
03151                           if ( gtype == GAMMA5 ) {
03152                               if ( j == GAMMA1 ) j = GAMMA5;
03153                               else if ( j == GAMMA5 ) j = GAMMA1;
03154                               else if ( j == GAMMA7 ) ncoef = -ncoef;
03155                               if ( nnum & 1 ) ncoef = -ncoef;
03156                           }
03157                           else if ( gtype == GAMMA6 || gtype == GAMMA7 ) {
03158                               if ( nnum & 1 ) {
03159                                   if ( gtype == GAMMA6 ) gtype = GAMMA7;
03160                                   else                   gtype = GAMMA6;
03161                               }
```

```
03162                               if ( j == GAMMA1 ) j = gtype;
03163                               else if ( j == GAMMA5 ) {
03164                                   j = gtype;
03165                                   if ( j == GAMMA7 ) ncoef = -ncoef;
03166                               }
03167                               else if ( j != gtype ) goto NormZero;
03168                               else {
03169                                   shortnum = 2;
03170                                   ncoef = REDLENG(ncoef);
03171                                   if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)(&shortnum),1) ) goto
      FromNorm;
03172                                   ncoef = INCLENG(ncoef);
03173                               }
03174                           }
03175                           else {
03176                               *m++ = gtype; nnum++;
03177                           }
03178                           t++;
03179                       }
03180
03181                   } while ( ( ++i < nnco ) && ( *(t = pnco[i]) >= GAMMA
03182                   && *t <= GAMMASEVEN ) && ( t[FUNHEAD] == stype ) );
03183                   i--;
03184                   if ( j ) {
03185                       k = WORDDIF(m,to) - FUNHEAD-1;
03186                       r = m;
03187                       from = m++;
03188                       while ( --k >= 0 ) *from-- = *--r;
03189                       *from = j;
03190                   }
03191                   to[1] = WORDDIF(m,to);
03192               }
03193               else if ( *t < 0 ) {
03194                   *m++ = -*t; *m++ = FUNHEAD; *m++ = 0;
03195                   FILLFUN3(m)
03196               }
03197               else {
03198                   if ( ( t[2] & DIRTYFLAG ) == DIRTYFLAG
03199                           && *t != REPLACEMENT && *t != DOLLAREXPRESSION
03200                           && TestFunFlag(BHEAD t) ) ReplaceVeto = 1;
03201                   k = t[1];
03202                   NCOPY(m,t,k);
03203               }
03204           }
03205
03206       }
03207   /*
03208       #] NonCommuting Functions :
03209       #[ Commuting Functions :
03210   */
03211       if ( ncom ) {
03212           for ( i = 0; i < ncom; i++ ) {
03213               t = pcom[i];
03214               if ( ( *t >= (FUNCTION+WILDOFFSET)
03215               && functions[*t-FUNCTION-WILDOFFSET].spec <= 0 )
03216               || ( *t >= FUNCTION && *t < (FUNCTION + WILDOFFSET)
03217               && functions[*t-FUNCTION].spec <= 0 ) ) {
03218                   DoRevert(t,m);
03219                   if ( didcontr ) {
03220                       r = t + FUNHEAD;
03221                       t += t[1];
03222                       while ( r < t ) {
03223                           if ( *r == -INDEX && r[1] >= 0 && r[1] < AM.OffsetIndex ) {
03224                               *r = -SNUMBER;
03225                               didcontr--;
03226                               pcom[i][2] |= DIRTYSYMFLAG;
03227                           }
03228                           NEXTARG(r)
03229                       }
03230                   }
03231               }
03232           }
03233
03234           /* Now we test for symmetric properties and the DIRTYSYMFLAG */
03235
03236           for ( i = 0; i < ncom; i++ ) {
03237               t = pcom[i];
03238               if ( *t > 0 && ( t[2] & DIRTYSYMFLAG ) ) {
03239                   l = 0; /* to make the compiler happy */
03240                   if ( ( *t >= (FUNCTION+WILDOFFSET)
03241                   && ( l = functions[*t-FUNCTION-WILDOFFSET].symmetric ) > 0 )
03242                   || ( *t >= FUNCTION && *t < (FUNCTION + WILDOFFSET)
03243                   && ( l = functions[*t-FUNCTION].symmetric ) > 0 ) ) {
03244                       if ( *t >= (FUNCTION+WILDOFFSET) ) {
03245                           *t -= WILDOFFSET;
03246                           j = FullSymmetrize(BHEAD t,l);
03247                           *t += WILDOFFSET;
```

```
03248                   }
03249                   else j = FullSymmetrize(BHEAD t,l);
03250                   if ( (l & ~REVERSEORDER) == ANTISYMMETRIC ) {
03251                       if ( ( j & 2 ) != 0 ) goto NormZero;
03252                       if ( ( j & 1 ) != 0 ) ncoef = -ncoef;
03253                   }
03254               }
03255               else t[2] &= ~DIRTYSYMFLAG;
03256           }
03257       }
03258 /*
03259       Sort the functions
03260       From a purists point of view this can be improved.
03261       There are slow and fast arguments and no conversions are
03262       taken into account here.
03263 */
03264       for ( i = 1; i < ncom; i++ ) {
03265           for ( j = i; j > 0; j-- ) {
03266               WORD jj,kk;
03267               jj = j-1;
03268               t = pcom[jj];
03269               r = pcom[j];
03270               if ( *t < 0 ) {
03271                   if ( *r < 0 ) { if ( *t >= *r ) goto NextI; }
03272                   else { if ( -*t <= *r ) goto NextI; }
03273                   goto jexch;
03274               }
03275               else if ( *r < 0 ) {
03276                   if ( *t < -*r ) goto NextI;
03277                   goto jexch;
03278               }
03279               else if ( *t != *r ) {
03280                   if ( *t < *r ) goto NextI;
03281 jexch:            t = pcom[j]; pcom[j] = pcom[jj]; pcom[jj] = t;
03282                   continue;
03283               }
03284               if ( AC.properorderflag ) {
03285                   if ( ( *t >= (FUNCTION+WILDOFFSET)
03286                   && functions[*t-FUNCTION-WILDOFFSET].spec >= TENSORFUNCTION )
03287                   || ( *t >= FUNCTION && *t < (FUNCTION + WILDOFFSET)
03288                   && functions[*t-FUNCTION].spec >= TENSORFUNCTION ) ) {}
03289                   else {
03290                       WORD *s1, *s2, *ss1, *ss2;
03291                       s1 = t+FUNHEAD; s2 = r+FUNHEAD;
03292                       ss1 = t + t[1]; ss2 = r + r[1];
03293                       while ( s1 < ss1 && s2 < ss2 ) {
03294                           k = CompArg(s1,s2);
03295                           if ( k > 0 ) goto jexch;
03296                           if ( k < 0 ) goto NextI;
03297                           NEXTARG(s1)
03298                           NEXTARG(s2)
03299                       }
03300                       if ( s1 < ss1 ) goto jexch;
03301                       goto NextI;
03302                   }
03303                   k = t[1] - FUNHEAD;
03304                   kk = r[1] - FUNHEAD;
03305                   t += FUNHEAD;
03306                   r += FUNHEAD;
03307                   while ( k > 0 && kk > 0 ) {
03308                       if ( *t < *r ) goto NextI;
03309                       else if ( *t++ > *r++ ) goto jexch;
03310                       k--; kk--;
03311                   }
03312                   if ( k > 0 ) goto jexch;
03313                   goto NextI;
03314               }
03315               else
03316               {
03317                   k = t[1] - FUNHEAD;
03318                   kk = r[1] - FUNHEAD;
03319                   t += FUNHEAD;
03320                   r += FUNHEAD;
03321                   while ( k > 0 && kk > 0 ) {
03322                       if ( *t < *r ) goto NextI;
03323                       else if ( *t++ > *r++ ) goto jexch;
03324                       k--; kk--;
03325                   }
03326                   if ( k > 0 ) goto jexch;
03327                   goto NextI;
03328               }
03329           }
03330 NextI:;
03331       }
03332       for ( i = 0; i < ncom; i++ ) {
03333           t = pcom[i];
03334           if ( *t == THETA || *t == THETA2 ) {
```

```
03335                 if ( ( k = DoTheta(BHEAD t) ) == 0 ) goto NormZero;
03336                 else if ( k < 0 ) {
03337                     k = t[1];
03338                     NCOPY(m,t,k);
03339                 }
03340             }
03341             else if ( *t == DELTA2 || *t == DELTAP ) {
03342                 if ( ( k = DoDelta(t) ) == 0 ) goto NormZero;
03343                 else if ( k < 0 ) {
03344                     k = t[1];
03345                     NCOPY(m,t,k);
03346                 }
03347             }
03348             else if ( *t == AR.PolyFunInv && AR.PolyFunType == 2 ) {
03349 /*
03350                 If there are two arguments, exchange them, change the
03351                 name of the function and go to dealing with PolyRatFun.
03352 */
03353                 WORD *mm, *tt = t, numt = 0;
03354                 tt += FUNHEAD;
03355                 while ( tt < t+t[1] ) { numt++; NEXTARG(tt) }
03356                 if ( numt == 2 ) {
03357                     tt = t; mm = m; k = t[1];
03358                     NCOPY(mm,tt,k)
03359                     mm = m+FUNHEAD;
03360                     NEXTARG(mm);
03361                     tt = t+FUNHEAD;
03362                     if ( *mm < 0 ) {
03363                         if ( *mm <= -FUNCTION ) { *tt++ = *mm++; }
03364                         else { *tt++ = *mm++;  *tt++ = *mm++; }
03365                     }
03366                     else {
03367                         k = *mm; NCOPY(tt,mm,k)
03368                     }
03369                     mm = m+FUNHEAD;
03370                     if ( *mm < 0 ) {
03371                         if ( *mm <= -FUNCTION ) { *tt++ = *mm++; }
03372                         else { *tt++ = *mm++;  *tt++ = *mm++; }
03373                     }
03374                     else {
03375                         k = *mm; NCOPY(tt,mm,k)
03376                     }
03377                     *t = AR.PolyFun;
03378                     t[2] |= MUSTCLEANPRF;
03379                     goto regularratfun;
03380                 }
03381             }
03382             else if ( *t == AR.PolyFun ) {
03383               if ( AR.PolyFunType == 1 ) { /* Regular PolyFun with one argument */
03384                 if ( t[FUNHEAD+1] == 0 && AR.Eside != LHSIDE &&
03385                 t[1] == FUNHEAD + 2 && t[FUNHEAD] == -SNUMBER ) goto NormZero;
03386                 if ( i > 0 && pcom[i-1][0] == AR.PolyFun ) {
03387                     if ( AN.PolyNormFlag == 0 ) {
03388                         AN.PolyNormFlag = 1;
03389                         AN.PolyFunTodo = 0;
03390                     }
03391                 }
03392                 k = t[1];
03393                 NCOPY(m,t,k);
03394               }
03395               else if ( AR.PolyFunType == 2 ) {
03396 /*
03397                 PolyRatFun.
03398                     Regular type: Two arguments
03399                     Power expanded: One argument. Here to be treated as
03400                         AR.PolyFunType == 1, but with power cutoff.
03401 */
03402 regularratfun:;
03403 /*
03404                 First check for zeroes.
03405 */
03406                 if ( t[FUNHEAD+1] == 0 && AR.Eside != LHSIDE &&
03407                 t[1] > FUNHEAD + 2 && t[FUNHEAD] == -SNUMBER ) {
03408                     u = t + FUNHEAD + 2;
03409                     if ( *u < 0 ) {
03410                         if ( *u <= -FUNCTION ) {}
03411                         else if ( t[1] == FUNHEAD+4 && t[FUNHEAD+2] == -SNUMBER
03412                             && t[FUNHEAD+3] == 0 ) goto NormPRF;
03413                         else if ( t[1] == FUNHEAD+4 ) goto NormZero;
03414                     }
03415                     else if ( t[1] == *u+FUNHEAD+2 ) goto NormZero;
03416                 }
03417                 else {
03418                     u = t+FUNHEAD; NEXTARG(u);
03419                     if ( *u == -SNUMBER && u[1] == 0 ) goto NormInf;
03420                 }
03421                 if ( i > 0 && pcom[i-1][0] == AR.PolyFun ) AN.PolyNormFlag = 1;
```

```
03422                 else if ( i < ncom-1 && pcom[i+1][0] == AR.PolyFun ) AN.PolyNormFlag = 1;
03423                 k = t[1];
03424                 if ( AN.PolyNormFlag ) {
03425                     if ( AR.PolyFunExp == 0 ) {
03426                         AN.PolyFunTodo = 0;
03427                         NCOPY(m,t,k);
03428                     }
03429                     else if ( AR.PolyFunExp == 1 ) { /* get highest divergence */
03430                         if ( PolyFunMode == 0 ) {
03431                             NCOPY(m,t,k);
03432                             AN.PolyFunTodo = 1;
03433                         }
03434                         else {
03435                             WORD *mmm = m;
03436                             NCOPY(m,t,k);
03437                             if ( TreatPolyRatFun(BHEAD mmm) != 0 )
03438                                     goto FromNorm;
03439                             m = mmm+mmm[1];
03440                         }
03441                     }
03442                     else {
03443                         if ( PolyFunMode == 0 ) {
03444                             NCOPY(m,t,k);
03445                                 AN.PolyFunTodo = 1;
03446                         }
03447                         else {
03448                             WORD *mmm = m;
03449                             NCOPY(m,t,k);
03450                             if ( ExpandRat(BHEAD mmm) != 0 )
03451                                     goto FromNorm;
03452                             m = mmm+mmm[1];
03453                         }
03454                     }
03455                 }
03456                 else {
03457                     if ( AR.PolyFunExp == 0 ) {
03458                         AN.PolyFunTodo = 0;
03459                         NCOPY(m,t,k);
03460                     }
03461                     else if ( AR.PolyFunExp == 1 ) { /* get highest divergence */
03462                         WORD *mmm = m;
03463                         NCOPY(m,t,k);
03464                         if ( TreatPolyRatFun(BHEAD mmm) != 0 )
03465                                 goto FromNorm;
03466                         m = mmm+mmm[1];
03467                     }
03468                     else {
03469                         WORD *mmm = m;
03470                         NCOPY(m,t,k);
03471                         if ( ExpandRat(BHEAD mmm) != 0 )
03472                                 goto FromNorm;
03473                         m = mmm+mmm[1];
03474                     }
03475                 }
03476               }
03477             }
03478             else if ( *t > 0 ) {
03479                 if ( ( t[2] & DIRTYFLAG ) == DIRTYFLAG
03480                     && *t != REPLACEMENT && TestFunFlag(BHEAD t) ) ReplaceVeto = 1;
03481                 k = t[1];
03482                 NCOPY(m,t,k);
03483             }
03484             else {
03485                 *m++ = -*t; *m++ = FUNHEAD; *m++ = 0;
03486                 FILLFUN3(m)
03487             }
03488         }
03489     }
03490 /*
03491     #] Commuting Functions :
03492     #[ Track Replace_ :
03493 */
03494     if ( ReplaceVeto < 0 ) {
03495 /*
03496         We found one (or more) replace_ functions and all other
03497         functions are 'clean' (no dirty flag).
03498         Now we check whether one of these functions can be used.
03499         Thus far the functions go from fillsetexp to m.
03500         Somewhere in there there are -ReplaceVeto occurrences of REPLACEMENT.
03501         Hunt for the first one that fits the bill.
03502         Note that replace_ is a commuting function.
03503 */
03504         WORD *ma = fillsetexp, *mb, *mc;
03505         while ( ma < m ) {
03506             mb = ma + ma[1];
03507             if ( *ma != REPLACEMENT ) {
03508                 ma = mb;
```

```
03509                 continue;
03510             }
03511             if ( *ma == REPLACEMENT && ReplaceType == -1 ) {
03512                 mc = ma;
03513                 ReplaceType = 0;
03514                 if ( AN.RSsize < 2*ma[1]+SUBEXPSIZE ) {
03515                     if ( AN.ReplaceScrat ) M_free(AN.ReplaceScrat,"AN.ReplaceScrat");
03516                     AN.RSsize = 2*ma[1]+SUBEXPSIZE+40;
03517                     AN.ReplaceScrat = (WORD *)Malloc1((AN.RSsize+1)*sizeof(WORD),"AN.ReplaceScrat");
03518                 }
03519                 ma += FUNHEAD;
03520                 ReplaceSub = AN.ReplaceScrat;
03521                 ReplaceSub += SUBEXPSIZE;
03522                 while ( ma < mb ) {
03523                     if ( *ma > 0 ) goto NoRep;
03524                     if ( *ma <= -FUNCTION ) {
03525                         *ReplaceSub++ = FUNTOFUN;
03526                         *ReplaceSub++ = 4;
03527                         *ReplaceSub++ = -*ma++;
03528                         if ( *ma > -FUNCTION ) goto NoRep;
03529                         *ReplaceSub++ = -*ma++;
03530                     }
03531                     else if ( ma+4 > mb ) goto NoRep;
03532                     else {
03533                         if ( *ma == -SYMBOL ) {
03534                             if ( ma[2] == -SYMBOL && ma+4 <= mb )
03535                                 *ReplaceSub++ = SYMTOSYM;
03536                             else if ( ma[2] == -SNUMBER && ma+4 <= mb ) {
03537                                 *ReplaceSub++ = SYMTONUM;
03538                                 if ( ReplaceType == 0 ) {
03539                                     oldtoprhs = C->numrhs;
03540                                     oldcpointer = C->Pointer - C->Buffer;
03541                                 }
03542                                 ReplaceType = 1;
03543                             }
03544                             else if ( ma[2] == ARGHEAD && ma+2+ARGHEAD <= mb ) {
03545                                 *ReplaceSub++ = SYMTONUM;
03546                                 *ReplaceSub++ = 4;
03547                                 *ReplaceSub++ = ma[1];
03548                                 *ReplaceSub++ = 0;
03549                                 ma += 2+ARGHEAD;
03550                                 continue;
03551                             }
03552 /*
03553                             Next is the subexpression. We have to test that
03554                             it isn't vector-like or index-like
03555 */
03556                             else if ( ma[2] > 0 ) {
03557                                 WORD *sstop, *ttstop, n;
03558                                 ss = ma+2;
03559                                 sstop = ss + *ss;
03560                                 ss += ARGHEAD;
03561                                 while ( ss < sstop ) {
03562                                     tt = ss + *ss;
03563                                     ttstop = tt - ABS(tt[-1]);
03564                                     ss++;
03565                                     while ( ss < ttstop ) {
03566                                         if ( *ss == INDEX ) goto NoRep;
03567                                         ss += ss[1];
03568                                     }
03569                                     ss = tt;
03570                                 }
03571                                 subtype = SYMTOSUB;
03572                                 if ( ReplaceType == 0 ) {
03573                                     oldtoprhs = C->numrhs;
03574                                     oldcpointer = C->Pointer - C->Buffer;
03575                                 }
03576                                 ReplaceType = 1;
03577                                 ss = AddRHS(AT.ebufnum,1);
03578                                 tt = ma+2;
03579                                 n = *tt - ARGHEAD;
03580                                 tt += ARGHEAD;
03581                                 while ( (ss + n + 10) > C->Top ) ss = DoubleCbuffer(AT.ebufnum,ss,14);
03582                                 while ( --n >= 0 ) *ss++ = *tt++;
03583                                 *ss++ = 0;
03584                                 C->rhs[C->numrhs+1] = ss;
03585                                 C->Pointer = ss;
03586                                 *ReplaceSub++ = subtype;
03587                                 *ReplaceSub++ = 4;
03588                                 *ReplaceSub++ = ma[1];
03589                                 *ReplaceSub++ = C->numrhs;
03590                                 ma += 2 + ma[2];
03591                                 continue;
03592                             }
03593                             else goto NoRep;
03594                         }
03595                         else if ( ( *ma == -VECTOR || *ma == -MINVECTOR ) && ma+4 <= mb ) {
```

```
03596                           if ( ma[2] == -VECTOR ) {
03597                               if ( *ma == -VECTOR ) *ReplaceSub++ = VECTOVEC;
03598                               else *ReplaceSub++ = VECTOMIN;
03599                           }
03600                           else if ( ma[2] == -MINVECTOR ) {
03601                               if ( *ma == -VECTOR ) *ReplaceSub++ = VECTOMIN;
03602                               else *ReplaceSub++ = VECTOVEC;
03603                           }
03604 /*
03605                           Next is a vector-like subexpression
03606                           Search for vector nature first
03607 */
03608                           else if ( ma[2] > 0 ) {
03609                               WORD *sstop, *ttstop, *w, *mm, n, count;
03610                               WORD *v1, *v2 = 0;
03611                               if ( *ma == -MINVECTOR ) {
03612                                   ss = ma+2;
03613                                   sstop = ss + *ss;
03614                                   ss += ARGHEAD;
03615                                   while ( ss < sstop ) {
03616                                       ss += *ss;
03617                                       ss[-1] = -ss[-1];
03618                                   }
03619                                   *ma = -VECTOR;
03620                               }
03621                               ss = ma+2;
03622                               sstop = ss + *ss;
03623                               ss += ARGHEAD;
03624                               while ( ss < sstop ) {
03625                                   tt = ss + *ss;
03626                                   ttstop = tt - ABS(tt[-1]);
03627                                   ss++;
03628                                   count = 0;
03629                                   while ( ss < ttstop ) {
03630                                       if ( *ss == INDEX ) {
03631                                           n = ss[1] - 2; ss += 2;
03632                                           while ( --n >= 0 ) {
03633                                               if ( *ss < MINSPEC ) count++;
03634                                               ss++;
03635                                           }
03636                                       }
03637                                       else ss += ss[1];
03638                                   }
03639                                   if ( count != 1 ) goto NoRep;
03640                                   ss = tt;
03641                               }
03642                               subtype = VECTOSUB;
03643                               if ( ReplaceType == 0 ) {
03644                                   oldtoprhs = C->numrhs;
03645                                   oldcpointer = C->Pointer - C->Buffer;
03646                               }
03647                               ReplaceType = 1;
03648                               mm = AddRHS(AT.ebufnum,1);
03649                               *ReplaceSub++ = subtype;
03650                               *ReplaceSub++ = 4;
03651                               *ReplaceSub++ = ma[1];
03652                               *ReplaceSub++ = C->numrhs;
03653                               w = ma+2;
03654                               n = *w - ARGHEAD;
03655                               w += ARGHEAD;
03656                               while ( (mm + n + 10) > C->Top )
03657                                   mm = DoubleCbuffer(AT.ebufnum,mm,15);
03658                               while ( --n >= 0 ) *mm++ = *w++;
03659                               *mm++ = 0;
03660                               C->rhs[C->numrhs+1] = mm;
03661                               C->Pointer = mm;
03662                               mm = AddRHS(AT.ebufnum,1);
03663                               w = ma+2;
03664                               n = *w - ARGHEAD;
03665                               w += ARGHEAD;
03666                               while ( (mm + n + 13) > C->Top )
03667                                   mm = DoubleCbuffer(AT.ebufnum,mm,16);
03668                               sstop = w + n;
03669                               while ( w < sstop ) {
03670                                   tt = w + *w; ttstop = tt - ABS(tt[-1]);
03671                                   ss = mm; mm++; w++;
03672                                   while ( w < ttstop ) {      /* Subterms */
03673                                       if ( *w != INDEX ) {
03674                                           n = w[1];
03675                                           NCOPY(mm,w,n);
03676                                       }
03677                                       else {
03678                                           v1 = mm;
03679                                           *mm++ = *w++;
03680                                           *mm++ = n = *w++;
03681                                           n -= 2;
03682                                           while ( --n >= 0 ) {
```

```
03683                                                   if ( *w >= MINSPEC ) *mm++ = *w++;
03684                                                   else v2 = w++;
03685                                               }
03686                                               n = WORDDIF(mm,v1);
03687                                               if ( n != v1[1] ) {
03688                                                   if ( n <= 2 ) mm -= 2;
03689                                                   else v1[1] = n;
03690                                                   *mm++ = VECTOR;
03691                                                   *mm++ = 4;
03692                                                   *mm++ = *v2;
03693                                                   *mm++ = FUNNYVEC;
03694                                               }
03695                                           }
03696                                       }
03697                                       while ( w < tt ) *mm++ = *w++;
03698                                       *ss = WORDDIF(mm,ss);
03699                                   }
03700                                   *mm++ = 0;
03701                                   C->rhs[C->numrhs+1] = mm;
03702                                   C->Pointer = mm;
03703                                   if ( mm > C->Top ) {
03704 /* INTERNAL_ERROR_EXCL_START */
03705                                       MLOCK(ErrorMessageLock);
03706                                       MesPrint("!>Internal error in Normalize with extra compiler
      buffer");
03707                                       MUNLOCK(ErrorMessageLock);
03708                                       Terminate(-1);
03709 /* INTERNAL_ERROR_EXCL_STOP */
03710                                   }
03711                                   ma += 2 + ma[2];
03712                                   continue;
03713                               }
03714                               else goto NoRep;
03715                           }
03716                           else if ( *ma == -INDEX ) {
03717                               if ( ( ma[2] == -INDEX || ma[2] == -VECTOR )
03718                               && ma+4 <= mb )
03719                                   *ReplaceSub++ = INDTOIND;
03720                               else if ( ma[1] >= AM.OffsetIndex ) {
03721                                   if ( ma[2] == -SNUMBER && ma+4 <= mb
03722                                   && ma[3] >= 0 && ma[3] < AM.OffsetIndex )
03723                                       *ReplaceSub++ = INDTOIND;
03724                                   else if ( ma[2] == ARGHEAD && ma+2+ARGHEAD <= mb ) {
03725                                       *ReplaceSub++ = INDTOIND;
03726                                       *ReplaceSub++ = 4;
03727                                       *ReplaceSub++ = ma[1];
03728                                       *ReplaceSub++ = 0;
03729                                       ma += 2+ARGHEAD;
03730                                       continue;
03731                                   }
03732                                   else goto NoRep;
03733                               }
03734                               else goto NoRep;
03735                           }
03736                           else goto NoRep;
03737                           *ReplaceSub++ = 4;
03738                           *ReplaceSub++ = ma[1];
03739                           *ReplaceSub++ = ma[3];
03740                           ma += 4;
03741                       }
03742
03743                   }
03744                   AN.ReplaceScrat[1] = ReplaceSub-AN.ReplaceScrat;
03745 /*
03746                   Success. This means that we have to remove the replace_
03747                   from the functions. It starts at mc and end at mb.
03748 */
03749                   while ( mb < m ) *mc++ = *mb++;
03750                   m = mc;
03751                   break;
03752 NoRep:
03753                   if ( ReplaceType > 0 ) {
03754                       C->numrhs = oldtoprhs;
03755                       C->Pointer = C->Buffer + oldcpointer;
03756                   }
03757                   ReplaceType = -1;
03758                   if ( ++ReplaceVeto >= 0 ) break;
03759               }
03760               ma = mb;
03761           }
03762       }
03763 /*
03764       #] Track Replace_ :
03765       #[ LeviCivita tensors :
03766 */
03767       if ( neps ) {
03768           to = m;
```

```
03769         for ( i = 0; i < neps; i++ ) {  /* Put the indices in order */
03770             t = peps[i];
03771             if ( ( t[2] & DIRTYSYMFLAG ) != DIRTYSYMFLAG ) continue;
03772             t[2] &= ~DIRTYSYMFLAG;
03773             if ( AR.Eside == LHSIDE || AR.Eside == LHSIDEX ) {
03774                         /* Potential problems with FUNNYWILD */
03775 /*
03776                 First make sure all FUNNIES are at the end.
03777                 Then sort separately
03778 */
03779                 r = t + FUNHEAD;
03780                 m = tt = t + t[1];
03781                 while ( r < m ) {
03782                     if ( *r != FUNNYWILD ) { r++; continue; }
03783                     k = r[1]; u = r + 2;
03784                     while ( u < tt ) {
03785                         u[-2] = *u;
03786                         if ( *u != FUNNYWILD ) ncoef = -ncoef;
03787                         u++;
03788                     }
03789                     tt[-2] = FUNNYWILD; tt[-1] = k; m -= 2;
03790                 }
03791                 t += FUNHEAD;
03792                 do {
03793                     for ( r = t + 1; r < m; r++ ) {
03794                         if ( *r < *t ) { k = *r; *r = *t; *t = k; ncoef = -ncoef; }
03795                         else if ( *r == *t ) goto NormZero;
03796                     }
03797                     t++;
03798                 } while ( t < m );
03799                 do {
03800                     for ( r = t + 2; r < tt; r += 2 ) {
03801                         if ( r[1] < t[1] ) {
03802                             k = r[1]; r[1] = t[1]; t[1] = k; ncoef = -ncoef; }
03803                         else if ( r[1] == t[1] ) goto NormZero;
03804                     }
03805                     t += 2;
03806                 } while ( t < tt );
03807             }
03808             else {
03809                 m = t + t[1];
03810                 t += FUNHEAD;
03811                 do {
03812                     for ( r = t + 1; r < m; r++ ) {
03813                         if ( *r < *t ) { k = *r; *r = *t; *t = k; ncoef = -ncoef; }
03814                         else if ( *r == *t ) goto NormZero;
03815                     }
03816                     t++;
03817                 } while ( t < m );
03818             }
03819         }
03820
03821         /* Sort the tensors */
03822
03823         for ( i = 0; i < (neps-1); i++ ) {
03824             t = peps[i];
03825             for ( j = i+1; j < neps; j++ ) {
03826                 r = peps[j];
03827                 if ( t[1] > r[1] ) {
03828                     peps[i] = m = r; peps[j] = r = t; t = m;
03829                 }
03830                 else if ( t[1] == r[1] ) {
03831                     k = t[1] - FUNHEAD;
03832                     m = t + FUNHEAD;
03833                     r += FUNHEAD;
03834                     do {
03835                         if ( *r < *m ) {
03836                             m = peps[j]; peps[j] = t; peps[i] = t = m;
03837                             break;
03838                         }
03839                         else if ( *r++ > *m++ ) break;
03840                     } while ( --k > 0 );
03841                 }
03842             }
03843         }
03844         m = to;
03845         for ( i = 0; i < neps; i++ ) {
03846             t = peps[i];
03847             k = t[1];
03848             NCOPY(m,t,k);
03849         }
03850     }
03851 /*
03852     #] LeviCivita tensors :
03853     #[ Delta :
03854 */
03855     if ( ndel ) {
```

```
03856         r = t = pdel;
03857         for ( i = 0; i < ndel; i += 2, r += 2 ) {
03858             if ( r[1] < r[0] ) { k = *r; *r = r[1]; r[1] = k; }
03859         }
03860         for ( i = 2; i < ndel; i += 2, t += 2 ) {
03861             r = t + 2;
03862             for ( j = i; j < ndel; j += 2 ) {
03863                 if ( *r > *t ) { r += 2; }
03864                 else if ( *r < *t ) {
03865                     k = *r; *r++ = *t; *t++ = k;
03866                     k = *r; *r++ = *t; *t-- = k;
03867                 }
03868                 else {
03869                     if ( *++r < t[1] ) {
03870                         k = *r; *r = t[1]; t[1] = k;
03871                     }
03872                     r++;
03873                 }
03874             }
03875         }
03876         t = pdel;
03877         *m++ = DELTA;
03878         *m++ = ndel + 2;
03879         i = ndel;
03880         NCOPY(m,t,i);
03881     }
03882 /*
03883     #] Delta :
03884     #[ Loose Vectors/Indices :
03885 */
03886     if ( nind ) {
03887         t = pind;
03888         for ( i = 0; i < nind; i++ ) {
03889             r = t + 1;
03890             for ( j = i+1; j < nind; j++ ) {
03891                 if ( *r < *t ) {
03892                     k = *r; *r = *t; *t = k;
03893                 }
03894                 r++;
03895             }
03896             t++;
03897         }
03898         t = pind;
03899         *m++ = INDEX;
03900         *m++ = nind + 2;
03901         i = nind;
03902         NCOPY(m,t,i);
03903     }
03904 /*
03905     #] Loose Vectors/Indices :
03906     #[ Vectors :
03907 */
03908     if ( nvec ) {
03909         t = pvec;
03910         for ( i = 2; i < nvec; i += 2 ) {
03911             r = t + 2;
03912             for ( j = i; j < nvec; j += 2 ) {
03913                 if ( *r == *t ) {
03914                     if ( *++r < t[1] ) {
03915                         k = *r; *r = t[1]; t[1] = k;
03916                     }
03917                     r++;
03918                 }
03919                 else if ( *r < *t ) {
03920                     k = *r; *r++ = *t; *t++ = k;
03921                     k = *r; *r++ = *t; *t-- = k;
03922                 }
03923                 else { r += 2; }
03924             }
03925             t += 2;
03926         }
03927         t = pvec;
03928         *m++ = VECTOR;
03929         *m++ = nvec + 2;
03930         i = nvec;
03931         NCOPY(m,t,i);
03932     }
03933 /*
03934     #] Vectors :
03935     #[ Dotproducts :
03936 */
03937     if ( ndot ) {
03938         to = m;
03939         m = t = pdot;
03940         i = ndot;
03941         while ( --i >= 0 ) {
03942             if ( *t > t[1] ) { j = *t; *t = t[1]; t[1] = j; }
```

```
03943             t += 3;
03944         }
03945         t = m;
03946         ndot *= 3;
03947         m += ndot;
03948         while ( t < (m-3) ) {
03949             r = t + 3;
03950             do {
03951                 if ( *r == *t ) {
03952                     if ( *++r == *++t ) {
03953                         r++;
03954                         if ( ( *r < MAXPOWER && t[1] < MAXPOWER )
03955                         || ( *r > -MAXPOWER && t[1] > -MAXPOWER ) ) {
03956                             t++;
03957                             *t += *r;
03958                             if ( *t > MAXPOWER || *t < -MAXPOWER ) {
03959                                 MLOCK(ErrorMessageLock);
03960                                 MesPrint("Exponent of dotproduct out of range: %d",*t);
03961                                 MUNLOCK(ErrorMessageLock);
03962                                 goto NormMin;
03963                             }
03964                             ndot -= 3;
03965                             *r-- = *--m;
03966                             *r-- = *--m;
03967                             *r   = *--m;
03968                             if ( !*t ) {
03969                                 ndot -= 3;
03970                                 *t-- = *--m;
03971                                 *t-- = *--m;
03972                                 *t   = *--m;
03973                                 t -= 3;
03974                                 break;
03975                             }
03976                         }
03977                         else if ( *r < *++t ) {
03978                             k = *r; *r++ = *t; *t = k;
03979                         }
03980                         else r++;
03981                         t -= 2;
03982                     }
03983                     else if ( *r < *t ) {
03984                         k = *r; *r++ = *t; *t++ = k;
03985                         k = *r; *r++ = *t; *t = k;
03986                         t -= 2;
03987                     }
03988                     else { r += 2; t--; }
03989                 }
03990                 else if ( *r < *t ) {
03991                     k = *r; *r++ = *t; *t++ = k;
03992                     k = *r; *r++ = *t; *t++ = k;
03993                     k = *r; *r++ = *t; *t   = k;
03994                     t -= 2;
03995                 }
03996                 else { r += 3; }
03997             } while ( r < m );
03998             t += 3;
03999         }
04000         m = to;
04001         t = pdot;
04002         if ( ( i = ndot ) > 0 ) {
04003             *m++ = DOTPRODUCT;
04004             *m++ = i + 2;
04005             NCOPY(m,t,i);
04006         }
04007     }
04008 /*
04009     #] Dotproducts :
04010     #[ Symbols :
04011 */
04012     if ( nsym ) {
04013         nsym «= 1;
04014         t = psym;
04015         *m++ = SYMBOL;
04016         r = m;
04017         *m++ = ( i = nsym ) + 2;
04018         if ( i ) { do {
04019             if ( !*t ) {
04020                 if ( t[1] < (2*MAXPOWER) ) {        /* powers of i */
04021                     if ( t[1] & 1 ) { *m++ = 0; *m++ = 1; }
04022                     else *r -= 2;
04023                     if ( *++t & 2 ) ncoef = -ncoef;
04024                     t++;
04025                 }
04026             }
04027             else if ( *t <= NumSymbols && *t > -2*MAXPOWER ) {  /* Put powers in range */
04028                 if ( ( ( ( t[1] > symbols[*t].maxpower ) && ( symbols[*t].maxpower < MAXPOWER ) ) ||
04029                        ( ( t[1] < symbols[*t].minpower ) && ( symbols[*t].minpower > -MAXPOWER ) ) )
```

```
    &&
04030                       ( t[1] < 2*MAXPOWER ) && ( t[1] > -2*MAXPOWER ) ) {
04031                   if ( i <= 2 || t[2] != *t ) goto NormZero;
04032               }
04033               if ( AN.ncmod == 1 && ( AC.modmode & ALSOPOWERS ) != 0 ) {
04034                   if ( AC.cmod[0] == 1 ) t[1] = 0;
04035                   else if ( t[1] >= 0 ) t[1] = 1 + (t[1]-1)%(AC.cmod[0]-1);
04036                   else {
04037                       t[1] = -1 - (-t[1]-1)%(AC.cmod[0]-1);
04038                       if ( t[1] < 0 ) t[1] += (AC.cmod[0]-1);
04039                   }
04040               }
04041               if ( ( t[1] < (2*MAXPOWER) && t[1] >= MAXPOWER )
04042               || ( t[1] > -(2*MAXPOWER) && t[1] <= -MAXPOWER ) ) {
04043                   MLOCK(ErrorMessageLock);
04044                   MesPrint("Exponent out of range: %d",t[1]);
04045                   MUNLOCK(ErrorMessageLock);
04046                   goto NormMin;
04047               }
04048               if ( AT.TrimPower && AR.PolyFunVar == *t && t[1] > AR.PolyFunPow ) {
04049                   goto NormZero;
04050               }
04051               else if ( t[1] ) {
04052                   *m++ = *t++;
04053                   *m++ = *t++;
04054               }
04055               else { *r -= 2; t += 2; }
04056           }
04057           else {
04058               *m++ = *t++; *m++ = *t++;
04059           }
04060       } while ( (i-=2) > 0 ); }
04061       if ( *r <= 2 ) m = r-1;
04062   }
04063 /*
04064   #] Symbols :
04065   #[ float_ :
04066
04067   Here we treat float_ functions and combined them with the regular
04068   coefficient.
04069 */
04070
04071 #ifdef WITHFLOAT
04072   if ( withfloat ) {
04073       WORD floatsign = 3;
04074 /*
04075       First check whether the coefficient is already 1/1
04076 */
04077       if ( ABS(ncoef) == 3 && n_coef[0] == 1 && n_coef[1] == 1 ) {
04078           if ( withfloat == 1 ) {
04079               t = firstfloat;
04080 /*
04081               We only interfere by making the sign positive.
04082               The float_ function has already been tested.
04083 */
04084               if ( t[FUNHEAD+3] < 0 ) {
04085                   t[FUNHEAD+3] = -t[FUNHEAD+3];
04086                   floatsign = -floatsign;
04087               }
04088               if ( ncoef < 0 ) floatsign = -floatsign;
04089 /*
04090               Copy to output;
04091 */
04092               AT.FloatPos = m-termout;
04093               i = t[1]; NCOPY(m,t,i)
04094           }
04095           else {
04096               PackFloat(m,aux4);
04097               if ( m[FUNHEAD+3] < 0 ) {
04098                   m[FUNHEAD+3] = -m[FUNHEAD+3];
04099                   floatsign = -floatsign;
04100               }
04101               if ( ncoef < 0 ) floatsign = -floatsign;
04102               AT.FloatPos = m-termout;
04103               m += m[1];
04104           }
04105       }
04106       else {
04107           if ( withfloat == 1 ) UnpackFloat(aux4,firstfloat);
04108           RatToFloat(aux5,(UWORD *)n_coef,ncoef);
04109           mpf_mul(aux4,aux4,aux5);
04110           PackFloat(m,aux4);
04111           AT.FloatPos = m-termout;
04112           if ( m[FUNHEAD+3] < 0 ) {
04113               m[FUNHEAD+3] = -m[FUNHEAD+3];
04114               floatsign = -floatsign;
04115           }
```

```
04116             m += m[1];
04117         }
04118         n_coef[0] = 1; n_coef[1] = 1; ncoef = floatsign;
04119     }
04120     else AT.FloatPos = 0;
04121 #endif
04122
04123 /*
04124     #] float_ :
04125     #[ Do Replace_ :
04126 */
04127     stop = (WORD *)(((UBYTE *)(termout)) + AM.MaxTer);
04128     i = ABS(ncoef);
04129     if ( ( m + i ) > stop ) {
04130         MLOCK(ErrorMessageLock);
04131         MesPrint("Term too complex during normalization");
04132         MUNLOCK(ErrorMessageLock);
04133         goto NormMin;
04134     }
04135     if ( AT.SS == AT.S0 ) {
04136         if ( ( m + i - termout ) > AT.SS->verbMaxTermSize ) {
04137             AT.SS->verbMaxTermSize = m+i-termout;
04138         }
04139     }
04140     if ( ReplaceType >= 0 ) {
04141         t = n_coef;
04142         i--;
04143         NCOPY(m,t,i);
04144         *m++ = ncoef;
04145         t = termout;
04146         *t = WORDDIF(m,t);
04147         if ( ReplaceType == 0 ) {
04148             AT.WorkPointer = termout+*termout;
04149             WildFill(BHEAD term,termout,AN.ReplaceScrat);
04150             termout = term + *term;
04151         }
04152         else {
04153             AT.WorkPointer = r = termout + *termout;
04154             WildFill(BHEAD r,termout,AN.ReplaceScrat);
04155             i = *r; m = term;
04156             NCOPY(m,r,i);
04157             termout = m;
04158
04159
04160             r = m = term;
04161             r += *term; r -= ABS(r[-1]);
04162             m++;
04163             while ( m < r ) {
04164                 if ( *m >= FUNCTION && m[1] > FUNHEAD &&
04165                 functions[*m-FUNCTION].spec != TENSORFUNCTION )
04166                     m[2] |= DIRTYFLAG;
04167                 m += m[1];
04168             }
04169         }
04170 /*
04171         The next 'reset' cannot be done. We still need the expression
04172         in the buffer. Note though that this may cause a runaway pointer
04173         if we are not very careful.
04174
04175         C->numrhs = oldtoprhs;
04176         C->Pointer = C->Buffer + oldcpointer;
04177 */
04178         AT.NormDepth--;
04179         TermFree(n_llnum,"n_llnum");
04180         TermFree(n_coef,"NormCoef");
04181         return(1);
04182     }
04183     else {
04184         t = termout;
04185         k = WORDDIF(m,t);
04186         *t = k + i;
04187         m = term;
04188         NCOPY(m,t,k);
04189         i--;
04190         t = n_coef;
04191         NCOPY(m,t,i);
04192         *m++ = ncoef;
04193     }
04194 /*
04195     #] Do Replace_ :
04196     #[ Errors and Finish :
04197 */
04198 RegEnd:
04199     if ( termout < term + *term && termout >= term ) AT.WorkPointer = term + *term;
04200     else AT.WorkPointer = termout;
04201 /*
04202     if ( termflag ) {   We have to assign the term to $variable(s)
```

```
04203         TermAssign(term);
04204     }
04205 */
04206     AT.NormDepth--;
04207     TermFree(n_llnum,"n_llnum");
04208     TermFree(n_coef,"NormCoef");
04209     return(regval);
04210
04211 NormInf:
04212     MLOCK(ErrorMessageLock);
04213     MesPrint("Division by zero during normalization");
04214     MUNLOCK(ErrorMessageLock);
04215     Terminate(-1);
04216
04217 NormZZ:
04218     MLOCK(ErrorMessageLock);
04219     MesPrint("0^0 during normalization of term");
04220     MUNLOCK(ErrorMessageLock);
04221     Terminate(-1);
04222
04223 NormPRF:
04224     MLOCK(ErrorMessageLock);
04225     MesPrint("0/0 in polyratfun during normalization of term");
04226     MUNLOCK(ErrorMessageLock);
04227     Terminate(-1);
04228
04229 NormZero:
04230     *term = 0;
04231     AT.WorkPointer = termout;
04232     AT.NormDepth--;
04233     TermFree(n_llnum,"n_llnum");
04234     TermFree(n_coef,"NormCoef");
04235     return(regval);
04236
04237 NormMin:
04238     AT.NormDepth--;
04239     TermFree(n_llnum,"n_llnum");
04240     TermFree(n_coef,"NormCoef");
04241     return(-1);
04242
04243 FromNorm:
04244     MLOCK(ErrorMessageLock);
04245     MesCall("Norm");
04246     MUNLOCK(ErrorMessageLock);
04247     AT.NormDepth--;
04248     TermFree(n_llnum,"n_llnum");
04249     TermFree(n_coef,"NormCoef");
04250     return(-1);
04251
04252 /*
04253     #] Errors and Finish :
04254 */
04255 }
04256
04257 /*
04258         #] Normalize :
04259         #[ ExtraSymbol :
04260 */
04261
04262 int ExtraSymbol(WORD sym, WORD pow, WORD nsym, WORD *ppsym, WORD *ncoef)
04263 {
04264     WORD *m, i;
04265     i = nsym;
04266     m = ppsym - 2;
04267     while   ( i > 0 ) {
04268         if ( sym == *m ) {
04269             m++;
04270             if  ( pow > 2*MAXPOWER || pow < -2*MAXPOWER
04271             ||  *m > 2*MAXPOWER || *m < -2*MAXPOWER ) {
04272                 MLOCK(ErrorMessageLock);
04273                 MesPrint("Illegal wildcard power combination.");
04274                 MUNLOCK(ErrorMessageLock);
04275                 Terminate(-1);
04276             }
04277             *m += pow;
04278
04279             if ( ( sym <= NumSymbols && sym > -MAXPOWER )
04280              && ( symbols[sym].complex & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY ) {
04281                 *m %= symbols[sym].maxpower;
04282                 if ( *m < 0 ) *m += symbols[sym].maxpower;
04283                 if ( ( symbols[sym].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
04284                     if ( ( ( symbols[sym].maxpower & 1 ) == 0 ) &&
04285                          ( *m >= symbols[sym].maxpower/2 ) ) {
04286                         *m -= symbols[sym].maxpower/2; *ncoef = -*ncoef;
04287                     }
04288                 }
04289             }
```

```
04290
04291             if  ( *m >= 2*MAXPOWER || *m <= -2*MAXPOWER ) {
04292                 MLOCK(ErrorMessageLock);
04293                 MesPrint("Power overflow during normalization");
04294                 MUNLOCK(ErrorMessageLock);
04295                 return(-1);
04296             }
04297             if ( !*m ) {
04298                 m--;
04299                 while ( i < nsym )
04300                     { *m = m[2]; m++; *m = m[2]; m++; i++; }
04301                 return(-1);
04302             }
04303             return(0);
04304         }
04305         else if ( sym < *m ) {
04306             m -= 2;
04307             i--;
04308         }
04309         else break;
04310     }
04311     m = ppsym;
04312     while ( i < nsym )
04313         { m--; m[2] = *m; m--; m[2] = *m; i++; }
04314     *m++ = sym;
04315     *m = pow;
04316     return(1);
04317 }
04318
04319 /*
04320         #] ExtraSymbol :
04321         #[ DoTheta :
04322 */
04323
04324 int DoTheta(PHEAD WORD *t)
04325 {
04326     GETBIDENTITY
04327     WORD k, *r1, *r2, *tstop, type;
04328     WORD ia, *ta, *tb, *stopa, *stopb;
04329     if ( AC.BracketNormalize ) return(-1);
04330     type = *t;
04331     k = t[1];
04332     tstop = t + k;
04333     t += FUNHEAD;
04334     if ( k <= FUNHEAD ) return(1);
04335     r1 = t;
04336     NEXTARG(r1)
04337     if ( r1 == tstop ) {
04338 /*
04339         One argument
04340 */
04341         if ( *t == ARGHEAD ) {
04342             if ( type == THETA ) return(1);
04343             else return(0);  /* THETA2 */
04344         }
04345         if ( *t < 0 ) {
04346             if ( *t == -SNUMBER ) {
04347                 if ( t[1] < 0 ) return(0);
04348                 else {
04349                     if ( type == THETA2 && t[1] == 0 ) return(0);
04350                     else return(1);
04351                 }
04352             }
04353             return(-1);
04354         }
04355         k = t[*t-1];
04356         if ( *t == ABS(k)+1+ARGHEAD ) {
04357             if ( k > 0 ) return(1);
04358             else return(0);
04359         }
04360         return(-1);
04361     }
04362 /*
04363     At least two arguments
04364 */
04365     r2 = r1;
04366     NEXTARG(r2)
04367     if ( r2 < tstop ) return(-1);   /* More than 2 arguments */
04368 /*
04369     Note now that zero has to be treated specially
04370     We take the criteria from the symmetrize routine
04371 */
04372     if ( *t == -SNUMBER && *r1 == -SNUMBER ) {
04373         if ( t[1] > r1[1] ) return(0);
04374         else if ( t[1] < r1[1] ) {
04375             return(1);
04376         }
```

```
04377         else if ( type == THETA ) return(1);
04378         else return(0);   /* THETA2 */
04379     }
04380     else if ( t[1] == 0 && *t == -SNUMBER ) {
04381         if ( *r1 > 0 ) { }
04382         else if ( *t < *r1 ) return(1);
04383         else if ( *t > *r1 ) return(0);
04384     }
04385     else if ( r1[1] == 0 && *r1 == -SNUMBER ) {
04386         if ( *t > 0 ) { }
04387         else if ( *t < *r1 ) return(1);
04388         else if ( *t > *r1 ) return(0);
04389     }
04390     r2 = AT.WorkPointer;
04391     if ( *t < 0 ) {
04392         ta = r2;
04393         ToGeneral(t,ta,0);
04394         r2 += *r2;
04395     }
04396     else ta = t;
04397     if ( *r1 < 0 ) {
04398         tb = r2;
04399         ToGeneral(r1,tb,0);
04400     }
04401     else tb = r1;
04402     stopa = ta + *ta;
04403     stopb = tb + *tb;
04404     ta += ARGHEAD; tb += ARGHEAD;
04405     while ( ta < stopa ) {
04406         if ( tb >= stopb ) return(0);
04407         if ( ( ia = CompareTerms(BHEAD ta,tb,(WORD)1) ) < 0 ) return(0);
04408         if ( ia > 0 ) return(1);
04409         ta += *ta;
04410         tb += *tb;
04411     }
04412     if ( type == THETA ) return(1);
04413     else return(0); /* THETA2 */
04414 }
04415
04416 /*
04417         #] DoTheta :
04418         #[ DoDelta :
04419 */
04420
04421 int DoDelta(WORD *t)
04422 {
04423     WORD k, *r1, *r2, *tstop, isnum, isnum2, type = *t;
04424     if ( AC.BracketNormalize ) return(-1);
04425     k = t[1];
04426     if ( k <= FUNHEAD ) goto argzero;
04427     if ( k == FUNHEAD+ARGHEAD && t[FUNHEAD] == ARGHEAD ) goto argzero;
04428     tstop = t + k;
04429     t += FUNHEAD;
04430     r1 = t;
04431     NEXTARG(r1)
04432     if ( *t < 0 ) {
04433         k = 1;
04434         if ( *t == -SNUMBER ) { isnum = 1; k = t[1]; }
04435         else isnum = 0;
04436     }
04437     else {
04438         k = t[*t-1];
04439         k = ABS(k);
04440         if ( k == *t-ARGHEAD-1 ) isnum = 1;
04441         else isnum = 0;
04442         k = 1;
04443     }
04444     if ( r1 >= tstop ) {        /* Single argument */
04445         if ( !isnum ) return(-1);
04446         if ( k == 0 ) goto argzero;
04447         goto argnonzero;
04448     }
04449     r2 = r1;
04450     NEXTARG(r2)
04451     if ( r2 < tstop ) return(-1);
04452     if ( *r1 < 0 ) {
04453         if ( *r1 == -SNUMBER ) { isnum2 = 1; }
04454         else isnum2 = 0;
04455     }
04456     else {
04457         k = r1[*r1-1];
04458         k = ABS(k);
04459         if ( k == *r1-ARGHEAD-1 ) isnum2 = 1;
04460         else isnum2 = 0;
04461     }
04462     if ( isnum != isnum2 ) return(-1);
04463     tstop = r1;
```

```
04464     while ( t < tstop && r1 < r2 ) {
04465         if ( *t != *r1 ) {
04466             if ( !isnum ) return(-1);
04467             goto argnonzero;
04468         }
04469         t++; r1++;
04470     }
04471     if ( t != tstop || r1 != r2 ) {
04472         if ( !isnum ) return(-1);
04473         goto argnonzero;
04474     }
04475 argzero:
04476     if ( type == DELTA2 ) return(1);
04477     else return(0);
04478 argnonzero:
04479     if ( type == DELTA2 ) return(0);
04480     else return(1);
04481 }
04482
04483 /*
04484         #] DoDelta :
04485         #[ DoRevert :
04486 */
04487
04488 void DoRevert(WORD *fun, WORD *tmp)
04489 {
04490     WORD *t, *r, *m, *to, *tt, *mm, i, j;
04491     to = fun + fun[1];
04492     r = fun + FUNHEAD;
04493     while ( r < to ) {
04494         if ( *r <= 0 ) {
04495             if ( *r == -REVERSEFUNCTION ) {
04496                 m = r; mm = m+1;
04497                 while ( mm < to ) *m++ = *mm++;
04498                 to--;
04499                 (fun[1])--;
04500                 fun[2] |= DIRTYSYMFLAG;
04501             }
04502             else if ( *r <= -FUNCTION ) r++;
04503             else {
04504                 if ( *r == -INDEX && r[1] < MINSPEC ) *r = -VECTOR;
04505                 r += 2;
04506             }
04507         }
04508         else {
04509             if ( ( *r > ARGHEAD )
04510             && ( r[ARGHEAD+1] == REVERSEFUNCTION )
04511             && ( *r == (r[ARGHEAD]+ARGHEAD) )
04512             && ( r[ARGHEAD] == (r[ARGHEAD+2]+4) )
04513             && ( *(r+*r-3) == 1 )
04514             && ( *(r+*r-2) == 1 )
04515             && ( *(r+*r-1) == 3 ) ) {
04516                 mm = r;
04517                 r += ARGHEAD + 1;
04518                 tt = r + r[1];
04519                 r += FUNHEAD;
04520                 m = tmp;
04521                 t = r;
04522                 j = 0;
04523                 while ( t < tt ) {
04524                     NEXTARG(t)
04525                     j++;
04526                 }
04527                 while ( --j >= 0 ) {
04528                     i = j;
04529                     t = r;
04530                     while ( --i >= 0 ) {
04531                         NEXTARG(t)
04532                     }
04533                     if ( *t > 0 ) {
04534                         i = *t;
04535                         NCOPY(m,t,i);
04536                     }
04537                     else if ( *t <= -FUNCTION ) *m++ = *t++;
04538                     else { *m++ = *t++; *m++ = *t++; }
04539                 }
04540                 i = WORDDIF(m,tmp);
04541                 m = tmp;
04542                 t = mm;
04543                 r = t + *t;
04544                 NCOPY(t,m,i);
04545                 m = r;
04546                 r = t;
04547                 i = WORDDIF(to,m);
04548                 NCOPY(t,m,i);
04549                 fun[1] = WORDDIF(t,fun);
04550                 to = t;
```

```
04551                 fun[2] |= DIRTYSYMFLAG;
04552             }
04553             else r += *r;
04554         }
04555     }
04556 }
04557
04558 /*
04559         #] DoRevert :
04560     #] Normalize :
04561     #[ DetCommu :
04562
04563     Determines the number of terms in an expression that contain
04564     noncommuting objects. This can be used to see whether products of
04565     this expression can be evaluated with binomial coefficients.
04566
04567     We don't try to be fancy. If a term contains noncommuting objects
04568     we are not looking whether they can commute with complete other
04569     terms.
04570
04571     If the number gets too large we cut it off.
04572 */
04573
04574 #define MAXNUMBEROFNONCOMTERMS 2
04575
04576 WORD DetCommu(WORD *terms)
04577 {
04578     WORD *t, *tnext, *tstop;
04579     WORD num = 0;
04580     if ( *terms == 0 ) return(0);
04581     if ( terms[*terms] == 0 ) return(0);
04582     t = terms;
04583     while ( *t ) {
04584         tnext = t + *t;
04585         tstop = tnext - ABS(tnext[-1]);
04586         t++;
04587         while ( t < tstop ) {
04588             if ( *t >= FUNCTION ) {
04589                 if ( functions[*t-FUNCTION].commute ) {
04590                     num++;
04591                     if ( num >= MAXNUMBEROFNONCOMTERMS ) return(num);
04592                     break;
04593                 }
04594             }
04595             else if ( *t == SUBEXPRESSION ) {
04596                 if ( cbuf[t[4]].CanCommu[t[2]] ) {
04597                     num++;
04598                     if ( num >= MAXNUMBEROFNONCOMTERMS ) return(num);
04599                     break;
04600                 }
04601             }
04602             else if ( *t == EXPRESSION ) {
04603                 num++;
04604                 if ( num >= MAXNUMBEROFNONCOMTERMS ) return(num);
04605                 break;
04606             }
04607             else if ( *t == DOLLAREXPRESSION ) {
04608 /*
04609                 Technically this is not correct. We have to test first
04610                 whether this is DollarLocalCopy (in TFORM) and if so, use the
04611                 local version. Anyway, this should be rare to never
04612                 occurring because dollars should be replaced.
04613 */
04614                 if ( cbuf[AM.dbufnum].CanCommu[t[2]] ) {
04615                     num++;
04616                     if ( num >= MAXNUMBEROFNONCOMTERMS ) return(num);
04617                     break;
04618                 }
04619             }
04620             t += t[1];
04621         }
04622         t = tnext;
04623     }
04624     return(num);
04625 }
04626
04627 /*
04628     #] DetCommu :
04629     #[ DoesCommu :
04630
04631     Determines the number of noncommuting objects in a term.
04632     If the number gets too large we cut it off.
04633 */
04634
04635 WORD DoesCommu(WORD *term)
04636 {
04637     WORD *tstop;
```

```
04638     WORD num = 0;
04639     if ( *term == 0 ) return(0);
04640     tstop = term + *term;
04641     tstop = tstop - ABS(tstop[-1]);
04642     term++;
04643     while ( term < tstop ) {
04644         if ( ( *term >= FUNCTION ) && ( functions[*term-FUNCTION].commute ) ) {
04645             num++;
04646             if ( num >= MAXNUMBEROFNONCOMTERMS ) return(num);
04647         }
04648         term += term[1];
04649     }
04650     return(num);
04651 }
04652
04653 /*
04654     #] DoesCommu :
04655     #[ PolyNormPoly :
04656
04657         Normalizes a polynomial
04658 */
04659
04660 #ifdef EVALUATEGCD
04661 WORD *PolyNormPoly (PHEAD WORD *Poly) {
04662
04663     GETBIDENTITY;
04664     WORD *buffer = AT.WorkPointer;
04665     WORD *p;
04666     if ( NewSort(BHEAD0) ) { Terminate(-1); }
04667     AR.CompareRoutine = (COMPAREDUMMY)(&CompareSymbols);
04668     while ( *Poly ) {
04669         p = Poly + *Poly;
04670         if ( SymbolNormalize(Poly) < 0 ) return(0);
04671         if ( StoreTerm(BHEAD Poly) ) {
04672             AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
04673             LowerSortLevel();
04674             Terminate(-1);
04675         }
04676         Poly = p;
04677     }
04678     if ( EndSort(BHEAD buffer,1) < 0 ) {
04679         AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
04680         Terminate(-1);
04681     }
04682     p = buffer;
04683     while ( *p ) p += *p;
04684     AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
04685     AT.WorkPointer = p + 1;
04686     return(buffer);
04687 }
04688 #endif
04689
04690 /*
04691     #] PolyNormPoly :
04692     #[ EvaluateGcd :
04693
04694     Try to evaluate the GCDFUNCTION gcd_.
04695     This function can have a number of arguments which can be numbers
04696     and/or polynomials. If there are objects that aren't SYMBOLS or numbers
04697     it cannot work currently.
04698
04699     To make this work properly we have to intervene in proces.c
04700      proces.c:                       if ( Normalize(BHEAD m) ) {
04701 1060 proces.c:                           if ( Normalize(BHEAD r) ) {
04702 1126?proces.c:                           if ( Normalize(BHEAD term) ) {
04703      proces.c:                 if ( Normalize(BHEAD AT.WorkPointer) ) goto PasErr;
04704 2308!proces.c:      if ( ( retnorm = Normalize(BHEAD term) ) != 0 ) {
04705      proces.c:                     ReNumber(BHEAD term); Normalize(BHEAD term);
04706      proces.c:                 if ( Normalize(BHEAD v) ) Terminate(-1);
04707      proces.c:      if ( Normalize(BHEAD w) ) { LowerSortLevel(); goto PolyCall; }
04708      proces.c:      if ( Normalize(BHEAD term) ) goto PolyCall;
04709 */
04710 #ifdef EVALUATEGCD
04711
04712 WORD *EvaluateGcd(PHEAD WORD *subterm)
04713 {
04714     GETBIDENTITY
04715     WORD *oldworkpointer = AT.WorkPointer, *work1, *work2, *work3;
04716     WORD *t, *tt, *ttt, *t1, *t2, *t3, *t4, *tstop;
04717     WORD ct, nnum;
04718     UWORD gcdnum, stor;
04719     WORD *lnum=n_llnum+1;
04720     WORD *num1, *num2, *num3, *den1, *den2, *den3;
04721     WORD sizenum1, sizenum2, sizenum3, sizeden1, sizeden2, sizeden3;
04722     int i, isnumeric = 0, numarg = 0 /*, sizearg */;
04723     LONG size;
04724 /*
```

```
04725     Step 1: Look for -SNUMBER or -SYMBOL arguments.
04726             If encountered, treat everybody with it.
04727 */
04728     tt = subterm + subterm[1]; t = subterm + FUNHEAD;
04729
04730     while ( t < tt ) {
04731         numarg++;
04732         if ( *t == -SNUMBER ) {
04733             if ( t[1] == 0 ) {
04734 gcdzero:;
04735                 MLOCK(ErrorMessageLock);
04736                 MesPrint("Trying to take the GCD involving a zero term.");
04737                 MUNLOCK(ErrorMessageLock);
04738                 return(0);
04739             }
04740             gcdnum = ABS(t[1]);
04741             t1 = subterm + FUNHEAD;
04742             while ( gcdnum > 1 && t1 < tt ) {
04743                 if ( *t1 == -SNUMBER ) {
04744                     stor = ABS(t1[1]);
04745                     if ( stor == 0 ) goto gcdzero;
04746                     if ( GcdLong(BHEAD (UWORD *)&stor,1,(UWORD *)&gcdnum,1,
04747                                 (UWORD *)lnum,&nnum) ) goto FromGCD;
04748                     gcdnum = lnum[0];
04749                     t1 += 2;
04750                     continue;
04751                 }
04752                 else if ( *t1 == -SYMBOL ) goto gcdisone;
04753                 else if ( *t1 < 0 ) goto gcdillegal;
04754 /*
04755                 Now we have to go through all the terms in the argument.
04756                 This includes long numbers.
04757 */
04758                 ttt = t1 + *t1;
04759                 ct = *ttt; *ttt = 0;
04760                 if ( t1[1] != 0 ) { /* First normalize the argument */
04761                     t1 = PolyNormPoly(BHEAD t1+ARGHEAD);
04762                 }
04763                 else t1 += ARGHEAD;
04764                 while ( *t1 ) {
04765                     t1 += *t1;
04766                     i = ABS(t1[-1]);
04767                     t2 = t1 - i;
04768                     i = (i-1)/2;
04769                     t3 = t2+i-1;
04770                     while ( t3 > t2 && *t3 == 0 ) { t3--; i--; }
04771                     if ( GcdLong(BHEAD (UWORD *)t2,(WORD)i,(UWORD *)&gcdnum,1,
04772                                 (UWORD *)lnum,&nnum) ) {
04773                         *ttt = ct;
04774                         goto FromGCD;
04775                     }
04776                     gcdnum = lnum[0];
04777                     if ( gcdnum == 1 ) {
04778                         *ttt = ct;
04779                         goto gcdisone;
04780                     }
04781                 }
04782                 *ttt = ct;
04783                 t1 = ttt;
04784                 AT.WorkPointer = oldworkpointer;
04785             }
04786             if ( gcdnum == 1 ) goto gcdisone;
04787             oldworkpointer[0] = 4;
04788             oldworkpointer[1] = gcdnum;
04789             oldworkpointer[2] = 1;
04790             oldworkpointer[3] = 3;
04791             oldworkpointer[4] = 0;
04792             AT.WorkPointer = oldworkpointer + 5;
04793             return(oldworkpointer);
04794         }
04795         else if ( *t == -SYMBOL ) {
04796             t1 = subterm + FUNHEAD;
04797             i = t[1];
04798             while ( t1 < tt ) {
04799                 if ( *t1 == -SNUMBER ) goto gcdisone;
04800                 if ( *t1 == -SYMBOL ) {
04801                     if ( t1[1] != i ) goto gcdisone;
04802                     t1 += 2; continue;
04803                 }
04804                 if ( *t1 < 0 ) goto gcdillegal;
04805                 ttt = t1 + *t1;
04806                 ct = *ttt; *ttt = 0;
04807                 if ( t1[1] != 0 ) { /* First normalize the argument */
04808                     t2 = PolyNormPoly(BHEAD t1+ARGHEAD);
04809                 }
04810                 else t2 = t1 + ARGHEAD;
04811                 while ( *t2 ) {
```

```
04812                     t3 = t2+1;
04813                     t2 = t2 + *t2;
04814                     tstop = t2 - ABS(t2[-1]);
04815                     while ( t3 < tstop ) {
04816                         if ( *t3 != SYMBOL ) {
04817                             *ttt = ct;
04818                             goto gcdillegal;
04819                         }
04820                         t4 = t3 + 2;
04821                         t3 += t3[1];
04822                         while ( t4 < t3 ) {
04823                             if ( *t4 == i && t4[1] > 0 ) goto nextterminarg;
04824                             t4 += 2;
04825                         }
04826                     }
04827                     *ttt = ct;
04828                     goto gcdisone;
04829 nextterminarg:;
04830                 }
04831                 *ttt = ct;
04832                 t1 = ttt;
04833                 AT.WorkPointer = oldworkpointer;
04834             }
04835             oldworkpointer[0] = 8;
04836             oldworkpointer[1] = SYMBOL;
04837             oldworkpointer[2] = 4;
04838             oldworkpointer[3] = t[1];
04839             oldworkpointer[4] = 1;
04840             oldworkpointer[5] = 1;
04841             oldworkpointer[6] = 1;
04842             oldworkpointer[7] = 3;
04843             oldworkpointer[8] = 0;
04844             AT.WorkPointer = oldworkpointer+9;
04845             return(oldworkpointer);
04846         }
04847         else if ( *t < 0 ) {
04848 gcdillegal:;
04849             MLOCK(ErrorMessageLock);
04850             MesPrint("Illegal object in gcd_ function. Object not a number or a symbol.");
04851             MUNLOCK(ErrorMessageLock);
04852             goto FromGCD;
04853         }
04854         else if ( ABS(t[*t-1]) == *t-ARGHEAD-1 ) isnumeric = numarg;
04855         else if ( t[1] != 0 ) {
04856             ttt = t + *t; ct = *ttt; *ttt = 0;
04857             t = PolyNormPoly(BHEAD t+ARGHEAD);
04858             *ttt = ct;
04859             if ( t[*t] == 0 && ABS(t[*t-1]) == *t-ARGHEAD-1 ) isnumeric = numarg;
04860             AT.WorkPointer = oldworkpointer;
04861             t = ttt;
04862         }
04863         t += *t;
04864     }
04865 /*
04866     At this point there are only generic arguments.
04867     There are however still two cases:
04868     1: There is an argument that is purely numerical
04869        In that case we have to take the gcd of all coefficients
04870     2: All arguments are nontrivial polynomials.
04871        Here we don't worry so much about the factor. (???)
04872     We know whether case 1 occurs when isnumeric > 0.
04873     We can look up numarg to get a good starting value.
04874 */
04875     AT.WorkPointer = oldworkpointer;
04876     if ( isnumeric ) {
04877         t = subterm + FUNHEAD;
04878         for ( i = 1; i < isnumeric; i++ ) {
04879             NEXTARG(t);
04880         }
04881         if ( t[1] != 0 ) {  /* First normalize the argument */
04882             ttt = t + *t; ct = *ttt; *ttt = 0;
04883             t = PolyNormPoly(BHEAD t+ARGHEAD);
04884             *ttt = ct;
04885         }
04886         t += *t;
04887         i = (ABS(t[-1])-1)/2;
04888         den1 = t - 1 - i;
04889         num1 = den1 - i;
04890         sizenum1 = sizeden1 = i;
04891         while ( sizenum1 > 1 && num1[sizenum1-1] == 0 ) sizenum1--;
04892         while ( sizeden1 > 1 && den1[sizeden1-1] == 0 ) sizeden1--;
04893         work1 = AT.WorkPointer+1; work2 = work1+sizenum1;
04894         for ( i = 0; i < sizenum1; i++ ) work1[i] = num1[i];
04895         for ( i = 0; i < sizeden1; i++ ) work2[i] = den1[i];
04896         num1 = work1; den1 = work2;
04897         AT.WorkPointer = work2 = work2 + sizeden1;
04898         t = subterm + FUNHEAD;
```

```
04899         while ( t < tt ) {
04900             ttt = t + *t; ct = *ttt; *ttt = 0;
04901             if ( t[1] != 0 ) {
04902                 t = PolyNormPoly(BHEAD t+ARGHEAD);
04903             }
04904             else t += ARGHEAD;
04905             while ( *t ) {
04906                 t += *t;
04907                 i = (ABS(t[-1])-1)/2;
04908                 den2 = t - 1 - i;
04909                 num2 = den2 - i;
04910                 sizenum2 = sizeden2 = i;
04911                 while ( sizenum2 > 1 && num2[sizenum2-1] == 0 ) sizenum2--;
04912                 while ( sizeden2 > 1 && den2[sizeden2-1] == 0 ) sizeden2--;
04913                 num3 = AT.WorkPointer;
04914                 if ( GcdLong(BHEAD (UWORD *)num2,sizenum2,(UWORD *)num1,sizenum1,
04915                                     (UWORD *)num3,&sizenum3) ) goto FromGCD;
04916                 sizenum1 = sizenum3;
04917                 for ( i = 0; i < sizenum1; i++ ) num1[i] = num3[i];
04918                 den3 = AT.WorkPointer;
04919                 if ( GcdLong(BHEAD (UWORD *)den2,sizeden2,(UWORD *)den1,sizeden1,
04920                                     (UWORD *)den3,&sizeden3) ) goto FromGCD;
04921                 sizeden1 = sizeden3;
04922                 for ( i = 0; i < sizeden1; i++ ) den1[i] = den3[i];
04923                 if ( sizenum1 == 1 && num1[0] == 1 && sizeden1 == 1 && den1[1] == 1 )
04924                     goto gcdisone;
04925             }
04926             *ttt = ct;
04927             t = ttt;
04928             AT.WorkPointer = work2;
04929         }
04930         AT.WorkPointer = oldworkpointer;
04931 /*
04932         Now copy the GCD to the 'output'
04933 */
04934         if ( sizenum1 > sizeden1 ) {
04935             while ( sizenum1 > sizeden1 ) den1[sizeden1++] = 0;
04936         }
04937         else if ( sizenum1 < sizeden1 ) {
04938             while ( sizenum1 < sizeden1 ) num1[sizenum1++] = 0;
04939         }
04940         t = oldworkpointer;
04941         i = 2*sizenum1+1;
04942         *t++ = i+1;
04943         if ( num1 != t ) { NCOPY(t,num1,sizenum1); }
04944         else t += sizenum1;
04945         if ( den1 != t ) { NCOPY(t,den1,sizeden1); }
04946         else t += sizeden1;
04947         *t++ = i;
04948         *t++ = 0;
04949         AT.WorkPointer = t;
04950         return(oldworkpointer);
04951     }
04952 /*
04953     Now the real stuff with only polynomials.
04954     Pick up the shortest term to start.
04955     We are a bit brutish about this.
04956 */
04957     t = subterm + FUNHEAD;
04958     AT.WorkPointer += AM.MaxTer/sizeof(WORD);
04959     work2 = AT.WorkPointer;
04960 /*
04961     sizearg = subterm[1];
04962 */
04963     i = 0; work3 = 0;
04964     while ( t < tt ) {
04965         i++;
04966         work1 = AT.WorkPointer;
04967         ttt = t + *t; ct = *ttt; *ttt = 0;
04968         t = PolyNormPoly(BHEAD t+ARGHEAD);
04969         if ( *work1 < AT.WorkPointer-work1 ) {
04970 /*
04971             sizearg = AT.WorkPointer-work1;
04972 */
04973             numarg = i;
04974             work3 = work1;
04975         }
04976         *ttt = ct; t = ttt;
04977     }
04978     *AT.WorkPointer++ = 0;
04979 /*
04980     We have properly normalized arguments and the shortest is indicated in work3
04981 */
04982     work1 = work3;
04983     while ( *work2 ) {
04984         if ( work2 != work3 ) {
04985             work1 = PolyGCD2(BHEAD work1,work2);
```

```
04986         }
04987         while ( *work2 ) work2 += *work2;
04988         work2++;
04989     }
04990     work2 = work1;
04991     while ( *work2 ) work2 += *work2;
04992     size = work2 - work1 + 1;
04993     t = oldworkpointer;
04994     NCOPY(t,work1,size);
04995     AT.WorkPointer = t;
04996     return(oldworkpointer);
04997
04998 gcdisone:;
04999     oldworkpointer[0] = 4;
05000     oldworkpointer[1] = 1;
05001     oldworkpointer[2] = 1;
05002     oldworkpointer[3] = 3;
05003     oldworkpointer[4] = 0;
05004     AT.WorkPointer = oldworkpointer+5;
05005     return(oldworkpointer);
05006 FromGCD:
05007     MLOCK(ErrorMessageLock);
05008     MesCall("EvaluateGcd");
05009     MUNLOCK(ErrorMessageLock);
05010     return(0);
05011 }
05012
05013 #endif
05014
05015 /*
05016     #] EvaluateGcd :
05017     #[ TreatPolyRatFun :
05018
05019     if ( AR.PolyFunExp == 1 ) we have to trim the contents of the polyratfun
05020     down to its most divergent term and give it coefficient +1. This is done
05021     by taking the terms with the least power in the variable in the numerator
05022     and in the denominator and then combine them.
05023     Answer is either PolyRatFun(ep^n,1) or PolyRatFun(1,1) or PolyRatFun(1,ep^n)
05024 */
05025
05026 int TreatPolyRatFun(PHEAD WORD *prf)
05027 {
05028     WORD *t, *tstop, *r, *rstop, *m, *mstop;
05029     WORD exp1 = MAXPOWER, exp2 = MAXPOWER;
05030     t = prf+FUNHEAD;
05031     if ( *t < 0 ) {
05032         if ( *t == -SYMBOL && t[1] == AR.PolyFunVar ) {
05033             if ( exp1 > 1 ) exp1 = 1;
05034             t += 2;
05035         }
05036         else {
05037             if ( exp1 > 0 ) exp1 = 0;
05038             NEXTARG(t)
05039         }
05040     }
05041     else {
05042         tstop = t + *t;
05043         t += ARGHEAD;
05044         while ( t < tstop ) {
05045 /*
05046             Now look for the minimum power of AR.PolyFunVar
05047 */
05048             r = t+1;
05049             t += *t;
05050             rstop = t - ABS(t[-1]);
05051             while ( r < rstop ) {
05052                 if ( *r != SYMBOL ) { r += r[1]; continue; }
05053                 m = r;
05054                 mstop = m + m[1];
05055                 m += 2;
05056                 while ( m < mstop ) {
05057                     if ( *m == AR.PolyFunVar ) {
05058                         if ( m[1] < exp1 ) exp1 = m[1];
05059                         break;
05060                     }
05061                     m += 2;
05062                 }
05063                 if ( m == mstop ) {
05064                     if ( exp1 > 0 ) exp1 = 0;
05065                 }
05066                 break;
05067             }
05068             if ( r == rstop ) {
05069                 if ( exp1 > 0 ) exp1 = 0;
05070             }
05071         }
05072         t = tstop;
```

```
05073     }
05074     if ( *t < 0 ) {
05075         if ( *t == -SYMBOL && t[1] == AR.PolyFunVar ) {
05076             if ( exp2 > 1 ) exp2 = 1;
05077         }
05078         else {
05079             if ( exp2 > 0 ) exp2 = 0;
05080         }
05081     }
05082     else {
05083         tstop = t + *t;
05084         t += ARGHEAD;
05085         while ( t < tstop ) {
05086 /*
05087             Now look for the minimum power of AR.PolyFunVar
05088 */
05089             r = t+1;
05090             t += *t;
05091             rstop = t - ABS(t[-1]);
05092             while ( r < rstop ) {
05093                 if ( *r != SYMBOL ) { r += r[1]; continue; }
05094                 m = r;
05095                 mstop = m + m[1];
05096                 m += 2;
05097                 while ( m < mstop ) {
05098                     if ( *m == AR.PolyFunVar ) {
05099                         if ( m[1] < exp2 ) exp2 = m[1];
05100                         break;
05101                     }
05102                     m += 2;
05103                 }
05104                 if ( m == mstop ) {
05105                     if ( exp2 > 0 ) exp2 = 0;
05106                 }
05107                 break;
05108             }
05109             if ( r == rstop ) {
05110                 if ( exp2 > 0 ) exp2 = 0;
05111             }
05112         }
05113     }
05114 /*
05115     Now we can compose the output.
05116     Notice that the output can never be longer than the input provided
05117     we never can have arguments that consist of just a function.
05118 */
05119     exp1 = exp1-exp2;
05120 /*  if ( exp1 > 0 ) exp1 = 0; */
05121     t = prf+FUNHEAD;
05122     if ( exp1 == 0 ) {
05123         *t++ = -SNUMBER; *t++ = 1;
05124         *t++ = -SNUMBER; *t++ = 1;
05125     }
05126     else if ( exp1 > 0 ) {
05127         if ( exp1 == 1 ) {
05128             *t++ = -SYMBOL; *t++ = AR.PolyFunVar;
05129         }
05130         else {
05131             *t++ = 8+ARGHEAD;
05132             *t++ = 0;
05133             FILLARG(t);
05134             *t++ = 8; *t++ = SYMBOL; *t++ = 4; *t++ = AR.PolyFunVar;
05135             *t++ = exp1; *t++ = 1; *t++ = 1; *t++ = 3;
05136         }
05137         *t++ = -SNUMBER; *t++ = 1;
05138     }
05139     else {
05140         *t++ = -SNUMBER; *t++ = 1;
05141         if ( exp1 == -1 ) {
05142             *t++ = -SYMBOL; *t++ = AR.PolyFunVar;
05143         }
05144         else {
05145             *t++ = 8+ARGHEAD;
05146             *t++ = 0;
05147             FILLARG(t);
05148             *t++ = 8; *t++ = SYMBOL; *t++ = 4; *t++ = AR.PolyFunVar;
05149             *t++ = -exp1; *t++ = 1; *t++ = 1; *t++ = 3;
05150         }
05151     }
05152     prf[2] = 0;  /* Clean */
05153     prf[1] = t - prf;
05154     return(0);
05155 }
05156
05157 /*
05158     #] TreatPolyRatFun :
05159     #[ DropCoefficient :
```

```
05160 */
05161
05162 void DropCoefficient(PHEAD WORD *term)
05163 {
05164     GETBIDENTITY
05165     WORD *t = term + *term;
05166     WORD n, na;
05167     n = t[-1]; na = ABS(n);
05168     t -= na;
05169     if ( n == 3 && t[0] == 1 && t[1] == 1 ) return;
05170     *AN.RepPoint = 1;
05171     t[0] = 1; t[1] = 1; t[2] = 3;
05172     *term -= (na-3);
05173 }
05174
05175 /*
05176     #] DropCoefficient :
05177     #[ DropSymbols :
05178 */
05179
05180 void DropSymbols(PHEAD WORD *term)
05181 {
05182     GETBIDENTITY
05183     WORD *tend = term + *term, *t1, *t2, *tstop;
05184     tstop = tend - ABS(tend[-1]);
05185     t1 = term+1;
05186     while ( t1 < tstop ) {
05187         if ( *t1 == SYMBOL ) {
05188             *AN.RepPoint = 1;
05189             t2 = t1+t1[1];
05190             while ( t2 < tend ) *t1++ = *t2++;
05191             *term = t1 - term;
05192             break;
05193         }
05194         t1 += t1[1];
05195     }
05196 }
05197
05198 /*
05199     #] DropSymbols :
05200     #[ SymbolNormalize :
05201 */
05210 int SymbolNormalize(WORD *term)
05211 {
05212     GETIDENTITY
05213     WORD *t, *b, *bb, *tt, *m, *tstop;
05214     // Here we use a stack-allocated array, since things are much smaller
05215     // compared to the full Normalize routine.
05216     WORD buffer[7*NORMSIZE];
05217     int i;
05218     b = buffer;
05219     *b++ = SYMBOL; *b++ = 2;
05220     t = term + *term;
05221     tstop = t - ABS(t[-1]);
05222     t = term + 1;
05223     while ( t < tstop ) {   /* Step 1: collect symbols */
05224       if ( *t == SYMBOL && t < tstop ) {
05225         for ( i = 2; i < t[1]; i += 2 ) {
05226             const WORD sym = t[i];
05227             const WORD pow = t[i+1];
05228             bb = buffer+2;
05229             while ( bb < b ) {
05230                 if ( bb[0] == sym ) {   /* add powers */
05231                     bb[1] += pow;
05232                     if ( bb[1] > MAXPOWER || bb[1] < -MAXPOWER ) {
05233                         MLOCK(ErrorMessageLock);
05234                         MesPrint("Power in SymbolNormalize out of range");
05235                         MUNLOCK(ErrorMessageLock);
05236                         return(-1);
05237                     }
05238                     if ( bb[1] == 0 ) {
05239                         b -= 2;
05240                         while ( bb < b ) {
05241                             bb[0] = bb[2]; bb[1] = bb[3]; bb += 2;
05242                         }
05243                     }
05244                     goto Nexti;
05245                 }
05246                 else if ( bb[0] > sym ) { /* insert it */
05247                     m = b;
05248                     while ( m > bb ) { m[1] = m[-1]; m[0] = m[-2]; m -= 2; }
05249                     b += 2;
05250                     bb[0] = sym;
05251                     bb[1] = pow;
05252                     goto Nexti;
05253                 }
05254                 bb += 2;
```

```
05255             }
05256             if ( bb >= b ) { /* add it to the end */
05257                 *b++ = sym; *b++ = pow;
05258             }
05259 Nexti:;
05260         }
05261       }
05262       else {
05263         MLOCK(ErrorMessageLock);
05264         MesPrint("Illegal term in SymbolNormalize");
05265         MUNLOCK(ErrorMessageLock);
05266         return(-1);
05267       }
05268       t += t[1];
05269     }
05270     buffer[1] = b - buffer;
05271 /*
05272     Veto negative powers
05273 */
05274     if ( AT.LeaveNegative == 0 ) {
05275         b = buffer; bb = b + b[1]; b += 3;
05276         while ( b < bb ) {
05277             if ( *b < 0 ) {
05278                 MLOCK(ErrorMessageLock);
05279                 MesPrint("Negative power in SymbolNormalize");
05280                 MUNLOCK(ErrorMessageLock);
05281                 return(-1);
05282             }
05283             b += 2;
05284         }
05285     }
05286 /*
05287     Now we use the fact that the new term will not be longer than the old one
05288     Actually it should be shorter when there is more than one subterm!
05289     Copy back.
05290 */
05291     i = buffer[1];
05292     b = buffer; tt = term + 1;
05293     if ( i > 2 ) { NCOPY(tt,b,i) }
05294     if ( tt < tstop ) {
05295         i = term[*term-1];
05296         if ( i < 0 ) i = -i;
05297         *term -= (tstop-tt);
05298         NCOPY(tt,tstop,i)
05299     }
05300     return(0);
05301 }
05302
05303 /*
05304     #] SymbolNormalize :
05305     #[ TestFunFlag :
05306
05307     Tests whether a function still has unsubstituted subexpressions
05308     This function has its dirtyflag on!
05309 */
05310
05311 int TestFunFlag(PHEAD WORD *tfun)
05312 {
05313     WORD *t, *tstop, *r, *rstop, *m, *mstop;
05314     if ( functions[*tfun-FUNCTION].spec <= 0 ) return(0);
05315     tstop = tfun + tfun[1];
05316     t = tfun + FUNHEAD;
05317     while ( t < tstop ) {
05318         if ( *t < 0 ) { NEXTARG(t); continue; }
05319         rstop = t + *t;
05320         if ( t[1] == 0 ) { t = rstop; continue; }
05321         r = t + ARGHEAD;
05322         while ( r < rstop ) { /* Here we loop over terms */
05323             m = r+1; mstop = r+*r; mstop -= ABS(mstop[-1]);
05324             while ( m < mstop ) {   /* Loop over the subterms */
05325                 if ( *m == SUBEXPRESSION || *m == EXPRESSION || *m == DOLLAREXPRESSION ) return(1);
05326                 if ( ( *m >= FUNCTION ) && ( ( m[2] & DIRTYFLAG ) == DIRTYFLAG )
05327                     && ( *m != REPLACEMENT ) && TestFunFlag(BHEAD m) ) return(1);
05328                 m += m[1];
05329             }
05330             r += *r;
05331         }
05332         t += *t;
05333     }
05334     return(0);
05335 }
05336
05337 /*
05338     #] TestFunFlag :
05339     #[ BracketNormalize :
05340 */
05341
```

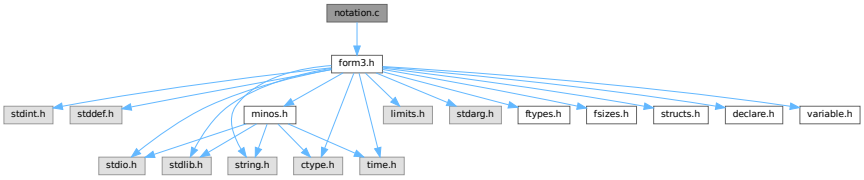
```
05342 int BracketNormalize(PHEAD WORD *term)
05343 {
05344     WORD *stop = term+*term-3, *t, *tt, *tstart, *r;
05345     WORD *oldwork = AT.WorkPointer;
05346     WORD *termout;
05347     WORD i, ii, j;
05348     termout = AT.WorkPointer = term+*term;
05349 /*
05350     First collect all functions and sort them
05351 */
05352     tt = termout+1; t = term+1;
05353     while ( t < stop ) {
05354         if ( *t >= FUNCTION ) { i = t[1]; NCOPY(tt,t,i); }
05355         else t += t[1];
05356     }
05357     if ( tt > termout+1 && tt-termout-1 > termout[2] ) { /* sorting */
05358         r = termout+1; ii = tt-r;
05359         for ( i = 0; i < ii-FUNHEAD; i += FUNHEAD ) {   /* Bubble sort */
05360             for ( j = i+FUNHEAD; j > 0; j -= FUNHEAD ) {
05361                 if ( functions[r[j-FUNHEAD]-FUNCTION].commute
05362                   && functions[r[j]-FUNCTION].commute == 0 ) break;
05363                 if ( r[j-FUNHEAD] > r[j] ) EXCH(r[j-FUNHEAD],r[j])
05364                 else break;
05365             }
05366         }
05367     }
05368
05369     tstart = tt; t = term + 1; *tt++ = DELTA; *tt++ = 2;
05370     while ( t < stop ) {
05371         if ( *t == DELTA ) { i = t[1]-2; t += 2; tstart[1] += i; NCOPY(tt,t,i); }
05372         else t += t[1];
05373     }
05374     if ( tstart[1] > 2 ) {
05375         for ( r = tstart+2; r < tstart+tstart[1]; r += 2 ) {
05376             if ( r[0] > r[1] ) EXCH(r[0],r[1])
05377         }
05378     }
05379     if ( tstart[1] > 4 ) { /* sorting */
05380         r = tstart+2; ii = tstart[1]-2;
05381         for ( i = 0; i < ii-2; i += 2 ) {   /* Bubble sort */
05382             for ( j = i+2; j > 0; j -= 2 ) {
05383                 if ( r[j-2] > r[j] ) {
05384                     EXCH(r[j-2],r[j])
05385                     EXCH(r[j-1],r[j+1])
05386                 }
05387                 else if ( r[j-2] < r[j] ) break;
05388                 else {
05389                     if ( r[j-1] > r[j+1] ) EXCH(r[j-1],r[j+1])
05390                     else break;
05391                 }
05392             }
05393         }
05394         tt = tstart+tstart[1];
05395     }
05396     else if ( tstart[1] == 2 ) { tt = tstart; }
05397     else tt = tstart+4;
05398
05399     tstart = tt; t = term + 1; *tt++ = INDEX; *tt++ = 2;
05400     while ( t < stop ) {
05401         if ( *t == INDEX ) { i = t[1]-2; t += 2; tstart[1] += i; NCOPY(tt,t,i); }
05402         else t += t[1];
05403     }
05404     if ( tstart[1] >= 4 ) { /* sorting */
05405         r = tstart+2; ii = tstart[1]-2;
05406         for ( i = 0; i < ii-1; i += 1 ) {   /* Bubble sort */
05407             for ( j = i+1; j > 0; j -= 1 ) {
05408                 if ( r[j-1] > r[j] ) EXCH(r[j-1],r[j])
05409                 else break;
05410             }
05411         }
05412         tt = tstart+tstart[1];
05413     }
05414     else if ( tstart[1] == 2 ) { tt = tstart; }
05415     else tt = tstart+3;
05416
05417     tstart = tt; t = term + 1; *tt++ = DOTPRODUCT; *tt++ = 2;
05418     while ( t < stop ) {
05419         if ( *t == DOTPRODUCT ) { i = t[1]-2; t += 2; tstart[1] += i; NCOPY(tt,t,i); }
05420         else t += t[1];
05421     }
05422     if ( tstart[1] > 5 ) { /* sorting */
05423         r = tstart+2; ii = tstart[1]-2;
05424         for ( i = 0; i < ii; i += 3 ) {
05425             if ( r[i] > r[i+1] ) EXCH(r[i],r[i+1])
05426         }
05427         for ( i = 0; i < ii-3; i += 3 ) {   /* Bubble sort */
05428             for ( j = i+3; j > 0; j -= 3 ) {
```

```
05429                 if ( r[j-3] < r[j] ) break;
05430                 if ( r[j-3] > r[j] ) {
05431                     EXCH(r[j-3],r[j])
05432                     EXCH(r[j-2],r[j+1])
05433                 }
05434                 else {
05435                     if ( r[j-2] > r[j+1] ) EXCH(r[j-2],r[j+1])
05436                     else break;
05437                 }
05438             }
05439         }
05440         tt = tstart+tstart[1];
05441     }
05442     else if ( tstart[1] == 2 ) { tt = tstart; }
05443     else {
05444         if ( tstart[2] > tstart[3] ) EXCH(tstart[2],tstart[3])
05445         tt = tstart+5;
05446     }
05447
05448     tstart = tt; t = term + 1; *tt++ = SYMBOL; *tt++ = 2;
05449     while ( t < stop ) {
05450         if ( *t == SYMBOL ) { i = t[1]-2; t += 2; tstart[1] += i; NCOPY(tt,t,i); }
05451         else t += t[1];
05452     }
05453     if ( tstart[1] > 4 ) { /* sorting */
05454         r = tstart+2; ii = tstart[1]-2;
05455         for ( i = 0; i < ii-2; i += 2 ) {   /* Bubble sort */
05456             for ( j = i+2; j > 0; j -= 2 ) {
05457                 if ( r[j-2] > r[j] ) EXCH(r[j-2],r[j])
05458                 else break;
05459             }
05460         }
05461         tt = tstart+tstart[1];
05462     }
05463     else if ( tstart[1] == 2 ) { tt = tstart; }
05464     else tt = tstart+4;
05465
05466     tstart = tt; t = term + 1; *tt++ = SETSET; *tt++ = 2;
05467     while ( t < stop ) {
05468         if ( *t == SETSET ) { i = t[1]-2; t += 2; tstart[1] += i; NCOPY(tt,t,i); }
05469         else t += t[1];
05470     }
05471     if ( tstart[1] > 4 ) { /* sorting */
05472         r = tstart+2; ii = tstart[1]-2;
05473         for ( i = 0; i < ii-2; i += 2 ) {   /* Bubble sort */
05474             for ( j = i+2; j > 0; j -= 2 ) {
05475                 if ( r[j-2] > r[j] ) {
05476                     EXCH(r[j-2],r[j])
05477                     EXCH(r[j-1],r[j+1])
05478                 }
05479                 else break;
05480             }
05481         }
05482         tt = tstart+tstart[1];
05483     }
05484     else if ( tstart[1] == 2 ) { tt = tstart; }
05485     else tt = tstart+4;
05486     *tt++ = 1; *tt++ = 1; *tt++ = 3;
05487     t = term; i = *termout = tt - termout; tt = termout;
05488     NCOPY(t,tt,i);
05489     AT.WorkPointer = oldwork;
05490     return(0);
05491 }
05492
05493 /*
05494     #] BracketNormalize :
05495 */
```

4.91 notation.c File Reference

```
#include "form3.h"
```

Include dependency graph for notation.c:



Functions

- int NormPolyTerm (PHEAD WORD ∗term)
- int ConvertToPoly (PHEAD WORD ∗term, WORD ∗outterm, WORD ∗comlist, WORD par)
- int LocalConvertToPoly (PHEAD WORD ∗term, WORD ∗outterm, WORD startebuf, WORD par)
- int ConvertFromPoly (PHEAD WORD ∗term, WORD ∗outterm, WORD from, WORD to, WORD offset, WORD par)
- int FindSubterm (WORD ∗subterm)
- int FindLocalSubterm (PHEAD WORD ∗subterm, WORD startebuf)
- void PrintSubtermList (int from, int to)
- void PrintExtraSymbol (int num, WORD ∗terms, int par)
- int FindSubexpression (WORD ∗subexpr)
- int ExtraSymFun (PHEAD WORD ∗term, WORD level)
- int PruneExtraSymbols (WORD downto)

4.91.1 Detailed Description

Contains the functions that deal with the rewriting and manipulation of expressions/terms in polynomial representation.

Definition in file notation.c.

4.91.2 Function Documentation

4.91.2.1 NormPolyTerm()

```
int NormPolyTerm (
            PHEAD WORD * term )
```

Definition at line 53 of file notation.c.

4.91.2.2 ConvertToPoly()

```
int ConvertToPoly (
            PHEAD WORD * term,
            WORD * outterm,
            WORD * comlist,
            WORD par )
```

Definition at line 309 of file notation.c.

4.91.2.3 LocalConvertToPoly()

```
int LocalConvertToPoly (
            PHEAD WORD * term,
            WORD * outterm,
            WORD startebuf,
            WORD par )
```

Converts a generic term to polynomial notation in which there are only symbols and brackets. During conversion there will be only symbols. Brackets are stripped. Objects that need 'translation' are put inside a special compiler buffer and represented by a symbol. The numbering of the extra symbols is down from the maximum. In principle there can be a problem when running into the already assigned ones. This uses the FindTree for searching in the global tree and then looks further in the AT.ebufnum. This allows fully parallel processing. Hence we need no locks. Cannot be used in the same module as ConvertToPoly.

Definition at line 514 of file notation.c.

Referenced by poly_factorize_expression().

4.91.2.4 ConvertFromPoly()

```
int ConvertFromPoly (
            PHEAD WORD * term,
            WORD * outterm,
            WORD from,
            WORD to,
            WORD offset,
            WORD par )
```

Definition at line 664 of file notation.c.

4.91.2.5 FindSubterm()

```
int FindSubterm (
            WORD * subterm )
```

Definition at line 777 of file notation.c.

4.91.2.6 FindLocalSubterm()

```
int FindLocalSubterm (
            PHEAD WORD * subterm,
            WORD startebuf )
```

Definition at line 847 of file notation.c.

4.91.2.7 PrintSubtermList()

```
void PrintSubtermList (
            int from,
            int to )
```

Definition at line 901 of file notation.c.

4.91.2.8 PrintExtraSymbol()

```
void PrintExtraSymbol (
            int num,
            WORD * terms,
            int par )
```

Definition at line 1013 of file notation.c.

4.91.2.9 FindSubexpression()

```
int FindSubexpression (
            WORD * subexpr )
```

Definition at line 1100 of file notation.c.

4.91.2.10 ExtraSymFun()

```
int ExtraSymFun (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 1153 of file notation.c.

4.91.2.11 PruneExtraSymbols()

```
int PruneExtraSymbols (
            WORD downto )
```

Definition at line 1221 of file notation.c.

4.92 notation.c

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032 /*
00033     #[ Includes :
00034 */
00035
00036 #include "form3.h"
00037
00038 /*
00039     #] Includes :
00040         #[ NormPolyTerm :
00041
00042         Brings a term to normal form.
00043
00044         This routine knows objects of the following types:
00045             SYMBOL
00046             HAAKJE
00047             SNUMBER
00048             LNUMBER
00049         The SNUMBER and LNUMBER are worked into the coefficient.
00050         One of the essences here is that everything can be done in place.
00051 */
00052
00053 int NormPolyTerm(PHEAD WORD *term)
00054 {
00055     WORD *tcoef, ncoef, *tstop, *tfill, *t, *tt;
00056     int equal, i;
00057     WORD *r1, *r2, *r3, *r4, *r5, *rfirst, rv;
00058     WORD *lnum, nnum;       /* Scratch, originally for factorials */
00059 /*
00060     One: find the coefficient
00061 */
00062     tcoef = term+*term;
00063     ncoef = tcoef[-1];
00064     tstop = tcoef - ABS(tcoef[-1]);
00065     tfill = t = term + 1;
00066     rfirst = 0;
00067     if ( t >= tstop ) { return(*term); }
00068     while ( t < tstop ) {
00069         switch ( *t ) {
00070             case SYMBOL:
00071                 if ( rfirst == 0 ) {
00072 /*
00073                     Here we only need to sort
00074                     1: assume no equals. Bubble.
00075 */
00076                     rfirst = t;
00077                     r2 = rfirst+4; tt = r3 = t + t[1]; equal = 0;
00078                     while ( r2 < r3 ) {
00079                         r1 = r2 - 2;
00080                         if ( *r2 > *r1 ) { r2 += 2; continue; }
00081                         if ( *r2 == *r1 ) { r2 += 2; equal = 1; continue; }
00082                         rv = *r1; *r1 = *r2; *r2 = rv;
00083                         r1 -= 2; r2 -= 2; r4 = r2 + 2;
00084                         while ( r1 > t ) {
00085                             if ( *r2 >= *r1 ) { r2 = r4; break; }
00086                             rv = *r1; *r1 = *r2; *r2 = rv;
```

```
00087                               r1 -= 2; r2 -= 2;
00088                           }
00089                       }
00090 /*
00091                       2: hunt down the equal objects
00092                          postpone eliminating zero powers.
00093 */
00094                       if ( equal ) {
00095                           r1 = t+2; r2 = r1+2;
00096                           while ( r2 < r3 ) {
00097                               if ( *r1 == *r2 ) {
00098                                   r1[1] += r2[1];
00099                                   r4 = r2+2;
00100                                  while ( r4 < r3 ) *r2++ = *r4++;
00101                                  t[1] -= 2;
00102                                  r2 = r1 + 2; r3 -= 2;
00103                              }
00104                          }
00105                      }
00106                  }
00107                  else {
00108 /*
00109                      Here we only need to insert
00110 */
00111                      r1 = t + 2; tt = r3 = t + t[1];
00112                      while ( r1 < r3 ) {
00113                          r2 = rfirst+2; r4 = rfirst + rfirst[1];
00114                          while ( r2 < r4 ) {
00115                              if ( *r1 == *r2 ) {
00116                                  r2[1] += r1[1];
00117                                  break;
00118                              }
00119                              else if ( *r2 > *r1 ) {
00120                                  r5 = r4;
00121                                  while ( r5 > r2 ) { r5[1] = r5[-1]; r5[0] = r5[-2]; r5 -= 2; }
00122                                  rfirst[1] += 2;
00123                                  *r2 = *r1; r2[1] = r1[1];
00124                                  break;
00125                              }
00126                              r2 += 2;
00127                          }
00128                          if ( r2 == r4 ) {
00129                              rfirst[1] += 2;
00130                              *r2++ = *r1++; *r2++ = *r1++;
00131                          }
00132                          else r1 += 2;
00133                      }
00134                  }
00135                  t = tt;
00136                  break;
00137              case HAAKJE:    /* Here we skip brackets */
00138                  t += t[1];
00139                  break;
00140              case SNUMBER:
00141                  if ( t[2] < 0 ) {
00142                      t[2] = -t[2];
00143                      if ( t[3] & 1 ) ncoef = -ncoef;
00144                  }
00145                  else if ( t[2] == 0 ) {
00146                      if ( t[3] < 0 ) goto NormInf;
00147                      goto NormZero;
00148                  }
00149                  lnum = TermMalloc("lnum");
00150                  lnum[0] = t[2];
00151                  nnum = 1;
00152                  if ( t[3] && RaisPow(BHEAD (UWORD *)lnum,&nnum,(UWORD)(ABS(t[3]))) ) goto FromNorm;
00153                  ncoef = REDLENG(ncoef);
00154                  if ( t[3] < 0 ) {
00155                      if ( Divvy(BHEAD (UWORD *)tstop,&ncoef,(UWORD *)lnum,nnum) )
00156                          goto FromNorm;
00157                  }
00158                  else if ( t[3] > 0 ) {
00159                      if ( Mully(BHEAD (UWORD *)tstop,&ncoef,(UWORD *)lnum,nnum) )
00160                          goto FromNorm;
00161                  }
00162                  ncoef = INCLENG(ncoef);
00163                  t += t[1];
00164                  TermFree(lnum,"lnum");
00165                  break;
00166              case LNUMBER:
00167                  ncoef = REDLENG(ncoef);
00168                  if ( Mully(BHEAD (UWORD *)tstop,&ncoef,(UWORD *)(t+3),t[2]) ) goto FromNorm;
00169                  ncoef = INCLENG(ncoef);
00170                  t += t[1];
00171                  break;
00172              default:
00173 /* INTERNAL_ERROR_EXCL_START */
```

```
00174                 MLOCK(ErrorMessageLock);
00175                 MesPrint("!>Illegal code in NormPolyTerm");
00176                 MUNLOCK(ErrorMessageLock);
00177                 Terminate(-1);
00178                 break;
00179 /* INTERNAL_ERROR_EXCL_STOP */
00180         }
00181     }
00182 /*
00183     Now we try to eliminate objects to the power zero.
00184 */
00185     if ( rfirst ) {
00186         r2 = rfirst+2;
00187         r3 = rfirst + rfirst[1];
00188         while ( r2 < r3 ) {
00189             if ( r2[1] == 0 ) {
00190                 r1 = r2 + 2;
00191                 while ( r1 < r3 ) { r1[-2] = r1[0]; r1[-1] = r1[1]; r1 += 2; }
00192                 r3 -= 2;
00193                 rfirst[1] -= 2;
00194             }
00195             else { r2 += 2; }
00196         }
00197         if ( rfirst[1] < 4 ) rfirst = 0;
00198     }
00199 /*
00200     Finally we put the term together
00201 */
00202     if ( rfirst ) {
00203         i = rfirst[1];
00204         NCOPY(tfill,rfirst,i)
00205     }
00206     i = ABS(ncoef)-1;
00207     NCOPY(tfill,tstop,i)
00208     *tfill++ = ncoef;
00209     *term = tfill - term;
00210     return(*term);
00211 NormZero:
00212     *term = 0;
00213     return(0);
00214 NormInf:
00215     MLOCK(ErrorMessageLock);
00216     MesPrint("0^0 in NormPolyTerm");
00217     MUNLOCK(ErrorMessageLock);
00218     Terminate(-1);
00219     return(-1);
00220 FromNorm:
00221     MLOCK(ErrorMessageLock);
00222     MesCall("NormPolyTerm");
00223     MUNLOCK(ErrorMessageLock);
00224     Terminate(-1);
00225     return(-1);
00226 }
00227
00228 /*
00229         #] NormPolyTerm :
00230         #[ ComparePoly :
00231 */
00253 #ifdef WITHCOMPAREPOLY
00254
00255 WORD ComparePoly(WORD *term1, WORD *term2, WORD level)
00256 {
00257     WORD *t1, *t2, *t3, *t4, *tstop1, *tstop2;
00258     tstop1 = term1 + *term1;
00259     tstop1 -= ABS(tstop1[-1]);
00260     tstop2 = term2 + *term2;
00261     tstop2 -= ABS(tstop2[-1]);
00262     t1 = term1+1;
00263     t2 = term2+1;
00264     while ( t1 < tstop1 && t2 < tstop2 ) {
00265         if ( *t1 == *t2 ) {
00266             if ( *t1 == HAAKJE ) {
00267                 if ( t1[2] != t2[2] ) return(t2[2]-t1[2]);
00268                 t1 += t1[1]; t2 += t2[1];
00269             }
00270             else {  /* must be type SYMBOL */
00271                 t3 = t1 + t1[1]; t4 = t2 + t2[1];
00272                 t1 += 2; t2 += 2;
00273                 while ( t1 < t3 && t2 < t4 ) {
00274                     if ( *t1 != *t2 ) return(*t2-*t1);
00275                     if ( t1[1] != t2[1] ) return(t2[1]-t1[1]);
00276                     t1 += 2; t2 += 2;
00277                 }
00278                 if ( t1 < t3 ) return(-1);
00279                 if ( t2 < t4 ) return(1);
00280             }
00281         }
```

```
00282         else return(*t2-*t1);
00283     }
00284     if ( t1 < tstop1 ) return(-1);
00285     if ( t2 < tstop2 ) return(1);
00286     return(0);
00287 }
00288
00289 #endif
00290
00291 /*
00292         #] ComparePoly :
00293         #[ ConvertToPoly :
00294 */
00307 static int FirstWarnConvertToPoly = 1;
00308
00309 int ConvertToPoly(PHEAD WORD *term, WORD *outterm, WORD *comlist, WORD par)
00310 {
00311     WORD *tout, *tstop, ncoef, *t, *r, *tt, *ttwo = 0;
00312     int i, action = 0;
00313     tt = term + *term;
00314     ncoef = ABS(tt[-1]);
00315     tstop = tt - ncoef;
00316     tout = outterm+1;
00317     t = term + 1;
00318     if ( comlist[2] == DOALL ) {
00319       while ( t < tstop ) {
00320         if ( *t == SYMBOL ) {
00321             r = t+2;
00322             t += t[1];
00323             while ( r < t ) {
00324                 if ( r[1] > 0 ) {
00325                     *tout++ = SYMBOL;
00326                     *tout++ = 4;
00327                     *tout++ = r[0];
00328                     *tout++ = r[1];
00329                 }
00330                 else {
00331                     tout[1] = SYMBOL;
00332                     tout[2] = 4;
00333                     tout[3] = r[0];
00334                     tout[4] = -1;
00335                     i = FindSubterm(tout+1);
00336                     *tout++ = SYMBOL;
00337                     *tout++ = 4;
00338                     *tout++ = MAXVARIABLES-i;
00339                     *tout++ = -r[1];
00340                     action = 1;
00341                 }
00342                 r += 2;
00343             }
00344         }
00345         else if ( *t == DOTPRODUCT ) {
00346             r = t + 2;
00347             t += t[1];
00348             while ( r < t ) {
00349                 tout[1] = DOTPRODUCT;
00350                 tout[2] = 5;
00351                 tout[3] = r[0];
00352                 tout[4] = r[1];
00353                 if ( r[2] < 0 ) {
00354                     tout[5] = -1;
00355                 }
00356                 else {
00357                     tout[5] = 1;
00358                 }
00359                 i = FindSubterm(tout+1);
00360                 *tout++ = SYMBOL;
00361                 *tout++ = 4;
00362                 *tout++ = MAXVARIABLES-i;
00363                 *tout++ = ABS(r[2]);
00364                 r += 3;
00365                 action = 1;
00366             }
00367         }
00368         else if ( *t == VECTOR ) {
00369             r = t + 2;
00370             t += t[1];
00371             while ( r < t ) {
00372                 tout[1] = VECTOR;
00373                 tout[2] = 4;
00374                 tout[3] = r[0];
00375                 tout[4] = r[1];
00376                 i = FindSubterm(tout+1);
00377                 *tout++ = SYMBOL;
00378                 *tout++ = 4;
00379                 *tout++ = MAXVARIABLES-i;
00380                 *tout++ = 1;
```

```
00381                 r += 2;
00382                 action = 1;
00383             }
00384         }
00385         else if ( *t == INDEX ) {
00386             r = t + 2;
00387             t += t[1];
00388             while ( r < t ) {
00389                 tout[1] = INDEX;
00390                 tout[2] = 3;
00391                 tout[3] = r[0];
00392                 i = FindSubterm(tout+1);
00393                 *tout++ = SYMBOL;
00394                 *tout++ = 4;
00395                 *tout++ = MAXVARIABLES-i;
00396                 *tout++ = 1;
00397                 r++;
00398                 action = 1;
00399             }
00400         }
00401         else if ( *t == HAAKJE) {
00402             if ( par ) {
00403                 tout[0] = 1; tout[1] = 1; tout[2] = 3;
00404                 *outterm = (tout+3)-outterm;
00405                 if ( NormPolyTerm(BHEAD outterm) < 0 ) return(-1);
00406                 tout = outterm + *outterm;
00407                 tout -= 3;
00408                 i = t[1]; NCOPY(tout,t,i);
00409                 ttwo = tout-1;
00410             }
00411             else { t += t[1]; }
00412         }
00413         else if ( *t >= FUNCTION ) {
00414             i = FindSubterm(t);
00415             t += t[1];
00416             *tout++ = SYMBOL;
00417             *tout++ = 4;
00418             *tout++ = MAXVARIABLES-i;
00419             *tout++ = 1;
00420             action = 1;
00421         }
00422         else {
00423             if ( FirstWarnConvertToPoly ) {
00424                 MLOCK(ErrorMessageLock);
00425                 MesPrint("Illegal object in conversion to polynomial notation");
00426                 MUNLOCK(ErrorMessageLock);
00427                 FirstWarnConvertToPoly = 0;
00428             }
00429             return(-1);
00430         }
00431       }
00432       NCOPY(tout,tstop,ncoef)
00433       if ( ttwo ) {
00434         WORD hh = *ttwo;
00435         *ttwo = tout-ttwo;
00436         if ( ( i = NormPolyTerm(BHEAD ttwo) ) >= 0 ) i = action;
00437         tout = ttwo + *ttwo;
00438         *ttwo = hh;
00439         *outterm = tout - outterm;
00440       }
00441       else {
00442         *outterm = tout-outterm;
00443         if ( ( i = NormPolyTerm(BHEAD outterm) ) >= 0 ) i = action;
00444       }
00445     }
00446     else if ( comlist[2] == ONLYFUNCTIONS ) {
00447       while ( t < tstop ) {
00448         if ( *t >= FUNCTION ) {
00449             if ( comlist[1] == 3 ) {
00450                 i = FindSubterm(t);
00451                 t += t[1];
00452                 *tout++ = SYMBOL;
00453                 *tout++ = 4;
00454                 *tout++ = MAXVARIABLES-i;
00455                 *tout++ = 1;
00456                 action = 1;
00457             }
00458             else {
00459                 for ( i = 3; i < comlist[1]; i++ ) {
00460                     if ( *t == comlist[i] ) break;
00461                 }
00462                 if ( i < comlist[1] ) {
00463                     i = FindSubterm(t);
00464                     t += t[1];
00465                     *tout++ = SYMBOL;
00466                     *tout++ = 4;
00467                     *tout++ = MAXVARIABLES-i;
```

```
00468                       *tout++ = 1;
00469                       action = 1;
00470                   }
00471                   else {
00472                       i = t[1]; NCOPY(tout,t,i);
00473                   }
00474               }
00475           }
00476           else {
00477               i = t[1]; NCOPY(tout,t,i);
00478           }
00479       }
00480       NCOPY(tout,tstop,ncoef)
00481       *outterm = tout-outterm;
00482       Normalize(BHEAD outterm);
00483       i = action;
00484     }
00485     else {
00486 /* INTERNAL_ERROR_EXCL_START */
00487         MLOCK(ErrorMessageLock);
00488         MesPrint("!>Illegal internal code in conversion to polynomial notation");
00489         MUNLOCK(ErrorMessageLock);
00490         i = -1;
00491 /* INTERNAL_ERROR_EXCL_STOP */
00492     }
00493     return(i);
00494 }
00495
00496 /*
00497         #] ConvertToPoly :
00498         #[ LocalConvertToPoly :
00499 */
00514 int LocalConvertToPoly(PHEAD WORD *term, WORD *outterm, WORD startebuf, WORD par)
00515 {
00516     WORD *tout, *tstop, ncoef, *t, *r, *tt, *ttwo = 0;
00517     int i, action = 0;
00518     tt = term + *term;
00519     ncoef = ABS(tt[-1]);
00520     tstop = tt - ncoef;
00521     tout = outterm+1;
00522     t = term + 1;
00523     while ( t < tstop ) {
00524         if ( *t == SYMBOL ) {
00525             r = t+2;
00526             t += t[1];
00527             while ( r < t ) {
00528                 if ( r[1] > 0 ) {
00529                     *tout++ = SYMBOL;
00530                     *tout++ = 4;
00531                     *tout++ = r[0];
00532                     *tout++ = r[1];
00533                 }
00534                 else {
00535                     tout[1] = SYMBOL;
00536                     tout[2] = 4;
00537                     tout[3] = r[0];
00538                     tout[4] = -1;
00539                     i = FindLocalSubterm(BHEAD tout+1,startebuf);
00540                     *tout++ = SYMBOL;
00541                     *tout++ = 4;
00542                     *tout++ = MAXVARIABLES-i;
00543                     *tout++ = -r[1];
00544                     action = 1;
00545                 }
00546                 r += 2;
00547             }
00548         }
00549         else if ( *t == DOTPRODUCT ) {
00550             r = t + 2;
00551             t += t[1];
00552             while ( r < t ) {
00553                 tout[1] = DOTPRODUCT;
00554                 tout[2] = 5;
00555                 tout[3] = r[0];
00556                 tout[4] = r[1];
00557                 if ( r[2] < 0 ) {
00558                     tout[5] = -1;
00559                 }
00560                 else {
00561                     tout[5] = 1;
00562                 }
00563                 i = FindLocalSubterm(BHEAD tout+1,startebuf);
00564                 *tout++ = SYMBOL;
00565                 *tout++ = 4;
00566                 *tout++ = MAXVARIABLES-i;
00567                 *tout++ = ABS(r[2]);
00568                 r += 3;
```

```
00569                   action = 1;
00570               }
00571           }
00572           else if ( *t == VECTOR ) {
00573               r = t + 2;
00574               t += t[1];
00575               while ( r < t ) {
00576                   tout[1] = VECTOR;
00577                   tout[2] = 4;
00578                   tout[3] = r[0];
00579                   tout[4] = r[1];
00580                   i = FindLocalSubterm(BHEAD tout+1,startebuf);
00581                   *tout++ = SYMBOL;
00582                   *tout++ = 4;
00583                   *tout++ = MAXVARIABLES-i;
00584                   *tout++ = 1;
00585                   r += 2;
00586                   action = 1;
00587               }
00588           }
00589           else if ( *t == INDEX ) {
00590               r = t + 2;
00591               t += t[1];
00592               while ( r < t ) {
00593                   tout[1] = INDEX;
00594                   tout[2] = 3;
00595                   tout[3] = r[0];
00596                   i = FindLocalSubterm(BHEAD tout+1,startebuf);
00597                   *tout++ = SYMBOL;
00598                   *tout++ = 4;
00599                   *tout++ = MAXVARIABLES-i;
00600                   *tout++ = 1;
00601                   r++;
00602                   action = 1;
00603               }
00604           }
00605           else if ( *t == HAAKJE) {
00606               if ( par ) {
00607                   tout[0] = 1; tout[1] = 1; tout[2] = 3;
00608                   *outterm = (tout+3)-outterm;
00609                   if ( NormPolyTerm(BHEAD outterm) < 0 ) return(-1);
00610                   tout = outterm + *outterm;
00611                   tout -= 3;
00612                   i = t[1]; NCOPY(tout,t,i);
00613                   ttwo = tout-1;
00614               }
00615               else { t += t[1]; }
00616           }
00617           else if ( *t >= FUNCTION ) {
00618               i = FindLocalSubterm(BHEAD t,startebuf);
00619               t += t[1];
00620               *tout++ = SYMBOL;
00621               *tout++ = 4;
00622               *tout++ = MAXVARIABLES-i;
00623               *tout++ = 1;
00624               action = 1;
00625           }
00626           else {
00627               if ( FirstWarnConvertToPoly ) {
00628                   MLOCK(ErrorMessageLock);
00629                   MesPrint("Illegal object in conversion to polynomial notation");
00630                   MUNLOCK(ErrorMessageLock);
00631                   FirstWarnConvertToPoly = 0;
00632               }
00633               return(-1);
00634           }
00635       }
00636       NCOPY(tout,tstop,ncoef)
00637       if ( ttwo ) {
00638           WORD hh = *ttwo;
00639           *ttwo = tout-ttwo;
00640           if ( ( i = NormPolyTerm(BHEAD ttwo) ) >= 0 ) i = action;
00641           tout = ttwo + *ttwo;
00642           *ttwo = hh;
00643           *outterm = tout - outterm;
00644       }
00645       else {
00646           *outterm = tout-outterm;
00647           if ( ( i = NormPolyTerm(BHEAD outterm) ) >= 0 ) i = action;
00648       }
00649       return(i);
00650   }
00651
00652   /*
00653           #] LocalConvertToPoly :
00654           #[ ConvertFromPoly :
00655
```

```
00656         Converts a generic term from polynomial notation to the original
00657         in which the extra symbols have been replaced by their values.
00658         The output is in outterm.
00659         We only deal with the extra symbols in the range from < i <= to
00660         The output has to be sent to TestSub because it may contain
00661         subexpressions when extra symbols have been replaced.
00662 */
00663
00664 int ConvertFromPoly(PHEAD WORD *term, WORD *outterm, WORD from, WORD to, WORD offset, WORD par)
00665 {
00666     WORD *tout, *tstop, *tstop1, ncoef, *t, *r, *tt;
00667     int i;
00668 /*  first = 1; */
00669     tt = term + *term;
00670     tout = outterm+1;
00671     ncoef = ABS(tt[-1]);
00672     tstop = tt - ncoef;
00673 /*
00674     r = t = term + 1;
00675     while ( t < tstop ) {
00676         if ( *t == SYMBOL ) {
00677             tstop1 = t + t[1];
00678             tt = t + 2;
00679             while ( tt < tstop1 ) {
00680                 if ( ( *tt < MAXVARIABLES - to )
00681                   || ( *tt >= MAXVARIABLES - from ) ) {
00682                     tt += 2;
00683                 }
00684                 else break;
00685             }
00686             if ( tt >= tstop1 ) { t = tstop1; continue; }
00687             while ( r < t ) *tout++ = *r++;
00688             t += 2;
00689             first = 0;
00690             while ( t < tstop1 ) {
00691                 if ( ( *t < MAXVARIABLES - to )
00692                   || ( *t >= MAXVARIABLES - from ) ) {
00693                     *tout++ = SYMBOL;
00694                     *tout++ = 4;
00695                     *tout++ = *t++;
00696                     *tout++ = *t++;
00697                 }
00698                 else {
00699                     *tout++ = SUBEXPRESSION;
00700                     *tout++ = SUBEXPSIZE;
00701                     *tout++ = MAXVARIABLES - *t++ + offset;
00702                     *tout++ = *t++;
00703                     if ( par ) *tout++ = AT.ebufnum;
00704                     else       *tout++ = AM.sbufnum;
00705                     FILLSUB(tout)
00706                 }
00707             }
00708             r = t;
00709         }
00710         else {
00711             t += t[1];
00712         }
00713     }
00714     if ( first ) {
00715         i = *term; t = term;
00716         NCOPY(outterm,t,i);
00717         return(*term);
00718     }
00719     while ( r < t ) *tout++ = *r++;
00720     NCOPY(tout,tstop,ncoef)
00721     *outterm = tout-outterm;
00722 */
00723     t = term + 1;
00724     while ( t < tstop ) {
00725         if ( *t == SYMBOL ) {
00726             tstop1 = t + t[1];
00727             tt = t + 2;
00728             while ( tt < tstop1 ) {
00729                 if ( ( *tt < MAXVARIABLES - to )
00730                   || ( *tt >= MAXVARIABLES - from ) ) {
00731                     tt += 2;
00732                 }
00733                 else {
00734                     *tout++ = SUBEXPRESSION;
00735                     *tout++ = SUBEXPSIZE;
00736                     *tout++ = MAXVARIABLES - *tt++ + offset;
00737                     *tout++ = *tt++;
00738                     if ( par ) *tout++ = AT.ebufnum;
00739                     else       *tout++ = AM.sbufnum;
00740                     FILLSUB(tout)
00741                 }
00742             }
```

```
00743             r = tout; t += 2;
00744             *tout++ = SYMBOL; *tout++ = 0;
00745             while ( t < tstop1 ) {
00746                 if ( ( *t < MAXVARIABLES - to )
00747                   || ( *t >= MAXVARIABLES - from ) ) {
00748                     *tout++ = *t++;
00749                     *tout++ = *t++;
00750                 }
00751                 else { t += 2; }
00752             }
00753             r[1] = tout - r;
00754             if ( r[1] <= 2 ) tout = r;
00755         }
00756         else {
00757             i = t[1]; NCOPY(tout,t,i)
00758         }
00759     }
00760     NCOPY(tout,tstop,ncoef)
00761     *outterm = tout-outterm;
00762     return(*outterm);
00763 }
00764
00765 /*
00766         #] ConvertFromPoly :
00767         #[ FindSubterm :
00768
00769         In this routine we look up a variable.
00770         If we don't find it we will enter it in the subterm compiler buffer
00771         Searching is by tree structure.
00772         Adding changes the tree.
00773
00774         Notice that in TFORM we should be in sequential mode.
00775 */
00776
00777 int FindSubterm(WORD *subterm)
00778 {
00779     WORD old[5], *ss, *term;
00780     int number;
00781     CBUF *C = cbuf + AM.sbufnum;
00782     LONG oldCpointer;
00783     term = subterm-1;
00784     ss = subterm+subterm[1];
00785 /*
00786         Convert to proper term
00787 */
00788     old[0] = *term; old[1] = ss[0]; old[2] = ss[1]; old[3] = ss[2]; old[4] = ss[3];
00789     ss[0] = 1; ss[1] = 1; ss[2] = 3; ss[3] = 0; *term = subterm[1]+4;
00790 /*
00791         We may have to add the term to the compiler
00792         buffer and then to the tree. This cannot be done in parallel and
00793         hence we have to set a lock.
00794 */
00795     LOCK(AM.sbuflock);
00796
00797     oldCpointer = C->Pointer-C->Buffer; /* Offset of course !!!!!*/
00798     AddRHS(AM.sbufnum,1);
00799     AddNtoC(AM.sbufnum,*term,term,8);
00800     AddToCB(C,0)
00801 /*
00802         See whether we have this one already. If not, insert it in the tree.
00803 */
00804     number = InsTree(AM.sbufnum,C->numrhs);
00805 /*
00806         Restore old values and return what is needed.
00807 */
00808     if ( number < (C->numrhs) ) {   /* It existed already */
00809         C->Pointer = oldCpointer + C->Buffer;
00810         C->numrhs--;
00811     }
00812     else {
00813         GETIDENTITY
00814         WORD dim = DimensionSubterm(subterm);
00815
00816         if ( dim == -MAXPOSITIVE ) {    /* Give error message but continue */
00817             WORD *old = AN.currentTerm;
00818             AN.currentTerm = term;
00819             MLOCK(ErrorMessageLock);
00820             MesPrint("Dimension out of range in %t");
00821             MUNLOCK(ErrorMessageLock);
00822             AN.currentTerm = old;
00823         }
00824 /*
00825         Store the dimension
00826 */
00827         C->dimension[number] = dim;
00828     }
00829     UNLOCK(AM.sbuflock);
```

```
00830
00831     *term = old[0]; ss[0] = old[1]; ss[1] = old[2]; ss[2] = old[3]; ss[3] = old[4];
00832     return(number);
00833 }
00834
00835 /*
00836         #] FindSubterm :
00837         #[ FindLocalSubterm :
00838
00839         In this routine we look up a variable.
00840         If we don't find it we will enter it in the subterm compiler buffer
00841         Searching is by tree structure.
00842         Adding changes the tree.
00843
00844         Notice that in TFORM we should be in sequential mode.
00845 */
00846
00847 int FindLocalSubterm(PHEAD WORD *subterm, WORD startebuf)
00848 {
00849     WORD old[5], *ss, *term, i, j, *t1, *t2;
00850     int number;
00851     CBUF *C = cbuf + AT.ebufnum;
00852     term = subterm-1;
00853     ss = subterm+subterm[1];
00854 /*
00855         Convert to proper term
00856 */
00857     old[0] = *term; old[1] = ss[0]; old[2] = ss[1]; old[3] = ss[2]; old[4] = ss[3];
00858     ss[0] = 1; ss[1] = 1; ss[2] = 3; ss[3] = 0; *term = subterm[1]+4;
00859 /*
00860         First see whether we have this one already in the global buffer.
00861 */
00862     number = FindTree(AM.sbufnum,term);
00863     if ( number > 0 ) goto wearehappy;
00864 /*
00865     Now look whether it is in the ebufnum between startebuf and numrhs
00866     Note however that we need an offset of (numxsymbol-startebuf)
00867 */
00868     for ( i = startebuf+1; i <= C->numrhs; i++ ) {
00869         t1 = C->rhs[i]; t2 = term;
00870         if ( *t1 == *t2 ) {
00871             j = *t1;
00872             while ( *t1 == *t2 && j > 0 ) { t1++; t2++; j--; }
00873             if ( j <= 0 ) {
00874                 number = i-startebuf+numxsymbol;
00875                 goto wearehappy;
00876             }
00877         }
00878     }
00879 /*
00880     Now we have to add it to cbuf[AT.ebufnum]
00881 */
00882     AddRHS(AT.ebufnum,1);
00883     AddNtoC(AT.ebufnum,*term,term,9);
00884     AddToCB(C,0)
00885     number = C->numrhs-startebuf+numxsymbol;
00886 wearehappy:
00887     *term = old[0]; ss[0] = old[1]; ss[1] = old[2]; ss[2] = old[3]; ss[3] = old[4];
00888     return(number);
00889 }
00890
00891 /*
00892         #] FindLocalSubterm :
00893         #[ PrintSubtermList :
00894
00895         Prints all the expressions in the subterm compiler buffer.
00896         The format is such that they give definitions of the temporary
00897         variables of which the contents are stored in this buffer.
00898         These variables have the names Z_123 etc.
00899 */
00900
00901 void PrintSubtermList(int from,int to)
00902 {
00903     UBYTE buffer[80], *out, outbuffer[300];
00904     int first, i, ii, inc = 1;
00905     WORD *term;
00906     CBUF *C = cbuf + AM.sbufnum;
00907 /*
00908     if ( to < from ) inc = -1;
00909     if ( to == from ) inc = 0;
00910 */
00911     if ( from <= to ) {
00912         inc = 1; to += inc;
00913     }
00914     else {
00915         inc = -1; to += inc;
00916     }
```

```
00917     AO.OutFill = AO.OutputLine = outbuffer;
00918     AO.OutStop = AO.OutputLine+AC.LineLength;
00919     AO.IsBracket = 0;
00920     AO.OutSkip = 3;
00921 
00922     if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {
00923         TokenToLine((UBYTE *)"      ");
00924         AO.OutSkip = 7;
00925     }
00926     else if ( ( AO.Optimize.debugflags & 1 ) == 1 ) {}
00927     else if ( AO.OutSkip > 0 ) {
00928         for ( i = 0; i < AO.OutSkip; i++ ) TokenToLine((UBYTE *)" ");
00929     }
00930     i = from;
00931     do {
00932         if ( ( AO.Optimize.debugflags & 1 ) == 1 ) {
00933             TokenToLine((UBYTE *)"id ");
00934             for ( ii = 3; ii < AO.OutSkip; ii++ ) TokenToLine((UBYTE *)" ");
00935         }
00936 /*
00937         if ( AC.OutputMode == NORMALFORMAT ) {
00938             TokenToLine((UBYTE *)"id ");
00939         }
00940 */
00941         else if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {}
00942         else { TokenToLine((UBYTE *)" "); }
00943 
00944         out = StrCopy((UBYTE *)AC.extrasym,buffer);
00945         if ( AC.extrasymbols == 0 ) {
00946             out = NumCopy(i,out);
00947             out = StrCopy((UBYTE *)"_",out);
00948         }
00949         else if ( AC.extrasymbols == 1 ) {
00950             out = AddArrayIndex(i,out);
00951         }
00952         out = StrCopy((UBYTE *)"=",out);
00953         TokenToLine(buffer);
00954         term = C->rhs[i];
00955         first = 1;
00956         if ( *term == 0 ) {
00957             out = StrCopy((UBYTE *)"0",buffer);
00958             if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE ) {
00959                 out = StrCopy((UBYTE *)";",out);
00960             }
00961             TokenToLine(buffer);
00962         }
00963         else {
00964             while ( *term ) {
00965                 if ( WriteInnerTerm(term,first) ) Terminate(-1);
00966                 term += *term;
00967                 first = 0;
00968             }
00969             if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE ) {
00970                 out = StrCopy((UBYTE *)";",buffer);
00971                 TokenToLine(buffer);
00972             }
00973         }
00974 /*
00975         There is a problem with FiniLine because it prepares for a
00976         continuation line in fortran mode.
00977         But the next statement should start on a blank line.
00978 */
00979 /*
00980         FiniLine();
00981         if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {
00982             AO.OutFill = AO.OutputLine;
00983             TokenToLine((UBYTE *)"      ");
00984             AO.OutSkip = 7;
00985         }
00986 */
00987         if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {
00988             AO.OutSkip = 6;
00989             FiniLine();
00990             AO.OutSkip = 7;
00991         }
00992         else {
00993             FiniLine();
00994         }
00995         i += inc;
00996     } while ( i != to );
00997 }
00998 
00999 /*
01000         #] PrintSubtermList :
01001         #[ PrintExtraSymbol :
01002 
01003         Prints the definition of extra symbol num as the contents
```

```
01004         of the expression in terms.
01005         The parameter par has three options:
01006             EXTRASYMBOL      num is interpreted as the number of an extra symbol
01007             REGULARSYMBOL    num is interpreted as the number of a symbol.
01008                              It could still be an extra symbol.
01009             EXPRESSIONNUMBER num is the number of an expression.
01010         terms contains the rhs expression.
01011 */
01012
01013 void PrintExtraSymbol(int num, WORD *terms,int par)
01014 {
01015     UBYTE buffer[80], *out, outbuffer[300];
01016     int first, i;
01017     WORD *term;
01018
01019     AO.OutFill = AO.OutputLine = outbuffer;
01020     AO.OutStop = AO.OutputLine+AC.LineLength;
01021     AO.IsBracket = 0;
01022
01023     if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {
01024         TokenToLine((UBYTE *)"      ");
01025         AO.OutSkip = 7;
01026     }
01027     else if ( ( AO.Optimize.debugflags & 1 ) == 1 ) {
01028         TokenToLine((UBYTE *)"id ");
01029         for ( i = 3; i < AO.OutSkip; i++ ) TokenToLine((UBYTE *)" ");
01030     }
01031     else if ( AO.OutSkip > 0 ) {
01032         for ( i = 0; i < AO.OutSkip; i++ ) TokenToLine((UBYTE *)" ");
01033     }
01034     out = buffer;
01035     switch ( par ) {
01036         case REGULARSYMBOL:
01037             if ( num >= MAXVARIABLES-cbuf[AM.sbufnum].numrhs ) {
01038                 num = MAXVARIABLES-num;
01039             }
01040             else {
01041                 out = StrCopy(FindSymbol(num),out);
01042 /*              out = StrCopy(VARNAME(symbols,num),out); */
01043                 break;
01044             }
01045             /* fall through */
01046         case EXTRASYMBOL:
01047             out = StrCopy(FindExtraSymbol(num),out);
01048 /*
01049             out = StrCopy((UBYTE *)AC.extrasym,out);
01050             if ( AC.extrasymbols == 0 ) {
01051                 out = NumCopy(num,out);
01052                 out = StrCopy((UBYTE *)"_",out);
01053             }
01054             else if ( AC.extrasymbols == 1 ) {
01055                 out = AddArrayIndex(num,out);
01056             }
01057 */
01058             break;
01059         case EXPRESSIONNUMBER:
01060             out = StrCopy(EXPRNAME(num),out);
01061             break;
01062         default:
01063             MesPrint("Illegal option in PrintExtraSymbol");
01064             Terminate(-1);
01065     }
01066     out = StrCopy((UBYTE *)"=",out);
01067     TokenToLine(buffer);
01068     term = terms;
01069     first = 1;
01070     if ( *term == 0 ) {
01071         out = StrCopy((UBYTE *)"0",buffer);
01072         TokenToLine(buffer);
01073     }
01074     else {
01075         while ( *term ) {
01076             if ( WriteInnerTerm(term,first) ) Terminate(-1);
01077             term += *term;
01078             first = 0;
01079         }
01080     }
01081     if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE ) {
01082         out = StrCopy((UBYTE *)";",buffer);
01083         TokenToLine(buffer);
01084     }
01085     FiniLine();
01086 }
01087
01088 /*
01089         #] PrintExtraSymbol :
01090         #[ FindSubexpression :
```

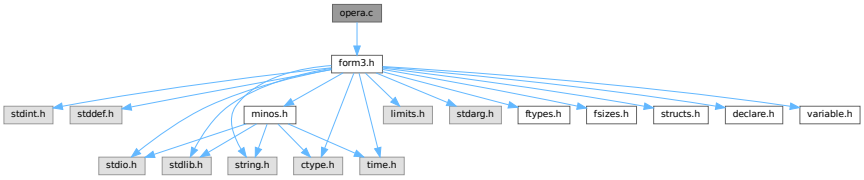
```
01091
01092         In this routine we look up a subexpression.
01093         If we don't find it we will enter it in the subterm compiler buffer
01094         Searching is by tree structure.
01095         Adding changes the tree.
01096
01097         Notice that in TFORM we should be in sequential mode.
01098 */
01099
01100 int FindSubexpression(WORD *subexpr)
01101 {
01102     WORD *term;
01103     int number;
01104     CBUF *C = cbuf + AM.sbufnum;
01105     LONG oldCpointer;
01106
01107     term = subexpr;
01108     while ( *term ) term += *term;
01109     number = term - subexpr;
01110 /*
01111         We may have to add the subexpression to the tree.
01112         This requires a lock.
01113 */
01114     LOCK(AM.sbuflock);
01115
01116     oldCpointer = C->Pointer-C->Buffer; /* Offset of course !!!!!*/
01117     AddRHS(AM.sbufnum,1);
01118 /*
01119         Add the terms to the compiler buffer. Paste on a zero.
01120 */
01121     AddNtoC(AM.sbufnum,number,subexpr,10);
01122     AddToCB(C,0)
01123 /*
01124         See whether we have this one already. If not, insert it in the tree.
01125 */
01126     number = InsTree(AM.sbufnum,C->numrhs);
01127 /*
01128         Restore old values and return what is needed.
01129 */
01130     if ( number < (C->numrhs) ) {   /* It existed already */
01131         C->Pointer = oldCpointer + C->Buffer;
01132         C->numrhs--;
01133     }
01134     else {
01135         GETIDENTITY
01136         WORD dim = DimensionExpression(BHEAD subexpr);
01137 /*
01138         Store the dimension
01139 */
01140         C->dimension[number] = dim;
01141     }
01142
01143     UNLOCK(AM.sbuflock);
01144
01145     return(number);
01146 }
01147
01148 /*
01149         #] FindSubexpression :
01150         #[ ExtraSymFun :
01151 */
01152
01153 int ExtraSymFun(PHEAD WORD *term,WORD level)
01154 {
01155     WORD *oldworkpointer = AT.WorkPointer;
01156     WORD *termout, *t1, *t2, *t3, *tstop, *tend, i;
01157     int retval = 0;
01158     tend = termout = term + *term;
01159     tstop = tend - ABS(tend[-1]);
01160     t3 = t1 = term+1; t2 = termout+1;
01161 /*
01162     First refind the function(s). There is at least one.
01163 */
01164     while ( t1 < tstop ) {
01165         if ( *t1 == EXTRASYMFUN && t1[1] == FUNHEAD+2 ) {
01166             if ( t1[FUNHEAD] == -SNUMBER && t1[FUNHEAD+1] <= numxsymbol
01167                         && t1[FUNHEAD+1] > 0 ) {
01168                 i = t1[FUNHEAD+1];
01169             }
01170             else if ( t1[FUNHEAD] == -SYMBOL && t1[FUNHEAD+1] < MAXVARIABLES
01171                         && t1[FUNHEAD+1] >= MAXVARIABLES-numxsymbol ) {
01172                 i = MAXVARIABLES - t1[FUNHEAD+1];
01173             }
01174             else goto nocase;
01175             while ( t3 < t1 ) *t2++ = *t3++;
01176 /*
01177                 Now inset the rhs pointer
```

```
01178 */
01179             *t2++ = SUBEXPRESSION;
01180             *t2++ = SUBEXPSIZE;
01181             *t2++ = i;
01182             *t2++ = 1;
01183             *t2++ = AM.sbufnum;
01184             FILLSUB(t2)
01185             t3 = t1 = t1 + t1[1];
01186         }
01187         else if ( *t1 == EXTRASYMFUN && t1[1] == FUNHEAD ) {
01188             while ( t3 < t1 ) *t2++ = *t3++;
01189             t3 = t1 = t1 + t1[1];
01190         }
01191         else {
01192 nocase:;
01193             t1 = t1 + t1[1];
01194         }
01195     }
01196     while ( t3 < tend ) *t2++ = *t3++;
01197     *termout = t2 - termout;
01198     AT.WorkPointer = t2;
01199     if ( AT.WorkPointer >= AT.WorkTop ) {
01200         MLOCK(ErrorMessageLock);
01201         MesWork();
01202         MUNLOCK(ErrorMessageLock);
01203         AT.WorkPointer = oldworkpointer;
01204         return(-1);
01205     }
01206     retval = Generator(BHEAD termout,level);
01207     AT.WorkPointer = oldworkpointer;
01208     if ( retval < 0 ) {
01209         MLOCK(ErrorMessageLock);
01210         MesCall("ExtraSymFun");
01211         MUNLOCK(ErrorMessageLock);
01212     }
01213     return(retval);
01214 }
01215
01216 /*
01217         #] ExtraSymFun :
01218         #[ PruneExtraSymbols :
01219 */
01220
01221 int PruneExtraSymbols(WORD downto)
01222 {
01223     CBUF *C = cbuf + AM.sbufnum;
01224     if ( downto < C->numrhs && downto >= 0 ) {  /* !!!!! */
01225         ClearTree(AM.sbufnum);
01226         C->numrhs = downto;
01227         if ( downto == 0 ) {
01228             C->Pointer = C->Buffer;
01229         }
01230         else {
01231             WORD *w = C->rhs[downto], i;
01232             while ( *w ) w += *w;
01233             C->Pointer = w+1;
01234             for ( i = 1; i <= downto; i++ ) {
01235                 InsTree(AM.sbufnum,i);
01236             }
01237         }
01238     }
01239     return(0);
01240 }
01241
01242 /*
01243         #] PruneExtraSymbols :
01244 */
```

4.93 opera.c File Reference

```
#include "form3.h"
```

Include dependency graph for opera.c:



Functions

- int EpfFind (PHEAD WORD *term, WORD *params)
- int EpfCon (PHEAD WORD *term, WORD *params, WORD num, WORD level)
- WORD EpfGen (WORD number, WORD *inlist, WORD *kron, WORD *perm, WORD sgn)
- WORD Trick (WORD *in, TRACES *t)
- int Trace4no (WORD number, WORD *kron, TRACES *t)
- int Trace4 (PHEAD WORD *term, WORD *params, WORD num, WORD level)
- int Trace4Gen (PHEAD TRACES *t, WORD number)
- WORD TraceNno (WORD number, WORD *kron, TRACES *t)
- int TraceN (PHEAD WORD *term, WORD *params, WORD num, WORD level)
- int TraceNgen (PHEAD TRACES *t, WORD number)
- int Traces (PHEAD WORD *term, WORD *params, WORD num, WORD level)
- int TraceFind (PHEAD WORD *term, WORD *params)
- int Chisholm (PHEAD WORD *term, WORD level)
- int TenVecFind (PHEAD WORD *term, WORD *params)
- int TenVec (PHEAD WORD *term, WORD *params, WORD num, WORD level)

4.93.1 Detailed Description

Contains the 'operations' These are the trace routines, the contractions of the Levi-Civita tensors and the tensor to vector/vector to tensor routines. The trace and contraction routines are done in a special way (see commentary with the FIXEDGLOBALS struct)

Definition in file opera.c.

4.93.2 Function Documentation

4.93.2.1 EpfFind()

```
int EpfFind (
            PHEAD WORD * term,
            WORD * params )
```

Definition at line 58 of file opera.c.

4.93.2.2 EpfCon()

```
int EpfCon (
              PHEAD WORD * term,
              WORD * params,
              WORD num,
              WORD level )
```

Definition at line 212 of file opera.c.

4.93.2.3 EpfGen()

```
WORD EpfGen (
              WORD number,
              WORD * inlist,
              WORD * kron,
              WORD * perm,
              WORD sgn )
```

Definition at line 284 of file opera.c.

4.93.2.4 Trick()

```
WORD Trick (
              WORD * in,
              TRACES * t )
```

Definition at line 333 of file opera.c.

4.93.2.5 Trace4no()

```
int Trace4no (
              WORD number,
              WORD * kron,
              TRACES * t )
```

Definition at line 420 of file opera.c.

4.93.2.6 Trace4()

```
int Trace4 (
              PHEAD WORD * term,
              WORD * params,
              WORD num,
              WORD level )
```

Definition at line 697 of file opera.c.

4.93.2.7 Trace4Gen()

```
int Trace4Gen (
             PHEAD TRACES * t,
             WORD number )
```

Definition at line 860 of file opera.c.

4.93.2.8 TraceNno()

```
WORD TraceNno (
             WORD number,
             WORD * kron,
             TRACES * t )
```

Definition at line 1345 of file opera.c.

4.93.2.9 TraceN()

```
int TraceN (
             PHEAD WORD * term,
             WORD * params,
             WORD num,
             WORD level )
```

Definition at line 1386 of file opera.c.

4.93.2.10 TraceNgen()

```
int TraceNgen (
             PHEAD TRACES * t,
             WORD number )
```

Definition at line 1451 of file opera.c.

4.93.2.11 Traces()

```
int Traces (
             PHEAD WORD * term,
             WORD * params,
             WORD num,
             WORD level )
```

Definition at line 1836 of file opera.c.

4.93.2.12 TraceFind()

```
int TraceFind (
             PHEAD WORD * term,
             WORD * params )
```

Definition at line 1858 of file opera.c.

4.93.2.13 Chisholm()

```
int Chisholm (
              PHEAD WORD * term,
              WORD level )
```

Definition at line 1968 of file opera.c.

4.93.2.14 TenVecFind()

```
int TenVecFind (
              PHEAD WORD * term,
              WORD * params )
```

Definition at line 2183 of file opera.c.

4.93.2.15 TenVec()

```
int TenVec (
              PHEAD WORD * term,
              WORD * params,
              WORD num,
              WORD level )
```

Definition at line 2317 of file opera.c.

4.94 opera.c

Go to the documentation of this file.

```
00001
00009 /* #[ License : */
00010 /*
00011  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00012  *   When using this file you are requested to refer to the publication
00013  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00014  *   This is considered a matter of courtesy as the development was paid
00015  *   for by FOM the Dutch physics granting agency and we would like to
00016  *   be able to track its scientific use to convince FOM of its value
00017  *   for the community.
00018  *
00019  *   This file is part of FORM.
00020  *
00021  *   FORM is free software: you can redistribute it and/or modify it under the
00022  *   terms of the GNU General Public License as published by the Free Software
00023  *   Foundation, either version 3 of the License, or (at your option) any later
00024  *   version.
00025  *
00026  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00027  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00028  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00029  *   details.
00030  *
00031  *   You should have received a copy of the GNU General Public License along
00032  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00033  */
00034 /* #] License : */
00035 /*
00036     #[ Includes : opera.c
00037 */
00038
00039 #include "form3.h"
00040 /*
00041     int hulp;
```

```
00042 */
00043
00044 /*
00045     #] Includes :
00046     #[ Operations :
00047         #[ EpfFind :            WORD EpfFind(term,params)
00048
00049         Searches for a pair of Levi-Civita tensors that should be
00050         contracted.
00051         If a match is found its settings are recorded in AT.TMout.
00052         type indicates the number of indices that is searched for,
00053         unless all are searched for (type = 0).
00054         number is the number of tensors that should survive contraction.
00055
00056 */
00057
00058 int EpfFind(PHEAD WORD *term, WORD *params)
00059 {
00060     GETBIDENTITY
00061     WORD *t, *m, *r, n1 = 0, n2, min = -1, count, fac;
00062     WORD *c1 = 0, *c2 = 0, sgn = 1;
00063     WORD *tstop, *mstop;
00064     UWORD *facto = (UWORD *)AT.WorkPointer;
00065     WORD ncoef,nfac;
00066     WORD number, type;
00067     if ( ( AT.WorkPointer = (WORD *)(facto + AM.MaxTal) ) > AT.WorkTop ) {
00068         MLOCK(ErrorMessageLock);
00069         MesWork();
00070         MUNLOCK(ErrorMessageLock);
00071         return(-1);
00072     }
00073     number = params[3];
00074     type = params[4];
00075     t = term;
00076     GETSTOP(t,tstop);
00077     t++;
00078     if ( !type ) {
00079         while ( *t != LEVICIVITA && t < tstop ) t += t[1];
00080         if ( t >= tstop ) return(0);
00081         m = t;
00082         while ( *m == LEVICIVITA && m < tstop ) { n1++; m += m[1]; }
00083 AllLev:
00084         if ( n1 <= (number+1) || n1 <= 1 ) return(0);
00085         mstop = m;
00086         m = t + t[1];
00087         do {
00088             while ( m[1] == t[1] ) {
00089                 m += FUNHEAD;
00090                 r = t+FUNHEAD;
00091                 count = fac = n1 = n2 = t[1] - FUNHEAD;
00092                 while ( n1 && n2 ) {
00093                     if ( *m > *r ) {
00094                         r++; n2--;
00095                     }
00096                     else if ( *m < *r ) {
00097                         m++; n1--;
00098                     }
00099                     else {
00100                         if ( *m >= AM.OffsetIndex &&
00101                         ( ( *m >= AM.IndDum && AC.lDefDim == fac ) ||
00102                         ( *m < AM.IndDum &&
00103                         indices[*m-AM.OffsetIndex].dimension == fac ) ) ) {
00104                             count--;
00105                         }
00106                         r++; m++; n1--; n2--;
00107                     }
00108                 }
00109                 m += n1;
00110                 if ( min < 0 || count < min ) {
00111                     c1 = t;
00112                     c2 = m - fac - FUNHEAD;
00113                     min = count;
00114                 }
00115                 if ( m >= mstop ) break;
00116             }
00117             t += t[1];
00118         } while ( ( m = t + t[1] ) < mstop );
00119     }
00120     else {
00121         fac = type + FUNHEAD;
00122         while ( *t != LEVICIVITA && t < tstop ) t += t[1];
00123         while ( *t == LEVICIVITA && t < tstop && t[1] != fac ) t += t[1];
00124         if ( t >= tstop ) return(0);
00125         m = t;
00126         while ( *m == LEVICIVITA && m < tstop && m[1] == fac ) { n1++; m += m[1]; }
00127         goto AllLev;
00128     }
```

```
00129 /*
00130     We have now the two tensors that give the minimum contraction
00131     in c1 and c2.
00132     Prepare the AT.TMout array;
00133 */
00134     if ( min < 0 ) return(0);   /* No matching pair! */
00135     t = c1;
00136     mstop = c2;
00137     fac = t[1] - FUNHEAD;
00138     m = AT.TMout;
00139     *m++ = 3 + (min*2);     /* The full length */
00140     *m++ = CONTRACT;
00141     if ( number < 0 ) *m++ = 1;
00142     else *m++ = 0;
00143     n1 = n2 = t[1] - FUNHEAD;
00144     r = c1 + FUNHEAD;
00145     c1 = m;
00146     m = c2 + FUNHEAD;
00147     c2 = c1 + min;
00148     while ( n1 && n2 ) {
00149         if ( *m > *r ) { *c1++ = *r++; n2--; }
00150         else if ( *m < *r ) { *c2++ = *m++; n1--; }
00151         else {
00152             if ( *m < AM.OffsetIndex || ( *m < AM.IndDum &&
00153             ( indices[*m-AM.OffsetIndex].dimension != fac ) ) ||
00154             ( *m >= AM.IndDum && AC.lDefDim != fac ) ) {
00155                 *c1++ = *r++; *c2++ = *m++;
00156             }
00157             else { if ( ( n1 ^ n2 ) & 1 ) sgn = -sgn; r++; m++; }
00158             n1--; n2--;
00159         }
00160     }
00161     if ( n1 ) { NCOPY(c2,m,n1); }
00162     else if ( n2 ) { NCOPY(c1,r,n2); }
00163     fac -= min;
00164     m = t + t[1];
00165     while ( m < mstop ) *t++ = *m++;
00166     m += m[1];
00167     while ( m < tstop ) *t++ = *m++;
00168     *t++ = SUBEXPRESSION;
00169     *t++ = SUBEXPSIZE;
00170     *t++ = -1;
00171     *t++ = 1;
00172     *t++ = DUMMYBUFFER;
00173     FILLSUB(t)
00174     r = term;
00175     r += *r - 1;
00176     mstop = r;
00177     ncoef = REDLENG(*r);
00178     tstop = t;
00179     while ( m < mstop ) *t++ = *m++;
00180     if ( Factorial(BHEAD fac,facto,&nfac) || Mully(BHEAD (UWORD *)tstop,&ncoef,facto,nfac) ) {
00181         MLOCK(ErrorMessageLock);
00182         MesCall("EpfFind");
00183         MUNLOCK(ErrorMessageLock);
00184         SETERROR(-1)
00185     }
00186     tstop += (ABS(ncoef))*2;
00187     if ( sgn < 0 ) ncoef = -ncoef;
00188     ncoef *= 2;
00189     *tstop++ = (ncoef<0)?(ncoef-1):(ncoef+1);
00190     *term = WORDDIF(tstop,term);
00191     return(1);
00192 }
00193
00194 /*
00195         #] EpfFind :
00196         #[ EpfCon :             WORD EpfCon(term,params,num,level)
00197
00198         Contraction of two strings of indices/vectors. They come
00199         from LeviCivita tensors that are being contracted.
00200         term is the term with the subterm to be replaced.
00201         params is the full indicator:
00202             Length, number, raise, parameters.
00203         Length is the length of the parameter field.
00204         number is the number of the operation.
00205         raise tells whether level should be raised afterwards.
00206         the parameters are the two strings.
00207         level is the id level.
00208         The factorial has been multiplied in already.
00209
00210 */
00211
00212 int EpfCon(PHEAD WORD *term, WORD *params, WORD num, WORD level)
00213 {
00214     GETBIDENTITY
00215     WORD *kron, *perm, *termout, *tstop, size2;
```

```
00216     WORD *m, *t, sizes, sgn = 0, i;
00217     sizes = *params - 3;
00218     kron = AT.WorkPointer;
00219     perm = (AT.WorkPointer += sizes);
00220     termout = (AT.WorkPointer += sizes);
00221     AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
00222     if ( AT.WorkPointer > AT.WorkTop ) {
00223         MLOCK(ErrorMessageLock);
00224         MesWork();
00225         MUNLOCK(ErrorMessageLock);
00226         return(-1);
00227     }
00228     params += 2;
00229     if ( !(*params++) ) level--;
00230     size2 = sizes>>1;
00231     if ( !size2 ) goto DoOnce;
00232     while ( ( sgn = EpfGen(size2,params,kron,perm,sgn) ) != 0 ) {
00233 DoOnce:
00234         t = term;
00235         GETSTOP(t,tstop);
00236         m = termout;
00237         tstop -= SUBEXPSIZE;
00238         while ( t < tstop ) *m++ = *t++;
00239         if ( t[2] != num || *t != SUBEXPRESSION ) {
00240 /* INTERNAL_ERROR_EXCL_START */
00241             MLOCK(ErrorMessageLock);
00242             MesPrint("!>Serious error in EpfCon");
00243             MUNLOCK(ErrorMessageLock);
00244             return(-1);
00245 /* INTERNAL_ERROR_EXCL_STOP */
00246         }
00247         tstop += SUBEXPSIZE;
00248         if ( sizes ) {
00249             *m++ = DELTA;
00250             *m++ = sizes + 2;
00251             t = kron;
00252             i = sizes;
00253             if ( i ) { NCOPY(m,t,i); }
00254         }
00255         t = tstop;
00256         tstop = term + *term;
00257         while ( t < tstop ) *m++ = *t++;
00258         *termout = WORDDIF(m,termout);
00259         m--;
00260         if ( sgn < 0 ) *m = - *m;
00261         if ( *termout ) {
00262             *AN.RepPoint = 1;
00263             AR.expchanged = 1;
00264             AT.WorkPointer = termout + *termout;
00265             if ( Generator(BHEAD termout,level) < 0 ) goto EpfCall;
00266         }
00267     }
00268     AT.WorkPointer = kron;
00269     return(0);
00270 EpfCall:
00271     if ( AM.tracebackflag ) {
00272         MLOCK(ErrorMessageLock);
00273         MesCall("EpfCon");
00274         MUNLOCK(ErrorMessageLock);
00275     }
00276     return(-1);
00277 }
00278
00279 /*
00280         #] EpfCon :
00281         #[ EpfGen :             WORD EpfGen(number,inlist,kron,perm,sgn)
00282 */
00283
00284 WORD EpfGen(WORD number, WORD *inlist, WORD *kron, WORD *perm, WORD sgn)
00285 {
00286     WORD i, *in2, k, a;
00287     if ( !sgn ) {
00288         in2 = inlist + number;
00289         number *= 2;
00290         for ( i = 1; i < number; i += 2 ) {
00291             *perm++ = i;
00292             *perm++ = i;
00293             *kron++ = *inlist++;
00294             *kron++ = *in2++;
00295         }
00296         if ( number <= 0 ) return(0);
00297         else return(1);
00298     }
00299     number *= 2;
00300     i = number - 1;
00301     while ( ( i -= 2 ) >= 0 ) {
00302         if ( ( k = perm[i] ) != i ) {
```

```
00303             sgn = -sgn;
00304             a = kron[i];
00305             kron[i] = kron[k];
00306             kron[k] = a;
00307         }
00308         if ( ( k = ( perm[i] += 2 ) ) < number ) {
00309             a = kron[i];
00310             kron[i] = kron[k];
00311             kron[k] = a;
00312             sgn = - sgn;
00313             for ( k = i + 2; k < number; k += 2 ) perm[k] = k;
00314             return(sgn);
00315         }
00316     }
00317     return(0);
00318 }
00319
00320 /*
00321         #] EpfGen :
00322         #[ Trick :              WORD Trick(in,t)
00323
00324         This routine implements the identity:
00325         g_(j,mu)*g_(j,nu)*g_(j,ro)=e_(mu,nu,ro,si)*g5_(j)*g_(j,si)
00326         +d_(mu,nu)*g_(j,ro)-d_(mu,ro)*g_(j,nu)+d_(nu,ro)*g_(j,mu)
00327         which is for 4 dimensions only!
00328
00329         Note that z->gamm = 1 if there is no g5 present.
00330
00331 */
00332
00333 WORD Trick(WORD *in, TRACES *t)
00334 {
00335     WORD n, n1;
00336     n = t->stap;
00337     n1 = t->step1;
00338     switch ( t->eers[n] ) {
00339         case 0: {
00340             WORD *p;
00341             p = t->pepf + t->mepf[n];
00342             *p++ = *in++;
00343             *p++ = *in++;
00344             *p++ = *in;
00345             *p   = ++(t->mdum);
00346             (t->mepf[n1]) += 4;
00347             *in  = t->mdum;
00348             t->gamm = - t->gamm;
00349             t->eers[n] = 5;
00350             break;
00351         }
00352         case 4: {
00353             WORD *p;
00354             p = t->pdel + t->mdel[n];
00355             (t->mepf[n1]) -= 4;
00356             (t->mdum)--;
00357             *p++ = *in++;
00358             *p   = *in++;
00359             *in  = *(t->pepf + t->mepf[n] + 2);
00360             (t->mdel[n1]) += 2;
00361             t->gamm = - t->gamm;
00362             break;
00363         }
00364         case 3: {
00365             t->sign1 = - t->sign1;
00366             *(t->pdel + t->mdel[n] + 1) = in[2];
00367             in[2] = in[1];
00368             break;
00369         }
00370         case 2: {
00371             t->sign1 = - t->sign1;
00372             in[2] = in[0];
00373             *(t->pdel + t->mdel[n]) = in[1];
00374             break;
00375         }
00376         case 1: {
00377             in[2] = *(t->pdel + t->mdel[n] + 1);
00378             (t->mdel[n1]) -= 2;
00379             break;
00380         }
00381         default: {
00382             return(0);
00383         }
00384     }
00385     return(--(t->eers[n]));
00386 }
00387
00388 /*
00389         #] Trick :
```

```
00390         #[ Trace4no :           WORD Trace4no(number,kron,t)
00391
00392         Takes the trace of a string of gamma matrices in 4 dimensions.
00393         There is no test for indices or vectors that are the same.
00394         The four dimensions refer to the contraction in the algebra:
00395         g_(i,a,b,c) = e_(a,b,c,d)*g_(i,5_,d) + g_(i,a)*d_(b,c)
00396                     - g_(i,b)*d_(a,c) + g_(i,c)*d_(a,b)
00397         This simplifies life very much and leads to shorter expressions
00398         than in the n dimensional case.
00399
00400         Parameters:
00401         number: the number of vectors/indices in inlist.
00402         inlist: the indices/vectors in the string.
00403         kron:   the output delta's.
00404         gamma5: the potential gamma5 in front.
00405         t:      the struct for scratch manipulations.
00406         stack:  the space to put all scratch arrays in.
00407
00408         The return value is zero if there are no more terms, 1 if a
00409         term was generated with a positive sign and -1 if a term was
00410         generated with a negative sign.
00411         The value of one is increased to two if the first 4 values
00412         in kron should be interpreted as a Levi-Civita tensor.
00413
00414         Note that kron should have more places reserved than the number
00415         of indices in inlist, because it will contain dummy indices
00416         temporarily. In principle there can be 'number*1/4' extra dummies.
00417
00418 */
00419
00420 int Trace4no(WORD number, WORD *kron, TRACES *t)
00421 {
00422     WORD i;
00423     WORD *p, *m;
00424     WORD *stop, oldsign;
00425     int retval;
00426     if ( !t->sgn ) {        /* Startup */
00427         if ( ( number < 0 ) || ( number & 1 ) ) return(0);
00428         if ( number <= 2 ) {
00429             if ( t->gamma5 == GAMMA5 ) return(0);
00430             if ( number == 2 ) {
00431                 *kron++ = *t->inlist;
00432                 *kron++ = t->inlist[1];
00433             }
00434             return(1);
00435         }
00436         t->sgn = 1;
00437         {
00438             WORD nhalf = number » 1;
00439             WORD ndouble = number * 2;
00440             p = t->eers;
00441             t->eers = p;    p += nhalf;
00442             t->mepf = p;    p += nhalf;
00443             t->mdel = p;    p += nhalf;
00444             t->pdel = p;    p += number + nhalf;
00445             t->pepf = p;    p += ndouble;
00446             t->e4 = p;      p += number;
00447             t->e3 = p;      p += ndouble;
00448             t->nt3 = p;     p += nhalf;
00449             t->nt4 = p;     p += nhalf;
00450             t->j3 = p;      p += ndouble;
00451             t->j4 = p;
00452         }
00453         t->mepf[0] = 0;
00454         t->mdel[0] = 0;
00455         t->mdum = AM.mTraceDum;
00456         t->kstep = -2;
00457         t->step1 = 0;
00458         t->sign1 = 1;
00459         t->lc3 = -1;
00460         t->lc4 = -1;
00461         t->gamm = 1;
00462
00463         do {
00464             t->stap = (t->step1)++;
00465             t->kstep += 2;
00466             t->eers[t->stap] = 0;
00467             t->mepf[t->step1] = t->mepf[t->stap];
00468             t->mdel[t->step1] = t->mdel[t->stap];
00469 CallTrick:  while ( !Trick(t->inlist+t->kstep,t) ) {
00470                 t->kstep -= 2;
00471                 t->step1 = (t->stap)--;
00472                 if ( t->stap < 0 ) {
00473                     return(0);
00474                 }
00475             }
00476         } while ( t->kstep < (number-4) );
```

```
00477 /*
00478         Take now the trace of the leftover matrices.
00479         If gamma5 causes the term to vanish there will be a
00480         renewed call to Trick for its next term.
00481 */
00482         t->sign2 = t->sign1;
00483         if ( ( t->gamma5 == GAMMA7 ) && ( t->gamm == -1 ) ) {
00484             t->sign2 = - t->sign2;
00485         }
00486         else if ( ( t->gamma5 == GAMMA5 ) && ( t->gamm == 1 ) ) {
00487             goto CallTrick;
00488         }
00489         else if ( ( t->gamma5 == GAMMA1 ) && ( t->gamm == -1 ) ) {
00490             goto CallTrick;
00491         }
00492         p = t->pdel + t->mdel[t->step1];
00493         *p++ = t->inlist[t->kstep+2];
00494         *p++ = t->inlist[t->kstep+3];
00495 /*
00496         Now the trace has been expressed in terms of Levi-Civita tensors
00497         and Kronecker delta's.
00498         The Levi-Civita tensors are in t->pepf
00499         and there are t->mepf[step1] elements in this array.
00500         The Kronecker delta's are in t->pdel
00501         and there are t->mdel[step1] elements in this array.
00502
00503         Next we rake the Levi-Civita tensors together such that there
00504         is an optimal use of the contractions.
00505 */
00506         {
00507             WORD ae;
00508             ae = t->mepf[t->step1];
00509             t->ad = t->mdel[t->step1]+2;
00510             t->a4 = 0;
00511             t->a3 = 0;
00512             while ( ( ae -= 4 ) >= 0 ) {
00513                 if ( t->pepf[ae] > AM.mTraceDum && t->pepf[ae] <= t->mdum ) {
00514                     p = t->e3 + t->a3;
00515                     m = t->pepf + ae;
00516                     for ( i = 0; i < 3; i++ ) {
00517                         p[3] = m[3-i];
00518                         *p++ = m[i-4];
00519                     }
00520                     t->a3 += 6;
00521                     ae -= 4;
00522                 }
00523                 else {
00524                     p = t->e4 + t->a4;
00525                     m = t->pepf + ae;
00526                     for ( i = 0; i < 4; i++ ) *p++ = *m++;
00527                     t->a4 += 4;
00528                 }
00529             }
00530         }
00531 /*
00532         Now e3 contains pairs of LeviCivita tensors that have
00533         three indices each and a3 is the total number of indices.
00534         e4 contains individual tensors with 4 indices.
00535         Some indices may be contracted with Kronecker delta's.
00536
00537         Contract the e3 tensors first.
00538 */
00539
00540         while ( t->a3 > 0 ) {
00541             t->nt3[++(t->lc3)] = 0;
00542             while ( ( t->nt3[t->lc3] = EpfGen(3,t->e3+t->a3-6,
00543             t->pdel+t->ad,t->j3+6*t->lc3,oldsign = t->nt3[t->lc3]) ) == 0 ) {
00544                 if ( oldsign < 0 ) t->sign2 = - t->sign2;
00545                 (t->lc3)--;
00546 NextE3:         if ( t->lc3 < 0 ) goto CallTrick;
00547                 t->ad -= 6;
00548                 t->a3 += 6;
00549             }
00550             if ( oldsign ) {
00551                 if ( oldsign != t->nt3[t->lc3] ) t->sign2 = - t->sign2;
00552             }
00553             else if ( t->nt3[t->lc3] < 0 ) t->sign2 = - t->sign2;
00554             t->a3 -= 6;
00555             t->ad += 6;
00556         }
00557 /*
00558         Contract the e4 tensors.
00559 */
00560         while ( t->a4 > 4 ) {
00561             t->nt4[++(t->lc4)] = 0;
00562             while ( ( t->nt4[t->lc4] = EpfGen(4,t->e4+t->a4-8,
00563             t->pdel+t->ad,t->j4+8*t->lc4,oldsign = t->nt4[t->lc4]) ) == 0 ) {
```

```
00564                 if ( oldsign < 0 ) t->sign2 = - t->sign2;
00565                 (t->lc4)--;
00566 NextE4:         if ( t->lc4 < 0 ) goto NextE3;
00567                 t->ad -= 8;
00568                 t->a4 += 8;
00569             }
00570             if ( oldsign ) {
00571                 if ( oldsign != t->nt4[t->lc4] ) t->sign2 = - t->sign2;
00572             }
00573             else if ( t->nt4[t->lc4] < 0 ) t->sign2 = - t->sign2;
00574             t->a4 -= 8;
00575             t->ad += 8;
00576         }
00577 /*
00578         Finally the extra dummy indices can be eliminated.
00579         Note that there are currently t->ad words in t->pdel forming
00580         t->ad / 2 Kronecker delta's. We are however not allowed to
00581         alter anything in these arrays, so the results should be
00582         copied to kron.
00583 */
00584         m = kron;
00585         if ( t->a4 == 4 ) {
00586             p = t->e4;
00587             *m++ = *p++; *m++ = *p++; *m++ = *p++; *m++ = *p++;
00588             retval = 2;
00589         }
00590         else retval = 1;
00591         if ( t->sign2 < 0 ) retval = - retval;
00592         p = t->pdel;
00593         for ( i = 0; i < t->ad; i++ ) *m++ = *p++;
00594         p = kron;
00595         if ( t->a4 == 4 ) {
00596 /*
00597             Test for dummies in the last position of the e_.
00598 */
00599             stop = p + t->ad + 4;
00600             p += 3;
00601             while ( *p >= AM.mTraceDum && *p <= t->mdum ) {
00602                 m = p + 1;
00603                 do {
00604                     if ( *m == *p ) {
00605                         *p = m[1];
00606                         *m = *--stop;
00607                         m[1] = *--stop;
00608                         break;
00609                     }
00610                     else if ( m[1] == *p ) {
00611                         *p = *m;
00612                         *m = *--stop;
00613                         m[1] = *--stop;
00614                         break;
00615                     }
00616                     else m += 2;
00617                 } while ( m < stop );
00618             }
00619             p++;
00620         }
00621         else stop = p + t->ad;
00622         while ( p < (stop-2) ) {
00623             while ( *p >= AM.mTraceDum && *p <= t->mdum ) {
00624                 m = p + 2;
00625                 do {
00626                     if ( *m == *p ) {
00627                         *p = m[1];
00628                         *m = *--stop;
00629                         m[1] = *--stop;
00630                         break;
00631                     }
00632                     else if ( m[1] == *p ) {
00633                         *p = *m;
00634                         *m = *--stop;
00635                         m[1] = *--stop;
00636                         break;
00637                     }
00638                     else m += 2;
00639                 } while ( m < stop );
00640             }
00641             p++;
00642             while ( *p >= AM.mTraceDum && *p <= t->mdum ) {
00643                 m = p + 1;
00644                 do {
00645                     if ( *m == *p ) {
00646                         *p = m[1];
00647                         *m = *--stop;
00648                         m[1] = *--stop;
00649                         break;
00650                     }
```

```
00651                     else if ( m[1] == *p ) {
00652                         *p = *m;
00653                         *m = *--stop;
00654                         m[1] = *--stop;
00655                         break;
00656                     }
00657                     else m += 2;
00658                 } while ( m < stop );
00659             }
00660             p++;
00661         }
00662         return(retval);
00663     }
00664     if ( number <= 2 ) return(0);
00665     else { goto NextE4; }
00666 }
00667
00668 /*
00669         #] Trace4no :
00670         #[ Trace4 :             WORD Trace4(term,params,num,level)
00671
00672         Generates traces of the string of gamma matrices in 'instring'.
00673
00674         The difference with the routine tracen ( for n dimensions )
00675         lies in the treatment of gamma 5 and the specific form
00676         of the Chisholm identities. The identities used here are
00677         g(mu)*g(a1)*...*g(an)*g(mu)=
00678         n=odd:  -2*g(an)*...*g(a1)    ( reversed order )
00679         n=even: 2*g(an)*g(a1)*...*g(a(n-1))
00680             +2*g(a(n-1))*...*g(a1)*g(an)
00681         There is a special case for n=2 : 4*d(a1,a2)*gi
00682
00683         The main difference with the old fortran version lies in
00684         the recursion that is used here. That cleans up the variables
00685         very much.
00686
00687         The contents of the AT.TMout array are:
00688         length,type,gamma5,gamma's
00689
00690         The space for the vectors in t is at most 14 * number.
00691
00692         The condition params[5] == 0 corresponds to finding gamma6*gamma7
00693         during the pick up of the matrices.
00694
00695 */
00696
00697 int Trace4(PHEAD WORD *term, WORD *params, WORD num, WORD level)
00698 {
00699     GETBIDENTITY
00700     TRACES *t;
00701     WORD *p, *m, number, i;
00702     WORD *OldW;
00703     WORD j, minimum, minimum2, *min, *stopper;
00704     int ret;
00705     OldW = AT.WorkPointer;
00706     if ( AN.numtracesctack >= AN.intracestack ) {
00707         number = AN.intracestack + 2;
00708         t = (TRACES *)Malloc1(number*sizeof(TRACES),"TRACES-struct");
00709         if ( AN.tracestack ) {
00710             for ( i = 0; i < AN.intracestack; i++ ) { t[i] = AN.tracestack[i]; }
00711             M_free(AN.tracestack,"TRACES-struct");
00712         }
00713         AN.tracestack = t;
00714         AN.intracestack = number;
00715     }
00716
00717     number = *params - 6;
00718     if ( number < 0 || ( number & 1 ) || !params[5] ) return(0);
00719
00720     t = AN.tracestack + AN.numtracesctack;
00721     AN.numtracesctack++;
00722
00723     t->finalstep = ( params[2] & 16 ) ? 1 : 0;
00724     t->gamma5 = params[3];
00725     if ( t->finalstep && t->gamma5 != GAMMA1 ) {
00726         MLOCK(ErrorMessageLock);
00727         MesPrint("Gamma5 not allowed in this option of the trace command");
00728         MUNLOCK(ErrorMessageLock);
00729         AN.numtracesctack--;
00730         SETERROR(-1)
00731     }
00732     t->inlist = AT.WorkPointer;
00733     t->accup = t->accu = t->inlist + number;
00734     t->perm = t->accu + (number*2);
00735     t->eers = t->perm + number;
00736     if ( ( AT.WorkPointer += 19 * number ) >= AT.WorkTop ) {
00737         MLOCK(ErrorMessageLock);
```

```
00738         MesWork();
00739         MUNLOCK(ErrorMessageLock);
00740         return(-1);
00741     }
00742     t->num = num;
00743     t->level = level;
00744     p = t->inlist;
00745     m = params+6;
00746     for ( i = 0; i < number; i++ ) *p++ = *m++;
00747     t->termp = term;
00748     t->factor = params[4];
00749     t->allsign = params[5];
00750     if ( number >= 10 || ( t->gamma5 != GAMMA1 && number > 4 ) ) {
00751 /*
00752         The next code should `normal order' the string
00753         We need the lexicographic smallest string, taking also the
00754         reverse strings into account.
00755 */
00756         minimum = 0; min = t->inlist;
00757         stopper = min + number;
00758         for ( i = 1; i < number; i++ ) {
00759             p = min;
00760             m = t->inlist + i;
00761             for ( j = 0; j < number; j++ ) {
00762                 if ( *p < *m ) break;
00763                 if ( *p > *m ) {
00764                     min = t->inlist+i;
00765                     minimum = i;
00766                     break;
00767                 }
00768                 p++; m++;
00769                 if ( m >= stopper ) m = t->inlist;
00770                 if ( p >= stopper ) p = t->inlist;
00771             }
00772         }
00773         p = min;
00774         min = m = AT.WorkPointer;
00775         i = number;
00776         while ( --i >= 0 ) {
00777             *m++ = *p++;
00778             if ( p >= stopper ) p = t->inlist;
00779         }
00780         p = t->inlist;
00781         m = min;
00782         i = number;
00783         while ( --i >= 0 ) *p++ = *m++;
00784         p = t->inlist;
00785         m = stopper;
00786         while ( p < m ) {   /* reverse string */
00787             i = *p; *p++ = *--m; *m = i;
00788         }
00789         minimum2 = 0;
00790         for ( i = 0; i < number; i++ ) {
00791             p = min;
00792             m = t->inlist + i;
00793             for ( j = 0; j < number; j++ ) {
00794                 if ( *p < *m ) break;
00795                 if ( *p > *m ) {
00796                     m = t->inlist + i;
00797                     p = min;
00798                     j = number;
00799                     while ( --j >= 0 ) {
00800                         *p++ = *m++;
00801                         if ( m >= stopper ) m = t->inlist;
00802                     }
00803                     minimum2 = i;
00804                     break;
00805                 }
00806                 p++; m++;
00807                 if ( m >= stopper ) m = t->inlist;
00808             }
00809         }
00810         minimum ^= minimum2;
00811         if ( ( minimum & 1 ) != 0 ) {
00812             if ( t->gamma5 == GAMMA5 ) t->allsign = - t->allsign;
00813             else if ( t->gamma5 != GAMMA1 )
00814                 t->gamma5 = GAMMA6 + GAMMA7 - t->gamma5;
00815         }
00816         p = min; m = t->inlist; i = number;
00817         while ( --i >= 0 ) *m++ = *p++;
00818 /*
00819         Now the trace is in normal order
00820 */
00821     }
00822     ret = Trace4Gen(BHEAD t,number);
00823     AT.WorkPointer = OldW;
00824     AN.numtracesctack--;
```

```
00825     return(ret);
00826 }
00827
00828 /*
00829         #] Trace4 :
00830         #[ Trace4Gen :          WORD Trace4Gen(t,number)
00831
00832         The recursive breakdown of a trace in 4 dimensions.
00833         We test first whether the trace has zero or two gamma's left.
00834         This case can be done quickly.
00835         Next we test whether we can eliminate adjacent objects that are
00836         the same.
00837         Then we test for Chisholm identities (I). First for identities
00838         with an odd number of gamma matrices (II), then for those with an
00839         even number of matrices (III). The special thing here is the demand
00840         that the contraction be between indices with 4 dimensions only.
00841         Then there is a scan for objects that are the same, not regarding
00842         their type (IV). This is exactly the same as in n dimensions.
00843         Finally we have a string left in which all objects are different (V).
00844         This case is treated by the routine Trace4no (no stands for no
00845         objects are the same).
00846
00847         In case I we have one d_ of which the result of the contraction
00848         has not yet been fixed.
00849         Case II gives just a reordering of the matrices and a factor -2.
00850         Case III gives two terms: one for the anti commutation, such that
00851         the number of intermediate matrices becomes odd and the other
00852         from the Chisholm rule for an odd number of matrices. Both have
00853         a factor 2.
00854         Case IV gives m+1 terms when m is the number of matrices inbetween.
00855         We take the shortest path. The sign alternates and all terms have
00856         a factor two, except for the last one.
00857
00858 */
00859
00860 int Trace4Gen(PHEAD TRACES *t, WORD number)
00861 {
00862     GETBIDENTITY
00863     WORD *termout, *stop;
00864     WORD *p, *m, oldval;
00865     WORD *pold, *mold, diff, *oldstring, cp;
00866 /*
00867             #[ Special cases :
00868 */
00869     if ( number <= 2 ) {    /* Special cases */
00870         if ( t->gamma5 == GAMMA5 ) return(0);
00871         termout = AT.WorkPointer;
00872         p = t->termp;
00873         stop = p + *p;
00874         m = termout;
00875         p++;
00876         if ( p < stop ) do {
00877             if ( *p == SUBEXPRESSION && p[2] == t->num ) {
00878                 oldstring = p;
00879                 p = t->termp;
00880                 do { *m++ = *p++; } while ( p < oldstring );
00881                 p += p[1];
00882                 *m++ = AC.lUniTrace[0];
00883                 *m++ = AC.lUniTrace[1];
00884                 *m++ = AC.lUniTrace[2];
00885                 *m++ = AC.lUniTrace[3];
00886                 if ( number == 2 || t->accup > t->accu ) {
00887                     oldstring = m;
00888                     *m++ = DELTA;
00889                     *m++ = 4;
00890                     if ( number == 2 ) {
00891                         *m++ = t->inlist[0];
00892                         *m++ = t->inlist[1];
00893                     }
00894                     if ( t->accup > t->accu ) {
00895                         pold = p;
00896                         p = t->accu;
00897                         while ( p < t->accup ) *m++ = *p++;
00898                         oldstring[1] = WORDDIF(m,oldstring);
00899                         p = pold;
00900                     }
00901                 }
00902                 if ( t->factor ) {
00903                     *m++ = SNUMBER;
00904                     *m++ = 4;
00905                     *m++ = 2;
00906                     *m++ = t->factor;
00907                 }
00908                 do { *m++ = *p++; } while ( p < stop );
00909                 *termout = WORDDIF(m,termout);
00910                 if ( t->allsign < 0 ) m[-1] = -m[-1];
00911                 if ( ( AT.WorkPointer = m ) > AT.WorkTop ) {
```

```
00912                     MLOCK(ErrorMessageLock);
00913                     MesWork();
00914                     MUNLOCK(ErrorMessageLock);
00915                     return(-1);
00916                 }
00917                 *AN.RepPoint = 1;
00918                 AR.expchanged = 1;
00919                 if ( *termout ) {
00920                     *AN.RepPoint = 1;
00921                     AR.expchanged = 1;
00922                     if ( Generator(BHEAD termout,t->level) ) goto TracCall;
00923                 }
00924                 AT.WorkPointer= termout;
00925                 return(0);
00926             }
00927             p += p[1];
00928         } while ( p < stop );
00929         return(0);
00930     }
00931 /*
00932             #] Special cases :
00933             #[ Adjacent objects :
00934 */
00935     p = t->inlist;
00936     stop = p + number - 1;
00937     if ( *p == *stop ) {        /* First and last of string */
00938         oldval = *p;
00939         *(t->accup)++ = *p;
00940         *(t->accup)++ = *p;
00941         m = p+1;
00942         while ( m < stop ) *p++ = *m++;
00943         if ( t->gamma5 != GAMMA1 ) {
00944             if ( t->gamma5 == GAMMA5 ) t->allsign = - t->allsign;
00945             else if ( t->gamma5 == GAMMA6 ) t->gamma5 = GAMMA7;
00946             else if ( t->gamma5 == GAMMA7 ) t->gamma5 = GAMMA6;
00947         }
00948         if ( Trace4Gen(BHEAD t,number-2) ) goto TracCall;
00949         t = AN.tracestack + AN.numtracesctack - 1;
00950         if ( t->gamma5 != GAMMA1 ) {
00951             if ( t->gamma5 == GAMMA5 ) t->allsign = - t->allsign;
00952             else if ( t->gamma5 == GAMMA6 ) t->gamma5 = GAMMA7;
00953             else if ( t->gamma5 == GAMMA7 ) t->gamma5 = GAMMA6;
00954         }
00955         while ( p > t->inlist ) *--m = *--p;
00956         *p = *stop = oldval;
00957         t->accup -= 2;
00958         return(0);
00959     }
00960     do {
00961         if ( *p == p[1] ) {     /* Adjacent in string */
00962             oldval = *p;
00963             pold = p;
00964             m = p+2;
00965             *(t->accup)++ = *p;
00966             *(t->accup)++ = *p;
00967             while ( m <= stop ) *p++ = *m++;
00968             if ( Trace4Gen(BHEAD t,number-2) ) goto TracCall;
00969             t = AN.tracestack + AN.numtracesctack - 1;
00970             while ( p > pold ) *--m = *--p;
00971             *p++ = oldval;
00972             *p++ = oldval;
00973             t->accup -= 2;
00974             return(0);
00975         }
00976         p++;
00977     } while ( p < stop );
00978 /*
00979             #] Adjacent objects :
00980             #[ Odd Contraction :
00981 */
00982     p = t->inlist;
00983     stop = p + number;
00984     do {
00985         if ( *p >= AM.OffsetIndex && (
00986         ( *p < WILDOFFSET + AM.OffsetIndex &&
00987          indices[*p-AM.OffsetIndex].dimension == 4 )
00988         || ( *p >= WILDOFFSET + AM.OffsetIndex && AC.lDefDim == 4 ) ) ) {
00989             m = p+2;
00990             while ( m < stop ) {
00991                 if ( *p == *m ) {
00992                     pold = p;
00993                     mold = m;
00994                     oldval = *p;
00995                     (t->factor)++;
00996                     t->allsign = - t->allsign;
00997                     *p++ = *--m;
00998                     m--;
```

```
00999                   while ( m > p ) { diff = *p; *p++ = *m; *m-- = diff; }
01000                   p = mold - 1;
01001                   m = mold + 1;
01002                   while ( m < stop ) *p++ = *m++;
01003                   if ( Trace4Gen(BHEAD t,number-2) ) goto TracCall;
01004                   t = AN.tracestack + AN.numtracesctack - 1;
01005                   m--;
01006                   while ( m > mold ) *m-- = *--p;
01007                   p = pold;
01008                   *m-- = oldval;
01009                   *m-- = *p;
01010                   *p++ = oldval;
01011                   while ( m > p ) { diff = *p; *p++ = *m; *m-- = diff; }
01012                   t->allsign = - t->allsign;
01013                   (t->factor)--;
01014                   return(0);
01015               }
01016               m += 2;
01017           }
01018       }
01019       p++;
01020   } while ( p < stop );
01021 /*
01022           #] Odd Contraction :
01023           #[ Even Contraction :
01024       First the case with two matrices inbetween.
01025 */
01026   p = t->inlist;
01027   stop = p + number;
01028   do {
01029       if ( *p >= AM.OffsetIndex && (
01030       ( *p < WILDOFFSET + AM.OffsetIndex &&
01031        indices[*p-AM.OffsetIndex].dimension == 4 )
01032       || ( *p >= WILDOFFSET + AM.OffsetIndex && AC.lDefDim == 4 ) ) ) {
01033           m = p+3;
01034           if ( m >= stop ) m -= number;
01035           if ( *p == *m ) {
01036               WORD oldfactor, old5;
01037               oldstring = AT.WorkPointer;
01038               AT.WorkPointer += number;
01039               oldfactor = t->allsign;
01040               old5 = t->gamma5;
01041               if ( m < p ) cp = (WORDDIF(m,t->inlist) + 1 ) & 1;
01042               else cp = 0;
01043               if ( cp && ( t->gamma5 != GAMMA1 ) ) {
01044                   if ( t->gamma5 == GAMMA5 ) t->allsign = -t->allsign;
01045                   else if ( t->gamma5 == GAMMA6 ) t->gamma5 = GAMMA7;
01046                   else if ( t->gamma5 == GAMMA7 ) t->gamma5 = GAMMA6;
01047               }
01048               mold = m;
01049               p = oldstring;
01050               m = t->inlist;
01051               while ( m < stop ) *p++ = *m++;
01052 /*
01053       Rotate the string
01054 */
01055               m = mold + 1;
01056               p = t->inlist;
01057               while ( m < stop ) *p++ = *m++;
01058               m = oldstring;
01059               if ( !cp && ((WORDDIF(stop,p))&1) != 0 && ( t->gamma5 != GAMMA1 ) ) {
01060                   if ( t->gamma5 == GAMMA5 ) t->allsign = -t->allsign;
01061                   else if ( t->gamma5 == GAMMA6 ) t->gamma5 = GAMMA7;
01062                   else if ( t->gamma5 == GAMMA7 ) t->gamma5 = GAMMA6;
01063               }
01064               while ( p < stop ) *p++ = *m++;
01065               t->factor += 2;
01066               m = p - 3;
01067               p = t->inlist;
01068               oldval = number - 4;
01069               while ( oldval > 0 ) {
01070                   if ( *p >= AM.OffsetIndex && (
01071                   ( *p < WILDOFFSET + AM.OffsetIndex &&
01072                    indices[*p-AM.OffsetIndex].dimension )
01073                   || ( *p >= WILDOFFSET + AM.OffsetIndex && AC.lDefDim ) ) ) {
01074                       if ( *p == *m ) {
01075                           *p = m[1];
01076                           break;
01077                       }
01078                       else if ( *p == m[1] ) {
01079                           *p = *m;
01080                           break;
01081                       }
01082                   }
01083                   p++; oldval--;
01084               }
01085               if ( oldval <= 0 ) {
```

```
01086                   *(t->accup)++ = *m++;
01087                   *(t->accup)++ = *m++;
01088               }
01089               if ( Trace4Gen(BHEAD t,number-4) ) goto TracCall;
01090               t = AN.tracestack + AN.numtracesctack - 1;
01091               t->factor -= 2;
01092               if ( oldval <= 0 ) t->accup -= 2;
01093               t->gamma5 = old5;
01094               t->allsign = oldfactor;
01095               AT.WorkPointer = p = oldstring;
01096               m = t->inlist;
01097               while ( m < stop ) *m++ = *p++;
01098               return(0);
01099           }
01100       }
01101       p++;
01102   } while ( p < stop );
01103 /*
01104       The case with at least 4 matrices inbetween.
01105 */
01106   p = t->inlist;
01107   stop = p + number;
01108   do {
01109       if ( *p >= AM.OffsetIndex && (
01110       ( *p < WILDOFFSET + AM.OffsetIndex &&
01111        indices[*p-AM.OffsetIndex].dimension == 4 )
01112       || ( *p >= WILDOFFSET + AM.OffsetIndex && AC.lDefDim == 4 ) ) ) {
01113           m = p+5;
01114           while ( m < stop ) {
01115               if ( *p == *m ) {
01116                   WORD *pex, *mex;
01117                   pold = p;
01118                   mold = m;
01119                   oldval = *p;
01120 /*
01121           g_(1,mu)*g_(1,a1)*...*g_(1,aj)*g_(1,an)*g_(1,mu) ->
01122           first:
01123           2*g_(1,an)*g_(1,a1)*...*g_(1,aj)
01124 */
01125                   (t->factor)++;
01126 /*
01127                   The variable hulp seems unnecessary
01128                   *p = hulp = m[-1];
01129 */
01130                   *p = m[-1];
01131                   p = m - 1;
01132                   m++;
01133                   while ( m < stop ) *p++ = *m++;
01134                   if ( Trace4Gen(BHEAD t,number-2) ) goto TracCall;
01135                   t = AN.tracestack + AN.numtracesctack - 1;
01136                   pex = p; mex = m;
01137                   p = pold;
01138                   m = mold - 2;
01139                   while ( m > p ) { diff = *p; *p++ = *m; *m-- = diff; }
01140 /*
01141           and then:
01142           2*g_(1,aj)*...*g_(1,a1)*g_(1,an)
01143 */
01144                   if ( Trace4Gen(BHEAD t,number-2) ) goto TracCall;
01145                   t = AN.tracestack + AN.numtracesctack - 1;
01146                   p = pold;
01147                   m = mold - 2;
01148                   while ( m > p ) { diff = *p; *p++ = *m; *m-- = diff; }
01149                   m = mex;
01150                   p = pex;
01151                   m--;
01152                   while ( m > mold ) *m-- = *--p;
01153                   m = mold;
01154                   *m-- = oldval;
01155                   p = pold;
01156                   *m = *p;
01157                   *p = oldval;
01158                   (t->factor)--;
01159                   return(0);
01160               }
01161               m += 2;
01162           }
01163       }
01164       p++;
01165   } while ( p < stop );
01166 /*
01167           #] Even Contraction :
01168           #[ Same Objects :
01169 */
01170   p = t->inlist;
01171   stop = p + number - 1;
01172   diff = 2;
```

```
01173     do {
01174         p = t->inlist;
01175         while ( p <= stop ) {
01176             m = p + diff;
01177             if ( m > stop ) m -= number;
01178             if ( *p == *m ) {
01179                 WORD oldfactor, c, old5;
01180                 oldfactor = t->allsign;
01181                 old5 = t->gamma5;
01182                 cp = (WORDDIF(m,t->inlist)) & 1;
01183                 if ( !cp && ( t->gamma5 != GAMMA1 ) ) {
01184                     if ( t->gamma5 == GAMMA5 ) t->allsign = -t->allsign;
01185                     else if ( t->gamma5 == GAMMA6 ) t->gamma5 = GAMMA7;
01186                     else if ( t->gamma5 == GAMMA7 ) t->gamma5 = GAMMA6;
01187                 }
01188                 oldstring = AT.WorkPointer;
01189                 AT.WorkPointer += number;
01190                 mold = m;
01191                 oldval = *p;
01192                 p = oldstring;
01193                 m = t->inlist;
01194                 while ( m <= stop ) *p++ = *m++;
01195 /*
01196         Rotate the string
01197 */
01198                 m = mold + 1;
01199                 p = t->inlist;
01200                 while ( m <= stop ) *p++ = *m++;
01201                 m = oldstring;
01202                 while ( p <= stop ) *p++ = *m++;
01203                 (t->factor)++;
01204                 p -= diff + 1;
01205                 m = stop;
01206                 *(t->accup) = oldval;
01207                 t->accup += 2;
01208                 m--;
01209                 while ( m > p ) {
01210                     c = t->accup[-1];
01211                     t->accup[-1] = *m;
01212                     *m = c;
01213                     if ( Trace4Gen(BHEAD t,number-2) ) goto Trac4Call;
01214                     t = AN.tracestack + AN.numtracesctack - 1;
01215                     m--;
01216                     t->allsign = - t->allsign;
01217                 }
01218                 c = t->accup[-1];
01219                 t->accup[-1] = *m;
01220                 *m = c;
01221                 (t->factor)--;
01222                 if ( Trace4Gen(BHEAD t,number-2) ) goto Trac4Call;
01223                 t = AN.tracestack + AN.numtracesctack - 1;
01224                 t->accup -= 2;
01225                 t->allsign = oldfactor;
01226                 AT.WorkPointer = p = oldstring;
01227                 m = t->inlist;
01228                 while ( m <= stop ) *m++ = *p++;
01229                 t->gamma5 = old5;
01230                 return(0);
01231             }
01232             p++;
01233         }
01234     } while ( ++diff <= (number>>1) );
01235 /*
01236             #] Same Objects :
01237             #[ All Different :
01238
01239         Here we have a string with all different objects.
01240
01241 */
01242     t->sgn = 0;
01243     termout = AT.WorkPointer;
01244     for(;;) {
01245         if ( t->finalstep == 0 ) diff = Trace4no(number,t->accup,t);
01246         else                     diff = TraceNno(number,t->accup,t);
01247 /*  while ( ( diff = Trace4no(number,t->accup,t) ) != 0 ) */
01248         if ( diff == 0 ) break;
01249         p = t->termp;
01250         stop = p + *p;
01251         m = termout;
01252         p++;
01253         if ( p < stop ) do {
01254             if ( *p == SUBEXPRESSION && p[2] == t->num ) {
01255                 oldstring = p;
01256                 p = t->termp;
01257                 do { *m++ = *p++; } while ( p < oldstring );
01258                 p += p[1];
01259                 pold = p;
```

```
01260                 *m++ = AC.lUniTrace[0];
01261                 *m++ = AC.lUniTrace[1];
01262                 *m++ = AC.lUniTrace[2];
01263                 *m++ = AC.lUniTrace[3];
01264                 *m++ = SNUMBER;
01265                 *m++ = 4;
01266                 *m++ = 2;
01267                 *m++ = t->factor;
01268                 p = t->accup;
01269                 oldval = number;
01270                 if ( diff == 2 || diff == -2 ) {
01271                     *m++ = LEVICIVITA;
01272                     *m++ = 4+FUNHEAD;
01273                     *m++ = DIRTYFLAG;
01274                     FILLFUN3(m)
01275                     *m++ = *p++; *m++ = *p++; *m++ = *p++; *m++ = *p++;
01276                     oldval -= 4;
01277                 }
01278                 if ( oldval > 0 || t->accup > t->accu ) {
01279                     oldstring = m;
01280                     *m++ = DELTA;
01281                     *m++ = oldval + 2;
01282                     if ( oldval > 0 ) NCOPY(m,p,oldval);
01283                     if ( t->accup > t->accu ) {
01284                         p = t->accu;
01285                         while ( p < t->accup ) *m++ = *p++;
01286                         oldstring[1] = WORDDIF(m,oldstring);
01287                     }
01288                 }
01289                 p = pold;
01290                 do { *m++ = *p++; } while ( p < stop );
01291                 *termout = WORDDIF(m,termout);
01292                 if ( ( diff ^ t->allsign ) < 0 ) m[-1] = - m[-1];
01293                 if ( ( AT.WorkPointer = m ) > AT.WorkTop ) {
01294                     MLOCK(ErrorMessageLock);
01295                     MesWork();
01296                     MUNLOCK(ErrorMessageLock);
01297                     return(-1);
01298                 }
01299                 if ( *termout ) {
01300                     *AN.RepPoint = 1;
01301                     AR.expchanged = 1;
01302                     if ( Generator(BHEAD termout,t->level) ) {
01303                         AT.WorkPointer = termout;
01304                         goto TracCall;
01305                     }
01306                     t = AN.tracestack + AN.numtracesctack - 1;
01307                 }
01308                 break;
01309             }
01310             p += p[1];
01311         } while ( p < stop );
01312     }
01313     AT.WorkPointer = termout;
01314     return(0);
01315
01316 /*
01317             #] All Different :
01318 */
01319 Trac4Call:
01320     AT.WorkPointer = oldstring;
01321 TracCall:
01322     if ( AM.tracebackflag ) {
01323         MLOCK(ErrorMessageLock);
01324         MesCall("Trace4Gen");
01325         MUNLOCK(ErrorMessageLock);
01326     }
01327     return(-1);
01328 }
01329
01330 /*
01331         #] Trace4Gen :
01332         #[ TraceNno :           WORD TraceNno(number,kron,t)
01333
01334         Routine takes the trace in N dimensions of a string
01335         of gamma matrices. It is assumed that there are no
01336         contractions and no vectors that are the same. For
01337         the treatment of those cases there are special routines,
01338         that call this routine as a final stage.
01339         The calling routine must reserve 'number' WORDs for perm
01340         and kron each.
01341         kron and perm may not be altered during the generation!
01342
01343 */
01344
01345 WORD TraceNno(WORD number, WORD *kron, TRACES *t)
01346 {
```

```
01347     WORD i, j, a, *p;
01348     if ( !t->sgn ) {
01349         if ( !number || ( number & 1 ) ) return(0);
01350         p = t->inlist;
01351         for ( i = 0; i < number; i++ ) {
01352             t->perm[i] = i;
01353             *kron++ = *p++;
01354         }
01355         t->sgn = 1;
01356         return(1);
01357     }
01358     else {
01359         i = number - 3;
01360         while ( i > 0 ) {
01361             a = kron[i];
01362             p = t->perm + i;
01363             for ( j = i + 1; j <= *p; j++ ) kron[j-1] = kron[j];
01364             kron[(*p)++] = a;
01365             if ( *p < number ) {
01366                 a = kron[*p];
01367                 j = *p;
01368                 while ( j >= (i+1) ) { kron[j] = kron[j-1]; j--; }
01369                 kron[i] = a;
01370                 number -= 2;
01371                 for ( j = i+2; j < number; j += 2 ) t->perm[j] = j;
01372                 t->sgn = - t->sgn;
01373                 return(t->sgn);
01374             }
01375             i -= 2;
01376         }
01377     }
01378     return(0);
01379 }
01380
01381 /*
01382         #] TraceNno :
01383         #[ TraceN :             WORD TraceN(term,params,num,level)
01384 */
01385
01386 int TraceN(PHEAD WORD *term, WORD *params, WORD num, WORD level)
01387 {
01388     GETBIDENTITY
01389     TRACES *t;
01390     WORD *p, *m, number, i;
01391     WORD *OldW;
01392     int ret;
01393     if ( params[3] != GAMMA1 ) {
01394         MLOCK(ErrorMessageLock);
01395         MesPrint("Gamma5 not allowed in n-trace");
01396         MUNLOCK(ErrorMessageLock);
01397         SETERROR(-1)
01398     }
01399     OldW = AT.WorkPointer;
01400     if ( AN.numtracesctack >= AN.intracestack ) {
01401         number = AN.intracestack + 2;
01402         t = (TRACES *)Malloc1(number*sizeof(TRACES),"TRACES-struct");
01403         if ( AN.tracestack ) {
01404             for ( i = 0; i < AN.intracestack; i++ ) { t[i] = AN.tracestack[i]; }
01405             M_free(AN.tracestack,"TRACES-struct");
01406         }
01407         AN.tracestack = t;
01408         AN.intracestack = number;
01409     }
01410     number = *params - 6;
01411     if ( number < 0 || ( number & 1 ) || !params[5] ) return(0);
01412
01413     t = AN.tracestack + AN.numtracesctack;
01414     AN.numtracesctack++;
01415
01416     t->inlist = AT.WorkPointer;
01417     t->accup = t->accu = t->inlist + number;
01418     t->perm = t->accu + number;
01419     if ( ( AT.WorkPointer += 3 * number ) >= AT.WorkTop ) {
01420         AN.numtracesctack--;
01421         MLOCK(ErrorMessageLock);
01422         MesWork();
01423         MUNLOCK(ErrorMessageLock);
01424         return(-1);
01425     }
01426     t->num = num;
01427     t->level = level;
01428     p = t->inlist;
01429     m = params+6;
01430     for ( i = 0; i < number; i++ ) *p++ = *m++;
01431     t->termp = term;
01432     t->factor = params[4];
01433     t->allsign = params[5];
```

```
01434     ret = TraceNgen(BHEAD t,number);
01435     AT.WorkPointer = OldW;
01436     AN.numtracesctack--;
01437     return(ret);
01438 }
01439
01440 /*
01441         #] TraceN :
01442         #[ TraceNgen :          WORD TraceNgen(t,number)
01443
01444         This routine is a simplified version of Trace4Gen. We know here
01445         only three cases: Adjacent objects, same objects and all different.
01446         The other difference lies of course in the struct which is now
01447         not of type TRACES, but of type TRACES.
01448
01449 */
01450
01451 int TraceNgen(PHEAD TRACES *t, WORD number)
01452 {
01453     GETBIDENTITY
01454     WORD *termout, *stop;
01455     WORD *p, *m, oldval;
01456     WORD *pold, *mold, diff, *oldstring;
01457 /*
01458             #[ Special cases :
01459 */
01460     if ( number <= 2 ) {    /* Special cases */
01461         termout = AT.WorkPointer;
01462         p = t->termp;
01463         stop = p + *p;
01464         m = termout;
01465         p++;
01466         if ( p < stop ) do {
01467             if ( *p == SUBEXPRESSION && p[2] == t->num ) {
01468                 oldstring = p;
01469                 p = t->termp;
01470                 do { *m++ = *p++; } while ( p < oldstring );
01471                 p += p[1];
01472                 *m++ = AC.lUniTrace[0];
01473                 *m++ = AC.lUniTrace[1];
01474                 *m++ = AC.lUniTrace[2];
01475                 *m++ = AC.lUniTrace[3];
01476                 if ( number == 2 || t->accup > t->accu ) {
01477                     oldstring = m;
01478                     *m++ = DELTA;
01479                     *m++ = 4;
01480                     if ( number == 2 ) {
01481                         *m++ = t->inlist[0];
01482                         *m++ = t->inlist[1];
01483                     }
01484                     if ( t->accup > t->accu ) {
01485                         pold = p;
01486                         p = t->accu;
01487                         while ( p < t->accup ) *m++ = *p++;
01488                         oldstring[1] = WORDDIF(m,oldstring);
01489                         p = pold;
01490                     }
01491                 }
01492                 if ( t->factor ) {
01493                     *m++ = SNUMBER;
01494                     *m++ = 4;
01495                     *m++ = 2;
01496                     *m++ = t->factor;
01497                 }
01498                 do { *m++ = *p++; } while ( p < stop );
01499                 *termout = WORDDIF(m,termout);
01500                 if ( t->allsign < 0 ) m[-1] = -m[-1];
01501                 if ( ( AT.WorkPointer = m ) > AT.WorkTop ) {
01502                     MLOCK(ErrorMessageLock);
01503                     MesWork();
01504                     MUNLOCK(ErrorMessageLock);
01505                     return(-1);
01506                 }
01507                 if ( *termout ) {
01508                     *AN.RepPoint = 1;
01509                     AR.expchanged = 1;
01510                     if ( Generator(BHEAD termout,t->level) ) goto TracCall;
01511                 }
01512                 AT.WorkPointer= termout;
01513                 return(0);
01514             }
01515             p += p[1];
01516         } while ( p < stop );
01517         return(0);
01518     }
01519 /*
01520             #] Special cases :
```

```
01521             #[ Adjacent objects :
01522 */
01523     p = t->inlist;
01524     stop = p + number - 1;
01525     if ( *p == *stop ) {        /* First and last of string */
01526         oldval = *p;
01527         *(t->accup)++ = *p;
01528         *(t->accup)++ = *p;
01529         m = p+1;
01530         while ( m < stop ) *p++ = *m++;
01531         if ( TraceNgen(BHEAD t,number-2) ) goto TracCall;
01532         t = AN.tracestack + AN.numtracesctack - 1;
01533         while ( p > t->inlist ) *--m = *--p;
01534         *p = *stop = oldval;
01535         t->accup -= 2;
01536         return(0);
01537     }
01538     do {
01539         if ( *p == p[1] ) {     /* Adjacent in string */
01540             oldval = *p;
01541             pold = p;
01542             m = p+2;
01543             *(t->accup)++ = *p;
01544             *(t->accup)++ = *p;
01545             while ( m <= stop ) *p++ = *m++;
01546             if ( TraceNgen(BHEAD t,number-2) ) goto TracCall;
01547             t = AN.tracestack + AN.numtracesctack - 1;
01548             while ( p > pold ) *--m = *--p;
01549             *p++ = oldval;
01550             *p++ = oldval;
01551             t->accup -= 2;
01552             return(0);
01553         }
01554         p++;
01555     } while ( p < stop );
01556 /*
01557             #] Adjacent objects :
01558             #[ Same Objects :
01559 */
01560     p = t->inlist;
01561     stop = p + number - 1;
01562     diff = 2;
01563     do {
01564         p = t->inlist;
01565         while ( p <= stop ) {
01566             m = p + diff;
01567             if ( m > stop ) m -= number;
01568             if ( *p == *m ) {
01569                 WORD oldfactor, c;
01570                 oldstring = AT.WorkPointer;
01571                 AT.WorkPointer += number;
01572                 mold = m;
01573                 oldval = *p;
01574                 p = oldstring;
01575                 m = t->inlist;
01576                 while ( m <= stop ) *p++ = *m++;
01577 /*
01578         Rotate the string
01579 */
01580                 {
01581                     m = mold + 1;
01582                     p = t->inlist;
01583                     while ( m <= stop ) *p++ = *m++;
01584                     m = oldstring;
01585                     while ( p <= stop ) *p++ = *m++;
01586                 }
01587                 oldfactor = t->allsign;
01588                 (t->factor)++;
01589                 p -= diff + 1;
01590                 m = stop;
01591                 if ( oldval >= ( AM.OffsetIndex + WILDOFFSET ) ||
01592                     ( oldval >= AM.OffsetIndex
01593                     && indices[oldval-AM.OffsetIndex].dimension ) ) {
01594                     m--;
01595 /*
01596         We distinguish 4 cases:
01597         m-p=1   Use g_(1,mu,a,mu) = (2-d_(mu,mu))*g_(1,a)
01598         m-p=2   Use g_(1,mu,a,b,mu) = 4*d_(a,b)+(d_(mu,mu)-4)*g_(1,a,b)
01599         m-p=3   Use g_(1,mu,a,b,c,mu) = -2*g_(1,c,b,a)
01600                         -(d_(mu,mu)-4)*g_(1,a,b,c)
01601         m-p>3   Reduce down to m-p=3 with the old technique
01602 */
01603                     while ( m > (p+3) ) {
01604                         c = *p;
01605                         *p = *m;
01606                         *m = c;
01607                         if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
```

```
01608                       t = AN.tracestack + AN.numtracesctack - 1;
01609                       m--;
01610                       t->allsign = - t->allsign;
01611                   }
01612                   switch ( WORDDIF(m,p) ) {
01613                       case 1:
01614                           c = *p;
01615                           *p = *m;
01616                           *m = c;
01617                           if ( oldval < ( AM.OffsetIndex + WILDOFFSET )
01618                           && indices[oldval-AM.OffsetIndex].nmin4
01619                           != -NMIN4SHIFT ) {
01620                               t->allsign = - t->allsign;
01621                               if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01622                               t = AN.tracestack + AN.numtracesctack - 1;
01623                               (t->factor)--;
01624                               *(t->accup)++ = SUMMEDIND;
01625                               *(t->accup)++ =
01626                                   indices[oldval-AM.OffsetIndex].nmin4;
01627                           }
01628                           else
01629                           {
01630                               if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01631                               t = AN.tracestack + AN.numtracesctack - 1;
01632                               t->allsign = - t->allsign;
01633                               (t->factor)--;
01634                               *(t->accup)++ = oldval;
01635                               *(t->accup)++ = oldval;
01636                           }
01637                           if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01638                           t = AN.tracestack + AN.numtracesctack - 1;
01639                           t->accup -= 2;
01640                           break;
01641                       case 2:
01642                           { WORD one, two;
01643                           one = *p = p[1];
01644                           two = p[1] = *m;
01645                           (t->factor)++;          /* 4 */
01646                           *(t->accup)++ = *p;     /* d_(a,b) */
01647                           *(t->accup)++ = *m;
01648                           if ( TraceNgen(BHEAD t,number-4) ) goto TracnCall;
01649                           t = AN.tracestack + AN.numtracesctack - 1;
01650                           *p = one; p[1] = two;
01651                           t->accup -= 2;
01652                           if ( oldval < ( AM.OffsetIndex + WILDOFFSET )
01653                           && indices[oldval-AM.OffsetIndex].nmin4
01654                           != -NMIN4SHIFT ) {
01655                               t->factor -= 2;
01656                               *(t->accup)++ = SUMMEDIND;
01657                               *(t->accup)++ =
01658                                   indices[oldval-AM.OffsetIndex].nmin4;
01659                           }
01660                           else {
01661                               t->allsign = - t->allsign;
01662                               if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01663                               t = AN.tracestack + AN.numtracesctack - 1;
01664                               t->allsign = - t->allsign;
01665                               t->factor -= 2;
01666                               *(t->accup)++ = oldval;
01667                               *(t->accup)++ = oldval;
01668                           }
01669                           if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01670                           t = AN.tracestack + AN.numtracesctack - 1;
01671                           t->accup -= 2;
01672                           }
01673                           break;
01674                       default:
01675                           c = *p;
01676                           *p = *m;
01677                           *m = c;
01678                           c = m[-1]; m[-1] = m[-2]; m[-2] = c;
01679                           t->allsign = - t->allsign;
01680                           if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01681                           t = AN.tracestack + AN.numtracesctack - 1;
01682                           m--;
01683                           c = *p;
01684                           *p = *m;
01685                           *m = c;
01686                           (t->factor)--;
01687                           if ( oldval < ( AM.OffsetIndex + WILDOFFSET )
01688                           && indices[oldval-AM.OffsetIndex].nmin4
01689                           != -NMIN4SHIFT ) {
01690                               *(t->accup)++ = SUMMEDIND;
01691                               *(t->accup)++ =
01692                                   indices[oldval-AM.OffsetIndex].nmin4;
01693                               if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01694                               t = AN.tracestack + AN.numtracesctack - 1;
```

```
01695                                   t->accup -= 2;
01696                                   t->allsign = - t->allsign;
01697                               }
01698                               else
01699                               {
01700                                   *(t->accup)++ = oldval;
01701                                   *(t->accup)++ = oldval;
01702                                   if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01703                                   t = AN.tracestack + AN.numtracesctack - 1;
01704                                   t->accup -= 2;
01705                                   t->allsign = - t->allsign;
01706                                   t->factor += 2;
01707                                   if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01708                                   t = AN.tracestack + AN.numtracesctack - 1;
01709                                   t->factor -= 2;
01710                               }
01711                               break;
01712                           }
01713                       }
01714                       else {
01715                           *(t->accup) = oldval;
01716                           t->accup += 2;
01717                           m--;
01718                           while ( m > p ) {
01719                               c = t->accup[-1];
01720                               t->accup[-1] = *m;
01721                               *m = c;
01722                               if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01723                               t = AN.tracestack + AN.numtracesctack - 1;
01724                               m--;
01725                               t->allsign = - t->allsign;
01726                           }
01727                           c = t->accup[-1];
01728                           t->accup[-1] = *m;
01729                           *m = c;
01730                           (t->factor)--;
01731                           if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01732                           t = AN.tracestack + AN.numtracesctack - 1;
01733                           t->accup -= 2;
01734                       }
01735                       t->allsign = oldfactor;
01736                       p = oldstring;
01737                       m = t->inlist;
01738                       while ( m <= stop ) *m++ = *p++;
01739                       AT.WorkPointer = oldstring;
01740                       return(0);
01741                   }
01742                   p++;
01743               }
01744               diff++;
01745           } while ( diff <= (number>>1) );
01746       /*
01747                   #] Same Objects :
01748                   #[ All Different :
01749
01750               Here we have a string with all different objects.
01751
01752       */
01753           t->sgn = 0;
01754           termout = AT.WorkPointer;
01755           while ( ( diff = TraceNno(number,t->accup,t) ) != 0 ) {
01756               p = t->termp;
01757               stop = p + *p;
01758               m = termout;
01759               p++;
01760               if ( p < stop ) do {
01761                   if ( *p == SUBEXPRESSION && p[2] == t->num ) {
01762                       oldstring = p;
01763                       p = t->termp;
01764                       do { *m++ = *p++; } while ( p < oldstring );
01765                       p += p[1];
01766                       pold = p;
01767                       *m++ = AC.lUniTrace[0];
01768                       *m++ = AC.lUniTrace[1];
01769                       *m++ = AC.lUniTrace[2];
01770                       *m++ = AC.lUniTrace[3];
01771                       *m++ = SNUMBER;
01772                       *m++ = 4;
01773                       *m++ = 2;
01774                       *m++ = t->factor;
01775                       p = t->accup;
01776                       oldval = number;
01777                       oldstring = m;
01778                       *m++ = DELTA;
01779                       *m++ = oldval + 2;
01780                       NCOPY(m,p,oldval);
01781                       if ( t->accup > t->accu ) {
```

```
01782                     p = t->accu;
01783                     while ( p < t->accup ) *m++ = *p++;
01784                     oldstring[1] = WORDDIF(m,oldstring);
01785                 }
01786                 p = pold;
01787                 do { *m++ = *p++; } while ( p < stop );
01788                 *termout = WORDDIF(m,termout);
01789                 if ( ( diff ^ t->allsign ) < 0 ) m[-1] = - m[-1];
01790                 if ( ( AT.WorkPointer = m ) > AT.WorkTop ) {
01791                     MLOCK(ErrorMessageLock);
01792                     MesWork();
01793                     MUNLOCK(ErrorMessageLock);
01794                     return(-1);
01795                 }
01796                 if ( *termout ) {
01797                     *AN.RepPoint = 1;
01798                     AR.expchanged = 1;
01799                     if ( Generator(BHEAD termout,t->level) ) {
01800                         AT.WorkPointer = termout;
01801                         goto TracCall;
01802                     }
01803                     t = AN.tracestack + AN.numtracesctack - 1;
01804                 }
01805                 break;
01806             }
01807             p += p[1];
01808         } while ( p < stop );
01809     }
01810     AT.WorkPointer = termout;
01811     return(0);
01812
01813 /*
01814             #] All Different :
01815 */
01816 TracnCall:
01817     AT.WorkPointer = oldstring;
01818 TracCall:
01819     if ( AM.tracebackflag ) {
01820         MLOCK(ErrorMessageLock);
01821         MesCall("TraceNGen");
01822         MUNLOCK(ErrorMessageLock);
01823     }
01824     return(-1);
01825 }
01826
01827 /*
01828         #] TraceNgen :
01829         #[ Traces :             WORD Traces(term,params,num,level)
01830
01831         The contents of the AT.TMout array are:
01832         length,type,subtype,gamma5,factor,sign,gamma's
01833
01834 */
01835
01836 int Traces(PHEAD WORD *term, WORD *params, WORD num, WORD level)
01837 {
01838     GETBIDENTITY
01839     switch ( AT.TMout[2] ) {    /* Subtype gives dimension */
01840         case 0:
01841             return(TraceN(BHEAD term,params,num,level));
01842         case 4:
01843             return(Trace4(BHEAD term,params,num,level));
01844         case 12:
01845             return(Trace4(BHEAD term,params,num,level));
01846         case 20:
01847             return(Trace4(BHEAD term,params,num,level));
01848         default:
01849             return(0);
01850     }
01851 }
01852
01853 /*
01854         #] Traces :
01855         #[ TraceFind :          WORD TraceFind(term,params)
01856 */
01857
01858 int TraceFind(PHEAD WORD *term, WORD *params)
01859 {
01860     GETBIDENTITY
01861     WORD *p, *m, *to;
01862     WORD *termout, *stop, *stop2, number = 0;
01863     WORD first = 1;
01864     WORD type, spinline, sp;
01865     type = params[3];
01866     spinline = params[4];
01867     if ( spinline < 0 ) {   /* $ variable. Evaluate */
01868         sp = DolToIndex(BHEAD -spinline);
```

```
01869         if ( AN.ErrorInDollar || sp < 0 ) {
01870             MLOCK(ErrorMessageLock);
01871             MesPrint("$%s does not have an index value in trace statement in module %l",
01872                 DOLLARNAME(Dollars,-spinline),AC.CModule);
01873             MUNLOCK(ErrorMessageLock);
01874             return(0);
01875         }
01876         spinline = sp;
01877     }
01878     to = AT.TMout;
01879     to++;
01880     *to++ = TAKETRACE;
01881     *to++ = type;
01882     *to++ = GAMMA1;
01883     *to++ = 0;              /* Powers of two */
01884     *to++ = 1;              /* sign */
01885     p = term;
01886     m = p + *p - 1;
01887     stop = m - ABS(*m);
01888     termout = m = AT.WorkPointer;
01889     m++;
01890     p++;
01891     while ( p < stop ) {
01892         stop2 = p + p[1];
01893         if ( *p == GAMMA && p[FUNHEAD] == spinline ) {
01894             if ( first ) {
01895                 *m++ = SUBEXPRESSION;
01896                 *m++ = SUBEXPSIZE;
01897                 *m++ = -1;
01898                 *m++ = 1;
01899                 *m++ = DUMMYBUFFER;
01900                 FILLSUB(m)
01901                 first = 0;
01902             }
01903             p += FUNHEAD+1;
01904             while ( p < stop2 ) {
01905                 if ( *p == GAMMA5 ) {
01906                     if ( AT.TMout[3] == GAMMA5 ) AT.TMout[3] = GAMMA1;
01907                     else if ( AT.TMout[3] == GAMMA1 ) AT.TMout[3] = GAMMA5;
01908                     else if ( AT.TMout[3] == GAMMA7 ) AT.TMout[5] = -AT.TMout[5];
01909                     if ( number & 1 ) AT.TMout[5] = - AT.TMout[5];
01910                     p++;
01911                 }
01912                 else if ( *p == GAMMA6 ) {
01913                     if ( number & 1 ) goto F7;
01914 F6:                  if ( AT.TMout[3] == GAMMA6 ) (AT.TMout[4])++;
01915                     else if ( AT.TMout[3] == GAMMA1 ) AT.TMout[3] = GAMMA6;
01916                     else if ( AT.TMout[3] == GAMMA5 ) AT.TMout[3] = GAMMA6;
01917                     else if ( AT.TMout[3] == GAMMA7 ) AT.TMout[5] = 0;
01918                     p++;
01919                 }
01920                 else if ( *p == GAMMA7 ) {
01921                     if ( number & 1 ) goto F6;
01922 F7:                  if ( AT.TMout[3] == GAMMA7 ) (AT.TMout[4])++;
01923                     else if ( AT.TMout[3] == GAMMA1 ) AT.TMout[3] = GAMMA7;
01924                     else if ( AT.TMout[3] == GAMMA5 ) {
01925                         AT.TMout[3] = GAMMA7;
01926                         AT.TMout[5] = -AT.TMout[5];
01927                     }
01928                     else if ( AT.TMout[3] == GAMMA6 ) AT.TMout[5] = 0;
01929                     p++;
01930                 }
01931                 else {
01932                     *to++ = *p++;
01933                     number++;
01934                 }
01935             }
01936         }
01937         else {
01938             while ( p < stop2 ) *m++ = *p++;
01939         }
01940     }
01941     if ( first ) return(0);
01942     AT.TMout[0] = WORDDIF(to,AT.TMout);
01943     to = term;
01944     to += *to;
01945     while ( p < to ) *m++ = *p++;
01946     *termout = WORDDIF(m,termout);
01947     to = term;
01948     p = termout;
01949     do { *to++ = *p++; } while ( p < m );
01950     AT.WorkPointer = term + *term;
01951     return(1);
01952 }
01953
01954 /*
01955         #] TraceFind :
```

```
01956         #[ Chisholm :           WORD Chisholm(term,level,num)
01957
01958         Routines for reorganizing traces.
01959         The command
01960             Chisholm,1;
01961         will collect the gamma matrices in spinline 1 and see whether
01962         they have an index in common with another gamma matrix. If this
01963         is the case the identity
01964             g_(2,mu)*Tr[g_(1,mu)*S(2)] = S(2)+SR(2)
01965         is applied (SR is the reversed string).
01966 */
01967
01968 int Chisholm(PHEAD WORD *term, WORD level)
01969 {
01970     GETBIDENTITY
01971     WORD *t, *r, *m, *s, *tt, *rr;
01972     WORD *mat, *matpoint, *termout, *rdo;
01973     CBUF *C = cbuf+AM.rbufnum;
01974     WORD i, j, num = C->lhs[level][2], gam5;
01975     WORD norm = 0, k, *matp;
01976 /*
01977     #[ Find : Find possible matrices
01978 */
01979     mat = matpoint = AT.WorkPointer;
01980     t = term;
01981     r = t + *t - 1; r -= ABS(*r);
01982     t++;
01983     i = 0;
01984     gam5 = GAMMA1;
01985     while ( t < r ) {
01986         if ( *t == GAMMA && t[FUNHEAD] == num ) {
01987             m = t + t[1];
01988             t += FUNHEAD+1;
01989             while ( t < m ) {
01990                 if ( *t >= 0 || *t < MINSPEC ) i++;
01991                 else {
01992                     if ( gam5 == GAMMA1 ) gam5 = *t;
01993                     else if ( gam5 == GAMMA5 ) {
01994                         if ( *t == GAMMA5 ) gam5 = GAMMA1;
01995                         else if ( *t != GAMMA1 ) gam5 = *t;
01996                     }
01997                 }
01998                 *matpoint++ = *t++;
01999             }
02000         }
02001         else t += t[1];
02002     }
02003     if ( ( i & 1 ) != 0 ) return(0);    /* odd trace */
02004 /*
02005     #] Find :
02006     #[ Test : Test for contracted index
02007
02008     This code should be modified.
02009
02010     We have to check for all possible matches if C->lhs[level][3] == 1
02011     and the trace contains a gamma5, gamma6 or gamma7.
02012     Then we normalize by the number of possible contractions (norm) and
02013     do all of them. This way the Levi-Civita tensors have a maximum
02014     chance of cancelling each other. This option is activated with
02015     `contract' and `symmetrize'. Defaults are that they are on, but
02016     they can be switched off with nocontract and nosymmetrize.
02017 */
02018     s = mat;
02019     while ( s < matpoint ) {
02020 /*
02021         if ( *s < AM.OffsetIndex || ( *s < ( AM.OffsetIndex + WILDOFFSET ) &&
02022         indices[*s-AM.OffsetIndex].dimension == 0 ) ) {
02023 */
02024         if ( *s < AM.OffsetIndex || ( *s < ( AM.OffsetIndex + WILDOFFSET ) &&
02025         indices[*s-AM.OffsetIndex].dimension != 4 )
02026         || ( ( AC.lDefDim != 4 ) && ( *s >= ( AM.OffsetIndex + WILDOFFSET ) ) ) ) {
02027             s++; continue;
02028         }
02029         t = term+1;
02030         while ( t < r ) {
02031             if ( *t == GAMMA && t[FUNHEAD] != num ) {
02032                 m = t + t[1];
02033                 t += FUNHEAD+1;
02034                 while ( t < m ) {
02035                     if ( *t == *s ) {
02036                         norm++;
02037                     }
02038                     t++;
02039                 }
02040             }
02041             else t += t[1];
02042         }
```

```
02043         s++;
02044     }
02045     if ( norm == 0 ) return(Generator(BHEAD term,level));  /* No Action */
02046 /*
02047     #] Test :
02048     #[ Do : Process the string
02049
02050     tt: The subterm
02051     t:  The matrix
02052     s:  The matrix in the relevant string
02053
02054     Cycle the string in mat so that s is at the end.
02055     Copy the part till the critical GAMMA.
02056     Copy inside the critical string, copy S, copy tail inside string.
02057     Important to remember where S is so that we can reverse it later.
02058     Add term UnitTrace/2/norm.
02059     Copy rest of term.
02060     Continue execution with S.
02061     Reverse S.
02062     Continue execution with SR.
02063 */
02064
02065     if ( C->lhs[level][3] == 0 /* || gam5 == GAMMA1 */ ) norm = 1;
02066
02067     matp = matpoint;
02068     for ( k = 0; k < norm; k++ ) {
02069         matpoint = matp;
02070         s = mat;
02071         while ( s < matpoint ) {
02072 /*
02073             if ( *s < AM.OffsetIndex || ( *s < ( AM.OffsetIndex + WILDOFFSET ) &&
02074             indices[*s-AM.OffsetIndex].dimension == 0 ) ) {
02075 */
02076             if ( *s < AM.OffsetIndex || ( *s < ( AM.OffsetIndex + WILDOFFSET ) &&
02077             indices[*s-AM.OffsetIndex].dimension != 4 ) ) {
02078                 s++; continue;
02079             }
02080             t = term+1;
02081             while ( t < r ) {
02082                 if ( *t == GAMMA && t[FUNHEAD] != num ) {
02083                     tt = t;
02084                     m = t + t[1];
02085                     t += FUNHEAD+1;
02086                     while ( t < m ) {
02087                         if ( *t == *s ) {
02088                             i = WORDDIF(t,tt);
02089                             m = mat;
02090                             while ( m <= s ) *matpoint++ = *m++;
02091                             t = mat;
02092                             while ( m < matpoint ) *t++ = *m++;
02093                             termout = t;
02094                             m = termout + 1;
02095                             t = term + 1;
02096                             while ( t < tt ) {
02097                                 if ( *t != GAMMA || t[FUNHEAD] != num ) {
02098                                     j = t[1];
02099                                     NCOPY(m,t,j);
02100                                 }
02101                                 else t += t[1];
02102                             }
02103
02104                             tt += tt[1];
02105                             rdo = m;
02106                             j = i;
02107                             while ( --j >= 0 ) *m++ = *t++;
02108                             matpoint = m;
02109                             s = mat;
02110                             while ( s < termout ) *m++ = *s++;
02111                             m--;
02112                             t++;
02113                             while ( t < tt ) *m++ = *t++;
02114                             rdo[1] = WORDDIF(m,rdo);
02115
02116                             *m++ = AC.lUniTrace[0];
02117                             *m++ = AC.lUniTrace[1];
02118                             *m++ = AC.lUniTrace[2];
02119                             *m++ = AC.lUniTrace[3];
02120                             *m++ = SNUMBER;
02121                             *m++ = 4;
02122                             *m++ = 2*norm;
02123                             *m++ = -1;
02124
02125                             while ( t < r ) {
02126                                 if ( *t != GAMMA || t[FUNHEAD] != num ) {
02127                                     j = t[1];
02128                                     NCOPY(m,t,j);
02129                                 }
```

```
02130                               else t += t[1];
02131                           }
02132                           rr = term + *term;
02133                           while ( t < rr ) *m++ = *t++;
02134
02135                           *termout = WORDDIF(m,termout);
02136                           rr = m;
02137                           t = termout;
02138                           j = *termout;
02139                           NCOPY(m,t,j);
02140                           AT.WorkPointer = m;
02141                           if ( Generator(BHEAD t,level) ) goto ChisCall;
02142
02143                           j = WORDDIF(termout,mat)-1;
02144                           t = matpoint;
02145                           m = t + j;
02146                           AT.WorkPointer = rr;
02147                           while ( m > t ) {
02148                               i = *--m; *m = *t; *t++ = i;
02149                           }
02150
02151                           if ( Generator(BHEAD termout,level) ) goto ChisCall;
02152                           AT.WorkPointer = mat;
02153
02154                           goto NextK;
02155                       }
02156                       t++;
02157                   }
02158               }
02159               else t += t[1];
02160           }
02161           s++;
02162       }
02163 NextK:;
02164   }
02165   return(0);
02166 /*
02167   #] Do :
02168 */
02169 ChisCall:
02170   if ( AM.tracebackflag ) {
02171       MLOCK(ErrorMessageLock);
02172       MesCall("Chisholm");
02173       MUNLOCK(ErrorMessageLock);
02174   }
02175   return(-1);
02176 }
02177
02178 /*
02179       #] Chisholm :
02180       #[ TenVecFind :         WORD TenVecFind(term,params)
02181 */
02182
02183 int TenVecFind(PHEAD WORD *term, WORD *params)
02184 {
02185   GETBIDENTITY
02186   WORD *t, *w, *m, *tstop;
02187   WORD i, mode, thevector, thetensor, spectator;
02188   thetensor = params[3];
02189   thevector = params[4];
02190   mode = params[5];
02191   if ( thetensor < 0 ) {  /* $-expression */
02192       thetensor = DolToTensor(BHEAD -thetensor);
02193       if ( thetensor < FUNCTION ) {
02194           if ( thevector > 0 ) {
02195               thetensor = DolToTensor(BHEAD thevector);
02196               if ( thetensor < FUNCTION ) {
02197                   MLOCK(ErrorMessageLock);
02198                   MesPrint("$%s should have been a tensor in module %l"
02199                       ,DOLLARNAME(Dollars,params[4]),AC.CModule);
02200                   MUNLOCK(ErrorMessageLock);
02201                   return(-1);
02202               }
02203               thevector = DolToVector(BHEAD -params[3]);
02204               if ( thevector >= 0 ) {
02205                   MLOCK(ErrorMessageLock);
02206                   MesPrint("$%s should have been a vector in module %l"
02207                       ,DOLLARNAME(Dollars,-params[3]),AC.CModule);
02208                   MUNLOCK(ErrorMessageLock);
02209                   return(-1);
02210               }
02211           }
02212           else {
02213               MLOCK(ErrorMessageLock);
02214               MesPrint("$%s should have been a tensor in module %l"
02215                   ,DOLLARNAME(Dollars,-params[3]),AC.CModule);
02216               MUNLOCK(ErrorMessageLock);
```

```
02217                 return(-1);
02218             }
02219         }
02220     }
02221     if ( thevector > 0 ) {  /* $-expression */
02222         thevector = DolToVector(BHEAD thevector);
02223         if ( thevector >= 0 ) {
02224             MLOCK(ErrorMessageLock);
02225             MesPrint("$%s should have been a vector in module %l"
02226                 ,DOLLARNAME(Dollars,params[4]),AC.CModule);
02227             MUNLOCK(ErrorMessageLock);
02228             return(-1);
02229         }
02230     }
02231     if ( ( mode & 1 ) != 0 ) {  /* Vector to tensor */
02232         GETSTOP(term,tstop);
02233         t = term + 1;
02234         while ( t < tstop ) {
02235             if ( *t == DOTPRODUCT ) {
02236                 i = t[1] - 2; t += 2;
02237                 while ( i > 0 ) {
02238                     spectator = 0;
02239                     if ( t[2] < 0 ) {}
02240                     else if ( *t == thevector && t[1] == thevector ) {
02241                         if ( ( mode & 2 ) == 0 ) spectator = thevector;
02242                     }
02243                     else if ( *t == thevector ) spectator = t[1];
02244                     else if ( t[1] == thevector ) spectator = *t;
02245                     if ( spectator ) {
02246                         if ( ( mode & 8 ) == 0 ) goto match;
02247                         w = SetElements + Sets[params[6]].first;
02248                         m = SetElements + Sets[params[6]].last;
02249                         while ( w < m ) {
02250                             if ( *w == spectator ) break;
02251                             w++;
02252                         }
02253                         if ( w >= m ) goto match;
02254                     }
02255                     t += 3;
02256                     i -= 3;
02257                 }
02258             }
02259             else if ( *t == VECTOR ) {
02260                 i = t[1] - 2; t += 2;
02261                 while ( i > 0 ) {
02262                     if ( *t == thevector ) goto match;
02263                     t += 2;
02264                     i -= 2;
02265                 }
02266             }
02267             else if ( *t == thetensor ) t += t[1];
02268             else if ( *t >= FUNCTION ) {
02269                 if ( functions[*t-FUNCTION].spec > 0 ) {
02270                     w = t + t[1];
02271                     t += FUNHEAD;
02272                     while ( t < w ) {
02273                         if ( *t == thevector ) goto match;
02274                         t++;
02275                     }
02276                 }
02277                 else if ( ( mode & 4 ) != 0 ) {
02278                     w = t + t[1];
02279                     t += FUNHEAD;
02280                     while ( t < w ) {
02281                         if ( *t == -VECTOR && t[1] == thevector ) goto match;
02282                         else if ( *t > 0 ) t += *t;
02283                         else if ( *t <= -FUNCTION ) t++;
02284                         else t += 2;
02285                     }
02286                 }
02287                 else t += t[1];
02288             }
02289             else t += t[1];
02290         }
02291     }
02292     else {                      /* Tensor to Vector */
02293         GETSTOP(term,tstop);
02294         t = term+1;
02295         while ( t < tstop ) {
02296             if ( *t == thetensor ) goto match;
02297             t += t[1];
02298         }
02299     }
02300     return(0);
02301 match:
02302     AT.TMout[0] = 5;
02303     AT.TMout[1] = TENVEC;
```

```
02304     AT.TMout[2] = thetensor;
02305     AT.TMout[3] = thevector;
02306     AT.TMout[4] = mode;
02307     if ( ( mode & 8 ) != 0 ) { AT.TMout[0] = 6; AT.TMout[5] = params[6]; }
02308     return(1);
02309
02310 }
02311
02312 /*
02313         #] TenVecFind :
02314         #[ TenVec :             WORD TenVec(term,params,num,level)
02315 */
02316
02317 int TenVec(PHEAD WORD *term, WORD *params, WORD num, WORD level)
02318 {
02319     GETBIDENTITY
02320     WORD *t, *m, *w, *termout, *tstop, *outlist, *ou, *ww, *mm;
02321     WORD i, j, k, x, mode, thevector, thetensor, DumNow, spectator;
02322     DUMMYUSE(num);
02323     thetensor = params[2];
02324     thevector = params[3];
02325     mode = params[4];
02326     termout = AT.WorkPointer;
02327     DumNow = AR.CurDum = DetCurDum(BHEAD term);
02328     if ( ( mode & 1 ) != 0 ) {  /* Vector to tensor */
02329         AT.WorkPointer += *term;
02330         ou = outlist = AT.WorkPointer;
02331         GETSTOP(term,tstop);
02332         t = term + 1;
02333         m = termout + 1;
02334         while ( t < tstop ) {
02335             if ( *t == DOTPRODUCT ) {
02336                 i = t[1] - 2;
02337                 w = m;
02338                 *m++ = *t++; *m++ = *t++;
02339                 while ( i > 0 ) {
02340                     spectator = 0;
02341                     if ( t[2] < 0 ) {
02342                         *m++ = *t++; *m++ = *t++; *m++ = *t++;
02343                     }
02344                     else if ( *t == thevector && t[1] == thevector ) {
02345                         if ( ( mode & 2 ) == 0 ) spectator = thevector;
02346                         else {
02347                             *m++ = *t++; *m++ = *t++; *m++ = *t++;
02348                         }
02349                     }
02350                     else if ( *t == thevector ) spectator = t[1];
02351                     else if ( t[1] == thevector ) spectator = *t;
02352                     else {
02353                         *m++ = *t++; *m++ = *t++; *m++ = *t++;
02354                     }
02355                     if ( spectator ) {
02356                         if ( ( mode & 8 ) == 0 ) goto noveto;
02357                         ww = SetElements + Sets[params[5]].first;
02358                         mm = SetElements + Sets[params[5]].last;
02359                         while ( ww < mm ) {
02360                             if ( *ww == spectator ) break;
02361                             ww++;
02362                         }
02363                         if ( ww < mm ) {
02364                             *m++ = *t++; *m++ = *t++; *m++ = *t++;
02365                         }
02366                         else {
02367 noveto:                 if ( spectator == thevector ) {
02368                             for ( j = 0; j < t[2]; j++ ) {
02369                                 *ou++ = ++AR.CurDum;
02370                                 *ou++ = AR.CurDum;
02371                             }
02372                             t += 3;
02373                         }
02374                         else {
02375                             for ( j = 0; j < t[2]; j++ ) *ou++ = spectator;
02376                             t += 3;
02377                         }}
02378                     }
02379                     i -= 3;
02380                 }
02381                 w[1] = WORDDIF(m,w);
02382                 if ( w[1] == 2 ) m = w;
02383             }
02384             else if ( *t == VECTOR ) {
02385                 i = t[1] - 2; w = m;
02386                 *m++ = *t++; *m++ = *t++;
02387                 while ( i > 0 ) {
02388                     if ( *t == thevector ) {
02389                         *ou++ = t[1];
02390                         t += 2;
```

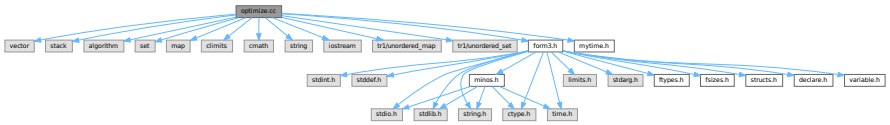
```
02391                   }
02392                   else { *m++ = *t++; *m++ = *t++; }
02393                   i -= 2;
02394               }
02395               w[1] = WORDDIF(m,w);
02396               if ( w[1] == 2 ) m = w;
02397           }
02398           else if ( *t == thetensor ) {
02399               i = t[1] - FUNHEAD;
02400               t += FUNHEAD;
02401               NCOPY(ou,t,i);
02402           }
02403           else if ( *t >= FUNCTION ) {
02404               if ( functions[*t-FUNCTION].spec > 0 ) {
02405                   w = t + t[1];
02406                   i = FUNHEAD;
02407                   NCOPY(m,t,i);
02408                   while ( t < w ) {
02409                       if ( *t == thevector ) {
02410                           *m++ = ++AR.CurDum;
02411                           *ou++ = AR.CurDum;
02412                           t++;
02413                       }
02414                       else *m++ = *t++;
02415                   }
02416               }
02417               else if ( ( mode & 4 ) != 0 ) {
02418                   w = t + t[1];
02419                   i = FUNHEAD;
02420                   NCOPY(m,t,i);
02421                   while ( t < w ) {
02422                       if ( *t == -VECTOR && t[1] == thevector ) {
02423                           *m++ = -INDEX;
02424                           *m++ = ++AR.CurDum;
02425                           *ou++ = AR.CurDum;
02426                           t += 2;
02427                       }
02428                       else if ( *t > 0 ) {
02429                           i = *t;
02430                           NCOPY(m,t,i);
02431                       }
02432                       else if ( *t <= -FUNCTION ) *m++ = *t++;
02433                       else { *m++ = *t++; *m++ = *t++; }
02434                   }
02435               }
02436               else goto docopy;
02437           }
02438           else {
02439 docopy:
02440               i = t[1];
02441               NCOPY(m,t,i);
02442           }
02443       }
02444       i = WORDDIF(ou,outlist);
02445       if ( i > 0 ) {
02446           for ( j = 1; j < i; j++ ) {
02447               if ( outlist[j-1] > outlist[j] ) {
02448                   x = outlist[j-1]; outlist[j-1] = outlist[j]; outlist[j] = x;
02449                   for ( k = j-1; k > 0; k-- ) {
02450                       if ( outlist[k-1] <= outlist[k] ) break;
02451                       x = outlist[k-1]; outlist[k-1] = outlist[k]; outlist[k] = x;
02452                   }
02453               }
02454           }
02455
02456           *m++ = thetensor;
02457           *m++ = FUNHEAD + i;
02458           *m++ = DIRTYSYMFLAG;
02459           FILLFUN3(m)
02460           ou = outlist;
02461           NCOPY(m,ou,i);
02462       }
02463       w = term + *term;
02464       while ( t < w ) *m++ = *t++;
02465   }
02466   else {                      /* Tensor to Vector */
02467       GETSTOP(term,tstop);
02468       t = term+1;
02469       m = termout+1;
02470       while ( t < tstop ) {
02471           if ( *t != thetensor ) {
02472               i = t[1];
02473               NCOPY(m,t,i);
02474           }
02475           else {
02476               i = t[1] - FUNHEAD;
02477               t += FUNHEAD;
```

```
02478                 if ( i > 0 ) {
02479                     w = m; m += 2;
02480                     while ( --i >= 0 ) {
02481                         *m++ = thevector;
02482                         *m++ = *t++;
02483                     }
02484                     *w = DELTA;
02485                     w[1] = WORDDIF(m,w);
02486                 }
02487             }
02488         }
02489         w = term + *term;
02490         while ( t < w ) *m++ = *t++;
02491     }
02492     *termout = WORDDIF(m,termout);
02493     AT.WorkPointer = m;
02494     *AT.TMout = 0;
02495     if ( Generator(BHEAD termout,level) ) goto fromTenVec;
02496     AR.CurDum = DumNow;
02497     AT.WorkPointer = termout;
02498     return(0);
02499 fromTenVec:
02500     if ( AM.tracebackflag ) {
02501         MLOCK(ErrorMessageLock);
02502         MesCall("TenVec");
02503         MUNLOCK(ErrorMessageLock);
02504     }
02505     return(-1);
02506 }
02507
02508 /*
02509         #] TenVec :
02510     #] Operations :
02511 */
```

4.95 optimize.cc File Reference

```
#include <vector>
#include <stack>
#include <algorithm>
#include <set>
#include <map>
#include <climits>
#include <cmath>
#include <string>
#include <iostream>
#include <tr1/unordered_map>
#include <tr1/unordered_set>
#include "form3.h"
#include "mytime.h"
```

Include dependency graph for optimize.cc:



Data Structures

- class tree_node
- struct CSEHash
- struct CSEEq
- struct node
- struct NodeHash
- struct NodeEq
- class optimization

Typedefs

- typedef struct node **NODE**

Functions

- void print_instr (const vector< WORD > &instr, WORD num)
- template<class RandomAccessIterator >
  void my_random_shuffle (PHEAD RandomAccessIterator fr, RandomAccessIterator to)
- LONG get_expression (int exprnr)
- vector< vector< WORD > > get_brackets ()
- int count_operators (const WORD *expr, bool print=false)
- int count_operators (const vector< WORD > &instr, bool print=false)
- vector< WORD > occurrence_order (const WORD *expr, bool rev)
- WORD getpower (const WORD *term, int var, int pos)
- void fixarg (UWORD *t, WORD &n)
- void GcdLong_fix_args (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- void DivLong_fix_args (UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc, UWORD *d, WORD *nd)
- void build_Horner_tree (const WORD **terms, int numterms, int var, int maxvar, int pos, vector< WORD > *res)
- bool term_compare (const WORD *a, const WORD *b)
- vector< WORD > Horner_tree (const WORD *expr, const vector< WORD > &order)
- void print_tree (const vector< WORD > &tree)
- template<typename T >
  size_t hash_range (T *array, int size)
- vector< WORD > generate_instructions (const vector< WORD > &tree, bool do_CSE)
- int count_operators_cse (const vector< WORD > &tree)
- NODE * buildTree (vector< WORD > &tree)
- int count_operators_cse_topdown (vector< WORD > &tree)
- vector< WORD > simulated_annealing ()
- void find_Horner_MCTS_expand_tree ()
- void find_Horner_MCTS ()
- vector< WORD > merge_operators (const vector< WORD > &all_instr, bool move_coeff)
- vector< optimization > find_optimizations (const vector< WORD > &instr)
- bool do_optimization (const optimization optim, vector< WORD > &instr, int newid)
- int partial_factorize (vector< WORD > &instr, int n, int improve)
- vector< WORD > optimize_greedy (vector< WORD > instr, LONG time_limit)
- vector< WORD > recycle_variables (const vector< WORD > &all_instr)
- void optimize_expression_given_Horner ()
- void generate_output (const vector< WORD > &instr, int exprnr, int extraoffset, const vector< vector< WORD > > &brackets)
- WORD generate_expression (WORD exprnr)
- void optimize_print_code (int print_expr)
- int Optimize (WORD exprnr, int do_print)
- int ClearOptimize ()

Variables

- const WORD OPER_ADD = -1
- const WORD OPER_MUL = -2
- const WORD OPER_COMMA = -3
- WORD * optimize_expr
- vector< vector< WORD > > optimize_best_Horner_schemes
- int optimize_num_vars
- int optimize_best_num_oper
- vector< WORD > optimize_best_instr
- vector< WORD > optimize_best_vars
- bool mcts_factorized
- bool mcts_separated
- vector< WORD > mcts_vars
- tree_node mcts_root
- int mcts_expr_score
- set< pair< int, vector< WORD > > > mcts_best_schemes

4.95.1 Detailed Description

experimental routines for the optimization of FORTRAN or C output.

Definition in file optimize.cc.

4.95.2 Function Documentation

4.95.2.1 print_instr()

```
void print_instr (
            const vector< WORD > & instr,
            WORD num )
```

Definition at line 179 of file optimize.cc.

4.95.2.2 my_random_shuffle()

```
template<class RandomAccessIterator >
void my_random_shuffle (
            PHEAD RandomAccessIterator fr,
            RandomAccessIterator to )
```

Random shuffle

4.95.3 Description

Randomly permutes elements in the range [fr,to). Functionality is the same as C++'s "random_shuffle", but it uses Form's "wranf".

Definition at line 201 of file optimize.cc.

Referenced by find_Horner_MCTS().

4.95.3.1 get_expression()

```
LONG get_expression (
            int exprnr )
```

Get expression

4.95.4 Description

Reads an expression from the input file into a buffer (called optimize_expr). This buffer is used during the optimization process. Non-symbols are removed by ConvertToPoly and are put in temporary symbols.

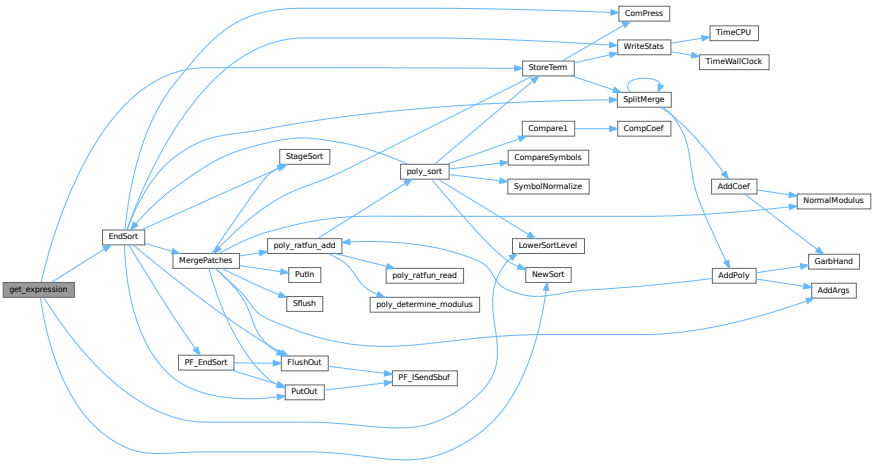
The return value is the length of the expression in WORDs, or a negative number if it failed.

Definition at line 225 of file optimize.cc.

References EndSort(), LowerSortLevel(), NewSort(), and StoreTerm().

Referenced by Optimize().

Here is the call graph for this function:



4.95.4.1 get_brackets()

```
vector< vector< WORD > > get_brackets ( )
```

Get brackets

4.95.5 Description

Checks whether the input expression (stored in optimize_expr) contains brackets. If so, this method replaces terms outside brackets by powers of SEPERATESYMBOL (equal brackets have equal powers) and the brackets are returned. If not, the result is empty.

Brackets are used for simultaneous optimization. The symbol SEPARATESYMBOL is always the first one used in a Horner scheme.

Definition at line 302 of file optimize.cc.

Referenced by Optimize().

4.95.5.1 count_operators() [1/2]

```
int count_operators (
            const WORD * expr,
            bool print = false )
```

Count operators

4.95.6 Description

Counts the number of operators in a Form-style expression.

Definition at line 422 of file optimize.cc.

Referenced by find_Horner_MCTS(), generate_instructions(), Optimize(), optimize_expression_given_Horner(), and optimize_greedy().

4.95.6.1 count_operators() [2/2]

```
int count_operators (
            const vector< WORD > & instr,
            bool print = false )
```

Count operators

4.95.7 Description

Counts the number of operators in a vector of instructions

Definition at line 476 of file optimize.cc.

4.95.7.1 occurrence_order()

```
vector< WORD > occurrence_order (
            const WORD * expr,
            bool rev )
```

Occurrence order

4.95.8 Description

Extracts all variables from an expression and sorts them with most occurring first (or last, with rev=true)

Definition at line 519 of file optimize.cc.

Referenced by Optimize().

4.95.8.1 getpower()

```
WORD getpower (
             const WORD * term,
             int var,
             int pos )
```

Horner tree building

4.95.9 Description

Given a Form-style expression (in a buffer in memory), this builds an expression tree. The tree is determined by a multivariate Horner scheme, i.e., something of the form:

1+y+x∗(2+y∗(1+y)+x∗(3-y∗(...)))

The order of the variables is given to the method "Horner_tree", which renumbers ad reorders the terms of the expression. Next, the recursive method "build_Horner_tree" does the actual tree construction.

The tree is represented in postfix notation. Tokens are of the following forms:

- SNUMBER tokenlength num den coefflength
- SYMBOL tokenlength variable power
- OPER_ADD or OPER_MUL

4.95.10 Note

Sets AN.poly_num_vars and allocates AN.poly_vars. The latter should be freed later. Get power of variable (helper function for build_Horner_tree)

4.95.11 Description

Returns the power of the variable "var", which is at position "pos" in this term, if it is present.

Definition at line 600 of file optimize.cc.

Referenced by build_Horner_tree().

4.95.11.1 fixarg()

```
void fixarg (
             UWORD * t,
             WORD & n )
```

Call GcdLong/DivLong with leading zeroes

4.95.12 Description

These method remove leading zeroes, so that GcdLong and DivLong can safely be called.

Definition at line 614 of file optimize.cc.

4.95.12.1 GcdLong_fix_args()

```
void GcdLong_fix_args (
             PHEAD UWORD * a,
             WORD na,
             UWORD * b,
             WORD nb,
             UWORD * c,
             WORD * nc )
```

Definition at line 620 of file optimize.cc.

4.95.12.2 DivLong_fix_args()

```
void DivLong_fix_args (
             UWORD * a,
             WORD na,
             UWORD * b,
             WORD nb,
             UWORD * c,
             WORD * nc,
             UWORD * d,
             WORD * nd )
```

Definition at line 626 of file optimize.cc.

4.95.12.3 build_Horner_tree()

```
void build_Horner_tree (
             const WORD ** terms,
             int numterms,
             int var,
             int maxvar,
             int pos,
             vector< WORD > * res )
```

Build the Horner tree

4.95.13 Description

Constructs the Horner tree. The method processes one variable and continues recursively until the Horner scheme is completed.

"terms" is a pointer to the starts of the terms. "numterms" is the number of terms to be processed. "var" is the next variable to be processed (index between 0 and #maxvar) and "maxvar" is the last variable to be processed, so that partial Horner trees can also be constructed. "pos" is the position that the power of "var" should be in (one level further in the recursion, "pos" has increased by 0 or 1 depending on whether the previous power was 0 or not). The result is written at the pointer "res".

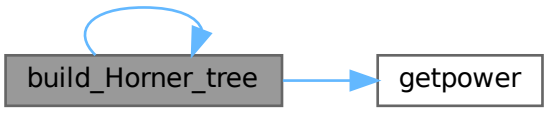
This method also factors out gcds of the coefficients. The result should end with "gcd OPER_MUL" at all times, so that one level higher gcds can be extracted again.

Definition at line 652 of file optimize.cc.

References build_Horner_tree(), and getpower().

Referenced by build_Horner_tree(), and Horner_tree().

Here is the call graph for this function:



4.95.13.1 term_compare()

```
bool term_compare (
            const WORD * a,
            const WORD * b )
```

Term compare (helper function for Horner_tree)

4.95.14 Description

Compares two terms of the form "L SYM 4 x n coeff" or "L coeff". Lower powers of lower-indexed symbols come first. This is used to sort the terms in correct order.

Definition at line 852 of file optimize.cc.

Referenced by Horner_tree().

4.95.14.1 Horner_tree()

```
vector< WORD > Horner_tree (
            const WORD * expr,
            const vector< WORD > & order )
```

Prepare Horner tree building

4.95.15 Description

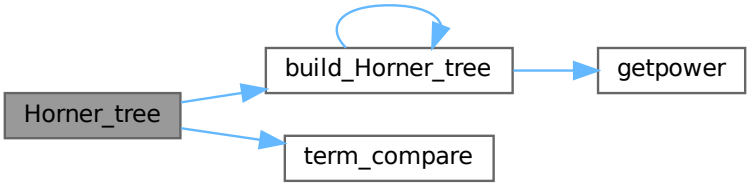
This method renumbers the variables to 0...#vars-1 according to the specified order. Next, it stored pointer to individual terms and sorts the terms with higher power first. Then the sorted lists of power is used for the construction of the Horner tree.

Definition at line 873 of file optimize.cc.

References build_Horner_tree(), and term_compare().

Referenced by optimize_expression_given_Horner().

Here is the call graph for this function:



4.95.15.1 print_tree()

```
void print_tree (
            const vector< WORD > & tree )
```

Definition at line 1001 of file optimize.cc.

4.95.15.2 hash_range()

```
template<typename T >
size_t hash_range (
            T * array,
            int size )
```

Definition at line 1070 of file optimize.cc.

4.95.15.3 generate_instructions()

```
vector< WORD > generate_instructions (
            const vector< WORD > & tree,
            bool do_CSE )
```

Generate instructions

4.95.16 Description

Converts the expression tree to a list of instructions that directly translate to code. Instructions are of the form:

expr.nr operator length [operands]+ trailing.zero

The operands are of the form:

length [(EXTRA)SYMBOL length variable power] coeff

This method only generates binary operators. Merging is done later. The method also checks for common subexpressions and eliminates them and the flag "do_CSE" is set.

4.95.17 Implementation details

The map "ID" keeps track of which subexpressions already exist. The key is formatted as one of the following:

SYMBOL x n SNUMBER coeff OPERATOR LHS RHS

with LHS/RHS formatted as one of the following:

SNUMBER idx 0 (EXTRA)SYMBOL x n

ID[symbol] or ID[operator] equals a subexpression number. ID[coeff] equals the position of the number in the input.

The stack s is used the process the postfix expression tree. Three-word tokens of the form:

SNUMBER idx.of.coeff 0 SYMBOL x n EXTRASYMBOL x 1

are pushed onto it. Operators pop two operands and push the resulting expression.

(Extra)symbols are 1-indexed, because -X is also needed to represent -1 times this term.

There is currently a bug. The notation cannot tell if there is a single bracket and then ignores the bracket.

TODO: check if this method performs properly if do_CSE=false

Definition at line 1132 of file optimize.cc.

References count_operators().

Referenced by optimize_expression_given_Horner().

Here is the call graph for this function:

4.95.17.1 count_operators_cse()

```
int count_operators_cse (
            const vector< WORD > & tree )
```

Count number of operators in a binary tree, while removing CSEs on the fly. The instruction set is not created, which makes this method slightly faster.

A hash is created on the fly and is passed through the stack. TODO: find better hash functions

Definition at line 1341 of file optimize.cc.

4.95.17.2 buildTree()

```
NODE * buildTree (
            vector< WORD > & tree )
```

Definition at line 1563 of file optimize.cc.

4.95.17.3 count_operators_cse_topdown()

```
int count_operators_cse_topdown (
            vector< WORD > & tree )
```

Definition at line 1625 of file optimize.cc.

4.95.17.4 simulated_annealing()

```
vector< WORD > simulated_annealing ( )
```

Definition at line 1680 of file optimize.cc.

4.95.17.5 find_Horner_MCTS_expand_tree()

```
void find_Horner_MCTS_expand_tree ( )
```

Definition at line 2053 of file optimize.cc.

4.95.17.6 find_Horner_MCTS()

```
void find_Horner_MCTS ( )
```

Find best Horner schemes using MCTS

4.95.18 Description

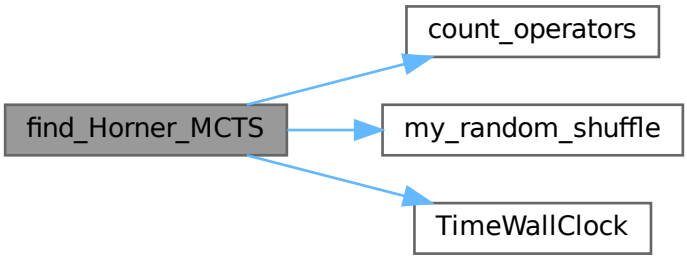
The method governs the MCTS for the best Horner schemes. It does some pre-processing, calls "find_Horner_↩ MCTS_expand_tree" a number of times and does some post-processing.

Definition at line 2254 of file optimize.cc.

References count_operators(), my_random_shuffle(), and TimeWallClock().

Referenced by Optimize().

Here is the call graph for this function:



4.95.18.1 merge_operators()

```
vector< WORD > merge_operators (
            const vector< WORD > & all_instr,
            bool move_coeff )
```

Merge operators

4.95.19 Description

The input instructions form a binary DAG. This method merges expressions like

Z1 = a+b; Z2 = Z1+c;

into

Z2 = a+b+c;

An instruction is merged iff it only has one parent and the operator equals its parent's operator.

This still leaves some freedom: where should the coefficients end up in cases as:

Z1 = Z2 + x <=> Z1 = 2∗Z2 + x Z2 = 2∗x∗y Z2 = x∗y

Both are relevant, e.g. for CSE of the form "2∗x" and "2∗Z2". The flag "move_coeff" moves coefficients from LHS-like expressions to RHS-like expressions.

Furthermore, this method removes empty equation (Z1=0), that are introduced by some "optimize_greedy" substitutions.

4.95.20 Implementation details

Expressions are mostly traversed via a stack, so that parents are evaluated before their children.

With "move_coeff" set coefficients are moved, but this leads to some tricky cases, e.g.

Z1 = Z2 + x Z2 = 2∗y

Here Z2 reduces to the trivial equation Z2=y, which should be eliminated. Here the array skip[i] comes in.

Furthermore in the case

Z1 = Z2 + x Z2 = 2∗Z3 Z3 = x∗Z4 Z4 = y∗z

after substituting Z1 = 2∗Z3 + x, the parent expression for Z4 becomes Z3 instead of Z2. This is where renum_par[i] comes in.

Finally, once a coefficient has been moved, skip_coeff[i] is set and this coefficient is copied into the new expression anymore.

Definition at line 2402 of file optimize.cc.

Referenced by optimize_expression_given_Horner().

4.95.20.1 find_optimizations()

```
vector< optimization > find_optimizations (
            const vector< WORD > & instr )
```

Find optimizations

4.95.21 Description

This method find all optimization of the form described in "class Optimization". It process every equation, looking for possible optimizations and stores them in a fast-access data structure to count the total improvement of an optimization.

Definition at line 2697 of file optimize.cc.

Referenced by optimize_greedy().

4.95.21.1 do_optimization()

```
bool do_optimization (
            const optimization optim,
            vector< WORD > & instr,
            int newid )
```

Do optimization

4.95.22 Description

This method performs an optimization. It scans through the equations of "optim.eqnidxs" and looks in which this optimization can still be performed (due to other performed optimizations this isn't always the case). If possible, it substitutes the common subexpression by a new extra symbol numbered "newid". Finally, the new extrasymbol is defined accordingly.

Substitutions may lead to trivial equations of the form "Zi=Zj", but these are removed in the end of the method. The method returns whether the substitution has been done once or more (or not).

Definition at line 2930 of file optimize.cc.

Referenced by optimize_greedy().

4.95.22.1 partial_factorize()

```
int partial_factorize (
            vector< WORD > & instr,
            int n,
            int improve )
```

Partial factorization of instructions

4.95.23 Description

This method performs partial factorization of instructions. In particular the following instructions

Z1 = x∗a∗b Z2 = x∗c∗d∗e Z3 = 2∗x + Z1 + Z2 + more

are replaced by

Z1 = a∗b Z2 = c∗d∗e Z3 = Zj + more Zi = 2 + Z1 + Z2 Zj = x∗Zi

Here it is necessary that no other equations refer to Z1 and Z2. The generation of trivial instructions (Zi=Zj or Zi=x) is prevented.

Definition at line 3479 of file optimize.cc.

Referenced by optimize_greedy().

4.95.23.1 optimize_greedy()

```
vector< WORD > optimize_greedy (
            vector< WORD > instr,
            LONG time_limit )
```

Optimize instructions greedily

4.95.24 Description

This method optimizes an expression greedily. It calls "find_optimizations" to obtain candidates and performs the best one(s) by calling "do_optimization".

How many different optimization are done, before "find_optimization" is called again, is determined by the settings "greedyminnum" and "greedymaxperc".

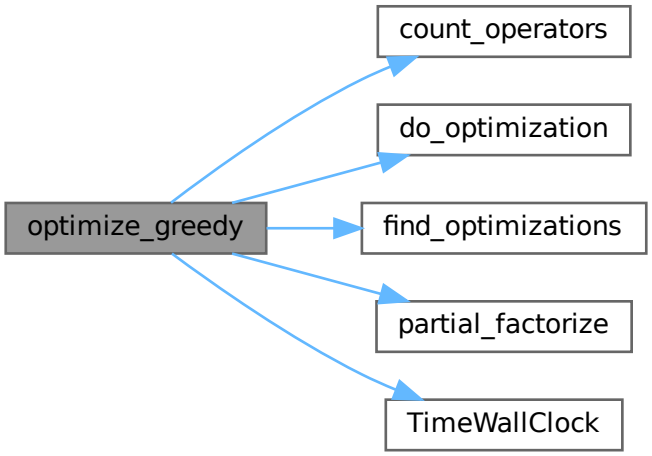
During the optimization process, sequences of zeroes are introduced in the instructions, since moving all instructions when one gets optimized, is very costly. Therefore, in the end, the instructions are "compressed" again to remove these extra zeroes.

Definition at line 3773 of file optimize.cc.

References count_operators(), do_optimization(), find_optimizations(), partial_factorize(), and TimeWallClock().

Referenced by optimize_expression_given_Horner().

Here is the call graph for this function:



4.95.24.1 recycle_variables()

```
vector< WORD > recycle_variables (
            const vector< WORD > & all_instr )
```

Recycle variables

4.95.25 Description

The current input uses many temporary variables. Many of them become obsolete at some point during the evaluation of the code, so can be recycled. This method renumbers the temporary variables, so that they are recycled. Furthermore, the input is order in depth-first order, so that the instructions can be performed consecutively.

4.95.26 Implementation details

First, for each subDAG, an estimate for the number of variables needed is made. This is done by the following recursive formula:

#vars(x) = max(#vars(ch_i(x)) + i),

with ch_i(x) the i-th child of x, where the childs are ordered w.r.t. #vars(ch_i). This formula is exact if the input forms a tree, and otherwise gives a reasonable estimate.

Then, the instructions are reordered in a depth-first order with childs ordered w.r.t. #vars. Next, the times that variables become obsolete are found. Each LHS of an instruction is renumbered to the lowest-numbered temporary variable that is available at that time.

Definition at line 3913 of file optimize.cc.

Referenced by optimize_expression_given_Horner().

4.95.26.1 optimize_expression_given_Horner()

```
void optimize_expression_given_Horner ( )
```

Optimize expression given a Horner scheme

4.95.27 Description

This method picks one Horner scheme from the list of best Horner schemes, applies this scheme to the expression and then, depending on optimize.settings, does a common subexpression elimination (CSE) or performs greedy optimizations.

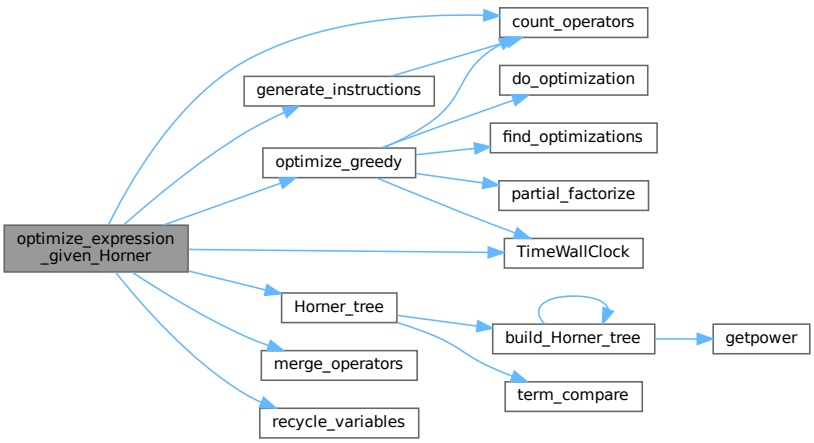
CSE is fast, while greedy might be slow. CSE followed by greedy is faster than greedy alone, but typically results in slightly worse code (not proven; just observed). eventually do greedy optimizations

Definition at line 4060 of file optimize.cc.

References count_operators(), generate_instructions(), Horner_tree(), merge_operators(), optimize_greedy(), recycle_variables(), and TimeWallClock().

Referenced by Optimize().

Here is the call graph for this function:



4.95.27.1 generate_output()

```
void generate_output (
             const vector< WORD > & instr,
             int exprnr,
             int extraoffset,
             const vector< vector< WORD > > & brackets )
```

Generate output

4.95.28 Description

This method prepares the instructions for printing. They are stored in Form format, so that they can be printed by "PrintExtraSymbol". The results are stored in the buffer AO.OptimizeResult.

Definition at line 4291 of file optimize.cc.

Referenced by Optimize().

4.95.28.1 generate_expression()

```
WORD generate_expression (
             WORD exprnr )
```

Generate expression

4.95.29 Description

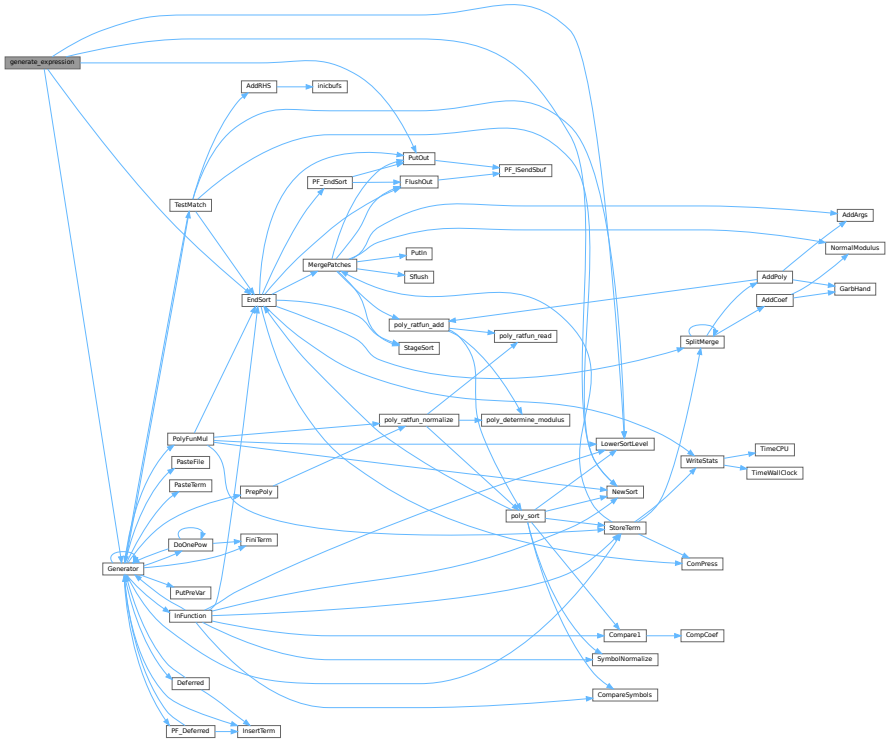
This method modifies the original optimized expression by an expression with extra symbols. This is used for "#Optimize".

Definition at line 4440 of file optimize.cc.

References EndSort(), Generator(), LowerSortLevel(), NewSort(), and PutOut().

Referenced by Optimize().

Here is the call graph for this function:



4.95.29.1 optimize_print_code()

```
void optimize_print_code (
            int print_expr )
```

Print optimized code

4.95.30 Description

This method prints the optimized code via "PrintExtraSymbol". Depending on the flag, the original expression is printed (for "Print") or not (for "#Optimize / #write "O").

Definition at line 4528 of file optimize.cc.

Referenced by Optimize().

4.95.30.1 Optimize()

```
int Optimize (
            WORD exprnr,
            int do_print )
```

Optimization of expression

4.95.31 Description

This method takes an input expression and generates optimized code to calculate its value. The following methods are called to do so:

(1) get_expression : to read to expression

(2) get_brackets : find brackets for simultaneous optimization

(3) occurrence_order or find_Horner_MCTS : to determine (the) Horner scheme(s) to use; this depends on AO.↩ optimize.horner

(4) optimize_expression_given_Horner : to do the optimizations for each Horner scheme; this method does either CSE or greedy optimizations dependings on AO.optimize.method

(5) generate_output : to format the output in Form notation and store it in a buffer

(6a) optimize_print_code : to print the expression (for "Print") or (6b) generate_expression : to modify the expression (for "#Optimize")

On ParFORM, all the processes must call this function at the same time. Then

(1) Because only the master can access to the expression to be optimized, the master broadcast the expression to all the slaves after reading the expression (PF_get_expression).

(2) get_brackets reads optimize_expr as the input and it works also on the slaves. We leave it although the bracket information is not needed on the slaves (used in (5) on the master).

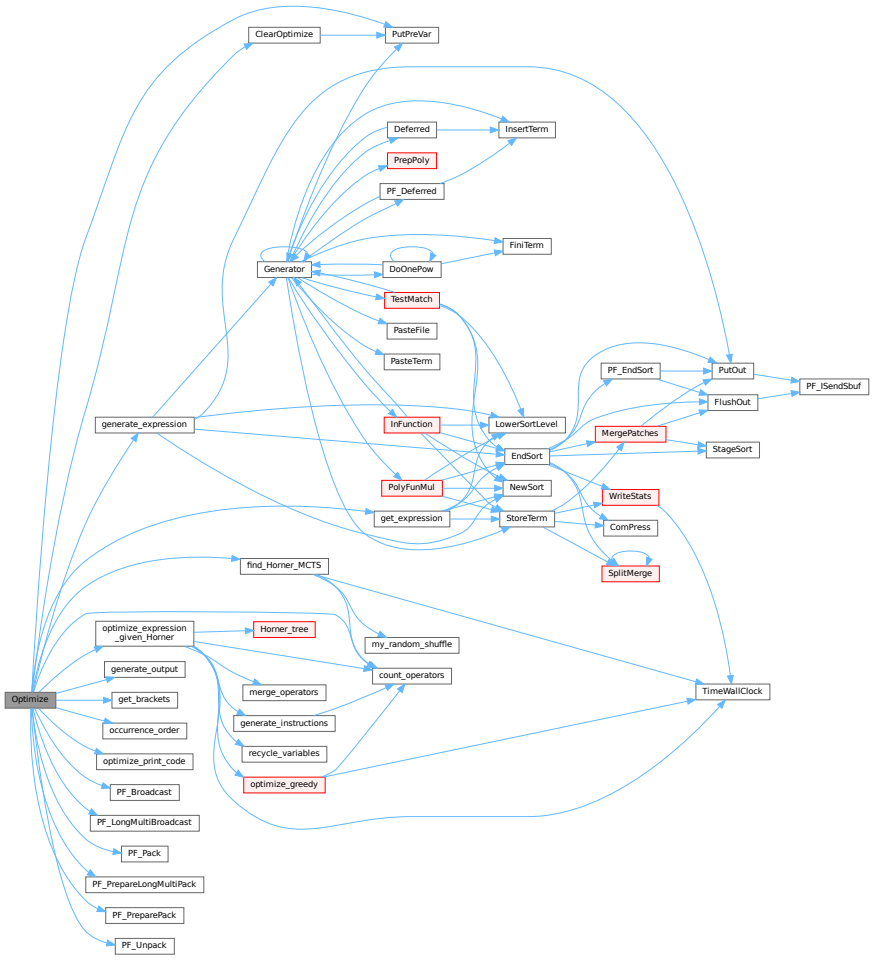
(3) and (4) find_Horner_MCTS and optimize_expression_given_Horner are parallelized.

(5), (6a) and (6b) are needed only on the master.

Definition at line 4641 of file optimize.cc.

References ClearOptimize(), count_operators(), find_Horner_MCTS(), generate_expression(), generate_output(), get_brackets(), get_expression(), occurrence_order(), optimize_expression_given_Horner(), optimize_print_code(), PF_Broadcast(), PF_LongMultiBroadcast(), PF_Pack(), PF_PrepareLongMultiPack(), PF_PreparePack(), PF_Unpack(), and PutPreVar().

Here is the call graph for this function:



4.95.31.1 ClearOptimize()

```
int ClearOptimize (
            void  )
```

Optimization of expression

4.95.32 Description

Clears the buffers that were used for optimization output. Clears the expression from the buffers (marks it to be dropped). Note: we need to use the expression by its name, because numbers may change if we drop other expressions between when we do the optimizations and clear the results (in execute.c). Also this is not 100% safe, because we could overwrite the optimized expression. But that can be done only in a Local or Global statement and hence we only have to test there that we might have to call ClearOptimize first. (in file comexpr.c)

Definition at line 4978 of file optimize.cc.

References PutPreVar().

Referenced by Optimize().

Here is the call graph for this function:

4.95.33 Variable Documentation

4.95.33.1 OPER_ADD

```
const WORD OPER_ADD = -1
```

Definition at line 120 of file optimize.cc.

4.95.33.2 OPER_MUL

```
const WORD OPER_MUL = -2
```

Definition at line 121 of file optimize.cc.

4.95.33.3 OPER_COMMA

```
const WORD OPER_COMMA = -3
```

Definition at line 122 of file optimize.cc.

4.95.33.4 optimize_expr

```
WORD* optimize_expr
```

Definition at line 156 of file optimize.cc.

4.95.33.5 optimize_best_Horner_schemes

```
vector<vector<WORD> > optimize_best_Horner_schemes
```

Definition at line 157 of file optimize.cc.

4.95.33.6 optimize_num_vars

```
int optimize_num_vars
```

Definition at line 158 of file optimize.cc.

4.95.33.7 optimize_best_num_oper

```
int optimize_best_num_oper
```

Definition at line 159 of file optimize.cc.

4.95.33.8 optimize_best_instr

```
vector<WORD> optimize_best_instr
```

Definition at line 160 of file optimize.cc.

4.95.33.9 optimize_best_vars

```
vector<WORD> optimize_best_vars
```

Definition at line 161 of file optimize.cc.

4.95.33.10 mcts_factorized

```
bool mcts_factorized
```

Definition at line 164 of file optimize.cc.

4.95.33.11 mcts_separated

```
bool mcts_separated
```

Definition at line 164 of file optimize.cc.

4.95.33.12 mcts_vars

```
vector<WORD> mcts_vars
```

Definition at line 165 of file optimize.cc.

4.95.33.13 mcts_root

```
tree_node mcts_root
```

Definition at line 166 of file optimize.cc.

4.95.33.14 mcts_expr_score

```
int mcts_expr_score
```

Definition at line 167 of file optimize.cc.

4.95.33.15 mcts_best_schemes

```
set<pair<int,vector<WORD> > > mcts_best_schemes
```

Definition at line 168 of file optimize.cc.

4.96 optimize.cc

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /*
00032     #] License :
00033     #[ includes :
00034 */
00035
00036 //#define DEBUG
00037 //#define DEBUG_MORE
00038 //#define DEBUG_MCTS
00039 //#define DEBUG_GREEDY
00040
00041 #ifdef HAVE_CONFIG_H
00042 #ifndef CONFIG_H_INCLUDED
00043 #define CONFIG_H_INCLUDED
00044 #include <config.h>
00045 #endif
00046 #endif
00047
00048 #include <vector>
00049 #include <stack>
00050 #include <algorithm>
00051 #include <set>
00052 #include <map>
00053 #include <climits>
00054 #include <cmath>
00055 #include <string>
00056 #include <algorithm>
00057 #include <iostream>
00058
00059 #ifdef APPLE64
00060 #ifndef HAVE_UNORDERED_MAP
00061 #define HAVE_UNORDERED_MAP
00062 #endif
```

```
00063 #ifndef HAVE_UNORDERED_SET
00064 #define HAVE_UNORDERED_SET
00065 #endif
00066 #endif
00067
00068 #ifdef HAVE_UNORDERED_MAP
00069     #include <unordered_map>
00070     using std::unordered_map;
00071 #elif !defined(HAVE_TR1_UNORDERED_MAP) && defined(HAVE_BOOST_UNORDERED_MAP_HPP)
00072     #include <boost/unordered_map.hpp>
00073     using boost::unordered_map;
00074 #else
00075     #include <tr1/unordered_map>
00076     using std::tr1::unordered_map;
00077 #endif
00078 #ifdef HAVE_UNORDERED_SET
00079     #include <unordered_set>
00080     using std::unordered_set;
00081 #elif !defined(HAVE_TR1_UNORDERED_SET) && defined(HAVE_BOOST_UNORDERED_SET_HPP)
00082     #include <boost/unordered_set.hpp>
00083     using boost::unordered_set;
00084 #else
00085     #include <tr1/unordered_set>
00086     using std::tr1::unordered_set;
00087 #endif
00088
00089 #if defined(HAVE_BUILTIN_POPCOUNT)
00090     static inline int popcount(unsigned int x) {
00091         return __builtin_popcount(x);
00092     }
00093 #elif defined(HAVE_POPCNT)
00094     #include <intrin.h>
00095     static inline int popcount(unsigned int x) {
00096         return __popcnt(x);
00097     }
00098 #else
00099     static inline int popcount(unsigned int x) {
00100         int count = 0;
00101         while (x > 0) {
00102             if ((x & 1) == 1) count++;
00103             x >>= 1;
00104         }
00105         return count;
00106     }
00107 #endif
00108
00109 extern "C" {
00110 #include "form3.h"
00111 }
00112
00113 //#ifdef DEBUG
00114 #include "mytime.h"
00115 //#endif
00116
00117 using namespace std;
00118
00119 // operators
00120 const WORD OPER_ADD = -1;
00121 const WORD OPER_MUL = -2;
00122 const WORD OPER_COMMA = -3;
00123
00124 // class for a node of the MCTS tree
00125 class tree_node {
00126 public:
00127     vector<tree_node> childs;
00128     double sum_results;
00129     int num_visits;
00130     WORD var;
00131     bool finished;
00132
00133     tree_node (int _var=0):
00134         sum_results(0), num_visits(0), var(_var), finished(false) {}
00135
00136     tree_node(const tree_node& other):
00137         childs(other.childs),
00138         sum_results(other.sum_results),
00139         num_visits(other.num_visits),
00140         var(other.var),
00141         finished(other.finished) {}
00142
00143     tree_node& operator=(const tree_node& other) {
00144         if (this != &other) {
00145             childs = other.childs;
00146             sum_results = other.sum_results;
00147             num_visits = other.num_visits;
00148             var = other.var;
00149             finished = other.finished;
```

```
00150         }
00151         return *this;
00152     }
00153 };
00154
00155 // global variables for multithreading
00156 WORD *optimize_expr;
00157 vector<vector<WORD> > optimize_best_Horner_schemes;
00158 int optimize_num_vars;
00159 int optimize_best_num_oper;
00160 vector<WORD> optimize_best_instr;
00161 vector<WORD> optimize_best_vars;
00162
00163 // global variables for MCTS
00164 bool mcts_factorized, mcts_separated;
00165 vector<WORD> mcts_vars;
00166 tree_node mcts_root;
00167 int mcts_expr_score;
00168 set<pair<int,vector<WORD> > > mcts_best_schemes;
00169
00170 #ifdef WITHPTHREADS
00171 pthread_mutex_t optimize_lock;
00172 #endif
00173
00174 /*
00175     #] includes :
00176     #[ print_instr :
00177 */
00178
00179 void print_instr (const vector<WORD> &instr, WORD num)
00180 {
00181     const WORD *tbegin = &*instr.begin();
00182     const WORD *tend = tbegin+instr.size();
00183     for (const WORD *t=tbegin; t!=tend; t+=*(t+2)) {
00184         MesPrint("<%d> %a",num,t[2],t);
00185     }
00186 }
00187
00188 /*
00189     #] print_instr :
00190     #[ my_random_shuffle :
00191 */
00192
00200 template <class RandomAccessIterator>
00201 void my_random_shuffle (PHEAD RandomAccessIterator fr, RandomAccessIterator to) {
00202   for (int i=to-fr-1; i>0; --i)
00203         swap (fr[i],fr[wranf(BHEAD0) % (i+1)]);
00204 }
00205
00206 /*
00207     #] my_random_shuffle :
00208     #[ get_expression :
00209 */
00210
00211 static WORD comlist[3] = {TYPETOPOLYNOMIAL,3,DOALL};
00212
00225 LONG get_expression (int exprnr) {
00226
00227     GETIDENTITY;
00228
00229   AR.NoCompress = 1;
00230
00231   NewSort(BHEAD0);
00232   EXPRESSIONS e = Expressions+exprnr;
00233   SetScratch(AR.infile,&(e->onfile));
00234
00235   // get header term
00236   WORD *term = AT.WorkPointer;
00237     GetTerm(BHEAD term);
00238
00239   NewSort(BHEAD0);
00240
00241   // get terms
00242   while (GetTerm(BHEAD term) > 0) {
00243     AT.WorkPointer = term + *term;
00244     WORD *t1 = term;
00245     WORD *t2 = term + *term;
00246     if (ConvertToPoly(BHEAD t1,t2,comlist,1) < 0) return -1;
00247     int n = *t2;
00248     NCOPY(t1,t2,n);
00249     AT.WorkPointer = term + *term;
00250     if (StoreTerm(BHEAD term)) return -1;
00251   }
00252
00253   // sort and store in buffer
00254   LONG len = EndSort(BHEAD (WORD *)((void *)(&optimize_expr)),2);
00255   LowerSortLevel();
```

```
00256   AT.WorkPointer = term;
00257 
00258     return len;
00259 }
00260 
00261 /*
00262     #] get_expression :
00263     #[ PF_get_expression :
00264 */
00265 #ifdef WITHMPI
00266 
00267 // get_expression for ParFORM
00268 LONG PF_get_expression (int exprnr) {
00269     LONG len;
00270     if (PF.me == MASTER) {
00271         len = get_expression(exprnr);
00272     }
00273     if (PF.numtasks > 1) {
00274         PF_BroadcastBuffer(&optimize_expr, &len);
00275     }
00276     return len;
00277 }
00278 
00279 // replace get_expression called in Optimize
00280 #define get_expression PF_get_expression
00281 
00282 #endif
00283 /*
00284     #] PF_get_expression :
00285     #[ get_brackets :
00286 */
00287 
00302 vector<vector<WORD> > get_brackets () {
00303 
00304   // check for brackets in expression
00305   bool has_brackets = false;
00306   for (WORD *t=optimize_expr; *t!=0; t+=*t) {
00307     WORD *tend=t+*t; tend-=ABS(*(tend-1));
00308     for (WORD *t1=t+1; t1<tend; t1+=*(t1+1))
00309       if (*t1 == HAAKJE)
00310         has_brackets=true;
00311   }
00312 
00313   // replace brackets by SEPARATESYMBOL
00314   vector<vector<WORD> > brackets;
00315 
00316   if (has_brackets) {
00317         int exprlen=10; // we need potential space for an empty bracket
00318         for (WORD *t=optimize_expr; *t!=0; t+=*t)
00319             exprlen += *t;
00320     WORD *newexpr = (WORD *)Malloc1(exprlen*sizeof(WORD), "optimize newexpr");
00321 
00322     int i=0;
00323     int sep_power = 0;
00324 
00325     for (WORD *t=optimize_expr; *t!=0; t+=*t) {
00326       WORD *t1 = t+1;
00327 
00328             // scan for bracket
00329       vector<WORD> bracket;
00330       for (; *t1!=HAAKJE; t1+=*(t1+1))
00331         bracket.insert(bracket.end(), t1, t1+*(t1+1));
00332 
00333       if (brackets.size()==0 || bracket!=brackets.back()) {
00334         sep_power++;
00335         brackets.push_back(bracket);
00336       }
00337       t1+=*(t1+1);
00338 
00339       WORD left = t + *t - t1;
00340       bool more_symbols = (left != ABS(*(t+*t-1)));
00341 
00342             // add power of SEPARATESYMBOL
00343       newexpr[i++] = 1 + left + (more_symbols ? 2 : 4);
00344       newexpr[i++] = SYMBOL;
00345       newexpr[i++] = (more_symbols ? *(t1+1) + 2 : 4);
00346       newexpr[i++] = SEPARATESYMBOL;
00347       newexpr[i++] = sep_power;
00348 
00349             // add remaining terms
00350       if (more_symbols) {
00351         t1+=2;
00352         left-=2;
00353       }
00354       while (left-->0)
00355         newexpr[i++] = *(t1++);
00356     }
```

```
00357 /*
00358     We insert here an empty bracket that is zero.
00359     It is used for the case that there is only a single bracket which is
00360     outside the notation for trees at a later stage.
00361
00362     There may be a problem now in that in the case of sep_power==1
00363     newexpr is bigger than optimize_expr. We have to check that.
00364 */
00365     if ( sep_power == 1 )
00366     {
00367       WORD *t;
00368       vector<WORD> bracket;
00369       bracket.push_back(0);
00370       bracket.push_back(0);
00371       bracket.push_back(0);
00372       bracket.push_back(0);
00373       sep_power++;
00374       brackets.push_back(bracket);
00375       newexpr[i++] = 8;
00376       newexpr[i++] = SYMBOL;
00377       newexpr[i++] = 4;
00378       newexpr[i++] = SEPARATESYMBOL;
00379       newexpr[i++] = sep_power;
00380       newexpr[i++] = 1;
00381       newexpr[i++] = 1;
00382       newexpr[i++] = 3;
00383       newexpr[i++] = 0;
00384       for (t=optimize_expr; *t!=0; t+=*t) {}
00385       if ( t-optimize_expr+1 < i ) {  // We have to redo this
00386         M_free(optimize_expr,"$-sort space");
00387         optimize_expr = (WORD *)Malloc1(i*sizeof(WORD),"$-sort space");
00388       }
00389     }
00390     else {
00391         newexpr[i++] = 0;
00392     }
00393     memcpy(optimize_expr, newexpr, i*sizeof(WORD));
00394     M_free(newexpr,"optimize newexpr");
00395
00396     // if factorized, replace SEP by FAC and remove brackets
00397     if (brackets[0].size()>0 && brackets[0][2]==FACTORSYMBOL) {
00398         for (WORD *t=optimize_expr; *t!=0; t+=*t) {
00399             if (*t == ABS(*(t+*t-1))+1) continue;
00400             if (t[1]==SYMBOL)
00401                 for (int i=3; i<t[2]; i+=2)
00402                     if (t[i]==SEPARATESYMBOL) t[i]=FACTORSYMBOL;
00403         }
00404         return vector<vector<WORD> >();
00405     }
00406   }
00407
00408   return brackets;
00409 }
00410
00411 /*
00412     #] get_brackets :
00413     #[ count_operators :
00414 */
00415
00422 int count_operators (const WORD *expr, bool print=false) {
00423
00424     int n=0;
00425     while (*(expr+n)!=0) n+=*(expr+n);
00426
00427     int cntpow=0, cntmul=0, cntadd=0, sumpow=0;
00428     WORD maxpowfac=1, maxpowsep=1;
00429
00430     for (const WORD *t=expr; *t!=0; t+=*t) {
00431         if (t!=expr) cntadd++;              // new term
00432         if (*t==ABS(*(t+*t-1))+1) continue; // only coefficient
00433
00434         int cntsym=0;
00435
00436         if (t[1]==SYMBOL)
00437             for (int i=3; i<t[2]; i+=2) {
00438                 if (t[i]==FACTORSYMBOL) {
00439                     maxpowfac = max(maxpowfac, t[i+1]);
00440                     continue;
00441                 }
00442                 if (t[i]==SEPARATESYMBOL) {
00443                     maxpowsep = max(maxpowsep, t[i+1]);
00444                     continue;
00445                 }
00446                 if (t[i+1]>2) {          // (extra)symbol power>2
00447                     cntpow++;
00448                     sumpow += (int)floor(log(t[i+1])/log(2.0)) + popcount(t[i+1]) - 1;
00449                 }
```

```
00450                 if (t[i+1]==2) cntmul++; // (extra)symbol squared
00451                 cntsym++;
00452             }
00453 
00454         if (ABS(*(t+*t-1))!=3 || *(t+*t-2)!=1 || *(t+*t-3)!=1) cntsym++; // non +/-1 coefficient
00455 
00456         if (cntsym > 0) cntmul+=cntsym-1;
00457     }
00458 
00459     cntadd -= maxpowfac-1;
00460     cntmul += maxpowfac-1;
00461 
00462     cntadd -= maxpowsep-1;
00463 
00464     if (print)
00465         MesPrint ("*** STATS: original  %lP %lM %lA : %l", cntpow,cntmul,cntadd,sumpow+cntmul+cntadd);
00466 
00467     return sumpow+cntmul+cntadd;
00468 }
00469 
00476 int count_operators (const vector<WORD> &instr, bool print=false) {
00477 
00478     int cntpow=0, cntmul=0, cntadd=0, sumpow=0;
00479 
00480     const WORD *ebegin = &*instr.begin();
00481     const WORD *eend = ebegin+instr.size();
00482 
00483     for (const WORD *e=ebegin; e!=eend; e+=*(e+2)) {
00484         for (const WORD *t=e+3; *t!=0; t+=*t) {
00485             if (t!=e+3) {
00486                 if (*(e+1)==OPER_ADD) cntadd++;     // new term
00487                 if (*(e+1)==OPER_MUL) cntmul++;     // new term
00488             }
00489             if (*t == ABS(*(t+*t-1))+1) continue;   // only coefficient
00490             if (*(t+1)==SYMBOL || *(t+1)==EXTRASYMBOL) {
00491                 if (*(t+4)==2) cntmul++;            // (extra)symbol squared
00492                 if (*(t+4)>2) {                     // (extra)symbol power>2
00493                     cntpow++;
00494                     sumpow += (int)floor(log(*(t+4))/log(2.0)) + popcount(*(t+4)) - 1;
00495                 }
00496             }
00497             if (ABS(*(t+*t-1))!=3 || *(t+*t-2)!=1 || *(t+*t-3)!=1) cntmul++; // non +/-1 coefficient
00498         }
00499     }
00500 
00501     if (print)
00502         MesPrint ("*** STATS: optimized %lP %lM %lA : %l", cntpow,cntmul,cntadd,sumpow+cntmul+cntadd);
00503 
00504     return sumpow+cntmul+cntadd;
00505 }
00506 
00507 /*
00508     #] count_operators :
00509     #[ occurrence_order :
00510 */
00511 
00519 vector<WORD> occurrence_order (const WORD *expr, bool rev) {
00520 
00521     // count the number of occurrences of variables
00522     map<WORD,int> cnt;
00523     for (const WORD *t=expr; *t!=0; t+=*t) {
00524         if (*t == ABS(*(t+*t-1))+1) continue; // skip constant term
00525         if (t[1] == SYMBOL)
00526             for (int i=3; i<t[2]; i+=2)
00527                 cnt[t[i]]++;
00528     }
00529 
00530     bool is_fac=false, is_sep=false;
00531     if (cnt.count(FACTORSYMBOL)) {
00532         is_fac=true;
00533         cnt.erase(FACTORSYMBOL);
00534     }
00535     if (cnt.count(SEPARATESYMBOL)) {
00536         is_sep=true;
00537         cnt.erase(SEPARATESYMBOL);
00538     }
00539 
00540     // determine the order of the variables
00541     vector<pair<int,WORD> > cnt_order;
00542     for (map<WORD,int>::iterator i=cnt.begin(); i!=cnt.end(); i++)
00543         cnt_order.push_back(make_pair(i->second, i->first));
00544     sort(cnt_order.rbegin(), cnt_order.rend());
00545 
00546     // create resulting order
00547     vector<WORD> order;
00548     for (int i=0; i<(int)cnt_order.size(); i++)
00549         order.push_back(cnt_order[i].second);
```

```
00550
00551     if (rev) reverse(order.begin(),order.end());
00552
00553     // add FACTORSYMBOL/SEPARATESYMBOL
00554     if (is_fac) order.insert(order.begin(), FACTORSYMBOL);
00555     if (is_sep) order.insert(order.begin(), SEPARATESYMBOL);
00556
00557     return order;
00558 }
00559
00560 /*
00561     #] occurrence_order :
00562     #[ Horner_tree :
00563 */
00564
00600 WORD getpower (const WORD *term, int var, int pos) {
00601     if (*term == ABS(*(term+*term-1))+1) return 0; // constant term
00602     if (2*pos+2 >= term[2]) return 0;              // too few symbols
00603     if (term[2*pos+3] != var) return 0;            // incorrect symbol
00604     return term[2*pos+4];                          // return power
00605 }
00606
00614 void fixarg (UWORD *t, WORD &n) {
00615     int an=ABS(n), sn=SGN(n);
00616     while (*(t+an-1)==0) an--;
00617     n=an*sn;
00618 }
00619
00620 void GcdLong_fix_args (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc) {
00621     fixarg(a,na);
00622     fixarg(b,nb);
00623     GcdLong(BHEAD a,na,b,nb,c,nc);
00624 }
00625
00626 void DivLong_fix_args(UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c,   WORD *nc, UWORD *d, WORD *nd)
    {
00627     fixarg(a,na);
00628     fixarg(b,nb);
00629     DivLong(a,na,b,nb,c,nc,d,nd);
00630 }
00631
00652 void build_Horner_tree (const WORD **terms, int numterms, int var, int maxvar, int pos, vector<WORD>
    *res) {
00653
00654     GETIDENTITY;
00655
00656     if (var == maxvar) {
00657         // Horner tree is finished, so add remaining terms unfactorized
00658         // (note: since only complete Horner schemes seem to be useful, numterms=1 here
00659
00660         for (int fr=0; fr<numterms; fr++) {
00661
00662             bool empty = true;
00663             const WORD *t = terms[fr];
00664
00665             // add symbols
00666             if (*t != ABS(*(t+*t-1))+1)
00667                 for (int i=3+2*pos; i<t[2]; i+=2) {
00668                     res->push_back(SYMBOL);
00669                     res->push_back(4);
00670                     res->push_back(t[i]);
00671                     res->push_back(t[i+1]);
00672                     if (!empty) res->push_back(OPER_MUL);
00673                     empty = false;
00674                 }
00675
00676             // if empty, add a 1, since the result should look like "... coeff *"
00677             if (empty) {
00678                 res->push_back(SNUMBER);
00679                 res->push_back(5);
00680                 res->push_back(1);
00681                 res->push_back(1);
00682                 res->push_back(3);
00683             }
00684
00685             // add coefficient
00686             res->push_back(SNUMBER);
00687             WORD n = ABS(*(t+*t-1));
00688             res->push_back(n+2);
00689             for (int i=0; i<n; i++)
00690                 res->push_back(*(t+*t-n+i));
00691             res->push_back(OPER_MUL);
00692
00693             if (fr>0) res->push_back(OPER_ADD);
00694         }
00695
00696         // result should end with gcd of the terms; right now this never
```

```
00697         // triggers, but if one would allow for incomplete Horner schemes,
00698         // one should extract the gcd here
00699         if (numterms > 1) {
00700             res->push_back(SNUMBER);
00701             res->push_back(5);
00702             res->push_back(1);
00703             res->push_back(1);
00704             res->push_back(3);
00705             res->push_back(OPER_MUL);
00706         }
00707     }
00708     else {
00709         // extract variable "var" and the gcd and proceed recursively
00710 
00711         WORD nnum = 0, nden = 0, ntmp = 0, ndum = 0;
00712         UWORD *num = NumberMalloc("build_Horner_tree");
00713         UWORD *den = NumberMalloc("build_Horner_tree");
00714         UWORD *tmp = NumberMalloc("build_Horner_tree");
00715         UWORD *dum = NumberMalloc("build_Horner_tree");
00716 
00717         // previous coefficient for gcd extraction or coefficient multiplication
00718         int prev_coeff_idx = -1;
00719 
00720         for (int fr=0; fr<numterms;) {
00721 
00722             // find power of current term
00723             WORD pow = getpower(terms[fr], var, pos);
00724 
00725             // find all terms with that power
00726             int to=fr+1;
00727             while (to<numterms && getpower(terms[to], var, pos) == pow) to++;
00728 
00729             // recursively build Horner tree of all terms proportional to var^pow
00730             build_Horner_tree (terms+fr, to-fr, var+1, maxvar, pow==0?pos:pos+1, res);
00731 
00732             if (AN.poly_vars[var] != FACTORSYMBOL && AN.poly_vars[var] != SEPARATESYMBOL) {
00733                 // if normal symbol, find gcd(numerators) and gcd(denominators)
00734                 WORD  n1 = res->at(res->size()-2) / 2;
00735                 WORD *t1 = &res->at(res->size()-2-2*ABS(n1));
00736 
00737                 WORD *t2 = fr==0 ? t1 : &res->at(prev_coeff_idx);
00738                 WORD  n2 = fr==0 ? n1 : *(t2+*(t2+1)-1) / 2;
00739                 if (fr>0) t2+=2;
00740 
00741                 GcdLong_fix_args(BHEAD (UWORD *)t1,n1,(UWORD *)t2,n2,num,&nnum);
00742                 GcdLong_fix_args(BHEAD (UWORD *)t1+ABS(n1),ABS(n1),(UWORD
      *)t2+ABS(n2),ABS(n2),den,&nden);
00743 
00744                 // divide out gcds; note: leading zeroes can be added here
00745                 for (int i=0; i<2; i++) {
00746                     if (i==1 && fr==0) break;
00747 
00748                     WORD *t = i==0 ? t1 : t2;
00749                     WORD n = i==0 ? n1 : n2;
00750 
00751                     DivLong_fix_args((UWORD *)t, n, num, nnum, tmp, &ntmp, dum, &ndum);
00752                     for (int j=0; j<ABS(ntmp); j++) *t++ = tmp[j];
00753                     for (int j=0; j<ABS(n)-ABS(ntmp); j++) *t++ = 0;
00754 
00755                     if (SGN(ntmp) != SGN(n)) n=-n;
00756 
00757                     DivLong_fix_args((UWORD *)t, n, den, nden, tmp, &ntmp, dum, &ndum);
00758                     for (int j=0; j<ABS(ntmp); j++) *t++ = tmp[j];
00759                     for (int j=0; j<ABS(n)-ABS(ntmp); j++) *t++ = 0;
00760 
00761                     *t++ = SGN(n) * (2*ABS(n)+1);
00762                 }
00763 
00764                 // add the addition operator
00765                 if (fr>0) res->push_back(OPER_ADD);
00766 
00767                 // add the power of "var"
00768                 WORD nextpow = (to==numterms ? 0 : getpower(terms[to], var, pos));
00769 
00770                 if (pow-nextpow > 0) {
00771                     res->push_back(SYMBOL);
00772                     res->push_back(4);
00773                     res->push_back(var);
00774                     res->push_back(pow-nextpow);
00775                     res->push_back(OPER_MUL);
00776                 }
00777 
00778                 // add the extracted gcd
00779                 res->push_back(SNUMBER);
00780                 WORD n = MaX(ABS(nnum),nden);
00781                 res->push_back(n*2+3);
00782                 for (int i=0; i<ABS(nnum); i++) res->push_back(num[i]);
```

```
00783                 for (int i=0; i<n-ABS(nnum); i++) res->push_back(0);
00784                 for (int i=0; i<nden; i++) res->push_back(den[i]);
00785                 for (int i=0; i<n-ABS(nden); i++) res->push_back(0);
00786                 res->push_back(SGN(nnum)*(2*n+1));
00787                 res->push_back(OPER_MUL);
00788 
00789                 prev_coeff_idx = res->size() - ABS(res->at(res->size()-2)) - 3;
00790             }
00791             else if (AN.poly_vars[var]==FACTORSYMBOL) {
00792 
00793                 // if factorsymbol, multiply overall integer contents
00794 
00795                 if (fr>0) {
00796                     WORD  n1 = res->at(res->size()-2) / 2;
00797                     WORD *t1 = &res->at(res->size()-2-2*ABS(n1));
00798                     WORD *t2 = &res->at(prev_coeff_idx);
00799                     WORD  n2 = *(t2+*(t2+1)-1) / 2;
00800                     t2+=2;
00801 
00802                     MulRat(BHEAD (UWORD *)t1,n1,(UWORD *)t2,n2,tmp,&ntmp);
00803 
00804                     // replace previous coefficient with 1
00805                     n2=ABS(n2);
00806                     for (int i=0; i<ABS(n2); i++)
00807                         t2[i] = t2[n2+i] = i==0 ? 1 : 0;
00808                     t2[2*n2] = 2*n2+1;
00809 
00810                     // remove this coefficient
00811                     for (int i=0; i<2*ABS(n1)+2; i++)
00812                         res->pop_back();
00813 
00814                     // add product
00815                     res->back() = 2*ABS(ntmp)+3;                    // adjust size of term
00816                     res->insert(res->end(), tmp, tmp+2*ABS(ntmp)); // num/den coefficient
00817                     res->push_back(SGN(ntmp)*(2*ABS(ntmp)+1));     // size of coefficient
00818                     res->push_back(OPER_MUL);                      // operator
00819                 }
00820 
00821                 prev_coeff_idx = res->size() - ABS(res->at(res->size()-2)) - 3;
00822 
00823                 // multiply previous factors with this factor
00824                 if (fr>0)
00825                     res->push_back(OPER_MUL);
00826             }
00827             else { // AN.poly_vars[var]==SEPARATESYMBOL
00828                 if (fr>0)
00829                     res->push_back(OPER_COMMA);
00830                 prev_coeff_idx = -1;
00831             }
00832 
00833             fr=to;
00834         }
00835 
00836         // cleanup
00837         NumberFree(dum,"build_Horner_tree");
00838         NumberFree(tmp,"build_Horner_tree");
00839         NumberFree(den,"build_Horner_tree");
00840         NumberFree(num,"build_Horner_tree");
00841     }
00842 }
00843 
00852 bool term_compare (const WORD *a, const WORD *b) {
00853     if (*a == ABS(*(a+*a-1))+1) return true; // coefficient comes first
00854     if (*b == ABS(*(b+*b-1))+1) return false;
00855     if (a[1]!=SYMBOL) return true;
00856     if (b[1]!=SYMBOL) return false;
00857     for (int i=3; i<a[2] && i<b[2]; i+=2) {
00858         if (a[i]  !=b[i]  ) return a[i]  >b[i];
00859         if (a[i+1]!=b[i+1]) return a[i+1]<b[i+1];
00860     }
00861     return a[2]<b[2];
00862 }
00863 
00873 vector<WORD> Horner_tree (const WORD *expr, const vector<WORD> &order) {
00874 #ifdef DEBUG
00875     MesPrint ("*** [%s, w=%w] CALL: Horner_tree(%a)", thetime_str().c_str(), order.size(), &order[0]);
00876 #endif
00877 
00878     GETIDENTITY;
00879 
00880     // find the renumbering scheme (new numbers are 0,1,...,#vars-1)
00881     map<WORD,WORD> renum;
00882     AN.poly_num_vars = order.size();
00883     AN.poly_vars = (WORD *)Malloc1(AN.poly_num_vars*sizeof(WORD), "AN.poly_vars");
00884     for (int i=0; i<AN.poly_num_vars; i++) {
00885         AN.poly_vars[i] = order[i];
00886         renum[order[i]] = i;
```

```
00887     }
00888
00889     // sort variables in individual terms using bubble sort
00890     WORD* sorted;
00891     WORD* dynamicAlloc = 0;
00892     LONG sumsize = 0;
00893
00894     for (const WORD *t=expr; *t!=0; t+=*t) {
00895         sumsize += *t;
00896     }
00897
00898     // We actually need sumsize + 1 WORDS available, due to the "*sorted = 0;"
00899     // at the end of the sort loop.
00900     if ( AT.WorkPointer + sumsize + 1 > AT.WorkTop ) {
00901         dynamicAlloc = (WORD*)Malloc1(sizeof(*dynamicAlloc)*(sumsize+1), "Horner_tree alloc");
00902         sorted = dynamicAlloc;
00903     }
00904     else {
00905         sorted = AT.WorkPointer;
00906     }
00907
00908     for (const WORD *t=expr; *t!=0; t+=*t) {
00909         memcpy(sorted, t, *t*sizeof(WORD));
00910
00911         if (*t != ABS(*(t+*t-1))+1) {
00912             // non-constant term
00913
00914             // renumber variables, adding new variables if necessary
00915             for (int i=3; i<sorted[2]; i+=2) {
00916                 if (!renum.count(sorted[i])) {
00917                     renum[sorted[i]] = AN.poly_num_vars;
00918
00919                     WORD *new_poly_vars = (WORD *)Malloc1((AN.poly_num_vars+1)*sizeof(WORD),
    "AN.poly_vars");
00920                     memcpy(new_poly_vars, AN.poly_vars, AN.poly_num_vars*sizeof(WORD));
00921                     M_free(AN.poly_vars,"poly_vars");
00922                     AN.poly_vars = new_poly_vars;
00923                     AN.poly_vars[AN.poly_num_vars] = sorted[i];
00924                     AN.poly_num_vars++;
00925                 }
00926                 sorted[i] = renum[sorted[i]];
00927             }
00928             // order variables
00929             for (int i=0; i<sorted[2]/2; i++)
00930                 for (int j=3; j+2<sorted[2]; j+=2)
00931                     if (sorted[j] > sorted[j+2]) {
00932                         swap(sorted[j]  , sorted[j+2]);
00933                         swap(sorted[j+1], sorted[j+3]);
00934                     }
00935         }
00936
00937         sorted += *sorted;
00938     }
00939
00940     *sorted = 0;
00941     if ( dynamicAlloc ) {
00942         sorted = dynamicAlloc;
00943     }
00944     else {
00945         sorted = AT.WorkPointer;
00946     }
00947
00948     // find pointers to all terms and sort them efficiently
00949     vector<const WORD *> terms;
00950     for (const WORD *t=sorted; *t!=0; t+=*t)
00951         terms.push_back(t);
00952     sort(terms.rbegin(),terms.rend(),term_compare);
00953
00954     // construct the Horner tree
00955     vector<WORD> res;
00956     build_Horner_tree(&terms[0], terms.size(), 0, AN.poly_num_vars, 0, &res);
00957
00958     // remove leading zeroes in coefficients
00959     int j=0;
00960     for (int i=0; i<(int)res.size();) {
00961         if (res[i]==OPER_ADD || res[i]==OPER_MUL || res[i]==OPER_COMMA)
00962             res[j++] = res[i++];
00963         else if (res[i]==SYMBOL) {
00964             memmove(&res[j], &res[i], res[i+1]*sizeof(WORD));
00965             i+=res[j+1];
00966             j+=res[j+1];
00967         }
00968         else if (res[i]==SNUMBER) {
00969             int n = (res[i+1]-2)/2;
00970             int dn = 0;
00971             while (res[i+1+n-dn]==0 && res[i+1+2*n-dn]==0) dn++;
00972             res[j++] = SNUMBER;
```

```
00973             res[j++] = 2*(n-dn) + 3;
00974             memmove(&res[j], &res[i+2], (n-dn)*sizeof(WORD));
00975             j+=n-dn;
00976             memmove(&res[j], &res[i+n+2], (n-dn)*sizeof(WORD));
00977             j+=n-dn;
00978             res[j++] = SGN(res[i+2*n+2])*(2*(n-dn)+1);
00979             i+=2*n+3;
00980         }
00981     }
00982     res.resize(j);
00983 
00984     if ( dynamicAlloc ) {
00985         M_free(dynamicAlloc, "Horner_tree alloc");
00986     }
00987 
00988 #ifdef DEBUG
00989     MesPrint ("*** [%s, w=%w] DONE: Horner_tree(%a)", thetime_str().c_str(), order.size(), &order[0]);
00990 #endif
00991 
00992     return res;
00993 }
00994 
00995 /*
00996     #] Horner_tree :
00997     #[ print_tree :
00998 */
00999 
01000 // print Horner tree (for debugging)
01001 void print_tree (const vector<WORD> &tree) {
01002 
01003     GETIDENTITY;
01004 
01005     for (int i=0; i<(int)tree.size();) {
01006         if (tree[i]==OPER_ADD) {
01007             MesPrint("+%");
01008             i++;
01009         }
01010         else if (tree[i]==OPER_MUL) {
01011             MesPrint("*%");
01012             i++;
01013         }
01014         else if (tree[i]==OPER_COMMA) {
01015             MesPrint(",%");
01016             i++;
01017         }
01018         else if (tree[i]==SNUMBER) {
01019             UBYTE buf[100];
01020             int n = tree[i+tree[i+1]-1]/2;
01021             PrtLong((UWORD *)&tree[i+2], n, buf);
01022             int l = strlen((char *)buf);
01023             buf[l]='/';
01024             n=ABS(n);
01025             PrtLong((UWORD *)&tree[i+2+n], n, buf+l+1);
01026             MesPrint("%s%",buf);
01027             i+=tree[i+1];
01028         }
01029         else if (tree[i]==SYMBOL) {
01030             if (AN.poly_vars[tree[i+2]] < 10000)
01031                 MesPrint("%s^%d%", VARNAME(symbols,AN.poly_vars[tree[i+2]]), tree[i+3]);
01032             else
01033                 MesPrint("Z%d^%d%", MAXVARIABLES-AN.poly_vars[tree[i+2]], tree[i+3]);
01034             i+=tree[i+1];
01035         }
01036         else {
01037             MesPrint("error");
01038             exit(1);
01039         }
01040 
01041         MesPrint(" %");
01042     }
01043 
01044     MesPrint("");
01045 }
01046 
01047 /*
01048     #] print_tree :
01049     #[ generate_instructions :
01050 */
01051 
01052 
01053 struct CSEHash {
01054     size_t operator()(const vector<WORD>& n) const {
01055         return n[0];
01056     }
01057 };
01058 
01059 struct CSEEq {
```

```
01060     bool operator()(const vector<WORD>& lhs, const vector<WORD>& rhs) const {
01061             if (lhs.size() != rhs.size()) return false;
01062             for (unsigned int i = 1; i < lhs.size(); i++) {
01063                 if (lhs[i] != rhs[i]) return false;
01064             }
01065             return true;
01066     }
01067 };
01068
01069
01070 template<typename T> size_t hash_range(T* array, int size) {
01071     size_t hash = 0;
01072
01073     for (int i = 0; i < size; i++) {
01074         hash ^= array[i] + 0x9e3779b9 + (hash « 6) + (hash » 2);
01075     }
01076
01077     return hash;
01078 }
01079
01132 vector<WORD> generate_instructions (const vector<WORD> &tree, bool do_CSE) {
01133 #ifdef DEBUG
01134     MesPrint ("*** [%s, w=%w] CALL: generate_instructions(cse=%d)",
01135                         thetime_str().c_str(), do_CSE?1:0);
01136 #endif
01137
01138     typedef unordered_map<vector<WORD>, int, CSEHash, CSEEq> csemap;
01139     csemap ID;
01140
01141     // reserve lots of space, to prevent later rehashes
01142     // TODO: what if this is too large? make a parameter?
01143     if (do_CSE) {
01144             ID.rehash(mcts_expr_score * 2);
01145     }
01146
01147     // s is a stack of operands to process when you encounter operators
01148     // in the postfix expression tree. Operands consist of three WORDs,
01149     // formatted as the LHS/RHS of the keys in ID.
01150     stack<int> s;
01151     vector<WORD> instr;
01152     WORD numinstr = 0;
01153     vector<WORD> x;
01154
01155     // process the expression tree
01156     for (int i=0; i<(int)tree.size();) {
01157         x.clear();
01158
01159         if (tree[i]==SNUMBER) {
01160             WORD hash = hash_range(&tree[i], tree[i + 1]);
01161             x.push_back(hash);
01162             x.push_back(SNUMBER);
01163             x.insert(x.end(),&tree[i],&tree[i]+tree[i+1]);
01164             int sign = SGN(x.back());
01165             x.back() *= sign;
01166
01167             std::pair<csemap::iterator, bool> suc = ID.insert(csemap::value_type(x, i + 1));
01168             s.push(0);
01169             s.push(sign * suc.first->second);
01170             s.push(SNUMBER);
01171             s.push(hash);
01172             i+=tree[i+1];
01173         }
01174         else if (tree[i]==SYMBOL) {
01175             WORD hash = hash_range(&tree[i], tree[i + 1]);
01176             if (tree[i+3]>1) {
01177                 x.push_back(hash);
01178                 x.push_back(SYMBOL);
01179                 x.push_back(tree[i+2]+1); // variable (1-indexed)
01180                 x.push_back(tree[i+3]);   // power
01181
01182                 csemap::iterator it = ID.find(x);
01183                 if (do_CSE && it != ID.end()) {
01184                     // already-seen power of a symbol
01185                     s.push(1);
01186                     s.push(it->second);
01187                     s.push(EXTRASYMBOL);
01188                 } else {
01189                     //MesPrint("strange");
01190                     if (numinstr == MAXPOSITIVE) {
01191                         MesPrint((char *)"ERROR: too many temporary variables needed in
      optimization");
01192                         Terminate(-1);
01193                     }
01194
01195                     // new power greater than 1 of a symbol
01196                     instr.push_back(numinstr);   // expr.nr
01197                     instr.push_back(OPER_MUL);   // operator
```

```
01198                     instr.push_back(9+OPTHEAD);  // length total
01199                     instr.push_back(8);          // length operand
01200                     instr.push_back(SYMBOL);     // SYMBOL
01201                     instr.push_back(4);          // length symbol
01202                     instr.push_back(tree[i+2]);  // variable
01203                     instr.push_back(tree[i+3]);  // power
01204                     instr.push_back(1);          // numerator
01205                     instr.push_back(1);          // denominator
01206                     instr.push_back(3);          // length coeff
01207                     instr.push_back(0);          // trailing 0
01208 
01209                     ID[x] = ++numinstr;
01210                     s.push(1);
01211                     s.push(numinstr);
01212                     s.push(EXTRASYMBOL);
01213                 }
01214             }
01215             else {
01216                 // power of 1
01217                 s.push(tree[i+3]);   // power
01218                 s.push(tree[i+2]+1); // variable (1-indexed)
01219                 s.push(SYMBOL);
01220             }
01221 
01222             s.push(hash); // push hash
01223             i+=tree[i+1];
01224         }
01225         else { // tree[i]==OPERATOR
01226             int oper = tree[i];
01227             i++;
01228 
01229             x.push_back(0); // placeholder for hash
01230             x.push_back(oper);
01231             vector<WORD> hash;
01232             hash.push_back(oper);
01233 
01234             // get two operands from the stack
01235             for (int operand=0; operand<2; operand++) {
01236                 hash.push_back(s.top()); s.pop();
01237                 x.push_back(s.top()); s.pop();
01238                 x.push_back(s.top()); s.pop();
01239                 x.push_back(s.top()); s.pop();
01240             }
01241 
01242             x[0] = hash_range(&hash[0], 3);
01243 
01244             // get rid of multiplications by +/-1
01245             if (oper==OPER_MUL) {
01246                 bool do_continue = false;
01247 
01248                 for (int operand=0; operand<2; operand++) {
01249                     int idx_oper1 = operand==0 ? 2 : 5;
01250                     int idx_oper2 = operand==0 ? 5 : 2;
01251 
01252                     // check whether operand 1 equals +/-1
01253                     if (x[idx_oper1]==SNUMBER) {
01254 
01255                         int idx = ABS(x[idx_oper1+1])-1;
01256                         if (tree[idx+2]==1 && tree[idx+3]==1 && ABS(tree[idx+4])==3) {
01257                             // push +/- other operand and continue
01258                             s.push(x[idx_oper2+2]);
01259                             s.push(x[idx_oper2+1]*SGN(x[idx_oper1+1]));
01260                             s.push(x[idx_oper2]);
01261                             s.push(hash[1 + (operand + 1) % 2]);
01262                             do_continue = true;
01263                             break;
01264                         }
01265                     }
01266                 }
01267 
01268                 if (do_continue) continue;
01269             }
01270 
01271             // check whether this subexpression has been seen before
01272             // if not, generate instruction to define it
01273             csemap::iterator it = ID.find(x);
01274             if (!do_CSE || it == ID.end()) {
01275                 if (numinstr == MAXPOSITIVE) {
01276                     MesPrint((char *)"ERROR: too many temporary variables needed in optimization");
01277                     Terminate(-1);
01278                 }
01279 
01280                 instr.push_back(numinstr); // expr.nr.
01281                 instr.push_back(x[1]);     // operator
01282                 instr.push_back(3);        // length
01283 
01284                 ID[x] = ++numinstr;
```

```
01285
01286                 int lenidx = instr.size()-1;
01287
01288                 for (int j=0; j<2; j++)
01289                     if (x[3*j+2]==SYMBOL || x[3*j+2]==EXTRASYMBOL) {
01290                         instr.push_back(8);                         // length total
01291                         instr.push_back(x[3*j+2]);                  // (extra)symbol
01292                         instr.push_back(4);                         // length (extra)symbol
01293                         instr.push_back(ABS(x[3*j+3])-1);           // variable (0-indexed)
01294                         instr.push_back(x[3*j+4]);                  // power
01295                         instr.push_back(1);                         // numerator
01296                         instr.push_back(1);                         // denominator
01297                         instr.push_back(3*SGN(x[3*j+3]));           // length coeff
01298                         instr[lenidx] += 8;
01299                     }
01300                     else { // x[3*j+1]==SNUMBER
01301                         int t = ABS(x[3*j+3])-1;
01302                         instr.push_back(tree[t+1]-1);                                   // length number
01303                         instr.insert(instr.end(), &tree[t+2], &tree[t]+tree[t+1]); // digits
01304                         instr.back() *= SGN(instr.back()) * SGN(x[3*j+3]);
01305                         instr[lenidx] += tree[t+1]-1;
01306                     }
01307
01308                 instr.push_back(0); // trailing 0
01309                 instr[lenidx]++;
01310             }
01311
01312             // push new expression on the stack
01313             s.push(1);
01314             s.push(ID[x]);
01315             s.push(EXTRASYMBOL);
01316             s.push(x[0]); // push hash
01317         }
01318     }
01319
01320 #ifdef DEBUG
01321     MesPrint ("*** [%s, w=%w] DONE: generate_instructions(cse=%d) : numoper=%d",
01322                         thetime_str().c_str(), do_CSE?1:0, count_operators(instr));
01323 #endif
01324
01325     return instr;
01326 }
01327
01328 /*
01329     #] generate_instructions :
01330     #[ count_operators_cse :
01331 */
01332
01333
01341 int count_operators_cse (const vector<WORD> &tree) {
01342     //MesPrint ("*** [%s] Starting CSEE", thetime_str().c_str());
01343
01344     typedef unordered_map<vector<WORD>, int, CSEHash, CSEEq> csemap;
01345     csemap ID;
01346
01347     // reserve lots of space, to prevent later rehashes
01348     // TODO: what if this is too large? make a parameter?
01349     ID.rehash(mcts_expr_score * 2);
01350
01351     // s is a stack of operands to process when you encounter operators
01352     // in the postfix expression tree. Operands consist of three WORDs,
01353     // formatted as the LHS/RHS of the keys in ID.
01354     stack<int> s;
01355     int numinstr = 0, numcommas = 0;
01356     vector<WORD> x;
01357
01358     // process the expression tree
01359     for (int i=0; i<(int)tree.size();) {
01360         x.clear();
01361
01362         if (tree[i]==SNUMBER) {
01363             WORD hash = hash_range(&tree[i], tree[i + 1]);
01364             x.push_back(hash);
01365             x.push_back(SNUMBER);
01366             x.insert(x.end(),&tree[i],&tree[i]+tree[i+1]);
01367             int sign = SGN(x.back());
01368             x.back() *= sign;
01369
01370             std::pair<csemap::iterator, bool> suc = ID.insert(csemap::value_type(x, i + 1));
01371             s.push(0);
01372             s.push(sign * suc.first->second);
01373             s.push(SNUMBER);
01374             s.push(hash);
01375             i+=tree[i+1];
01376         }
01377         else if (tree[i]==SYMBOL) {
01378             WORD hash = hash_range(&tree[i], tree[i + 1]);
```

```
01379             if (tree[i+3]>1) {
01380                 x.push_back(hash);
01381                 x.push_back(SYMBOL);
01382                 x.push_back(tree[i+2]+1); // variable (1-indexed)
01383                 x.push_back(tree[i+3]);   // power
01384 
01385                 csemap::iterator it = ID.find(x);
01386                 if (it != ID.end()) {
01387                     // already-seen power of a symbol
01388                     s.push(1);
01389                     s.push(it->second);
01390                     s.push(EXTRASYMBOL);
01391                 } else {
01392                     if (tree[i + 3] == 2)
01393                         numinstr++;
01394                     else
01395                         numinstr += (int)floor(log(tree[i+3])/log(2.0)) + popcount(tree[i+3]) - 1;
01396 
01397                     ID[x] = numinstr;
01398                     s.push(1);
01399                     s.push(numinstr);
01400                     s.push(EXTRASYMBOL);
01401                 }
01402             }
01403             else {
01404                 // power of 1
01405                 s.push(tree[i+3]);   // power
01406                 s.push(tree[i+2]+1); // variable (1-indexed)
01407                 s.push(SYMBOL);
01408             }
01409 
01410             s.push(hash); // push hash
01411 
01412             i+=tree[i+1];
01413         }
01414         else { // tree[i]==OPERATOR
01415             int oper = tree[i];
01416             i++;
01417 
01418             x.push_back(0); // placeholder for hash
01419             x.push_back(oper);
01420             vector<WORD> hash;
01421             hash.push_back(oper);
01422 
01423             // get two operands from the stack
01424             for (int operand=0; operand<2; operand++) {
01425                 hash.push_back(s.top()); s.pop();
01426                 x.push_back(s.top()); s.pop();
01427                 x.push_back(s.top()); s.pop();
01428                 x.push_back(s.top()); s.pop();
01429             }
01430 
01431 
01432             x[0] = hash_range(&hash[0], 3);
01433 
01434             // get rid of multiplications by +/-1
01435             if (oper==OPER_MUL) {
01436                 bool do_continue = false;
01437 
01438                 for (int operand=0; operand<2; operand++) {
01439                     int idx_oper1 = operand==0 ? 2 : 5;
01440                     int idx_oper2 = operand==0 ? 5 : 2;
01441 
01442                     // check whether operand 1 equals +/-1
01443                     if (x[idx_oper1]==SNUMBER) {
01444 
01445                         int idx = ABS(x[idx_oper1+1])-1;
01446                         if (tree[idx+2]==1 && tree[idx+3]==1 && ABS(tree[idx+4])==3) {
01447                             // push +/- other operand and continue
01448                             s.push(x[idx_oper2+2]);
01449                             s.push(x[idx_oper2+1]*SGN(x[idx_oper1+1]));
01450                             s.push(x[idx_oper2]);
01451                             s.push(hash[1 + (operand + 1) % 2]);
01452                             do_continue = true;
01453                             break;
01454                         }
01455                     }
01456                 }
01457 
01458                 if (do_continue) continue;
01459             }
01460 
01461             // check whether this subexpression has been seen before
01462             // if not, generate instruction to define it
01463             csemap::iterator it = ID.find(x);
01464             if (it == ID.end()) {
01465                 if (numinstr == MAXPOSITIVE) {
```

```
01466                     MesPrint((char *)"ERROR: too many temporary variables needed in optimization");
01467                     Terminate(-1);
01468                 }
01469 
01470                 if (oper == OPER_COMMA) numcommas++;
01471                 ID[x] = ++numinstr;
01472                 s.push(1);
01473                 s.push(numinstr);
01474                 s.push(EXTRASYMBOL);
01475             } else {
01476                 // push new expression on the stack
01477                 s.push(1);
01478                 s.push(it->second);
01479                 s.push(EXTRASYMBOL);
01480             }
01481 
01482             s.push(x[0]); // push hash
01483         }
01484     }
01485 
01486     //MesPrint ("*** [%s] Stopping CSEE", thetime_str().c_str());
01487     return numinstr - numcommas;
01488 }
01489 
01490 /*
01491     #] count_operators_cse :
01492     #[ count_operators_cse_topdown :
01493 */
01494 
01495 typedef struct node {
01496     const WORD* data;
01497     struct node* l;
01498     struct node* r; // TODO: add l,r to data?
01499     WORD sign; // TODO: use data for this?
01500     UWORD hash;
01501 
01502     node() : l(NULL), r(NULL), sign(1), hash(0) {};
01503     node(const WORD* data) : data(data), l(NULL), r(NULL), sign(1), hash(0) {};
01504 
01505     // a minus sign in the tree should only count as a different entry if
01506     // it is a compound expression: a = -a, but T+-V != T+V
01507     int cmp(const struct node* rhs) const {
01508         if (this == rhs) return 0;
01509 
01510         if (data[0] != rhs->data[0]) return data[0] < rhs->data[0] ? -1 : 1;
01511         int mod = data[0] == SNUMBER ? -1 : 0; // don't check sign, for numbers
01512         if (data[0] == SYMBOL || data[0] == SNUMBER) {
01513             for (int i = 0; i < data[1] + mod; i++) {
01514                 if (data[i] != rhs->data[i]) return data[i] < rhs->data[i] ? -1 : 1;
01515                 }
01516         } else {
01517             int lv = l->cmp(rhs->l);
01518             if (lv != 0) return lv;
01519             int rv = r->cmp(rhs->r);
01520             if (rv != 0) return rv;
01521 
01522             // TODO: only for ADD operation
01523             if (l->sign != rhs->l->sign) return l->sign < rhs->l->sign ? -1 : 1;
01524             if (r->sign != rhs->r->sign) return r->sign < rhs->r->sign ? -1 : 1;
01525         }
01526 
01527         return 0;
01528     }
01529 
01530         // less than operator
01531     bool operator() (const struct node* lhs, const struct node* rhs) const
01532     {
01533         return lhs->cmp(rhs) < 0;
01534     }
01535 
01536     void calcHash() {
01537         int mod = data[0] == SNUMBER ? -1 : 0; // don't check sign, for numbers
01538         if (data[0] == SYMBOL || data[0] == SNUMBER) {
01539             hash = hash_range(data, data[1] + mod);
01540         } else {
01541             if (l->hash == 0) l->calcHash();
01542             if (r->hash == 0) r->calcHash();
01543 
01544             // signs only matter for compound expressions
01545             size_t newr[] = {(size_t)data[0], l->hash, (size_t)l->sign, r->hash, (size_t)r->sign};
01546             hash = hash_range(newr, 5);
01547         }
01548     }
01549 } NODE;
01550 
01551 struct NodeHash {
01552     size_t operator()(const NODE* n) const {
```

```
01553         return n->hash; // already computed
01554     }
01555 };
01556
01557 struct NodeEq {
01558     bool operator()(const NODE* lhs, const NODE* rhs) const {
01559         return lhs->cmp(rhs) == 0;
01560     }
01561 };
01562
01563 NODE* buildTree(vector<WORD> &tree) {
01564     //MesPrint ("*** [%s] Starting CSEE topdown", thetime_str().c_str());
01565
01566     // allocate spaces for the tree, cannot be more nodes than tree size
01567     NODE* ar = (NODE*)Malloc1(tree.size() * sizeof(NODE), "CSE tree");
01568     NODE* c = 0;
01569     unsigned int curIndex = 0;
01570
01571     stack<NODE*> st;
01572     for (int i=0; i<(int)tree.size();) {
01573         c = ar + curIndex;
01574         new (c) NODE(&tree[i]); // placement new
01575         curIndex++;
01576
01577         if (tree[i]==SYMBOL || tree[i] == SNUMBER) {
01578             // extract the sign to a new class member
01579             if (tree[i] == SNUMBER) {
01580                 c->sign = SGN(tree[i + tree[i + 1] -1]);
01581             }
01582
01583             c->calcHash();
01584             st.push(c);
01585             i+=tree[i+1];
01586         } else {
01587             c->r = st.top(); st.pop();
01588             c->l = st.top(); st.pop();
01589
01590             // filter *1 and *-1
01591             // TODO: also multiply if there are two numbers?
01592             if (c->data[0] == OPER_MUL) {
01593                 NODE* ch[] = {c->r, c->l};
01594                 for (int j = 0; j < 2; j++)
01595                     if (ch[j]->data[0] == SNUMBER && ch[j]->data[1] == 5 && ch[j]->data[2]==1 &&
    ch[j]->data[3]==1) {
01596                         ch[(j+1)%2]->sign *= ch[j]->sign; // transfer sign
01597                         c = ch[(j+1)%2];
01598                         break;
01599                     }
01600             }
01601
01602             c->calcHash();
01603             st.push(c);
01604             i++;
01605         }
01606     }
01607
01608     // TODO: reallocate to smaller size? Could save memory
01609     //MesPrint("Memory difference: %d vs %d", curIndex, tree.size());
01610
01611     // we want to make the root of the tree the first element
01612     // so that we can easily free the array.
01613     // we swap the first element with the root
01614     // we need to change the pointer in the operator node that has this element as a child
01615     // TODO: check performance
01616     for (unsigned int i = 0; i < curIndex; i++) {
01617         if (ar[i].l == ar) ar[i].l = st.top();
01618         if (ar[i].r == ar) ar[i].r = st.top();
01619     }
01620
01621     swap(ar[0], *st.top());
01622     return ar;
01623 }
01624
01625 int count_operators_cse_topdown (vector<WORD> &tree) {
01626     typedef unordered_set<NODE*, NodeHash, NodeEq> nodeset;
01627     nodeset ID;
01628
01629     // reserve lots of space, to prevent later rehashes
01630     // TODO: what if this is too large? make a parameter?
01631     ID.rehash(mcts_expr_score * 2);
01632
01633     int numinstr = 0;
01634
01635     NODE* root = buildTree(tree);
01636
01637     stack<NODE*> stack;
01638     stack.push(root);
```

```
01639     while (!stack.empty())
01640     {
01641         NODE* c = stack.top();
01642         stack.pop();
01643 
01644         if (c->data[0] == SYMBOL) {
01645             if (c->data[3] > 1) {
01646                 std::pair<nodeset::iterator, bool> suc = ID.insert(c);
01647                 if (suc.second) { // new
01648                     if (c->data[3] == 2)
01649                         numinstr++;
01650                     else
01651                         numinstr += (int)floor(log(c->data[3])/log(2.0)) + popcount(c->data[3]) - 1;
01652                 }
01653             }
01654         } else {
01655             if (c->data[0] != SNUMBER) {
01656                 // operator
01657                 std::pair<nodeset::iterator, bool> suc = ID.insert(c);
01658                 if (suc.second) {
01659                     stack.push(c->r);
01660                     stack.push(c->l);
01661 
01662                     // ignore OPER_COMMA
01663                     if (c->data[0] == OPER_MUL || c->data[0] == OPER_ADD)
01664                         numinstr++;
01665                 }
01666             }
01667         }
01668     }
01669 
01670     //MesPrint ("*** [%s] Stopping CSEE", thetime_str().c_str());
01671     M_free(root, "CSE tree");
01672 
01673     return numinstr;
01674 }
01675 
01676 /*
01677     #] count_operators_cse_topdown :
01678     #[ simulated_annealing :
01679 */
01680 vector<WORD> simulated_annealing() {
01681     float minT = AO.Optimize.saMinT.fval;
01682     float maxT = AO.Optimize.saMaxT.fval;
01683     float T = maxT;
01684     float coolrate = pow(minT / maxT, 1 / (float)AO.Optimize.saIter);
01685 
01686     GETIDENTITY;
01687 
01688     // create a valid state where FACTORSYMBOL/SEPARATESYMBOL remains first
01689     vector<WORD> state = occurrence_order(optimize_expr, false);
01690     int startindex = 0;
01691     if ((state.size() > 0 && state[0] == SEPARATESYMBOL) || (state.size() > 1 && state[1] ==
      FACTORSYMBOL)) startindex++;
01692     if (state.size() > 1 && state[1] == FACTORSYMBOL) startindex++;
01693 
01694     my_random_shuffle(BHEAD state.begin() + startindex, state.end()); // start from random scheme
01695 
01696     vector<WORD> tree = Horner_tree(optimize_expr, state);
01697     int curscore = count_operators_cse_topdown(tree);
01698 
01699     // clean poly_vars, that are allocated by Horner_tree
01700     AN.poly_num_vars = 0;
01701     M_free(AN.poly_vars,"poly_vars");
01702 
01703     std::vector<WORD> best = state; // best state
01704     int bestscore = curscore;
01705 
01706     if (startindex == (int)state.size() || (int)state.size() - startindex < 2) {
01707         return best;
01708     }
01709 
01710     for (int o = 0; o < AO.Optimize.saIter; o++) {
01711         int inda = iranf(BHEAD state.size() - startindex) + startindex;
01712         int indb = iranf(BHEAD state.size() - startindex) + startindex;
01713 
01714         if (inda == indb) {
01715             continue;
01716         }
01717 
01718         swap(state[inda], state[indb]); // swap works best for Horner
01719 
01720         vector<WORD> tree = Horner_tree(optimize_expr, state);
01721         int newscore = count_operators_cse_topdown(tree);
01722 
01723         // clean poly_vars, that are allocated by Horner_tree
01724         AN.poly_num_vars = 0;
```

```
01725         M_free(AN.poly_vars,"poly_vars");
01726
01727         if (newscore <= curscore || 2.0 * wranf(BHEAD0) / (float)(UWORD)(-1) < exp((curscore -
      newscore) / T)) {
01728             curscore = newscore;
01729
01730             if (curscore < bestscore) {
01731                 bestscore = curscore;
01732                 best = state;
01733             }
01734         } else {
01735             swap(state[inda], state[indb]);
01736         }
01737
01738 #ifdef DEBUG_SA
01739     MesPrint("Score at step %d: %d", o, curscore);
01740 #endif
01741         T *= coolrate;
01742     }
01743
01744 #ifdef DEBUG_SA
01745     MesPrint("Simulated annealing score: %d", bestscore);
01746 #endif
01747
01748     return best;
01749 }
01750
01751 /*
01752     #] simulated_annealing :
01753     #[ printpstree :
01754 */
01755
01756 /*
01757 // print MCTS tree with LaTeX/pstricks (for analysis)
01758 void printpstree_rec (tree_node x, string pre="") {
01759
01760     if (x.num_visits==1) {
01761         MesPrint("%s\\TR{%d}",pre.c_str(),x.var);
01762     }
01763     else {
01764         MesPrint("%s\\pstree%s{\\TR{%d}}{",pre.c_str(),
01765                         pre==""  "?"[nodesep=0, levelsep=40]":"",
01766                         x.var);
01767         for (int i=0; i<(int)x.childs.size(); i++)
01768             if (x.childs[i].num_visits>0)
01769                 printpstree_rec(x.childs[i], pre+"  ");
01770         MesPrint("%s}",pre.c_str());
01771     }
01772 }
01773
01774 void printpstree () {
01775     // draw tree with pstricks
01776     MesPrint ("\\documentclass{article}");
01777     MesPrint ("\\usepackage{pstricks,pst-node,pst-tree,graphicx}");
01778     MesPrint ("\\begin{document}");
01779     MesPrint ("\\scalebox{0.02}{");
01780     printpstree_rec(mcts_root,"  ");
01781     MesPrint ("}");
01782     MesPrint ("\\end{document}");
01783 }
01784 */
01785
01786 /*
01787     #] printpstree :
01788     #[ find_Horner_MCTS_expand_tree :
01789 */
01790
01822 /*
01823         #[ next_MCTS_scheme :
01824 */
01825
01826 // find a Horner scheme to be used for the next simulation
01827 inline static void next_MCTS_scheme (PHEAD vector<WORD> *porder, vector<WORD> *pscheme,
      vector<tree_node *> *ppath) {
01828
01829     vector<WORD> &order = *porder;
01830     vector<WORD> &schemev = *pscheme;
01831     vector<tree_node *> &path = *ppath;
01832     int depth = 0, nchild0;
01833     float slide_down_factor = 1.0;
01834
01835     order.clear();
01836     path.clear();
01837
01838     // MCTS step I: select
01839     tree_node *select = &mcts_root;
01840     path.push_back(select);
```

```
01841     nchild0 = select->childs.size();
01842     while (select->childs.size() > 0) {
01843         // add virtual loss
01844         select->num_visits++;
01845         select->sum_results+=mcts_expr_score;
01846 
01847 //----------------------------------------------------------------------
01848                 switch ( AO.Optimize.mctsdecaymode ) {
01849                     case 1:  // Based on http://arxiv.org/abs/arXiv:1312.0841
01850                         slide_down_factor = 1.0-(1.0*AT.optimtimes)/(1.0*AO.Optimize.mctsnumexpand);
01851                         break;
01852                     case 2:  // This gives a bit more cleanup time at the end.
01853                         if ( 2*AT.optimtimes < AO.Optimize.mctsnumexpand ) {
01854                             slide_down_factor = 1.0*(AO.Optimize.mctsnumexpand-2*AT.optimtimes);
01855                             slide_down_factor /= 1.0*AO.Optimize.mctsnumexpand;
01856                         }
01857                         else {
01858                             slide_down_factor = 0.0001;
01859                         }
01860                         break;
01861                     case 3:  // depth dependent factor combined with case 1
01862                         float dd = 1.0-(1.0*depth)/(1.0*nchild0);
01863                         slide_down_factor = 1.0-(1.0*AT.optimtimes)/(1.0*AO.Optimize.mctsnumexpand);
01864                         if ( dd <= 0.000001 ) slide_down_factor = 1.0;
01865                         else slide_down_factor /= dd;
01866                         if ( slide_down_factor > 1.0 ) slide_down_factor = 1.0;
01867                         break;
01868                 }
01869 //----------------------------------------------------------------------
01870 
01871 #ifdef DEBUG_MCTS
01872         MesPrint("select %d",select->var);
01873 #endif
01874 
01875         // find most-promising node
01876         double best=0;
01877         tree_node *next=NULL;
01878         for (vector<tree_node>::iterator p=select->childs.begin(); p<select->childs.end(); p++) {
01879             double score;
01880             if (p->num_visits >= 1) {
01881 
01882                 // there are results calculated, so select with the UCT formula
01883                 score = mcts_expr_score / (p->sum_results/p->num_visits) +
01884 //------------------------------------------------------------------------
01885                     slide_down_factor *
01886 //------------------------------------------------------------------------
01887                     2 * AO.Optimize.mctsconstant.fval * sqrt(2*log(select->num_visits) /
      p->num_visits);
01888 
01889 #ifdef DEBUG_MCTS
01890                 printf("%d: %.2lf [x=%.2lf n=%d fin=%i]\n",p->var,score,mcts_expr_score /
      (p->sum_results/p->num_visits),
01891                                  p->num_visits,p->finished?1:0);
01892                 fflush(stdout);
01893 #endif
01894             }
01895             else {
01896         // no results yet, so select this node by setting score=infinite
01897                 score = 1e100;
01898 
01899 #ifdef DEBUG_MCTS
01900                 printf("%d: inf\n",p->var); fflush(stdout);
01901 #endif
01902             }
01903 
01904             // update best candidate
01905             if (!p->finished && score>best) {
01906                 best=score;
01907                 next=&*p;
01908             }
01909         }
01910 
01911         // if no node is found, this node is finished
01912         if (next==NULL) {
01913             select->finished=true;
01914             break;
01915         }
01916 
01917         // traverse down the tree
01918         select = next;
01919         path.push_back(select);
01920         order.push_back(select->var);
01921         depth++;
01922     }
01923 
01924     // MCTS step II: expand
01925 
```

```
01926 #ifdef DEBUG_MCTS
01927     MesPrint("expand %d",select->var);
01928 #endif
01929 
01930     // variables used so far
01931     set<WORD> var_used;
01932 
01933     for (int i=0; i<(int)order.size(); i++)
01934         var_used.insert(ABS(order[i])-1);
01935 
01936     // if this a new node, create node and add children
01937     if (!select->finished && select->childs.size()==0) {
01938         tree_node new_node(select->var);
01939         int sign = (order.size() == 0) ? 1 : SGN(order.back());
01940         for (int i=0; i<(int)mcts_vars.size(); i++)
01941             if (!var_used.count(mcts_vars[i])) {
01942                 new_node.childs.push_back(tree_node(sign*(mcts_vars[i]+1)));
01943                 if (AO.Optimize.hornerdirection==O_FORWARDANDBACKWARD)
01944                     new_node.childs.push_back(tree_node(-sign*(mcts_vars[i]+1)));
01945             }
01946         my_random_shuffle(BHEAD new_node.childs.begin(), new_node.childs.end());
01947 
01948         // here locking is necessary, since operator=(tree_node) is a
01949         // non-atomic operation (using pointers makes this lock obsolete)
01950         LOCK(optimize_lock);
01951         *select = new_node;
01952         UNLOCK(optimize_lock);
01953     }
01954     // set finished if necessary
01955     if (select->childs.size()==0)
01956         select->finished = true;
01957 
01958     // add virtual loss of number of operators in original expression
01959     select->num_visits++;
01960     select->sum_results+=mcts_expr_score;
01961 
01962     // MCTS step III: simulation
01963 
01964     // create complete Horner scheme
01965     deque<WORD> scheme;
01966 
01967     for (int i=0; i<(int)mcts_vars.size(); i++)
01968         if (!var_used.count(mcts_vars[i]))
01969             scheme.push_back(mcts_vars[i]);
01970     my_random_shuffle(BHEAD scheme.begin(), scheme.end());
01971 
01972     for (int i=(int)order.size()-1; i>=0; i--) {
01973         if (order[i] > 0)
01974             scheme.push_front(order[i]-1);
01975         else
01976             scheme.push_back(-order[i]-1);
01977     }
01978 
01979     // add FACTORSYMBOL/SEPARATESYMBOL is necessary
01980     if (mcts_factorized)
01981         scheme.push_front(FACTORSYMBOL);
01982     if (mcts_separated)
01983         scheme.push_front(SEPARATESYMBOL);
01984 
01985     // Horner scheme as a vector
01986     schemev = vector<WORD>(scheme.begin(), scheme.end());
01987 
01988 }
01989 
01990 /*
01991         #] next_MCTS_scheme :
01992         #[ try_MCTS_scheme :
01993 */
01994 
01995 // count the number of operators in the given Horner scheme
01996 inline static void try_MCTS_scheme (PHEAD const vector<WORD> &scheme, int *pnum_oper) {
01997 
01998     // do Horner, CSE and count the number of operators
01999     vector<WORD> tree = Horner_tree(optimize_expr, scheme);
02000     //vector<WORD> instr = generate_instructions(tree, true);
02001     //int num_oper = count_operators(instr);
02002     //int num_oper2 = count_operators_cse(tree);
02003     //int num_oper2 = count_operators_cse_topdown(tree);
02004     //MesPrint("%d %d", num_oper, num_oper2);
02005 
02006     int num_oper = count_operators_cse_topdown(tree);
02007 
02008     // clean poly_vars, that is allocated by Horner_tree
02009     AN.poly_num_vars = 0;
02010     M_free(AN.poly_vars,"poly_vars");
02011 
02012     *pnum_oper = num_oper;
```

```
02013
02014 }
02015
02016 /*
02017         #] try_MCTS_scheme :
02018         #[ update_MCTS_scheme :
02019 */
02020
02021 // update the best score and the search tree
02022 inline static void update_MCTS_scheme (int num_oper, const vector<WORD> &scheme, vector<tree_node *>
      *ppath) {
02023
02024     vector<tree_node *> &path = *ppath;
02025
02026     // update the (global) list of best Horner scheme
02027     if ((int)mcts_best_schemes.size() < AO.Optimize.mctsnumkeep ||
02028             (--mcts_best_schemes.end())->first > num_oper) {
02029         // here locking is necessary, for otherwise best_schemes may
02030         // become corrupted; lock can be prevented if each thread keeps
02031         // track of it's own list and those lists are merged in the end,
02032         // but this seems not useful to implement
02033         LOCK(optimize_lock);
02034         mcts_best_schemes.insert(make_pair(num_oper,scheme));
02035         if ((int)mcts_best_schemes.size() > AO.Optimize.mctsnumkeep)
02036             mcts_best_schemes.erase(--mcts_best_schemes.end());
02037         UNLOCK(optimize_lock);
02038     }
02039
02040     // MCTS step IV: backpropagate
02041
02042     // add number of operator and subtract mcts_expr_score, which
02043     // behaves as a "virtual loss"
02044     for (vector<tree_node *>::iterator p=path.begin(); p<path.end(); p++)
02045         (*p)->sum_results += num_oper - mcts_expr_score;
02046
02047 }
02048
02049 /*
02050         #] update_MCTS_scheme :
02051 */
02052
02053 void find_Horner_MCTS_expand_tree () {
02054
02055 #ifdef DEBUG
02056     MesPrint ("*** [%s, w=%w] CALL: find_Horner_MCTS_expand_tree", thetime_str().c_str());
02057 #endif
02058
02059     GETIDENTITY;
02060
02061     // the order for the Horner scheme up to the selected node, with signs
02062     // indicating forward or backward.
02063     vector<WORD> order;
02064
02065     // complete Horner scheme
02066     vector<WORD> scheme;
02067
02068     // path to the selected node
02069     vector<tree_node *> path;
02070
02071     // the number of operations obtained by the simulation
02072     int num_oper;
02073
02074     next_MCTS_scheme(BHEAD &order, &scheme, &path);
02075     try_MCTS_scheme(BHEAD scheme, &num_oper);
02076 #ifdef DEBUG_MCTS
02077     // Actually "order" is needed only for this debug output.
02078     MesPrint ("{%a} -> {%a} -> %d", order.size(), &order[0], scheme.size(), &scheme[0], num_oper);
02079 #endif
02080     update_MCTS_scheme(num_oper, scheme, &path);
02081
02082 #ifdef DEBUG
02083     MesPrint ("*** [%s, w=%w] DONE: find_Horner_MCTS_expand_tree(%a-> %d)",
02084                         thetime_str().c_str(), scheme.size(), &scheme[0], num_oper);
02085 #endif
02086
02087 }
02088
02089 /*
02090     #] find_Horner_MCTS_expand_tree :
02091     #[ PF_find_Horner_MCTS_expand_tree :
02092 */
02093 #ifdef WITHMPI
02094
02095 // To remember which task is assigned to each slave. A task is represented by
02096 // a pair of a Horner scheme and the selected path for the scheme.
02097 // The index range is from 1 to PF.numtasks-1.
02098 vector<pair<vector<WORD>, vector<tree_node *> > > PF_opt_MCTS_tasks;
```

```
02099
02100 // Initialization.
02101 void PF_find_Horner_MCTS_expand_tree_master_init () {
02102
02103     PF_opt_MCTS_tasks.resize(PF.numtasks);
02104     for (int i = 1; i < PF.numtasks; i++) {
02105         pair<vector<WORD>, vector<tree_node *> > &p = PF_opt_MCTS_tasks[i];
02106         p.first.clear();
02107         p.second.clear();
02108     }
02109
02110 }
02111
02112 // Wait for an idle slave and return the process number.
02113 int PF_find_Horner_MCTS_expand_tree_master_next () {
02114
02115     // Find an idle slave.
02116     int next;
02117     PF_Receive(PF_ANY_SOURCE, PF_OPT_MCTS_MSGTAG, &next, NULL);
02118
02119     // Check if the slave had a task.
02120     pair<vector<WORD>, vector<tree_node *> > &p = PF_opt_MCTS_tasks[next];
02121     if (!p.first.empty()) {
02122         // If so, update the result.
02123         int num_oper;
02124         PF_Unpack(&num_oper, 1, PF_INT);
02125         update_MCTS_scheme(num_oper, p.first, &p.second);
02126
02127         // Clear the task.
02128         p.first.clear();
02129         p.second.clear();
02130     }
02131
02132     return next;
02133
02134 }
02135
02136 // The main function on the master.
02137 void PF_find_Horner_MCTS_expand_tree_master () {
02138
02139 #ifdef DEBUG
02140     MesPrint ("*** [%s, w=%w] CALL: PF_find_Horner_MCTS_expand_tree_master", thetime_str().c_str());
02141 #endif
02142
02143     vector<WORD> order;
02144     vector<WORD> scheme;
02145     vector<tree_node *> path;
02146
02147     next_MCTS_scheme(BHEAD &order, &scheme, &path);
02148
02149     // Find an idle slave.
02150     int next = PF_find_Horner_MCTS_expand_tree_master_next();
02151
02152     // Send a new task to the slave.
02153     PF_PrepareLongSinglePack();
02154     int len = scheme.size();
02155     PF_LongSinglePack(&len, 1, PF_INT);
02156     PF_LongSinglePack(&scheme[0], len, PF_WORD);
02157     PF_LongSingleSend(next, PF_OPT_MCTS_MSGTAG);
02158
02159     // Remember the task.
02160     pair<vector<WORD>, vector<tree_node *> > &p = PF_opt_MCTS_tasks[next];
02161     p.first = scheme;
02162     p.second = path;
02163
02164 #ifdef DEBUG
02165     MesPrint ("*** [%s, w=%w] DONE: PF_find_Horner_MCTS_expand_tree_master", thetime_str().c_str());
02166 #endif
02167
02168 }
02169
02170 // Wait for all the slaves to finish their tasks.
02171 void PF_find_Horner_MCTS_expand_tree_master_wait () {
02172
02173 #ifdef DEBUG
02174     MesPrint ("*** [%s, w=%w] CALL: PF_find_Horner_MCTS_expand_tree_master_wait",
      thetime_str().c_str());
02175 #endif
02176
02177     // Wait for all the slaves.
02178     for (int i = 1; i < PF.numtasks; i++) {
02179         int next = PF_find_Horner_MCTS_expand_tree_master_next();
02180         // Send a null task.
02181         PF_PrepareLongSinglePack();
02182         int len = 0;
02183         PF_LongSinglePack(&len, 1, PF_INT);
02184         PF_LongSingleSend(next, PF_OPT_MCTS_MSGTAG);
```

```
02185     }
02186
02187 #ifdef DEBUG
02188     MesPrint ("*** [%s, w=%w] DONE: PF_find_Horner_MCTS_expand_tree_master_wait",
      thetime_str().c_str());
02189 #endif
02190
02191 }
02192
02193 // The main function on the slaves.
02194 void PF_find_Horner_MCTS_expand_tree_slave () {
02195
02196 #ifdef DEBUG
02197     MesPrint ("*** [%s, w=%w] CALL: PF_find_Horner_MCTS_expand_tree_slave", thetime_str().c_str());
02198 #endif
02199
02200     vector<WORD> scheme;
02201
02202     {
02203         // Send the first message to the master, which indicates I am idle.
02204         PF_PreparePack();
02205         int dummy = 0;
02206         PF_Pack(&dummy, 1, PF_INT);
02207         PF_Send(MASTER, PF_OPT_MCTS_MSGTAG);
02208     }
02209
02210     for (;;) {
02211         // Get a task from the master.
02212         PF_LongSingleReceive(MASTER, PF_OPT_MCTS_MSGTAG, NULL, NULL);
02213
02214         // Length of the task.
02215         int len;
02216         PF_LongSingleUnpack(&len, 1, PF_INT);
02217
02218         // No task remains.
02219         if (len == 0) break;
02220
02221         // Perform the given task.
02222         scheme.resize(len);
02223         PF_LongSingleUnpack(&scheme[0], len, PF_WORD);
02224         int num_oper;
02225         try_MCTS_scheme(scheme, &num_oper);
02226
02227         // Send the result to the master.
02228         PF_PreparePack();
02229         PF_Pack(&num_oper, 1, PF_INT);
02230         PF_Send(MASTER, PF_OPT_MCTS_MSGTAG);
02231     }
02232
02233 #ifdef DEBUG
02234     MesPrint ("*** [%s, w=%w] DONE: PF_find_Horner_MCTS_expand_tree_slave", thetime_str().c_str());
02235 #endif
02236
02237 }
02238
02239 #endif
02240 /*
02241     #] PF_find_Horner_MCTS_expand_tree :
02242     #[ find_Horner_MCTS :
02243 */
02244
02253 //vector<vector<WORD> > find_Horner_MCTS () {
02254 void find_Horner_MCTS () {
02255
02256 #ifdef WITHMPI
02257     if (PF.me != MASTER) {
02258         if (PF.numtasks <= 1) return;
02259         PF_find_Horner_MCTS_expand_tree_slave();
02260         return;
02261     }
02262 #endif
02263
02264 #ifdef DEBUG
02265     MesPrint ("*** [%s, w=%w] CALL: find_Horner_MCTS", thetime_str().c_str());
02266 #endif
02267
02268     GETIDENTITY;
02269
02270     LONG start_time = TimeWallClock(1);
02271
02272     // initialize the used global variables
02273     mcts_expr_score = count_operators(optimize_expr);
02274     mcts_root = tree_node();
02275
02276     // extract all symbols from the expression
02277     set<WORD> var_set;
02278     for (WORD *t=optimize_expr; *t!=0; t+=*t)
```

```
02279         if (t[1] == SYMBOL)
02280             for (int i=3; i<t[2]; i+=2)
02281                 var_set.insert(t[i]);
02282 
02283     // check for factorized/separated expression and make sure that
02284     // FACTORSYMBOL/SEPARATESYMBOL isn't included in the MCTS
02285     mcts_factorized = var_set.count(FACTORSYMBOL);
02286     if (mcts_factorized) var_set.erase(FACTORSYMBOL);
02287     mcts_separated = var_set.count(SEPARATESYMBOL);
02288     if (mcts_separated) var_set.erase(SEPARATESYMBOL);
02289 
02290     mcts_vars = vector<WORD>(var_set.begin(), var_set.end());
02291     optimize_num_vars = (int)mcts_vars.size();
02292     // initialize MCTS tree root
02293     for (int i=0; i<(int)mcts_vars.size(); i++) {
02294         if (AO.Optimize.hornerdirection != O_BACKWARD)
02295             mcts_root.childs.push_back(tree_node(+(mcts_vars[i]+1)));
02296         if (AO.Optimize.hornerdirection != O_FORWARD)
02297             mcts_root.childs.push_back(tree_node(-(mcts_vars[i]+1)));
02298     }
02299     my_random_shuffle(BHEAD mcts_root.childs.begin(), mcts_root.childs.end());
02300 
02301 #if defined(WITHMPI)
02302     PF_find_Horner_MCTS_expand_tree_master_init();
02303 #endif
02304 
02305     // initialize a potential variable mctsconstant scheme.
02306     AT.optimtimes = 0;
02307 
02308     // call expand_tree until it is called "mctsnumexpand" times, the
02309     // time limit is reached or the tree is fully finished
02310     for (int times=0; times<AO.Optimize.mctsnumexpand && !mcts_root.finished &&
02311                   (AO.Optimize.mctstimelimit==0 ||    (TimeWallClock(1)-start_time)/100 <
    AO.Optimize.mctstimelimit);
02312              times++) {
02313         AT.optimtimes = times;
02314     // call expand_tree routine depending on threading mode
02315 #if defined(WITHPTHREADS)
02316         if (AM.totalnumberofthreads > 1)
02317             find_Horner_MCTS_expand_tree_threaded();
02318         else
02319 #elif defined(WITHMPI)
02320         if (PF.numtasks > 1)
02321             PF_find_Horner_MCTS_expand_tree_master();
02322         else
02323 #endif
02324             find_Horner_MCTS_expand_tree();
02325     }
02326 
02327     // if TForm or ParForm, wait for everyone to finish
02328 #ifdef WITHPTHREADS
02329     MasterWaitAll();
02330 #endif
02331 #ifdef WITHMPI
02332     PF_find_Horner_MCTS_expand_tree_master_wait();
02333 #endif
02334 #ifdef DEBUG
02335     MesPrint ("*** [%s, w=%w] DONE: find_Horner_MCTS", thetime_str().c_str());
02336 #endif
02337 
02338 }
02339 
02340 /*
02341     #] find_Horner_MCTS :
02342     #[ merge_operators :
02343 */
02344 
02402 vector<WORD> merge_operators (const vector<WORD> &all_instr, bool move_coeff) {
02403 
02404 #ifdef DEBUG_MORE
02405     MesPrint ("*** [%s, w=%w] CALL: merge_operators", thetime_str().c_str());
02406 #endif
02407 
02408     // get starting positions of instructions
02409     vector<const WORD *> instr;
02410     const WORD *tbegin = &*all_instr.begin();
02411     const WORD *tend = tbegin+all_instr.size();
02412     // copy all instructions to temp space. There will be n of them in instr.
02413     for (const WORD *t=tbegin; t!=tend; t+=*(t+2)) {
02414         instr.push_back(t);
02415     }
02416     int n = instr.size();
02417 
02418     // find parents and number of parents of instructions
02419     vector<int> par(n), numpar(n,0);
02420     for (int i=0; i<n; i++) par[i]=i;
02421 
```

```
02422     for (int i=0; i<n; i++) {
02423         for (const WORD *t=instr[i]+OPTHEAD; *t!=0; t+=*t) {
02424             if ( *(t+1)==EXTRASYMBOL && *t!=1+ABS(*(t+*t-1)) ) {
02425                 // extra symbol t[3] is referred to in instr i.
02426                 par[*(t+3)]=i;
02427                 numpar[*(t+3)]++;
02428 
02429                 // if coefficient isn't +/-1 or power > 1, increase numpar,
02430                 // so that this is not merged
02431                 if (*(t+*t-3)!=1 || *(t+*t-2)!=1 || ABS(*(t+*t-1))!=3) numpar[*(t+3)]++;
02432                 if (*(t+4)>1) numpar[*(t+3)]++;
02433             }
02434         }
02435     }
02436     // determine which instructions to merge
02437     stack<int> s;
02438     s.push(n-1);
02439     vector<bool> vis(n,false);
02440 
02441     while (!s.empty()) {
02442 
02443         int i=s.top(); s.pop();
02444         if (vis[i]) continue;
02445         vis[i]=true;
02446 
02447         for (const WORD *t=instr[i]+OPTHEAD; *t!=0; t+=*t)
02448             if ( *(t+1)==EXTRASYMBOL && *t!=1+ABS(*(t+*t-1)) )
02449                 s.push(*(t+3));
02450 
02451         // condition: one parent and equal operator as parent
02452         if (numpar[i]==1 && *(instr[i]+1)==*(instr[par[i]]+1))
02453             par[i] = par[par[i]]; // The expr into which we subst par[i] to get i
02454         else
02455             par[i] = i;
02456     }
02457 
02458     // merge instructions into new instructions
02459     vector<WORD> newinstr;
02460 
02461     // stack of new expressions, all 0-indexed
02462     stack<WORD> new_expr;
02463     new_expr.push(n-1);
02464     vis = vector<bool>(n,false);
02465 
02466     // skip empty equations (might be introduced by greedy optimizations)
02467     vector<bool> skip(n,false), skipcoeff(n,false);
02468     for (int i=0; i<n; i++)
02469         if (*(instr[i]+OPTHEAD) == 0) skip[i]=skipcoeff[i]=true;
02470 
02471     // for renumbering merged parents
02472     vector<int> renum_par(n);
02473     for (int i=0; i<n; i++)
02474         renum_par[i]=i;
02475 
02476     while (!new_expr.empty()) {
02477         int x = new_expr.top(); new_expr.pop();
02478         if (vis[x]) continue;
02479         vis[x] = true;
02480 
02481         // find all instructions with parent=x and copy their arguments
02482         // into a new expression
02483         bool first_copy=true;
02484         int lenidx = newinstr.size()+2;
02485 
02486         // 1-indexed, since signs may occur
02487         stack<WORD> this_expr;
02488         this_expr.push(x+1);
02489 
02490         while (!this_expr.empty()) {
02491             // pop from stack, determine expr.nr and sign
02492             int i = this_expr.top(); this_expr.pop();
02493             int sign = SGN(i);
02494             i = ABS(i)-1;
02495             for (const WORD *t=instr[i]+OPTHEAD; *t!=0; t+=*t) { // terms in i
02496                 // don't copy a term if:
02497                 // (1) skip=true, since then it's merged into the parent
02498                 // (2) extrasymbol with parent=x, because its children should be copied
02499                 // (3) coefficient with skipcoeff=true, since it's already copied
02500                 bool copy = !skip[i];
02501                 if (*t!=1+ABS(*(t+*t-1)) && *(t+1)==EXTRASYMBOL) {
02502                     if (par[*(t+3)] == x) {
02503                         // parent of term refers to x. we push it with its sign if no skip is true
02504                         // and the sign of the expr.
02505                         this_expr.push(sign * (skip[i]||skipcoeff[i] ? 1 : SGN(*(t+*t-1))) *
      (1+*(t+3)));
02506                         if (*(instr[i]+1) == OPER_MUL) sign=1;
02507                         copy=false;
```

```
02508                     }
02509                     else {
02510                         new_expr.push(*(t+3));
02511                     }
02512                 }
02513
02514                 if (*t == 1+ABS(*(t+*t-1)) && skipcoeff[i]) {
02515                     copy=false;
02516                 }
02517
02518                 if (copy) {
02519                     // first term, so add header
02520                     if (first_copy) {
02521                         newinstr.push_back(renum_par[x]);  // expr.nr.
02522                         newinstr.push_back(*(instr[x]+1)); // operator
02523                         newinstr.push_back(3);             // length   OPTHEAD?
02524                         first_copy=false;
02525                     }
02526
02527                     // copy term and adjust sign
02528                     int thislenidx = newinstr.size();
02529                     newinstr.insert(newinstr.end(), t, t+*t); // Put the whole term in newinstr
02530                     newinstr.back() *= sign;
02531                     if (*(instr[i]+1) == OPER_MUL) sign=1;
02532                     newinstr[lenidx] += *t;
02533
02534                     // check for moving coefficients up
02535                     // necessary condition: MUL-expression with 1 parent
02536                     if (move_coeff && *t!=1+ABS(*(t+*t-1)) && *(instr[i]+1)!=OPER_COMMA &&
02537                             *(t+1)==EXTRASYMBOL && numpar[*(t+3)]==1 && *(instr[*(t+3)]+1)==OPER_MUL)
    {
02538
02539                         // coefficient is always the first term (that's how Horner+generate works)
02540                         const WORD *t1 = instr[*(t+3)]+OPTHEAD;
02541                         const WORD *t2 = t1+*t1;
02542
02543                         if (*t1 == 1+ABS(*(t1+*t1-1))) {
02544                             // t1 pointer to a coefficient, so move it
02545
02546                             // remove old coefficient of 1
02547                             WORD *t3 = &*newinstr.end();                    //
02548                             int sign2 = SGN(t3[-1]);                         //
02549                             newinstr.erase(newinstr.end()-3, newinstr.end());
02550                             // count number of arguments; iff it is 2 move the (extra)symbol too
02551                             int numargs=0;
02552                             for (const WORD *tt=t1; *tt!=0; tt+=*tt) {
02553                                 numargs++;
02554                             }
02555                             if (numargs==2 && *(t2+4)==1) {
02556                                 // replace (extra)symbol
02557                                 newinstr[newinstr.size()-4] = *(t2+1);
02558                                 newinstr[newinstr.size()-3] = *(t2+2);
02559                                 newinstr[newinstr.size()-2] = *(t2+3);
02560                                 newinstr[newinstr.size()-1] = *(t2+4);
02561                                 sign2 *= SGN(*(t2+*t2-1));          // was t2[7]
02562
02563                                 // ignore this expression from now on
02564                                 skip[*(t+3)]=true;
02565                                 if (*(t2+1)==EXTRASYMBOL)
02566                                     renum_par[*(t+3)] = *(t2+3);
02567                             }
02568                             else {
02569                                 // otherwise, ignore coefficient from now on
02570                                 // we need to collect the signs of the terms
02571                                 // first and set them to one. This was forgotten
02572                                 // before. Gave occasional errors.
02573                                 if ( numargs > 2 || ( numargs == 2 && t2[4] > 1 ) ) {
02574                                     for (WORD *tt=(WORD *)t2; *tt!=0; tt+=*tt) {
02575                                         if ( tt[*tt-1] < 0 ) {
02576                                             tt[*tt-1] = -tt[*tt-1];
02577                                             sign2 = -sign2;
02578                                         }
02579                                     }
02580                                 }
02581                                 skipcoeff[*(t+3)]=true;
02582                             }
02583
02584                             // add new coefficient
02585                             newinstr.insert(newinstr.end(), t1+1, t1+*t1);
02586                             newinstr.back() *= sign2;
02587                             newinstr[thislenidx] += ABS(newinstr.back()) - 3;
02588                             newinstr[lenidx] += ABS(newinstr.back()) - 3;
02589                         }
02590                     }
02591                 }
02592             }
02593         }
```

```
02594
02595         // if something has been copied, add trailing zero
02596         if (!first_copy) {
02597             newinstr.push_back(0);
02598             newinstr[lenidx]++;
02599         }
02600     }
02601
02602     // renumber the expressions to 0,1,2,..,; only keep expressions with
02603     // skip=false which are their own parent after a renumbering in case
02604     // of moved coefficients
02605
02606     // find renumber scheme
02607     vector<int> renum(n,-1);
02608     int next=0;
02609     for (int i=0; i<n; i++)
02610         if (!skip[i] && renum_par[par[i]]==i) renum[renum_par[i]]=next++;
02611
02612     // find new instruction index
02613     tbegin = &*newinstr.begin();
02614     tend = tbegin+newinstr.size();
02615     for (const WORD *t=tbegin; t!=tend; t+=*(t+2))
02616         instr[renum[*t]] = t;
02617
02618     // renumbering expressions and sort them lexicographically (in
02619     // new_instr they are in the preorder of the expression tree)
02620     vector<WORD> sortinstr;
02621     for (int i=0; i<next; i++) {
02622         int idx = sortinstr.size();
02623         sortinstr.insert(sortinstr.end(), instr[i], instr[i]+*(instr[i]+2));
02624
02625         sortinstr[idx] = renum[sortinstr[idx]];
02626
02627         // renumber content of an expression
02628         for (WORD *t2=&sortinstr[idx]+3; *t2!=0; t2+=*t2)
02629             if (*t2!=1+ABS(*(t2+*t2-1)) && *(t2+1)==EXTRASYMBOL)
02630                 *(t2+3) = renum[*(t2+3)];
02631     }
02632
02633 #ifdef DEBUG_MORE
02634     MesPrint ("*** [%s, w=%w] DONE: merge_operators", thetime_str().c_str());
02635 #endif
02636
02637     return sortinstr;
02638 }
02639
02640 /*
02641     #] merge_operators :
02642     #[ class Optimization :
02643 */
02644
02669 class optimization {
02670 public:
02671     int type, arg1, arg2, improve;
02672     vector<WORD> coeff;
02673     vector<int> eqnidxs;
02674
02675     bool operator< (const optimization &a) const {
02676         if (arg1 != a.arg1) return arg1 < a.arg1;
02677         if (arg2 != a.arg2) return arg2 < a.arg2;
02678         if (type != a.type) return type < a.type;
02679         return coeff < a.coeff;
02680     }
02681 };
02682
02683 /*
02684     #] class Optimization :
02685     #[ find_optimizations :
02686 */
02687
02697 vector<optimization> find_optimizations (const vector<WORD> &instr) {
02698
02699 #ifdef DEBUG_GREEDY
02700     MesPrint ("*** [%s, w=%w] CALL: find_optimizations", thetime_str().c_str());
02701 #endif
02702
02703 //  #[ Startup :
02704     // the resulting vector of optimizations
02705     vector<optimization> res;
02706
02707     // a map to count the improvement of an optimization; the
02708     // improvement is stored as a vector<int> with equation numbers
02709     map<optimization, vector<int> > cnt;
02710
02711     // a map to identify coefficients
02712     map<vector<int>,int> idx_coeff;
02713
```

```
02714     const WORD *ebegin = &*instr.begin();
02715     const WORD *eend = ebegin+instr.size();
02716
02717     for (int optim_type=0; optim_type<=4; optim_type++) {
02718
02719         cnt.clear();
02720         idx_coeff.clear();
02721
02722       optimization optim;
02723         optim.type = optim_type;
02724 //  #] Startup :
02725
02726 //  #[ type 0 :   find optimizations of the form z=x^n (optim.type==0)
02727         if (optim_type == 0) {
02728             for (const WORD *e=ebegin; e!=eend; e+=*(e+2)) {
02729                 if (*(e+1) != OPER_MUL) continue;
02730                 for (const WORD *t=e+3; *t!=0; t+=*t) {
02731                     if (*t == ABS(*(t+*t-1))+1) continue;
02732                     if (*(t+4) > 1) {
02733                         optim.arg1 = (*(t+1)==SYMBOL ? 1 : -1) * (*(t+3) + 1);
02734                         optim.arg2 = *(t+4);
02735                         cnt[optim].push_back(e-ebegin);
02736                     }
02737                 }
02738             }
02739         }
02740 //  #] type 0 :
02741 //  #[ type 1 :  find optimizations of the form z=x*y (optim.type==1)
02742         if (optim_type == 1) {
02743             for (const WORD *e=ebegin; e!=eend; e+=*(e+2)) {
02744                 if (*(e+1) != OPER_MUL) continue;
02745
02746                 for (const WORD *t1=e+3; *t1!=0; t1+=*t1) {
02747                     if (*t1 == ABS(*(t1+*t1-1))+1) continue;
02748                     int x1 = (*(t1+1)==SYMBOL ? 1 : -1) * (*(t1+3) + 1);
02749
02750                     for (const WORD *t2=t1+*t1; *t2!=0; t2+=*t2) {
02751                         if (*t2 == ABS(*(t2+*t2-1))+1) continue;
02752                         int x2 = (*(t2+1)==SYMBOL ? 1 : -1) * (*(t2+3) + 1);
02753
02754                         if (*(t1+4) == *(t2+4)) {
02755                             optim.arg1 = x1;
02756                             optim.arg2 = x2;
02757                             if (optim.arg1 > optim.arg2)
02758                                 swap(optim.arg1, optim.arg2);
02759
02760                             if (*(t1+4) == 1)
02761                                 cnt[optim].push_back(e-ebegin);
02762                             else {
02763                                 // E=x^n*y^n -> z=x*y; E=z^n is double improvement
02764                                 cnt[optim].push_back(e-ebegin);
02765                                 cnt[optim].push_back(e-ebegin);
02766                             }
02767                         }
02768                     }
02769                 }
02770             }
02771         }
02772 //  #] type 1 :
02773 //  #[ type 2 :  find optimizations of the form z=c*x (optim.type==2)
02774         if (optim_type == 2) {
02775             for (const WORD *e=ebegin; e!=eend; e+=*(e+2)) {
02776
02777                 if (*(e+1) == OPER_ADD) {
02778                     // in ADD-equation
02779
02780                     for (const WORD *t=e+3; *t!=0; t+=*t) {
02781                         if (*t == ABS(*(t+*t-1))+1) continue;
02782
02783                         if (*(t+4)==1) {
02784                             if (ABS(*(t+*t-1))==3 && *(t+*t-2)==1 && *(t+*t-3)==1) continue;
02785
02786                             optim.coeff = vector<WORD>(t+*t-ABS(*(t+*t-1)), t+*t);
02787                             optim.coeff.back() = ABS(optim.coeff.back());
02788
02789                             optim.arg1 = (*(t+1)==SYMBOL ? 1 : -1) * (*(t+3) + 1);
02790                             optim.arg2 = 0;
02791
02792                             cnt[optim].push_back(e-ebegin);
02793                         }
02794                     }
02795                 }
02796                 else if (*(e+1) == OPER_MUL) {
02797                     // in MUL-equation
02798                     optim.coeff.clear();
02799
02800                     for (const WORD *t=e+3; *t!=0; t+=*t)
```

```
02801                           if (*t == ABS(*(t+*t-1))+1) {
02802                               if (ABS(*(t+*t-1))==3 && *(t+*t-2)==1 && *(t+*t-3)==1) continue;
02803                               optim.coeff = vector<WORD>(t+*t-ABS(*(t+*t-1)), t+*t);
02804                               optim.coeff.back() = ABS(optim.coeff.back());
02805                           }
02806
02807                       if (!optim.coeff.empty())
02808                           for (const WORD *t=e+3; *t!=0; t+=*t) {
02809                               if (*t == ABS(*(t+*t-1))+1) continue;
02810                               if (*(t+4) != 1) continue;
02811                               optim.arg1 = (*(t+1)==SYMBOL ? 1 : -1) * (*(t+3) + 1);
02812                               optim.arg2 = 0;
02813                               cnt[optim].push_back(e-ebegin);
02814                           }
02815                   }
02816               }
02817           }
02818 //  #] type 2 :
02819 //  #[ type 3 :  find optimizations of the form z=x+c (optim.type==3)
02820           if (optim_type == 3) {
02821               for (const WORD *e=ebegin; e!=eend; e+=*(e+2)) {
02822
02823                   if (*(e+1) != OPER_ADD) continue;
02824
02825                   optim.coeff.clear();
02826
02827                   for (const WORD *t=e+3; *t!=0; t+=*t)
02828                       if (*t == ABS(*(t+*t-1))+1) {
02829                           optim.coeff = vector<WORD>(t+*t-ABS(*(t+*t-1)), t+*t);
02830                       }
02831
02832                   if (!optim.coeff.empty()) {
02833                       for (const WORD *t=e+3; *t!=0; t+=*t) {
02834                           if (*t == ABS(*(t+*t-1))+1) continue;
02835                           if (ABS(*(t+*t-1))!=3 || *(t+*t-2)!=1 || *(t+*t-3)!=1) continue;
02836                           if (*(t+*t-1)==-3) optim.coeff.back() *= -1;
02837
02838                           optim.arg1 = (*(t+1)==SYMBOL ? 1 : -1) * (*(t+3) + 1);
02839                           optim.arg2 = 0;
02840                           cnt[optim].push_back(e-ebegin);
02841                           if (*(t+*t-1)==-3) optim.coeff.back() *= -1;
02842                       }
02843                   }
02844               }
02845           }
02846 //  #] type 3 :
02847 //  #[ type 4,5 : find optimizations of the form z=x+y or z=x-y (optim.type==4 or 5)
02848           if (optim_type == 4) {
02849               for (const WORD *e=ebegin; e!=eend; e+=*(e+2)) {
02850                   if (*(e+1) != OPER_ADD) continue;
02851
02852                   for (const WORD *t1=e+3; *t1!=0; t1+=*t1) {
02853                       if (*t1 == ABS(*(t1+*t1-1))+1) continue;
02854                       int x1 = (*(t1+1)==SYMBOL ? 1 : -1) * (*(t1+3) + 1);
02855
02856                       for (const WORD *t2=t1+*t1; *t2!=0; t2+=*t2) {
02857                           if (*t2 == ABS(*(t2+*t2-1))+1) continue;
02858                           int x2 = (*(t2+1)==SYMBOL ? 1 : -1) * (*(t2+3) + 1);
02859
02860                           int sign1 = SGN(*(t1+*t1-1));
02861                           int sign2 = SGN(*(t2+*t2-1));
02862
02863                           if (BigLong((UWORD *)t1+5, ABS(*(t1+*t1-1))-1, (UWORD *)t2+5,
      ABS(*(t2+*t2-1))-1) == 0) {
02864
02865                               optim.type = (sign1 * sign2 == 1 ? 4 : 5); // optimization type
02866                               optim.arg1 = x1;
02867                               optim.arg2 = x2;
02868                               if (optim.arg1 > optim.arg2) {
02869                                   swap(optim.arg1, optim.arg2);
02870                               }
02871
02872                               if (ABS(*(t1+*t1-1))==3 && *(t1+*t1-2)==1 && *(t1+*t1-3)==1)
02873                                   cnt[optim].push_back(e-ebegin);
02874                               else {
02875                                   // E=2x+2y -> z=x+y; E=2z is improvement bby itself
02876                                   cnt[optim].push_back(e-ebegin);
02877                                   cnt[optim].push_back(e-ebegin);
02878                               }
02879                           }
02880                       }
02881                   }
02882               }
02883           }
02884 //  #] type 4,5 :
02885 //  #[ add :
02886
```

```
02887         // add optimizations with positive improvement to the result
02888         for (map<optimization, vector<int> >::iterator i=cnt.begin(); i!=cnt.end(); i++) {
02889             int improve = i->second.size() - 1;
02890             if (improve > 0) {
02891                 res.push_back(i->first);
02892                 res.back().improve = improve;
02893                 res.back().eqnidxs = i->second;
02894 
02895                 // remove duplicates, that were add to get the correct improvement
02896                 res.back().eqnidxs.erase(unique(res.back().eqnidxs.begin(), res.back().eqnidxs.end()),
      res.back().eqnidxs.end());
02897             }
02898         }
02899     }
02900 //  #] add :
02901 
02902 #ifdef DEBUG_GREEDY
02903     MesPrint ("*** [%s, w=%w] DONE: find_optimizations",thetime_str().c_str());
02904 #endif
02905 
02906     return res;
02907 }
02908 
02909 /*
02910     #] find_optimizations :
02911     #[ do_optimization :
02912 */
02913 
02930 bool do_optimization (const optimization optim, vector<WORD> &instr, int newid) {
02931 
02932 //  #[ Debug code :
02933 #ifdef DEBUG_GREEDY
02934     if (optim.type==0)
02935         MesPrint ("*** [%s, w=%w] CALL: do_optimization(improve=%d, %c%d^%d)",
02936           thetime_str().c_str(), optim.improve,
02937         optim.arg1>0?'x':'Z', ABS(optim.arg1)-1, optim.arg2);
02938     else if (optim.type==1 || optim.type>=4)
02939         MesPrint ("*** [%s, w=%w] CALL: do_optimization(improve=%d, %c%d%c%c%d)",
02940         thetime_str().c_str(), optim.improve,
02941         optim.arg1>0?'x':'Z', ABS(optim.arg1)-1,
02942         optim.type==1 ? '*' : optim.type==4 ? '+' : '-',
02943         optim.arg2>0?'x':'Z', ABS(optim.arg2)-1);
02944     else {
02945     WORD n = optim.coeff.back()/2;
02946     UBYTE num[BITSINWORD*ABS(n)], den[BITSINWORD*ABS(n)];
02947     PrtLong((UWORD *)&optim.coeff[0], n, num);
02948     PrtLong((UWORD *)&optim.coeff[ABS(n)], ABS(n), den);
02949         MesPrint ("*** [%s, w=%w] CALL: do_optimization(improve=%d, %c%d%c%s/%s)",
02950         thetime_str().c_str(), optim.improve,
02951         optim.arg1>0?'x':'Z', ABS(optim.arg1)-1,
02952         optim.type==2 ? '*' : '+', num,den);
02953   }
02954 #endif
02955 //  #] Debug code :
02956 
02957     bool substituted = false;
02958     WORD *ebegin = &*instr.begin();
02959 
02960 //  #[ type 0 : substitution of the form z=x^n (optim.type==0)
02961     if (optim.type == 0) {
02962 
02963         int vartypex = optim.arg1>0 ? SYMBOL : EXTRASYMBOL;
02964         int varnumx  = ABS(optim.arg1) - 1;
02965         int n = optim.arg2;
02966 
02967         for (int i=0; i<(int)optim.eqnidxs.size(); i++) {
02968             WORD *e = ebegin + optim.eqnidxs[i];
02969             if (*(e+1) != OPER_MUL) continue;
02970 
02971             // scan through equation
02972             for (WORD *t=e+3; *t!=0; t+=*t) {
02973                 if (*t == ABS(*(t+*t-1))+1) continue;
02974                 if (*(t+1)==vartypex &&
02975                         *(t+3)==varnumx &&
02976                         *(t+4) % n == 0) {
02977 
02978                     // substitute
02979                     *(t+1) = EXTRASYMBOL;
02980                     *(t+3) = newid;
02981                     *(t+4) /= n;
02982 
02983                     substituted = true;
02984                 }
02985             }
02986         }
02987 
02988         if (!substituted) {
```

```
02989 #ifdef DEBUG_GREEDY
02990             MesPrint ("*** [%s, w=%w] DONE: do_optimization : res=false", thetime_str().c_str(),
      optim.improve);
02991 #endif
02992             return false;
02993         }
02994 
02995         // add extra equation (Tnew = x^n)
02996         instr.push_back(newid);     // eqn.nr
02997         instr.push_back(OPER_MUL); // operator
02998         instr.push_back(12);        // total length
02999         instr.push_back(8);         // term length
03000         instr.push_back(vartypex); // (extra)symbol
03001         instr.push_back(4);         // symbol length
03002         instr.push_back(varnumx);   // symbol id
03003         instr.push_back(n);         // power
03004         instr.push_back(1);
03005         instr.push_back(1);         // coefficient 1
03006         instr.push_back(3);
03007         instr.push_back(0);         // trailing 0
03008     }
03009 //  #] type 0 :
03010 //  #[ type 1 : substitution of the form z=x*y (optim.type==1)
03011     if (optim.type == 1) {
03012 
03013         int vartypex = optim.arg1>0 ? SYMBOL : EXTRASYMBOL;
03014         int varnumx  = ABS(optim.arg1) - 1;
03015         int vartypey = optim.arg2>0 ? SYMBOL : EXTRASYMBOL;
03016         int varnumy  = ABS(optim.arg2) - 1;
03017 
03018         for (int i=0; i<(int)optim.eqnidxs.size(); i++) {
03019             WORD *e = ebegin + optim.eqnidxs[i];
03020             if (*(e+1) != OPER_MUL) continue;
03021 
03022             // scan through equation
03023             int powx=0, powy=0;
03024             for (WORD *t=e+3; *t!=0; t+=*t) {
03025                 if (*t == ABS(*(t+*t-1))+1) continue;
03026                 if (*(t+1)==vartypex && *(t+3)==varnumx) powx = *(t+4);
03027                 if (*(t+1)==vartypey && *(t+3)==varnumy) powy = *(t+4);
03028             }
03029 
03030             // substitute if found
03031             if (powx>0 && powy>0 && powx==powy) {
03032 
03033                 WORD sign = 1;
03034                 WORD *newt = e+3;
03035 
03036                 for (WORD *t=e+3; *t!=0;) {
03037                     int dt=*t;
03038 
03039                     if (*t == ABS(*(t+*t-1))+1 ||
03040                             (!(*(t+1)==vartypex && *(t+3)==varnumx) &&
03041                              !(*(t+1)==vartypey && *(t+3)==varnumy))) {
03042                         memmove(newt, t, *t*sizeof(WORD));
03043                         newt += dt;
03044                     }
03045                     else {
03046                         sign *= SGN(*(t+*t-1));
03047                     }
03048 
03049                     t+=dt;
03050                 }
03051 
03052                 *newt++ = 8;           // term length
03053                 *newt++ = EXTRASYMBOL; // extrasymbol
03054                 *newt++ = 4;           // symbol length
03055                 *newt++ = newid;       // symbol id
03056                 *newt++ = powx;        // power
03057                 *newt++ = 1;
03058                 *newt++ = 1;           // coefficient +/-1
03059                 *newt++ = 3*sign;
03060                 *newt++ = 0;           // trailing 0
03061 
03062                 substituted = true;
03063             }
03064         }
03065 
03066         if (!substituted) {
03067 #ifdef DEBUG_GREEDY
03068             MesPrint ("*** [%s, w=%w] DONE: do_optimization : res=false", thetime_str().c_str(),
      optim.improve);
03069 #endif
03070             return false;
03071         }
03072 
03073         // add extra equation (Tnew = x*y)
```

```
03074         instr.push_back(newid);     // eqn.nr
03075         instr.push_back(OPER_MUL); // operator
03076         instr.push_back(20);        // total length
03077         instr.push_back(8);         // LHS length
03078         instr.push_back(vartypex); // (extra)symbol
03079         instr.push_back(4);         // symbol length
03080         instr.push_back(varnumx);  // symbol id
03081         instr.push_back(1);         // power 1
03082         instr.push_back(1);
03083         instr.push_back(1);         // coefficient 1
03084         instr.push_back(3);
03085         instr.push_back(8);         // RHS length
03086         instr.push_back(vartypey); // (extra)symbol
03087         instr.push_back(4);         // symbol length
03088         instr.push_back(varnumy);  // symbol id
03089         instr.push_back(1);         // power 1
03090         instr.push_back(1);
03091         instr.push_back(1);         // coefficient 1
03092         instr.push_back(3);
03093         instr.push_back(0);         // trailing 0
03094     }
03095 //  #] type 1 :
03096 //  #[ type 2 : substitution of the form z=c*x (optim.type==2)
03097
03098     if (optim.type == 2) {
03099         int vartype = optim.arg1>0 ? SYMBOL : EXTRASYMBOL;
03100         int varnum  = ABS(optim.arg1) - 1;
03101
03102         WORD ncoeff = optim.coeff.back();
03103
03104         // scan through equations
03105         for (int i=0; i<(int)optim.eqnidxs.size(); i++) {
03106             WORD *e = ebegin + optim.eqnidxs[i];
03107
03108             if (*(e+1) == OPER_ADD) {
03109                 // scan through ADD-equation
03110                 for (WORD *t=e+3; *t!=0; t+=*t) {
03111
03112                     if (*t == ABS(*(t+*t-1))+1) continue;
03113
03114                     if (*(t+1)==vartype && ABS(*(t+3))==varnum && *(t+4)==1 &&
03115                             BigLong((UWORD *)&optim.coeff[0],ncoeff-1,
03116                                     (UWORD *)t+*t-ABS(*(t+*t-1)),ABS(*(t+*t-1))-1) == 0) {
03117                         // substitute
03118
03119                         int sign = SGN(*(t+*t-1));
03120
03121                         WORD *tend = t;
03122                         while (*tend!=0) tend+=*tend;
03123                         WORD nmove = tend - t - *t;
03124                         memmove(t, t+*t, nmove*sizeof(WORD));
03125                         t += nmove;
03126
03127                         *t++ = 8;           // term length
03128                         *t++ = EXTRASYMBOL; // (extra)symbol
03129                         *t++ = 4;           // symbol length
03130                         *t++ = newid;       // symbol id
03131                         *t++ = 1;           // power of 1
03132                         *t++ = 1;
03133                         *t++ = 1;           // coefficient of +/-1
03134                         *t++ = 3 * sign;
03135                         *t++ = 0;           // trailing 0
03136
03137                         substituted = true;
03138                         break;
03139                     }
03140                 }
03141             }
03142             else if (*(e+1) == OPER_MUL) {
03143
03144                 bool coeff_match=false, var_match=false;
03145                 int sign = 1;
03146
03147                 // scan through MUL-equation
03148                 for (WORD *t=e+3; *t!=0; t+=*t) {
03149                     if (*t == ABS(*(t+*t-1))+1 && BigLong((UWORD *)&optim.coeff[0],ncoeff-1,
03150                                                                                                 (UWORD
    *)t+*t-ABS(*(t+*t-1)),ABS(*(t+*t-1))-1) == 0) {
03151                         coeff_match = true;
03152                         sign *= SGN(*(t+*t-1));
03153                     }
03154                     else if (*(t+1)==vartype && ABS(*(t+3))==varnum && *(t+4)==1) {
03155                         var_match = true;
03156                         sign *= SGN(*(t+*t-1));
03157                     }
03158                 }
03159
```

```
03160                 // substitute if found
03161                 if (coeff_match && var_match) {
03162 
03163                     WORD *newt = e+3;
03164 
03165                     for (WORD *t=e+3; *t!=0;) {
03166 
03167                         int dt=*t;
03168 
03169                         if (*t!=ABS(*(t+*t-1))+1 && !(*(t+1)==vartype && ABS(*(t+3))==varnum &&
      *(t+4)==1)) {
03170                             memmove(newt, t, dt*sizeof(WORD));
03171                             newt += dt;
03172                         }
03173 
03174                         t+=dt;
03175                     }
03176 
03177                     *newt++ = 8;            // term length
03178                     *newt++ = EXTRASYMBOL; // extrasymbol
03179                     *newt++ = 4;            // symbol length
03180                     *newt++ = newid;        // symbol id
03181                     *newt++ = 1;            // power of 1
03182                     *newt++ = 1;
03183                     *newt++ = 1;            // coefficient of +/-1
03184                     *newt++ = 3 * sign;
03185                     *newt++ = 0;            // trailing 0
03186 
03187                     substituted = true;
03188                 }
03189             }
03190         }
03191 
03192         if (!substituted) {
03193 #ifdef DEBUG_GREEDY
03194             MesPrint ("*** [%s, w=%w] DONE: do_optimization : res=false", thetime_str().c_str(),
      optim.improve);
03195 #endif
03196             return false;
03197         }
03198 
03199         // add extra equation (Tnew = c*y)
03200         instr.push_back(newid);             // eqn.nr
03201         instr.push_back(OPER_ADD);          // operator
03202         instr.push_back(9+ABS(ncoeff));     // total length
03203         instr.push_back(5+ABS(ncoeff));     // term length
03204         instr.push_back(vartype);           // (extra)symbol
03205         instr.push_back(4);                 // symbol length
03206         instr.push_back(varnum);            // symbol id
03207         instr.push_back(1);                 // power of 1
03208         for (int i=0; i<ABS(ncoeff); i++)  // coefficient
03209             instr.push_back(optim.coeff[i]);
03210         instr.push_back(0);                 // trailing 0
03211     }
03212 //  #] type 2 :
03213 //  #[ type 3 : substitution of the form z=x+c (optim.type==3)
03214     if (optim.type == 3) {
03215         int vartype = optim.arg1>0 ? SYMBOL : EXTRASYMBOL;
03216         int varnum  = ABS(optim.arg1) - 1;
03217 
03218         WORD ncoeff = optim.coeff.back();
03219 
03220         // scan through equation
03221         for (int i=0; i<(int)optim.eqnidxs.size(); i++) {
03222             WORD *e = ebegin + optim.eqnidxs[i];
03223 
03224             if (*(e+1) != OPER_ADD) continue;
03225 
03226             int coeff_match=0, var_match=0;
03227 
03228             for (WORD *t=e+3; *t!=0; t+=*t) {
03229                 if (*t == ABS(*(t+*t-1))+1 && BigLong((UWORD *)&optim.coeff[0],ABS(ncoeff)-1,
03230                                                                                                 (UWORD
      *)t+*t-ABS(*(t+*t-1)),ABS(*(t+*t-1))-1) == 0)
03231                     coeff_match = SGN(ncoeff) * SGN(*(t+*t-1));
03232                 else if (*(t+1)==vartype && ABS(*(t+3))==varnum && *(t+4)==1)
03233                     var_match = SGN(*(t+7));
03234             }
03235 
03236             // substitute if found (x+c and -x-c and matches)
03237             if (coeff_match * var_match == 1) {
03238 
03239                 WORD *newt = e+3;
03240 
03241                 for (WORD *t=e+3; *t!=0;) {
03242                     int dt=*t;
03243                     if (*t!=ABS(*(t+*t-1))+1 && !(*(t+1)==vartype && ABS(*(t+3))==varnum &&
```

```
      *(t+4)==1)) {
03244                         memmove(newt, t, dt*sizeof(WORD));
03245                         newt += dt;
03246                     }
03247                     t+=dt;
03248                 }
03249
03250                 *newt++ = 8;            // term length
03251                 *newt++ = EXTRASYMBOL;  // extrasymbol
03252                 *newt++ = 4;            // symbol length
03253                 *newt++ = newid;        // symbol id
03254                 *newt++ = 1;            // power of 1
03255                 *newt++ = 1;
03256                 *newt++ = 1;            // coefficient of +/-1
03257                 *newt++ = 3*coeff_match;
03258                 *newt++ = 0;            // trailing zero
03259
03260                 substituted = true;
03261             }
03262         }
03263
03264         if (!substituted) {
03265 #ifdef DEBUG_GREEDY
03266             MesPrint ("*** [%s, w=%w] DONE: do_optimization : res=false", thetime_str().c_str(),
      optim.improve);
03267 #endif
03268             return false;
03269         }
03270
03271         // add extra equation (Tnew = x+c)
03272         instr.push_back(newid);           // eqn.nr
03273         instr.push_back(OPER_ADD);        // operator
03274         instr.push_back(13+ABS(ncoeff));  // total length
03275         instr.push_back(8);               // x-term length
03276         instr.push_back(vartype);         // (extra)symbol
03277         instr.push_back(4);               // symbol length
03278         instr.push_back(varnum);          // symbol id
03279         instr.push_back(1);               // power of 1
03280         instr.push_back(1);
03281         instr.push_back(1);               // coefficient of 1
03282         instr.push_back(3);
03283         instr.push_back(ABS(ncoeff)+1);   // c-term length
03284         for (int i=0; i<ABS(ncoeff); i++) // coefficient
03285             instr.push_back(optim.coeff[i]);
03286         instr.push_back(0);               // trailing zero
03287     }
03288 //  #] type 3 :
03289 //  #[ type 4,5 : substitution of the form z=x+y or z=x-y (optim.type=4 or 5)
03290     if (optim.type >= 4) {
03291
03292         int vartypex = optim.arg1>0 ? SYMBOL : EXTRASYMBOL;
03293         int varnumx  = ABS(optim.arg1) - 1;
03294         int vartypey = optim.arg2>0 ? SYMBOL : EXTRASYMBOL;
03295         int varnumy  = ABS(optim.arg2) - 1;
03296
03297         // scan through equations
03298         for (int i=0; i<(int)optim.eqnidxs.size(); i++) {
03299             WORD *e = ebegin + optim.eqnidxs[i];
03300
03301             if (*(e+1) != OPER_ADD) continue;
03302
03303             const WORD *coeffx=NULL, *coeffy=NULL;
03304             WORD ncoeffx=0,ncoeffy=0;
03305
03306             // looks for terms
03307             for (WORD *t=e+3; *t!=0; t+=*t) {
03308                 if (*t == ABS(*(t+*t-1))+1) continue; // constant
03309                 if (*(t+1)==vartypex && *(t+3)==varnumx && *(t+4)==1) {
03310                     coeffx = t+5;
03311                     ncoeffx = *(t+*t-1);
03312                 }
03313                 if (*(t+1)==vartypey && *(t+3)==varnumy && *(t+4)==1) {
03314                     coeffy = t+5;
03315                     ncoeffy = *(t+*t-1);
03316                 }
03317             }
03318
03319             // check signs (type=4: x+y and -x-y, type=5: x-y and -x+y) ??????
03320             // check signs (type=4: x+y, type=5: x-y) !!!!!!!!!!
03321             if (SGN(ncoeffx) * SGN(ncoeffy) * (optim.type==4 ? 1 : -1) == 1) {
03322                 // check absolute value of coefficients
03323                 if (BigLong((UWORD *)coeffx, ABS(ncoeffx)-1, (UWORD *)coeffy, ABS(ncoeffy)-1) == 0) {
03324                     // substitute
03325                     vector<WORD> coeff(coeffx, coeffx+ABS(ncoeffx));
03326
03327                     WORD *newt = e+3;
03328 /*
```

```
03329 if ( optim.type == 5 ) {
03330     while ( *newt ) newt+=*newt;
03331     int i = (newt - e) - 3;
03332     MesPrint("  < %a",i,e+3);
03333     newt = e+3;
03334 }
03335 */
03336                     for (WORD *t=e+3; *t!=0;) {
03337                         int dt=*t;
03338                         if (*t == ABS(*(t+*t-1))+1 ||
03339                                 (!(*(t+1)==vartypex && *(t+3)==varnumx) &&
03340                                  !(*(t+1)==vartypey && *(t+3)==varnumy))) {
03341                             memmove(newt, t, dt*sizeof(WORD));
03342                             newt += dt;
03343                         }
03344                         t+=dt;
03345                     }
03346 
03347                     *newt++ = 5 + ABS(ncoeffx);       // term length
03348                     *newt++ = EXTRASYMBOL;            // extrasymbol
03349                     *newt++ = 4;                      // symbol length
03350                     *newt++ = newid;                  // symbol id
03351                     *newt++ = 1;                      // power of 1
03352                     for (int j=0; j<ABS(ncoeffx); j++)// coefficient
03353                         *newt++ = coeff[j];
03354                     *newt++ = 0;                      // trailing 0
03355                     substituted = true;
03356 /*
03357 if ( optim.type == 5 ) {
03358     int i = (newt - e) - 4;
03359     MesPrint("  > %a",i,e+3);
03360 }
03361 */
03362                 }
03363             }
03364         }
03365 
03366         if (!substituted) {
03367 #ifdef DEBUG_GREEDY
03368             MesPrint ("*** [%s, w=%w] DONE: do_optimization : res=false", thetime_str().c_str(),
      optim.improve);
03369 #endif
03370             return false;
03371         }
03372 /*
03373 if ( optim.type == 5 )
03374 MesPrint ("improve=%d, %c%d%c%c%d)", optim.improve,
03375     optim.arg1>0?'x':'Z', ABS(optim.arg1)-1,
03376     optim.type==1 ? '*' : optim.type==4 ? '+' : '-',
03377     optim.arg2>0?'x':'Z', ABS(optim.arg2)-1);
03378 */
03379         // add extra equation (Tnew = x+/-y)
03380         instr.push_back(newid);     // eqn.nr
03381         instr.push_back(OPER_ADD); // operator
03382         instr.push_back(20);        // total length
03383         instr.push_back(8);         // term length
03384         instr.push_back(vartypex); // (extra)symbol
03385         instr.push_back(4);         // symbol length
03386         instr.push_back(varnumx);  // symbol id
03387         instr.push_back(1);         // power of 1
03388         instr.push_back(1);
03389         instr.push_back(1);         // coefficient of 1
03390         instr.push_back(3);
03391         instr.push_back(8);         // term length
03392         instr.push_back(vartypey); // (extra)symbol
03393         instr.push_back(4);         // symbol length
03394         instr.push_back(varnumy);  // symbol id
03395         instr.push_back(1);         // power of 1
03396         instr.push_back(1);
03397         instr.push_back(1);         // coefficient of +/-1
03398         instr.push_back(3*(optim.type==4?1:-1));
03399         instr.push_back(0);         // trailing 0
03400     }
03401 //  #] type 4,5 :
03402 //  #[ trivial :  remove trivial equations of the form Zi = +/-Zj
03403     vector<int> renum(newid+1, 0);
03404     bool do_renum=false;
03405 
03406     // vector may be moved when it is extended
03407     ebegin = &*instr.begin();
03408 
03409     for (int i=0; i<(int)optim.eqnidxs.size(); i++) {
03410         WORD *e = ebegin + optim.eqnidxs[i];
03411         WORD *t = e+3;
03412         if (*t==0) continue;       // empty (removed) equation
03413         if (*(t+*t)!=0) continue; // more than 1 term
03414         if (*t!=8) continue;       // term not of correct form
```

```
03415         if (*(t+4)!=1) continue;  // power != 1
03416         if (*(t+5)!=1 || *(t+6)!=1) continue; // coeff != +/-1
03417 
03418         // trivial term, so renumber this one
03419         renum[*e] = SGN(*(t+7)) * (*(t+3) + 1);
03420         do_renum = true;
03421 
03422         // remove equation
03423         *t=0;
03424     }
03425 //  #] trivial :
03426 //  #[ renumbering :
03427 
03428     // there are renumberings to be done, so loop through all equations
03429     if (do_renum) {
03430         WORD *eend = ebegin+instr.size();
03431 
03432         for (WORD *e=ebegin; e!=eend; e+=*(e+2)) {
03433             for (WORD *t=e+3; *t!=0; t+=*t) {
03434                 if (*t == ABS(*(t+*t-1))+1) continue;
03435                 if (*(t+1)==EXTRASYMBOL && renum[*(t+3)]!=0) {
03436                     *(t+*t-1) *= SGN(renum[*(t+3)]);
03437                     *(t+3) = ABS(renum[*(t+3)]) - 1;
03438                 }
03439             }
03440         }
03441     }
03442 //  #] renumbering :
03443 
03444 #ifdef DEBUG_GREEDY
03445     MesPrint ("*** [%s, w=%w] DONE: do_optimization : res=true", thetime_str().c_str(),
    optim.improve);
03446 #endif
03447 
03448     return true;
03449 }
03450 
03451 /*
03452     #] do_optimization :
03453     #[ partial_factorize :
03454 */
03455 
03479 int partial_factorize (vector<WORD> &instr, int n, int improve) {
03480 
03481 #ifdef DEBUG_GREEDY
03482     MesPrint ("*** [%s, w=%w] CALL: partial_factorize (n=%d)", thetime_str().c_str(), n);
03483 #endif
03484 
03485     GETIDENTITY;
03486 
03487     // get starting positions of instructions
03488     vector<int> instr_idx(n);
03489     WORD *ebegin = &*instr.begin();
03490     WORD *eend = ebegin+instr.size();
03491     for (WORD *e=ebegin; e!=eend; e+=*(e+2)) {
03492         instr_idx[*e] = e - ebegin;
03493     }
03494 
03495     // get reference counts
03496 /*
03497  *  The next construction replaces the vector construction which is
03498  *  rather costly for valgrind (and maybe also in normal running)
03499  */
03500     int nmax = 2*n;
03501     WORD *numpar = (WORD *)Malloc1(nmax*sizeof(WORD),"numpar");
03502     for ( int i = 0; i < nmax; i++ ) numpar[i] = 0;
03503 //  vector<WORD> numpar(n);
03504     for (WORD *e=ebegin; e!=eend; e+=*(e+2))
03505         for (WORD *t=e+3; *t!=0; t+=*t) {
03506             if (*t == ABS(*(t+*t-1))+1) continue;
03507             if (*(t+1) == EXTRASYMBOL) numpar[*(t+3)]++;
03508         }
03509 
03510     // find factorizable expressions
03511     for (int i=0; i<n; i++) {
03512         WORD *e = &*instr.begin() + instr_idx[i];
03513         if (*(e+1) != OPER_ADD) continue;
03514 
03515         // count symbol occurrences
03516         map<WORD,WORD> cnt; // 1-indexed, <0:EXTRASYMBOL, >0:SYMBOL
03517 
03518         for (WORD *t=e+3; *t!=0; t+=*t) {
03519             if (*t==ABS(*(t+*t-1))+1) continue;
03520 
03521             // count symbols in t
03522             if (*(t+4)==1)
03523                 cnt[(*(t+1)==SYMBOL ? 1 : -1) * (*(t+3)+1)]++;
```

```
03524
03525             // count symbols in extrasymbols of t
03526             if (*(t+1)==EXTRASYMBOL && *(t+4)==1 && numpar[*(t+3)]==1) {
03527                 WORD *t2 = &*instr.begin() + instr_idx[*(t+3)];
03528                 if (*(t2+1) != OPER_MUL) continue;
03529                 for (t2+=3; *t2!=0; t2+=*t2) {
03530                     if (*t2 == ABS(*(t2+*t2-1))+1) continue;
03531                     if (*(t2+4)==1)
03532                         cnt[(*(t2+1)==SYMBOL ? 1 : -1) * (*(t2+3)+1)]++;
03533                 }
03534             }
03535         }
03536
03537         // find most-occurring symbol
03538         WORD x=0, best=0;
03539         for (map<WORD,WORD>::iterator it=cnt.begin(); it!=cnt.end(); it++)
03540             if (it->second > best) { x=it->first; best=it->second; }
03541
03542         // occurrence>=2 and occurrence>improve, so factorize
03543
03544         if (best>=2 && best>improve) {
03545             // initialize new equation (Zi from example above)
03546             vector<WORD> new_eqn;
03547             new_eqn.push_back(n);
03548             new_eqn.push_back(OPER_ADD);
03549             new_eqn.push_back(0); // length
03550
03551             WORD dt;
03552             WORD *newt=e+3;
03553             for (WORD *t=e+3; *t!=0; t+=dt) {
03554                 dt = *t;
03555                 bool keep=true;
03556
03557                 if (*t!=ABS(*(t+*t-1))+1) {
03558
03559                     // factorized symbol is in t itself
03560                     if (*(t+4)==1) {
03561                         WORD y = (*(t+1)==SYMBOL ? 1 : -1) * (*(t+3)+1);
03562                         if (y==x) {
03563                             new_eqn.push_back(*t-4);
03564                             new_eqn.insert(new_eqn.end(), t+5, t+dt);
03565                             keep=false;
03566                         }
03567                     }
03568
03569                     // look in extrasymbol of t with ref.count=1
03570                     if (*(t+1)==EXTRASYMBOL && *(t+4)==1 && numpar[*(t+3)]==1) {
03571                         WORD *t2 = &*instr.begin() + instr_idx[*(t+3)];
03572                         if (*(t2+1) == OPER_MUL) {
03573                             bool has_x=false;
03574                             for (t2+=3; *t2!=0; t2+=*t2) {
03575                                 if (*t2 == ABS(*(t2+*t2-1))+1) continue;
03576                                 WORD y = (*(t2+1)==SYMBOL ? 1 : -1) * (*(t2+3)+1);
03577                                 // extrasymbol has factorized symbol
03578                                 if (y==x && *(t2+4)==1) {
03579                                     has_x=true;
03580                                     // copy remaining part
03581                                     WORD *tend=t2+*t2;
03582                                     WORD sign = SGN(*(tend-1));
03583                                     while (*tend!=0) tend+=*tend;
03584                                     int dt2 = tend - (t2+*t2);
03585                                     memmove(t2, t2+*t2, (dt2+1)*sizeof(WORD));
03586                                     t2 += dt2;
03587                                     *(t2-1) *= sign;
03588                                     break;
03589                                 }
03590                             }
03591                             if (has_x) {
03592                                 // extrasymbol has x, so add it to new equation
03593                                 keep=false;
03594                                 int thisidx=new_eqn.size();
03595                                 new_eqn.insert(new_eqn.end(), t, t+dt);
03596                                 t2 = &*instr.begin() + instr_idx[*(t+3)] + 3;
03597                                 // if becomes trivial, substitute the term
03598                                 if (*(t2+*t2)==0) {
03599                                     // it's a number
03600                                     if (*t2 == ABS(*(t2+*t2-1))+1) {
03601                                         if (ABS(new_eqn[new_eqn.size()-1])==3 &&
      new_eqn[new_eqn.size()-2]==1 && new_eqn[new_eqn.size()-3]==1) {
03602                                             // original equation has coefficient of +/-1, so replace
      it
03603                                             WORD sign = SGN(new_eqn.back());
03604                                             new_eqn.erase(new_eqn.begin()+thisidx, new_eqn.end());
03605                                             new_eqn.insert(new_eqn.end(), t2, t2+*t2);
03606                                             new_eqn.back() *= sign;
03607                                             *t2 = 0;
03608                                         }
```

```
03609                                       else {
03610                                           // two non-trivial coefficients, so multiply them
03611                                           // note: untested code (found no way to trigger it)
03612                                           UWORD *tmp = NumberMalloc("partial_factorize");
03613                                           WORD ntmp=0;
03614                                           MulRat(BHEAD (UWORD *)t2+*t2-ABS(*(t2+*t2-1)),
     *(t2+*t2-1),
03615                                                                                   (UWORD
     *)&*(new_eqn.end()-ABS(new_eqn.back())), new_eqn.back(),
03616                                                                                   tmp, &ntmp);
03617                                           new_eqn.erase(new_eqn.begin()+thisidx, new_eqn.end());
03618                                           new_eqn.push_back(ABS(ntmp)+1);
03619                                           new_eqn.insert(new_eqn.end(), tmp, tmp+ABS(ntmp));
03620                                           NumberFree(tmp,"partial_factorize");
03621                                           *t2 = 0;
03622                                       }
03623                                   }
03624                                   else if (*(t2+4)==1) {
03625                                       // it's a variable
03626                                       new_eqn.back() *= SGN(*(t2+*t2-1));
03627                                       new_eqn[thisidx+1] = *(t2+1);
03628                                       new_eqn[thisidx+2] = *(t2+2);
03629                                       new_eqn[thisidx+3] = *(t2+3);
03630                                       new_eqn[thisidx+4] = *(t2+4);
03631                                       *t2 = 0;
03632                                   }
03633                               }
03634                           }
03635                       }
03636                   }
03637               }
03638
03639               // no x, so copy it
03640               if (keep) {
03641                   memmove(newt, t, dt*sizeof(WORD));
03642                   newt += dt;
03643               }
03644           }
03645
03646           // finalize new equation
03647           new_eqn.push_back(0);
03648           new_eqn[2] = new_eqn.size();
03649
03650           bool empty = newt == e+3;
03651           if ( n+1 >= nmax ) {
03652               int i, newnmax = nmax*2;
03653               WORD *newnumpar = (WORD *)Malloc1(newnmax*sizeof(WORD),"newnumpar");
03654               for ( i = 0; i < n; i++ ) newnumpar[i] = numpar[i];
03655               for ( ; i < newnmax; i++ ) newnumpar[i] = 0;
03656               M_free(numpar,"numpar");
03657               numpar = newnumpar;
03658               nmax = newnmax;
03659           }
03660 //          numpar.push_back(0);
03661           n++;
03662
03663           // if original is not empty, add new equation (Zj) to it
03664           // otherwise replace it later
03665           if (!empty) {
03666               *newt++ = 8;
03667               *newt++ = EXTRASYMBOL;
03668               *newt++ = 4;
03669               *newt++ = n;
03670               *newt++ = 1;
03671               *newt++ = 1;
03672               *newt++ = 1;
03673               *newt++ = 3;
03674               *newt++ = 0;
03675           }
03676
03677           // add new equation to instructions
03678           instr_idx.push_back(instr.size());
03679           instr.insert(instr.end(), new_eqn.begin(), new_eqn.end());
03680
03681           // generate another new equation (Zj=x*Zi)
03682           new_eqn.clear();
03683           new_eqn.push_back(n);
03684           new_eqn.push_back(OPER_MUL);
03685           new_eqn.push_back(20);
03686           new_eqn.push_back(8);
03687
03688           // add factorized symbol
03689           if (x>0) {
03690               new_eqn.push_back(SYMBOL);
03691               new_eqn.push_back(4);
03692               new_eqn.push_back(x-1);
03693               new_eqn.push_back(1);
```

```
03694             }
03695             else {
03696                 new_eqn.push_back(EXTRASYMBOL);
03697                 new_eqn.push_back(4);
03698                 new_eqn.push_back(-x-1);
03699                 new_eqn.push_back(1);
03700             }
03701             new_eqn.push_back(1);
03702             new_eqn.push_back(1);
03703             new_eqn.push_back(3);
03704             new_eqn.push_back(8);
03705             // add new equation (Zi)
03706             new_eqn.push_back(EXTRASYMBOL);
03707             new_eqn.push_back(4);
03708             new_eqn.push_back(n-1);
03709             new_eqn.push_back(1);
03710             new_eqn.push_back(1);
03711             new_eqn.push_back(1);
03712             new_eqn.push_back(3);
03713             new_eqn.push_back(0);
03714 
03715             if (!empty) {
03716                 // add new equation (Zj) to instructions
03717                 instr_idx.push_back(instr.size());
03718                 instr.insert(instr.end(), new_eqn.begin(), new_eqn.end());
03719                 if ( n+1 >= nmax ) {
03720                     int i, newnmax = nmax*2;
03721                     WORD *newnumpar = (WORD *)Malloc1(newnmax*sizeof(WORD),"newnumpar");
03722                     for ( i = 0; i < n; i++ ) newnumpar[i] = numpar[i];
03723                     for ( ; i < newnmax; i++ ) newnumpar[i] = 0;
03724                     M_free(numpar,"numpar");
03725                     numpar = newnumpar;
03726                     nmax = newnmax;
03727                 }
03728 //              numpar.push_back(0);
03729                 n++;
03730             }
03731             else {
03732                 // replace e with Zj
03733                 e = &*instr.begin() + instr_idx[i];
03734                 e[1] = OPER_MUL;
03735                 memcpy(e+3, &new_eqn[3], (new_eqn.size()-3)*sizeof(WORD));
03736             }
03737 
03738             // decrease i, so this expression is factorized again if possible
03739             i--;
03740         }
03741     }
03742 
03743 #ifdef DEBUG_GREEDY
03744     MesPrint ("*** [%s, w=%w] DONE: partial_factorize (n=%d)", thetime_str().c_str(), n);
03745 #endif
03746     M_free(numpar,"numpar");
03747     return n;
03748 }
03749 
03750 /*
03751     #] partial_factorize :
03752     #[ optimize_greedy :
03753 */
03754 
03773 vector<WORD> optimize_greedy (vector<WORD> instr, LONG time_limit) {
03774 
03775 #ifdef DEBUG
03776     int old_num_oper = count_operators(instr);
03777     MesPrint ("*** [%s, w=%w] CALL: optimize_greedy(numoper=%d)",
03778                         thetime_str().c_str(), old_num_oper);
03779 #endif
03780 
03781     LONG start_time = TimeWallClock(1);
03782 
03783     WORD *ebegin = &*instr.begin();
03784     WORD *eend = ebegin+instr.size();
03785 
03786     // store final equation, since it must be the last equation later
03787     int final_eqn_idx = 0;
03788     int next_eqn = 0;
03789 
03790     for (WORD *e=ebegin; e!=eend; e+=*(e+2)) {
03791         next_eqn = *e + 1;
03792         final_eqn_idx = e-ebegin;
03793     }
03794     // optimize instructions
03795     while (TimeWallClock(1)-start_time < time_limit) {
03796         int old_next_eqn = next_eqn;
03797 
03798         // find optimizations
```

```
03799         vector<optimization> optim = find_optimizations(instr);
03800 
03801         // add_eqnidxs contains modified equations, which might have to be updated later again
03802         vector<int> add_eqnidxs;
03803 
03804         // number of optimizations to do
03805         int num_do_optims = max(AO.Optimize.greedyminnum,
      (int)optim.size()*AO.Optimize.greedymaxperc/100);
03806 
03807         // if best improvement is one, do all optimizations
03808         int best_improve=0;
03809         for (int i=0; i<(int)optim.size(); i++)
03810             best_improve = max(best_improve, optim[i].improve);
03811         if (best_improve <= 1)
03812             num_do_optims = MAXPOSITIVE;
03813         // do a number of optimizations
03814         while (optim.size() > 0 && num_do_optims-- > 0) {
03815 
03816             // find best optimization
03817             int best=0;
03818             best_improve=0;
03819             for (int i=0; i<(int)optim.size(); i++)
03820                 if (optim[i].improve > best_improve) {
03821                     best=i;
03822                     best_improve=optim[i].improve;
03823                 }
03824 
03825             // add extra equations
03826             for (int i=0; i<(int)add_eqnidxs.size(); i++)
03827                 optim[best].eqnidxs.push_back(add_eqnidxs[i]);
03828 
03829             // do optimization, update next_eqn if successful
03830             int next_idx = instr.size();
03831             if (do_optimization(optim[best], instr, next_eqn)) {
03832                 next_eqn++;
03833                 add_eqnidxs.push_back(next_idx);
03834             }
03835 
03836             optim.erase(optim.begin()+best);
03837         }
03838 
03839         // partially factorize with improve >= best_improve
03840         next_eqn = partial_factorize(instr, next_eqn, best_improve);
03841 
03842         // check whether nothing has changed
03843         if (next_eqn == old_next_eqn) break;
03844     }
03845 
03846     // add final equation to the back (must be by definition)
03847     instr.push_back(next_eqn);
03848     instr.insert(instr.end(), instr.begin()+final_eqn_idx+1,
      instr.begin()+final_eqn_idx+instr[final_eqn_idx+2]);
03849 
03850     // removed original final equation
03851     instr[final_eqn_idx+3] = 0;
03852 
03853     // remove extra zeroes
03854     WORD *t = &instr[0];
03855 
03856     ebegin = &*instr.begin();
03857     eend = ebegin+instr.size();
03858     int de=0;
03859 
03860     for (WORD *e=ebegin; e!=eend; e+=de) {
03861         de = *(e+2);
03862         int n=3;
03863         while (*(e+n) != 0) n+=*(e+n);
03864         n++;
03865         memmove (t, e, n*sizeof(WORD));
03866         *(t+2) = n;
03867         t += n;
03868     }
03869 
03870     instr.resize(t - &instr[0]);
03871 
03872 #ifdef DEBUG
03873     MesPrint ("*** [%s, w=%w] DONE: optimize_greedy(numoper=%d) : numoper=%d",
03874                         thetime_str().c_str(), old_num_oper, count_operators(instr));
03875 #endif
03876 
03877     return instr;
03878 }
03879 
03880 /*
03881     #] optimize_greedy :
03882     #[ recycle_variables :
03883 */
```

```
03884
03913 vector<WORD> recycle_variables (const vector<WORD> &all_instr) {
03914
03915 #ifdef DEBUG_MORE
03916     MesPrint ("*** [%s, w=%w] CALL: recycle_variables", thetime_str().c_str());
03917 #endif
03918
03919     // get starting positions of instructions
03920     vector<const WORD *> instr;
03921     const WORD *tbegin = &*all_instr.begin();
03922     const WORD *tend = tbegin+all_instr.size();
03923     for (const WORD *t=tbegin; t!=tend; t+=*(t+2))
03924         instr.push_back(t);
03925     int n = instr.size();
03926
03927     // determine with expressions are connected, how many intermediate
03928     // are needed (assuming it's a expression tree instead of a DAG) and
03929     // sort the leaves such that you need a minimal number of variables
03930     vector<int> vars_needed(n);
03931     vector<bool> vis(n,false);
03932     vector<vector<int> > conn(n);
03933
03934     stack<int> s;
03935     s.push(n);
03936
03937     while (!s.empty()) {
03938         int i=s.top(); s.pop();
03939         if (i>0) {
03940             i--;
03941             if (vis[i]) continue;
03942             vis[i]=true;
03943             s.push(-(i+1));
03944
03945             // find all connections
03946             for (const WORD *t=instr[i]+3; *t!=0; t+=*t)
03947                 if (*t!=1+ABS(*(t+*t-1)) && *(t+1)==EXTRASYMBOL) {
03948                     int k = *(t+3);
03949                     conn[i].push_back(k);
03950                     s.push(k+1);
03951                 }
03952         }
03953         else {
03954             i=-i-1;
03955
03956             // sort the childs w.r.t. needed variables
03957             vector<pair<int,int> > need;
03958             for (int j=0; j<(int)conn[i].size(); j++)
03959                 need.push_back(make_pair(vars_needed[conn[i][j]], conn[i][j]));
03960
03961             // keep the comma expression in proper order
03962             if (*(instr[i]+1) != OPER_COMMA)
03963                 sort(need.rbegin(), need.rend());
03964
03965             vars_needed[i] = 1;
03966             for (int j=0; j<(int)need.size(); j++) {
03967                 vars_needed[i] = max(vars_needed[i], need[j].first+j);
03968                 conn[i][j] = need[j].second;
03969             }
03970         }
03971     }
03972
03973     // order the instructions in depth-first order and determine the first
03974     // and last occurrences of variables
03975     vector<int> order, first(n,0), last(n,0);
03976     vis = vector<bool>(n,false);
03977     s.push(n);
03978
03979     while (!s.empty()) {
03980
03981         int i=s.top(); s.pop();
03982
03983         if (i>0) {
03984             i--;
03985             if (vis[i]) continue;
03986             vis[i]=true;
03987             s.push(-(i+1));
03988             for (int j=(int)conn[i].size()-1; j>=0; j--)
03989                 s.push(conn[i][j]+1);
03990         }
03991         else {
03992             i=-i-1;
03993             first[i] = last[i] = order.size();
03994             order.push_back(i);
03995             for (int j=0; j<(int)conn[i].size(); j++) {
03996                 int k = conn[i][j];
03997                 last[k] = max(last[k], first[i]);
03998             }
```

```
03999         }
04000     }
04001
04002     // find the renumbering to recycled variables, where at any time the
04003     // lowest-indexed variable that can be used is chosen
04004     int numvar=0;
04005     set<int> var;
04006     vector<int> renum(n);
04007
04008     for (int i=0; i<(int)order.size(); i++) {
04009         for (int j=0; j<(int)conn[order[i]].size(); j++) {
04010             int k = conn[order[i]][j];
04011             if (last[k] == i) var.insert(renum[k]);
04012         }
04013
04014         if (var.empty()) var.insert(numvar++);
04015         renum[order[i]] = *var.begin(); var.erase(var.begin());
04016     }
04017
04018     // put the number of variables used in a preprocessor variable
04019
04020     // generate new instructions with the renumbering
04021     vector<WORD> newinstr;
04022
04023     for (int i=0; i<(int)order.size(); i++) {
04024         int x = order[i];
04025         int j = newinstr.size();
04026         newinstr.insert(newinstr.end(), instr[x], instr[x]+*(instr[x]+2));
04027
04028         newinstr[j] = renum[newinstr[j]];
04029
04030         for (WORD *t=&newinstr[j+3]; *t!=0; t+=*t)
04031             if (*t!=1+ABS(*(t++t-1)) && *(t+1)==EXTRASYMBOL)
04032                 *(t+3) = renum[*(t+3)];
04033     }
04034
04035 #ifdef DEBUG_MORE
04036     MesPrint ("*** [%s, w=%w] DONE: recycle_variables", thetime_str().c_str());
04037 #endif
04038
04039     return newinstr;
04040 }
04041
04042 /*
04043     #] recycle_variables :
04044     #[ optimize_expression_given_Horner :
04045 */
04046
04060 void optimize_expression_given_Horner () {
04061
04062 #ifdef DEBUG
04063     MesPrint ("*** [%s, w=%w] CALL: optimize_expression_given_Horner", thetime_str().c_str());
04064 #endif
04065
04066     GETIDENTITY;
04067
04068     // initialize timer
04069     LONG start_time = TimeWallClock(1);
04070     LONG time_limit = 100 * AO.Optimize.greedytimelimit / (AO.Optimize.horner == O_MCTS ?
      AO.Optimize.mctsnumkeep : 1);
04071     if (time_limit == 0) time_limit=MAXPOSITIVE;
04072
04073     // pick a Horner scheme from the list
04074     LOCK(optimize_lock);
04075     vector<WORD> Horner_scheme = optimize_best_Horner_schemes.back();
04076     optimize_best_Horner_schemes.pop_back();
04077     UNLOCK(optimize_lock);
04078
04079 //  if ( ( AO.Optimize.debugflags&2 ) == 2 ) {
04080 //      MesPrint ("Scheme: %a",Horner_scheme.size(),&(Horner_scheme[0]));
04081 //  }
04082
04083     // apply Horner scheme
04084     vector<WORD> tree = Horner_tree(optimize_expr, Horner_scheme);
04085
04086     // generate instructions, eventually with CSE
04087     vector<WORD> instr;
04088
04089     if (AO.Optimize.method == O_CSE || AO.Optimize.method == O_CSEGREEDY)
04090         instr = generate_instructions(tree, true);
04091     else
04092         instr = generate_instructions(tree, false);
04094     if (AO.Optimize.method == O_CSEGREEDY || AO.Optimize.method == O_GREEDY) {
04095         instr = merge_operators(instr, false);
04096         instr = optimize_greedy(instr, time_limit-(TimeWallClock(1)-start_time));
04097         instr = merge_operators(instr, true);
04098         instr = optimize_greedy(instr, time_limit-(TimeWallClock(1)-start_time));
```

```
04099     }
04100     instr = merge_operators(instr, true);
04101 
04102     // recycle the temporary variables
04103     instr = recycle_variables(instr);
04104 
04105     // determine the quality of the code and possibly update the best code
04106     int num_oper = count_operators(instr);
04107 
04108     LOCK(optimize_lock);
04109     if (num_oper < optimize_best_num_oper) {
04110         optimize_num_vars = Horner_scheme.size();
04111         optimize_best_num_oper = num_oper;
04112         optimize_best_instr = instr;
04113         optimize_best_vars = vector<WORD>(AN.poly_vars, AN.poly_vars+AN.poly_num_vars);
04114     }
04115     UNLOCK(optimize_lock);
04116 
04117   // clean poly_vars, that are allocated by Horner_tree
04118     AN.poly_num_vars = 0;
04119     M_free(AN.poly_vars,"poly_vars");
04120 
04121 #ifdef DEBUG
04122     MesPrint ("*** [%s, w=%w] DONE: optimize_expression_given_Horner", thetime_str().c_str());
04123 #endif
04124 }
04125 
04126 /*
04127     #] optimize_expression_given_Horner :
04128     #[ PF_optimize_expression_given_Horner :
04129 */
04130 #ifdef WITHMPI
04131 
04132 // Initialization.
04133 void PF_optimize_expression_given_Horner_master_init () {
04134     // Nothing to do for now.
04135 }
04136 
04137 // Wait for an idle slave and return the process number.
04138 int PF_optimize_expression_given_Horner_master_next() {
04139 
04140     // Find an idle slave.
04141     int next;
04142     PF_LongSingleReceive(PF_ANY_SOURCE, PF_OPT_HORNER_MSGTAG, &next, NULL);
04143     return next;
04144 
04145 }
04146 
04147 // The main function on the master.
04148 void PF_optimize_expression_given_Horner_master () {
04149 
04150 #ifdef DEBUG
04151     MesPrint ("*** [%s, w=%w] CALL: PF_optimize_expression_given_Horner_master",
      thetime_str().c_str());
04152 #endif
04153 
04154     // pick a Horner scheme from the list
04155     vector<WORD> Horner_scheme = optimize_best_Horner_schemes.back();
04156     optimize_best_Horner_schemes.pop_back();
04157 
04158     // Find an idle slave.
04159     int next = PF_optimize_expression_given_Horner_master_next();
04160 
04161     // Send a new task to the slave.
04162     PF_PrepareLongSinglePack();
04163     int len = Horner_scheme.size();
04164     PF_LongSinglePack(&len, 1, PF_INT);
04165     PF_LongSinglePack(&Horner_scheme[0], len, PF_WORD);
04166     PF_LongSingleSend(next, PF_OPT_HORNER_MSGTAG);
04167 
04168 #ifdef DEBUG
04169     MesPrint ("*** [%s, w=%w] DONE: PF_optimize_expression_given_Horner_master",
      thetime_str().c_str());
04170 #endif
04171 
04172 }
04173 
04174 // Wait for all the slaves to finish their tasks.
04175 void PF_optimize_expression_given_Horner_master_wait () {
04176 
04177 #ifdef DEBUG
04178     MesPrint ("*** [%s, w=%w] CALL: PF_optimize_expression_given_Horner_master_wait",
      thetime_str().c_str());
04179 #endif
04180 
04181     // Wait for all the slaves.
04182     for (int i = 1; i < PF.numtasks; i++) {
```

```
04183         int next = PF_optimize_expression_given_Horner_master_next();
04184         // Send a null task.
04185         PF_PrepareLongSinglePack();
04186         int len = 0;
04187         PF_LongSinglePack(&len, 1, PF_INT);
04188         PF_LongSingleSend(next, PF_OPT_HORNER_MSGTAG);
04189     }
04190
04191     // Combine the result from all the slaves.
04192     optimize_best_num_oper = INT_MAX;
04193     for (int i = 1; i < PF.numtasks; i++) {
04194         PF_LongSingleReceive(PF_ANY_SOURCE, PF_OPT_COLLECT_MSGTAG, NULL, NULL);
04195
04196         int len;
04197
04198         // The first integer is the number of operations.
04199         PF_LongSingleUnpack(&len, 1, PF_INT);
04200
04201         if (len >= optimize_best_num_oper) continue;
04202
04203         // Update the best result.
04204         optimize_best_num_oper = len;
04205         PF_LongSingleUnpack(&len, 1, PF_INT);
04206         optimize_best_instr.resize(len);
04207         PF_LongSingleUnpack(&optimize_best_instr[0], len, PF_WORD);
04208         PF_LongSingleUnpack(&len, 1, PF_INT);
04209         optimize_best_vars.resize(len);
04210         PF_LongSingleUnpack(&optimize_best_vars[0], len, PF_WORD);
04211
04212         optimize_num_vars = optimize_best_vars.size();  // TODO
04213     }
04214
04215 #ifdef DEBUG
04216     MesPrint ("*** [%s, w=%w] DONE: PF_optimize_expression_given_Horner_master_wait",
      thetime_str().c_str());
04217 #endif
04218
04219 }
04220
04221 // The main function on the slaves.
04222 void PF_optimize_expression_given_Horner_slave () {
04223
04224 #ifdef DEBUG
04225     MesPrint ("*** [%s, w=%w] CALL: PF_optimize_expression_given_Horner_slave",
      thetime_str().c_str());
04226 #endif
04227
04228     optimize_best_Horner_schemes.clear();
04229     optimize_best_num_oper = INT_MAX;
04230
04231     int dummy = 0;
04232     int len;
04233
04234     for (;;) {
04235         // Ask the master for the next task.
04236         PF_PrepareLongSinglePack();
04237         PF_LongSinglePack(&dummy, 1, PF_INT);
04238         PF_LongSingleSend(MASTER, PF_OPT_HORNER_MSGTAG);
04239
04240         // Get a task from the master.
04241         PF_LongSingleReceive(MASTER, PF_OPT_HORNER_MSGTAG, NULL, NULL);
04242
04243         // Length of the task.
04244         PF_LongSingleUnpack(&len, 1, PF_INT);
04245
04246         // No task remains.
04247         if (len == 0) break;
04248
04249         // Perform the given task.
04250         optimize_best_Horner_schemes.push_back(vector<WORD>());
04251         vector<WORD> &Horner_scheme = optimize_best_Horner_schemes.back();
04252         Horner_scheme.resize(len);
04253         PF_LongSingleUnpack(&Horner_scheme[0], len, PF_WORD);
04254         optimize_expression_given_Horner ();
04255     }
04256
04257     // Send the result to the master.
04258     PF_PrepareLongSinglePack();
04259     PF_LongSinglePack(&optimize_best_num_oper, 1, PF_INT);
04260     if (optimize_best_num_oper != INT_MAX) {
04261         len = optimize_best_instr.size();
04262         PF_LongSinglePack(&len, 1, PF_INT);
04263         PF_LongSinglePack(&optimize_best_instr[0], len, PF_WORD);
04264         len = optimize_best_vars.size();
04265         PF_LongSinglePack(&len, 1, PF_INT);
04266         PF_LongSinglePack(&optimize_best_vars[0], len, PF_WORD);
04267     }
```

```
04268     PF_LongSingleSend(MASTER, PF_OPT_COLLECT_MSGTAG);
04269
04270 #ifdef DEBUG
04271     MesPrint ("*** [%s, w=%w] DONE: PF_optimize_expression_given_Horner_slave",
      thetime_str().c_str());
04272 #endif
04273
04274 }
04275
04276 #endif
04277 /*
04278     #] PF_optimize_expression_given_Horner :
04279     #[ generate_output :
04280 */
04281
04291 void generate_output (const vector<WORD> &instr, int exprnr, int extraoffset, const
      vector<vector<WORD> > &brackets) {
04292
04293 #ifdef DEBUG
04294     MesPrint ("*** [%s, w=%w] CALL: generate_output", thetime_str().c_str());
04295 #endif
04296
04297     GETIDENTITY;
04298     vector<WORD> output;
04299
04300     // one-indexed instead of zero-indexed
04301     extraoffset++;
04302     int num = 0;
04303     int maxsize = (int)instr.size();
04304     for (int i=0; i<maxsize; i+=instr[i+2]) num++;
04305     int *tstart = (int *)Malloc1(num*sizeof(int),"nplaces");
04306     num = 0;
04307     for (int i=0; i<maxsize; i+=instr[i+2]) tstart[num++] = i;
04308     for (int j=0; j<num; j++) {
04309         int i = tstart[j];
04310
04311         // copy arguments
04312         WCOPY(AT.WorkPointer, &instr[i+3], (instr[i+2]-3));
04313
04314         // update maximal variable number
04315         AO.OptimizeResult.maxvar = MaX(AO.OptimizeResult.maxvar, instr[i]+extraoffset);
04316
04317         // renumber symbols and extrasymbols to correct values
04318         for (WORD *t=AT.WorkPointer; *t!=0; t+=*t) {
04319             if (*t == ABS(*(t+*t-1))+1) continue;
04320             if (*(t+1)==SYMBOL) *(t+3) = optimize_best_vars[*(t+3)];
04321             if (*(t+1)==EXTRASYMBOL) {
04322                 *(t+1) = SYMBOL;
04323                 *(t+3) = MAXVARIABLES-*(t+3)-extraoffset;
04324             }
04325         }
04326
04327         // reformat multiplication instructions, since their current
04328         // format is "expr.nr OPER_MUL length arguments", which differs
04329         // from Form's format for a product of symbols
04330         if (instr[i+1]==OPER_MUL) {
04331
04332             WORD *now=AT.WorkPointer+1;
04333             int dt;
04334             bool coeff=false;
04335             int coeff_sign=1;
04336
04337             for (WORD *t=AT.WorkPointer; *t!=0; t+=dt) {
04338                 dt = *t;
04339                 if (*t == ABS(*(t+*t-1))+1) {
04340                     // copy coefficient
04341                     memmove(AT.WorkPointer+instr[i+2], t, *t*sizeof(WORD));
04342                     coeff = true;
04343                 }
04344                 else {
04345                     // move symbol
04346                     int n = *(t+2);
04347                     memmove(now, t+1, n*sizeof(WORD));
04348                     now += n;
04349                     coeff_sign *= SGN(*(t+dt-1));
04350                 }
04351             }
04352
04353             if (coeff) {
04354                 // add existing coefficient
04355                 int n = *(AT.WorkPointer + instr[i+2]) - 1;
04356                 memmove(now, AT.WorkPointer+instr[i+2]+1, n*sizeof(WORD));
04357                 now += n;
04358             }
04359             else {
04360                 // add coefficient of one
04361                 *now++=1;
```

```
04362                 *now++=1;
04363                 *now++=3;
04364             }
04365 
04366             *(now-1) *= coeff_sign;
04367 
04368             *AT.WorkPointer = now - AT.WorkPointer;
04369             *now++ = 0;
04370         }
04371 
04372         // in the case of simultaneous optimization of expressions, add the
04373         // brackets to the final expression
04374         if (instr[i+1]==OPER_COMMA) {
04375             WORD *start = AT.WorkPointer + instr[i+2];
04376             WORD *now = start;
04377             int b=0;
04378             for (const WORD *t=AT.WorkPointer; *t!=0; t+=*t) {
04379                 if ( ( brackets[b].size() != 0 ) && ( brackets[b][0] == 0 ) ) break;
04380                 *now++ = *t + brackets[b].size();
04381                 if (brackets[b].size() != 0) {
04382                     memcpy(now, &brackets[b][0], brackets[b].size()*sizeof(WORD));
04383                     now += brackets[b].size();
04384                 }
04385                 memcpy(now, t+1, (*t-1)*sizeof(WORD));
04386                 now += *t-1;
04387                 b++;
04388             }
04389             *now++ = 0;
04390             memmove(AT.WorkPointer, start, (now-start)*sizeof(WORD));
04391         }
04392 
04393         // add the number of the extra symbol; if it is the last one,
04394         // replace the extra symbol number with the expression number
04395         if (i+instr[i+2]<(int)instr.size())
04396             output.push_back(instr[i]+extraoffset);
04397         else {
04398             output.push_back(-(exprnr+1));
04399         }
04400 
04401         // add code for this symbol
04402         int n=0;
04403         while (*(AT.WorkPointer+n)!=0)
04404             n += *(AT.WorkPointer+n);
04405         n++;
04406         output.insert(output.end(), AT.WorkPointer, AT.WorkPointer+n);
04407     }
04408 
04409     // trailing zero
04410     output.push_back(0);
04411 
04412     // clear buffer
04413     if (AO.OptimizeResult.code != NULL)
04414         M_free(AO.OptimizeResult.code, "optimize output");
04415 
04416     M_free(tstart,"nplaces");
04417 
04418     // copy buffer
04419     AO.OptimizeResult.codesize = output.size();
04420     AO.OptimizeResult.code = (WORD *)Malloc1(output.size()*sizeof(WORD), "optimize output");
04421     memcpy(AO.OptimizeResult.code, &output[0], output.size()*sizeof(WORD));
04422 
04423 #ifdef DEBUG
04424     MesPrint ("*** [%s, w=%w] DONE: generate_output", thetime_str().c_str());
04425 #endif
04426 }
04427 
04428 /*
04429     #] generate_output :
04430     #[ generate_expression :
04431 */
04432 
04440 WORD generate_expression (WORD exprnr) {
04441 
04442 #ifdef DEBUG
04443     MesPrint ("*** [%s, w=%w] CALL: generate_expression", thetime_str().c_str());
04444 #endif
04445 
04446     GETIDENTITY;
04447 
04448     WORD *oldWorkPointer = AT.WorkPointer;
04449 
04450     CBUF *C = cbuf+AC.cbufnum;
04451     WORD *term = AT.WorkPointer, oldcurexpr = AR.CurExpr;
04452     POSITION position;
04453     EXPRESSIONS e = Expressions+exprnr;
04454     SetScratch(AR.infile,&(e->onfile));
04455 
```

```
04456     if ( GetTerm(BHEAD term) <= 0 ) {
04457 /* INTERNAL_ERROR_EXCL_START */
04458         MesPrint("!>Expression %d has problems in scratchfile",exprnr);
04459         Terminate(-1);
04460 /* INTERNAL_ERROR_EXCL_STOP */
04461     }
04462     SeekScratch(AR.outfile,&position);
04463     e->onfile = position;
04464     if ( PutOut(BHEAD term,&position,AR.outfile,0) < 0 ) {
04465 /* INTERNAL_ERROR_EXCL_START */
04466         MesPrint("!>Expression %d has problems in output scratchfile",exprnr);
04467         Terminate(-1);
04468 /* INTERNAL_ERROR_EXCL_STOP */
04469     }
04470
04471     AR.CurExpr = exprnr;
04472     NewSort(BHEAD0);
04473
04474     // scan for the original expression (marked by *t<0) and give the
04475     // terms to Generator
04476     WORD *t = AO.OptimizeResult.code;
04477     {
04478         WORD old = AR.Cnumlhs; AR.Cnumlhs = 0;
04479         // We can use the remaining part of the WorkSpace for every term that
04480         // goes through Generator. We have WorkSpace problems here if we use
04481         // the (modified) AT.WorkPointer every time in the loop.
04482         WORD* currentWorkPointer = AT.WorkPointer;
04483         while (*t!=0) {
04484             bool is_expr = *t < 0;
04485             t++;
04486             while (*t!=0) {
04487                 if (is_expr) {
04488                     memcpy(currentWorkPointer, t, *t*sizeof(WORD));
04489                     Generator(BHEAD currentWorkPointer, C->numlhs);
04490                 }
04491                 t+=*t;
04492             }
04493             t++;
04494         }
04495         AR.Cnumlhs = old;
04496     }
04497
04498     // final sorting
04499     if (EndSort(BHEAD NULL,0) < 0) {
04500         LowerSortLevel();
04501         Terminate(-1);
04502     }
04503
04504     AT.WorkPointer = oldWorkPointer;
04505     AR.CurExpr = oldcurexpr;
04506
04507 #ifdef DEBUG
04508     MesPrint ("*** [%s, w=%w] DONE: generate_expression", thetime_str().c_str());
04509 #endif
04510
04511     return 0;
04512 }
04513
04514 /*
04515     #] generate_expression :
04516     #[ optimize_print_code :
04517 */
04518
04528 void optimize_print_code (int print_expr) {
04529
04530 #ifdef DEBUG
04531     MesPrint ("*** [%s, w=%w] CALL: optimize_print_code", thetime_str().c_str());
04532 #endif
04533     if ( ( AO.Optimize.debugflags & 1 ) != 0 ) {
04534 /*
04535  *      The next code is for debugging purposes. We may want the statements
04536  *      in reverse order to substitute them all back.
04537  *      Jan used a Mathematica program to do this. Here we make that
04538  *          Format Ox,debugflag=1;
04539  *      Creates reverse order during printing.
04540  *      All we have to do is put id in front of the statements. This is done
04541  *      in PrintExtraSymbol.
04542  */
04543         WORD *t = AO.OptimizeResult.code;
04544         WORD num = 0;   // First we count the number of objects.
04545         while (*t!=0) {
04546             num++;
04547             t++; while (*t!=0) t+=*t; t++;
04548         }
04549         WORD **tstart = (WORD **)Malloc1(num*sizeof(WORD *),"print objects");
04550         t = AO.OptimizeResult.code; num = 0; // Now we get the addresses
04551         while (*t!=0) {
```

```
04552             tstart[num++] = t;
04553             t++; while (*t!=0) t+=*t; t++;
04554         }
04555         // Flip the addresses
04556         int halfnum = num/2;
04557         for (int i=0; i<halfnum; i++) { swap(tstart[i],tstart[num-1-i]); }
04558         for ( int i = 0; i < num; i++ ) {
04559             t = tstart[i];
04560             if (*t > 0)
04561                 PrintExtraSymbol(*t, t+1, EXTRASYMBOL);
04562             else if (print_expr)
04563                 PrintExtraSymbol(-*t-1, t+1, EXPRESSIONNUMBER);
04564         }
04565         CBUF *C = cbuf + AM.sbufnum;
04566         if (C->numrhs >= AO.OptimizeResult.minvar)
04567             PrintSubtermList(AO.OptimizeResult.minvar, C->numrhs);
04568     }
04569     else {
04570         // print extra symbols from ConvertToPoly in optimization
04571         CBUF *C = cbuf + AM.sbufnum;
04572         if (C->numrhs >= AO.OptimizeResult.minvar)
04573             PrintSubtermList(AO.OptimizeResult.minvar, C->numrhs);
04574 
04575         WORD *t = AO.OptimizeResult.code;
04576         while (*t!=0) {
04577             if (*t > 0) {
04578                 PrintExtraSymbol(*t, t+1, EXTRASYMBOL);
04579             }
04580             else if (print_expr)
04581                 PrintExtraSymbol(-*t-1, t+1, EXPRESSIONNUMBER);
04582             t++;
04583             while (*t!=0) t+=*t;
04584             t++;
04585         }
04586     }
04587 
04588 #ifdef DEBUG
04589     MesPrint ("*** [%s, w=%w] DONE: optimize_print_code", thetime_str().c_str());
04590 #endif
04591 }
04592 
04593 /*
04594     #] optimize_print_code :
04595     #[ Optimize :
04596 */
04597 
04641 int Optimize (WORD exprnr, int do_print) {
04642 
04643 #if defined(WITHMPI) && (defined(DEBUG) || defined(DEBUG_MORE) || defined(DEBUG_MCTS) ||
      defined(DEBUG_GREEDY))
04644     // set AS.printflag negative temporary.
04645     struct save_printflag {
04646         save_printflag() {
04647             oldprintflag = AS.printflag;
04648             AS.printflag = -1;
04649         }
04650         ~save_printflag() {
04651             AS.printflag = oldprintflag;
04652         }
04653         int oldprintflag;
04654     } save_printflag_;
04655 #endif
04656 
04657 #ifdef DEBUG
04658     MesPrint ("*** [%s, w=%w] CALL: Optimize", thetime_str().c_str());
04659     MesPrint ("*** %"); PrintRunningTime();
04660 #endif
04661 
04662 #ifdef WITHPTHREADS
04663     optimize_lock = dummylock;
04664 #endif
04665 
04666     AO.OptimizeResult.minvar = (cbuf + AM.sbufnum)->numrhs + 1;
04667 
04668     if (get_expression(exprnr) < 0) return -1;
04669     vector<vector<WORD> > brackets = get_brackets();
04670 
04671 #ifdef DEBUG
04672 #ifdef WITHMPI
04673         if (PF.me == MASTER)
04674 #endif
04675         MesPrint ("*** runtime after preparing the expression: %"); PrintRunningTime();
04676 #endif
04677 
04678     if (optimize_expr[0]==0 ||
04679             (optimize_expr[optimize_expr[0]]==0 &&
      optimize_expr[0]==ABS(optimize_expr[optimize_expr[0]-1])+1) ||
```

```
04680             (optimize_expr[optimize_expr[0]]==0 && optimize_expr[0]==8 &&
04681              optimize_expr[5]==1 && optimize_expr[6]==1 && ABS(optimize_expr[7])==3)) {
04682         if (AO.OptimizeResult.code != NULL)
04683             M_free(AO.OptimizeResult.code, "optimize output");
04684 
04685         // zero terms or one trivial term (number or +/-variable), so no
04686         // optimization, so copy expression; special case because without
04687         // operators the optimization crashes
04688         AO.OptimizeResult.code = (WORD *)Malloc1((optimize_expr[0]+3)*sizeof(WORD), "optimize
      output");
04689         AO.OptimizeResult.code[0] = -(exprnr+1);
04690         memcpy(AO.OptimizeResult.code+1, optimize_expr, (optimize_expr[0]+1)*sizeof(WORD));
04691         AO.OptimizeResult.code[optimize_expr[0]+2] = 0;
04692     }
04693     else {
04694         // find Horner scheme(s)
04695         optimize_best_Horner_schemes.clear();
04696         if (AO.Optimize.horner == O_OCCURRENCE) {
04697             if (AO.Optimize.hornerdirection != O_BACKWARD)
04698                 optimize_best_Horner_schemes.push_back(occurrence_order(optimize_expr, false));
04699             if (AO.Optimize.hornerdirection != O_FORWARD)
04700                 optimize_best_Horner_schemes.push_back(occurrence_order(optimize_expr, true));
04701         }
04702         else {
04703             if (AO.Optimize.horner == O_SIMULATED_ANNEALING) {
04704                 optimize_best_Horner_schemes.push_back(simulated_annealing());
04705             }
04706             else {
04707                 mcts_best_schemes.clear();
04708                 if ( AO.inscheme ) {
04709                     optimize_best_Horner_schemes.push_back(vector<WORD>(AO.schemenum));
04710                     for ( int i=0; i < AO.schemenum; i++ )
04711                         optimize_best_Horner_schemes[0][i] = AO.inscheme[i];
04712                 }
04713                 else {
04714                     for ( int i = 0; i < AO.Optimize.mctsnumrepeat; i++ )
04715                         find_Horner_MCTS();
04716                     // generate results
04717                     for (set<pair<int,vector<WORD> > >::iterator i=mcts_best_schemes.begin();
      i!=mcts_best_schemes.end(); i++) {
04718                         optimize_best_Horner_schemes.push_back(i->second);
04719 #ifdef DEBUG_MCTS
04720                         MesPrint ("{%a} -> %d",i->second.size(), &i->second[0], i->first);
04721 #endif
04722                     }
04723                 }
04724                 // clear the tree by making a new empty one.
04725                 mcts_root = tree_node();
04726             }
04727         }
04728 #ifdef DEBUG
04729 #ifdef WITHMPI
04730         if (PF.me == MASTER)
04731 #endif
04732         MesPrint ("*** runtime after Horner: %"); PrintRunningTime();
04733 #endif
04734 
04735 #ifdef WITHMPI
04736         if (PF.me == MASTER ) {
04737             PF_optimize_expression_given_Horner_master_init();
04738 #endif
04739 
04740         // find best Horner scheme and results
04741         optimize_best_num_oper = INT_MAX;
04742 
04743         int imax = (int)optimize_best_Horner_schemes.size();
04744 
04745         for (int i=0; i<imax; i++) {
04746 #if defined(WITHPTHREADS)
04747             if (AM.totalnumberofthreads > 1)
04748                 optimize_expression_given_Horner_threaded();
04749             else
04750 #elif defined(WITHMPI)
04751             if (PF.numtasks > 1)
04752                 PF_optimize_expression_given_Horner_master();
04753             else
04754 #endif
04755                 optimize_expression_given_Horner();
04756         }
04757 
04758 #ifdef WITHMPI
04759             PF_optimize_expression_given_Horner_master_wait();
04760         }
04761         else {
04762             if (PF.numtasks > 1)
04763                 PF_optimize_expression_given_Horner_slave();
04764         }
```

```
04765 #endif
04766
04767 #ifdef WITHPTHREADS
04768         MasterWaitAll();
04769 #endif
04770         // format results, then print it (for "Print") or modify
04771         // expression (for "#Optimize")
04772 #ifdef WITHMPI
04773         if (PF.me == MASTER)
04774 #endif
04775         generate_output(optimize_best_instr, exprnr, cbuf[AM.sbufnum].numrhs, brackets);
04776 #ifdef WITHMPI
04777         else {
04778             // non-null dummy code for slaves
04779             if (AO.OptimizeResult.code == NULL) {
04780                 AO.OptimizeResult.code = (WORD *)Malloc1(sizeof(WORD), "optimize output");
04781             }
04782         }
04783 #endif
04784     }
04785
04786 #ifdef WITHMPI
04787     if (PF.me == MASTER) {
04788         PF_PreparePack();
04789         PF_Pack(&AO.OptimizeResult.minvar, 1, PF_WORD);
04790         PF_Pack(&AO.OptimizeResult.maxvar, 1, PF_WORD);
04791     }
04792     PF_Broadcast();
04793     if (PF.me != MASTER) {
04794         PF_Unpack(&AO.OptimizeResult.minvar, 1, PF_WORD);
04795         PF_Unpack(&AO.OptimizeResult.maxvar, 1, PF_WORD);
04796     }
04797 #endif
04798
04799     // set preprocessor variables
04800     char str[100];
04801     snprintf (str,100,"%d",AO.OptimizeResult.minvar);
04802     PutPreVar((UBYTE *)"optimminvar_",(UBYTE *)str,0,1);
04803     snprintf (str,100,"%d",AO.OptimizeResult.maxvar);
04804     PutPreVar((UBYTE *)"optimmaxvar_",(UBYTE *)str,0,1);
04805
04806     if (do_print) {
04807 #ifdef WITHMPI
04808         if (PF.me == MASTER)
04809 #endif
04810         optimize_print_code(1);
04811         ClearOptimize();
04812     }
04813     else {
04814 #ifdef WITHMPI
04815         if (PF.me == MASTER)
04816 #endif
04817         generate_expression(exprnr);
04818     }
04819
04820 #ifdef WITHMPI
04821     if (PF.me == MASTER) {
04822 #endif
04823
04824     if ( AO.Optimize.printstats > 0 ) {
04825         char str[20];
04826         MesPrint("");
04827         count_operators(optimize_expr,true);
04828         int numop = count_operators(optimize_best_instr,true);
04829         snprintf(str,20,"%d",numop);
04830         PutPreVar((UBYTE *)"optimvalue_",(UBYTE *)str,0,1);
04831     }
04832     else {
04833         char str[20];
04834         int numop = count_operators(optimize_best_instr,false);
04835         snprintf(str,20,"%d",numop);
04836         PutPreVar((UBYTE *)"optimvalue_",(UBYTE *)str,0,1);
04837     }
04838
04839     if ( ( AO.Optimize.schemeflags&1 ) == 1 ) {
04840         GETIDENTITY
04841         UBYTE *OutScr, *Out, *old1 = AO.OutputLine, *old2 = AO.OutFill;
04842         int i, sym;
04843         AO.OutputLine = AO.OutFill = (UBYTE *)AT.WorkPointer;
04844         FiniLine();
04845         OutScr = (UBYTE *)AT.WorkPointer + ( TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer) ) /2;
04846         TokenToLine((UBYTE *)" Scheme selected: ");
04847         for ( i = 0; i < optimize_num_vars; i++ ) {
04848             Out = OutScr;
04849             sym = optimize_best_vars[i];
04850             if ( i > 0 ) TokenToLine((UBYTE *)",");
04851             if ( sym < NumSymbols ) {
```

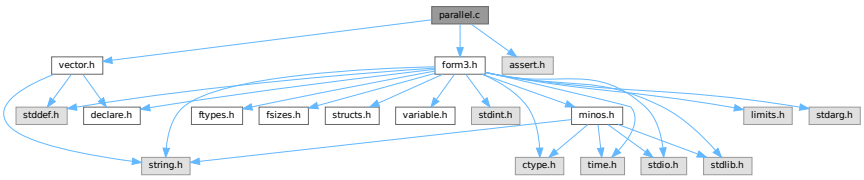
```
04852                 StrCopy(FindSymbol(sym),OutScr);
04853 /*              StrCopy(VARNAME(symbols,sym),OutScr); */
04854             }
04855             else {
04856                 Out = StrCopy((UBYTE *)AC.extrasym,Out);
04857                 if ( AC.extrasymbols == 0 ) {
04858                     Out = NumCopy((MAXVARIABLES-sym),Out);
04859                     Out = StrCopy((UBYTE *)"_",Out);
04860                 }
04861                 else if ( AC.extrasymbols == 1 ) {
04862                     Out = AddArrayIndex((MAXVARIABLES-sym),Out);
04863                 }
04864             }
04865             TokenToLine(OutScr);
04866         }
04867         TokenToLine((UBYTE *)";");
04868         FiniLine();
04869         AO.OutFill = old2;
04870         AO.OutputLine = old1;
04871     }
04872
04873     {
04874         GETIDENTITY
04875         UBYTE *OutScr, *Out, *outstring = 0;
04876         int i, sym;
04877         AO.OutputLine = AO.OutFill = (UBYTE *)AT.WorkPointer;
04878         OutScr = (UBYTE *)AT.WorkPointer + ( TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer) ) /2;
04879         for ( i = 0; i < optimize_num_vars; i++ ) {
04880             Out = OutScr;
04881             sym = optimize_best_vars[i];
04882             if ( sym < NumSymbols ) {
04883                 StrCopy(FindSymbol(sym),OutScr);
04884 /*              StrCopy(VARNAME(symbols,sym),OutScr); */
04885             }
04886             else {
04887                 Out = StrCopy((UBYTE *)AC.extrasym,Out);
04888                 if ( AC.extrasymbols == 0 ) {
04889                     Out = NumCopy((MAXVARIABLES-sym),Out);
04890                     Out = StrCopy((UBYTE *)"_",Out);
04891                 }
04892                 else if ( AC.extrasymbols == 1 ) {
04893                     Out = AddArrayIndex((MAXVARIABLES-sym),Out);
04894                 }
04895             }
04896             outstring = AddToString(outstring,OutScr,1);
04897         }
04898         if ( outstring == 0 ) {
04899             PutPreVar((UBYTE *)"optimscheme_",(UBYTE *)"",0,1);
04900         }
04901         else {
04902             PutPreVar((UBYTE *)"optimscheme_",(UBYTE *)outstring,0,1);
04903             M_free(outstring,"AddToString");
04904         }
04905     }
04906
04907 #ifdef WITHMPI
04908     }
04909
04910     // synchronize optimvalue_ and optimscheme_
04911     if ( PF.me == MASTER ) {
04912         UBYTE *value;
04913         int bytes;
04914
04915         PF_PrepareLongMultiPack();
04916
04917         value = GetPreVar((UBYTE *)"optimvalue_", WITHERROR);
04918         bytes = strlen((char *)value);
04919         PF_LongMultiPack(&bytes, 1, PF_INT);
04920         PF_LongMultiPack(value, bytes, PF_BYTE);
04921
04922         value = GetPreVar((UBYTE *)"optimscheme_", WITHERROR);
04923         bytes = strlen((char *)value);
04924         PF_LongMultiPack(&bytes, 1, PF_INT);
04925         PF_LongMultiPack(value, bytes, PF_BYTE);
04926     }
04927     PF_LongMultiBroadcast();
04928     if ( PF.me != MASTER ) {
04929         static vector<UBYTE> prevarbuf;
04930         UBYTE *value;
04931         int bytes;
04932
04933         PF_LongMultiUnpack(&bytes, 1, PF_INT);
04934         prevarbuf.reserve(bytes + 1);
04935         value = &prevarbuf[0];
04936         PF_LongMultiUnpack(value, bytes, PF_BYTE);
04937         value[bytes] = '\0';  // null terminator
04938         PutPreVar((UBYTE *)"optimvalue_", value, NULL, 1);
```

```
04939
04940         PF_LongMultiUnpack(&bytes, 1, PF_INT);
04941         prevarbuf.reserve(bytes + 1);
04942         value = &prevarbuf[0];
04943         PF_LongMultiUnpack(value, bytes, PF_BYTE);
04944         value[bytes] = '\0';  // null terminator
04945         PutPreVar((UBYTE *)"optimscheme_", value, NULL, 1);
04946     }
04947 #endif
04948
04949     // cleanup
04950     M_free(optimize_expr,"LoadOptim");
04951
04952 #ifdef DEBUG
04953     MesPrint ("*** [%s, w=%w] DONE: Optimize", thetime_str().c_str());
04954 #endif
04955
04956     return 0;
04957 }
04958
04959 /*
04960     #] Optimize :
04961     #[ ClearOptimize :
04962 */
04963
04978 int ClearOptimize()
04979 {
04980     char str[20];
04981     WORD numexpr, *w;
04982     int error = 0;
04983     if ( AO.OptimizeResult.code != NULL ) {
04984         M_free(AO.OptimizeResult.code, "optimize output");
04985         AO.OptimizeResult.code = NULL;
04986         AO.OptimizeResult.codesize = 0;
04987         PutPreVar((UBYTE *)"optimminvar_",(UBYTE *)("0"),0,1);
04988         PutPreVar((UBYTE *)"optimmaxvar_",(UBYTE *)("0"),0,1);
04989         PruneExtraSymbols(AO.OptimizeResult.minvar-1);
04990         cbuf[AM.sbufnum].numrhs = AO.OptimizeResult.minvar-1;
04991         AO.OptimizeResult.minvar = AO.OptimizeResult.maxvar = 0;
04992     }
04993     if ( AO.OptimizeResult.nameofexpr != NULL ) {
04994 /*
04995         We have to pick up the expression by its name. Numbers may change.
04996         Note that this requires that when we overwrite an expression, we
04997         check that it is not an optimized expression. See execute.c and
04998         comexpr.c
04999 */
05000         if ( GetName(AC.exprnames,AO.OptimizeResult.nameofexpr,&numexpr,NOAUTO) != CEXPRESSION ) {
05001 /* INTERNAL_ERROR_EXCL_START */
05002             MesPrint("!>Internal error while clearing optimized expression %s
    ",AO.OptimizeResult.nameofexpr);
05003             Terminate(-1);
05004 /* INTERNAL_ERROR_EXCL_STOP */
05005         }
05006         M_free(AO.OptimizeResult.nameofexpr, "optimize expression name");
05007         AO.OptimizeResult.nameofexpr = NULL;
05008         w = &(Expressions[numexpr].status);
05009         *w = SetExprCases(DROP,1,*w);
05010         if ( *w < 0 ) error = 1;
05011     }
05012     snprintf (str,20,"%d",cbuf[AM.sbufnum].numrhs);
05013     PutPreVar(AM.oldnumextrasymbols,(UBYTE *)str,0,1);
05014     PutPreVar((UBYTE *)"optimvalue_",(UBYTE *)("0"),0,1);
05015     PutPreVar((UBYTE *)"optimscheme_",(UBYTE *)("0"),0,1);
05016     return(error);
05017 }
05018
05019 /*
05020     #] ClearOptimize :
05021 */
```

4.97 parallel.c File Reference

```
#include "form3.h"
#include "vector.h"
#include <assert.h>
```

Include dependency graph for parallel.c:



Data Structures

- struct NoDe
- struct dollar_buf
- struct bufIPstruct

Macros

- #define PRINTFBUF(TEXT, TERM, SIZE) {}
- #define SWAP(x, y)
- #define PACK_LONG(p, n)
- #define UNPACK_LONG(p, n)
- #define CHECK(condition) _CHECK(condition, __FILE__, __LINE__)
- #define _CHECK(condition, file, line) __CHECK(condition, file, line)
- #define __CHECK(condition, file, line)
- #define DBGOUT(lv1, lv2, a) do { if (lv1 >= lv2) { printf a; fflush(stdout); } } while (0)
- #define DBGOUT_NINTERMS(lv, a)
- #define PF_STATS_SIZE 5
- #define PF_SNDFILEBUFSIZE 4096
- #define recvBuffer logBuffer /* (master) The buffer for receiving messages. */

Typedefs

- typedef struct NoDe NODE
- typedef struct bufIPstruct **bufIPstruct_t**

Functions

- LONG PF_RealTime (int)
- int PF_LibInit (int *, char ***)
- int PF_LibTerminate (int)
- int PF_Probe (int *)
- int PF_RecvWbuf (WORD *, LONG *, int *)
- int PF_IRecvRbuf (PF_BUFFER *, int, int)
- int PF_WaitRbuf (PF_BUFFER *, int, LONG *)
- int PF_RawSend (int dest, void *buf, LONG l, int tag)
- LONG PF_RawRecv (int *src, void *buf, LONG thesize, int *tag)
- int PF_RawProbe (int *src, int *tag, int *bytesize)
- int PF_RawIsend (int dest, const void *buf, int count, MPI_Datatype type, int tag, MPI_Request *request)
- int PF_RawWaitAll (int count, MPI_Request *request, MPI_Status *status)

- int PF_Discard (int *src, int *tag)
- int PF_EndSort (void)
- WORD PF_Deferred (WORD *term, WORD level)
- int PF_Processor (EXPRESSIONS e, WORD i, WORD LastExpression)
- int PF_Init (int *argc, char ***argv)
- void PF_PreTerminate (int errorcode)
- int PF_Terminate (int errorcode)
- LONG PF_GetSlaveTimes (void)
- LONG PF_BroadcastNumber (LONG x)
- void PF_BroadcastBuffer (WORD **buffer, LONG *length)
- int PF_BroadcastString (UBYTE *str)
- int PF_BroadcastPreDollar (WORD **dbuffer, LONG *newsize, int *numterms)
- int PF_CollectModifiedDollars (void)
- int PF_BroadcastModifiedDollars (void)
- int PF_BroadcastRedefinedPreVars (void)
- int PF_StoreInsideInfo (void)
- int PF_RestoreInsideInfo (void)
- int PF_BroadcastCBuf (int bufnum)
- int PF_BroadcastExpFlags (void)
- int PF_BroadcastExpr (EXPRESSIONS e, FILEHANDLE *file)
- int PF_BroadcastRHS (void)
- int PF_InParallelProcessor (void)
- int PF_SendFile (int to, FILE *fd)
- int PF_RecvFile (int from, FILE *fd)
- void PF_MLock (void)
- void PF_MUnlock (void)
- LONG PF_WriteFileToFile (int handle, UBYTE *buffer, LONG size)
- void PF_FlushStdOutBuffer (void)
- void PF_FreeErrorMessageBuffers (void)
- void PF_ReceiveRuntimeError (void)

Variables

- PARALLELVARS PF

4.97.1 Detailed Description

Message passing library independent functions of parform

This file contains functions needed for the parallel version of form3 these functions need no real link to the message passing libraries, they only need some interface dependent preprocessor definitions (check parallel.h). So there still need two different objectfiles to be compiled for mpi and pvm!

Definition in file parallel.c.

4.97.2 Macro Definition Documentation

4.97.2.1 PRINTFBUF

```
#define PRINTFBUF(
                TEXT,
                TERM,
                SIZE ) {}
```

Definition at line 124 of file parallel.c.

4.97.2.2 SWAP

```
#define SWAP(
              x,
              y )
```

Value:

```
    do { \
        char swap_tmp__[sizeof(x) == sizeof(y) ? (int)sizeof(x) : -1]; \
        memcpy(swap_tmp__, &y, sizeof(x)); \
        memcpy(&y, &x, sizeof(x)); \
        memcpy(&x, swap_tmp__, sizeof(x)); \
    } while (0)
```

Swaps the variables *x* and *y*. If sizeof(x) != sizeof(y) then a compilation error will occur. A set of memcpy calls with constant sizes is expected to be inlined by the optimisation.

Definition at line 131 of file parallel.c.

4.97.2.3 PACK_LONG

```
#define PACK_LONG(
              p,
              n )
```

Value:

```
    do { \
        *(p)++ = (UWORD)((ULONG)(n) & (ULONG)WORDMASK); \
        *(p)++ = (UWORD)(((ULONG)(n) » BITSINWORD) & (ULONG)WORDMASK); \
    } while (0)
```

Packs a LONG value *n* to a WORD buffer *p*.

Definition at line 142 of file parallel.c.

4.97.2.4 UNPACK_LONG

```
#define UNPACK_LONG(
              p,
              n )
```

Value:

```
    do { \
        (n) = (LONG)((((ULONG)(p)[1] & (ULONG)WORDMASK) « BITSINWORD) | ((ULONG)(p)[0] & (ULONG)WORDMASK));
      \
        (p) += 2; \
    } while (0)
```

Unpacks a LONG value *n* from a WORD buffer *p*.

Definition at line 151 of file parallel.c.

4.97.2.5 CHECK

```
#define CHECK(
              condition ) _CHECK(condition, __FILE__, __LINE__)
```

A simple check for unrecoverable errors.

Definition at line 160 of file parallel.c.

4.97.2.6 _CHECK

```
#define _CHECK(
               condition,
               file,
               line ) __CHECK(condition, file, line)
```

Definition at line 161 of file parallel.c.

4.97.2.7 __CHECK

```
#define __CHECK(
               condition,
               file,
               line )
```

Value:

```
    do { \
        if ( !(condition) ) { \
            Error0("Fatal error at " file ":" #line); \
            Terminate(-1); \
        } \
    } while (0)
```

Definition at line 162 of file parallel.c.

4.97.2.8 DBGOUT

```
#define DBGOUT(
               lv1,
               lv2,
               a ) do { if ( lv1 >= lv2 ) { printf a; fflush(stdout); } } while (0)
```

Definition at line 173 of file parallel.c.

4.97.2.9 DBGOUT_NINTERMS

```
#define DBGOUT_NINTERMS(
               lv,
               a )
```

Definition at line 176 of file parallel.c.

4.97.2.10 PF_STATS_SIZE

```
#define PF_STATS_SIZE 5
```

Definition at line 185 of file parallel.c.

4.97.2.11 PF_SNDFILEBUFSIZE

```
#define PF_SNDFILEBUFSIZE 4096
```

Definition at line 4225 of file parallel.c.

4.97.2.12 recvBuffer

```
#define recvBuffer logBuffer /* (master) The buffer for receiving messages.  */
```

Definition at line 4333 of file parallel.c.

4.97.3 Typedef Documentation

4.97.3.1 NODE

```
typedef struct NoDe NODE
```

A node for the tree of losers in the final sorting on the master.

4.97.4 Function Documentation

4.97.4.1 PF_RealTime()

```
LONG PF_RealTime (
            int i )
```

Returns the realtime in 1/100 sec. as a LONG.

Parameters

i	the timer will be reset if i == 0.

Returns

the real elapsed time in 1/100 second.

Definition at line 106 of file mpi.c.

Referenced by PF_Init().

4.97.4.2 PF_LibInit()

```
int PF_LibInit (
            int * argcp,
            char *** argvp )
```

Performs all library dependent initializations.

Parameters

argcp	pointer to the number of arguments.
argvp	pointer to the arguments.

Returns

0 if OK, nonzero on error.

Definition at line 128 of file mpi.c.

Referenced by PF_Init().

4.97.4.3 PF_LibTerminate()

```
int PF_LibTerminate (
            int error )
```

Exits mpi, when there is an error either indicated or happening, returnvalue is 1, else returnvalue is 0.

Parameters

error	an error code.

Returns

0 if OK, nonzero on error.

Definition at line 214 of file mpi.c.

Referenced by PF_Terminate().

4.97.4.4 PF_Probe()

```
int PF_Probe (
            int * src )
```

Probes the next incoming message. If src == PF_ANY_SOURCE this operation is blocking, otherwise nonblocking.

Parameters

in,out	*src*	the source process number. In output, the process number of actual found source.

Returns

the tag value of the next incoming message if found, 0 if a nonblocking probe (input src != PF_ANY_SOURCE) did not find any messages. A negative returned value indicates an error.

Definition at line 235 of file mpi.c.

4.97.4.5 PF_RecvWbuf()

```
int PF_RecvWbuf (
            WORD * b,
            LONG * s,
            int * src )
```

Blocking receive of a WORD buffer.

Parameters

out	*b*	the buffer to store the received data.
in,out	*s*	the size of the buffer. The output value is the actual size of the received data.
in,out	*src*	the source process number. The output value is the process number of actual source.

Returns

the received message tag. A negative value indicates an error.

Definition at line 342 of file mpi.c.

References PF_ReceiveRuntimeError().

Here is the call graph for this function:

4.97.4.6 PF_IRecvRbuf()

```
int PF_IRecvRbuf (
            PF_BUFFER * r,
            int bn,
            int from )
```

Posts nonblocking receive for the active receive buffer. The buffer is filled from full to stop.

Parameters

r	the PF_BUFFER struct for the nonblocking receive.
bn	the index of the cyclic buffer.
from	the source process number.

Returns

0 if OK, nonzero on error.

Definition at line 378 of file mpi.c.

4.97.4.7 PF_WaitRbuf()

```
int PF_WaitRbuf (
            PF_BUFFER * r,
            int bn,
            LONG * size )
```

Waits for the buffer *bn* to finish a pending nonblocking receive. It returns the received tag and in `*size` the number of field received. If the receive is already finished, just return the flag and size, else wait for it to finish, but also check for other pending receives.

Parameters

	r	the PF_BUFFER struct for the pending nonblocking receive.
	bn	the index of the cyclic buffer.
out	*size*	the actual size of received data.

Returns

the received message tag. A negative value indicates an error.

Definition at line 412 of file mpi.c.

4.97.4.8 PF_RawSend()

```
int PF_RawSend (
            int dest,
            void * buf,
            LONG l,
            int tag )
```

Sends *l* bytes from *buf* to *dest*. Returns 0 on success, or -1.

Parameters

dest	the destination process number.
buf	the send buffer.
l	the size of data in the send buffer in bytes.
tag	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 497 of file mpi.c.

Referenced by PF_MUnlock(), PF_ReceiveRuntimeError(), and PF_SendFile().

4.97.4.9 PF_RawRecv()

```
LONG PF_RawRecv (
            int * src,
            void * buf,
            LONG thesize,
            int * tag )
```

Receives not more than *thesize* bytes from *src*, returns the actual number of received bytes, or -1 on failure.

Parameters

in,out	*src*	the source process number. In output, that of the actual received message.
out	*buf*	the receive buffer.
	thesize	the size of the receive buffer in bytes.
out	*tag*	the message tag of the actual received message.

Returns

the actual sizeof received data in bytes, or -1 on failure.

Definition at line 518 of file mpi.c.

Referenced by PF_ReceiveRuntimeError(), and PF_RecvFile().

4.97.4.10 PF_RawProbe()

```
int PF_RawProbe (
            int * src,
            int * tag,
            int * bytesize )
```

Probes an incoming message.

Parameters

in,out	*src*	the source process number. In output, that of the actual received message.
in,out	*tag*	the message tag. In output, that of the actual received message.
out	*bytesize*	the size of incoming data in bytes.

Returns

0 if OK, nonzero on error.

Definition at line 542 of file mpi.c.

4.97.4.11 PF_RawIsend()

```
int PF_RawIsend (
            int dest,
            const void * buf,
            int count,
            MPI_Datatype type,
            int tag,
            MPI_Request * request )
```

Performs a nonblocking send.

Parameters

	dest	the destination process number.
in	*buf*	the send buffer.
	count	the number of elements in the send buffer.
	type	the datatype of the data in the send buffer.
	tag	the message tag.
out	*request*	the request handle for the nonblocking operation.

Returns

0 if OK, nonzero on error.

Definition at line 574 of file mpi.c.

4.97.4.12 PF_RawWaitAll()

```
int PF_RawWaitAll (
            int count,
            MPI_Request * request,
            MPI_Status * status )
```

Waits for all the given requests to complete.

Parameters

	count	the number of requests.
in,out	*request*	the array of request handles.
out	*status*	the array of status objects to store the status of each completed request.

Returns

0 if OK, nonzero on error.

Definition at line 594 of file mpi.c.

4.97.4.13 PF_Discard()

```
int PF_Discard (
            int * src,
            int * tag )
```

Discards an incoming message.

Parameters

in,out	*src*	the source process number. On output, that of the actual received message.
in,out	*tag*	the message tag. On output, that of the actual received message.

Returns

0 if OK, nonzero on error.

Definition at line 615 of file mpi.c.

4.97.4.14 PF_EndSort()

```
int PF_EndSort (
            void  )
```

Finishes a master sorting with collecting terms from slaves. Called by EndSort().

If this is not the masterprocess, just initialize the sendbuffers and return 0, else PF_EndSort() sends the rest of the terms in the sendbuffer to the next slave and a dummy message to all slaves with tag PF_ENDSORT_MSGTAG. Then it receives the sorted terms, sorts them using a recursive 'tree of losers' (PF_GetLoser()) and writes them to the outputfile.

Returns

1 if the sorting on the master was done. 0 if EndSort() still must perform a regular sorting because it is not at the ground level or not on the master or in the sequential mode or in the InParallel mode. -1 if an error occurred.

Remarks

The slaves will send the sorted terms back to the master in the regular sorting (after the initialization of the send buffer in PF_EndSort()). See PutOut() and FlushOut().

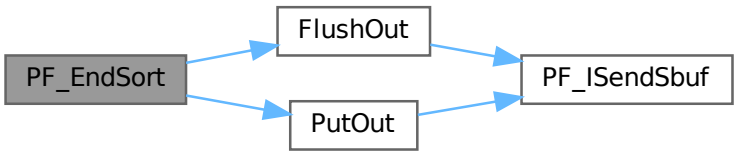
This function has been changed such that when it returns 1, AM.S0->TermsLeft is set correctly. But AM.S0->GenTerms is not set: it will be set after collecting the statistics from the slaves at the end of PF_Processor(). (TU 30 Jun 2011)

Definition at line 874 of file parallel.c.

References FlushOut(), and PutOut().

Referenced by EndSort().

Here is the call graph for this function:



4.97.4.15 PF_Deferred()

```
WORD PF_Deferred (
            WORD * term,
            WORD level )
```

Replaces Deferred() on the slaves.

Parameters

term	the term that must be multiplied by the contents of the current bracket.
level	the compiler level.

Returns

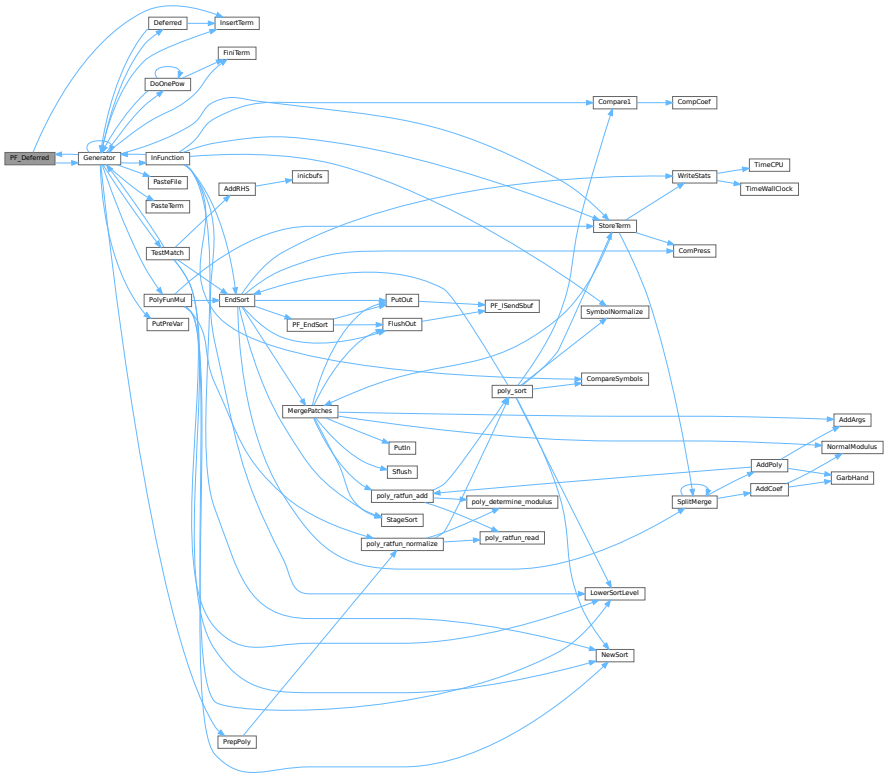
0 if OK, nonzero on error.

Definition at line 1201 of file parallel.c.

References Generator(), and InsertTerm().

Referenced by Generator().

Here is the call graph for this function:



4.97.4.16 PF_Processor()

```
int PF_Processor (
            EXPRESSIONS e,
            WORD i,
            WORD LastExpression )
```

Replaces parts of Processor() on the masters and slaves. On the master PF_Processor() is responsible for proper distribution of terms from the input file to the slaves. On the slaves it calls Generator() for all the terms that this process gets, but PF_GetTerm() gets terms from the master (not directly from infile).

Parameters

e	The pointer to the current expression.
i	The index for the current expression.
LastExpression	The flag indicating whether it is the last expression.

Returns

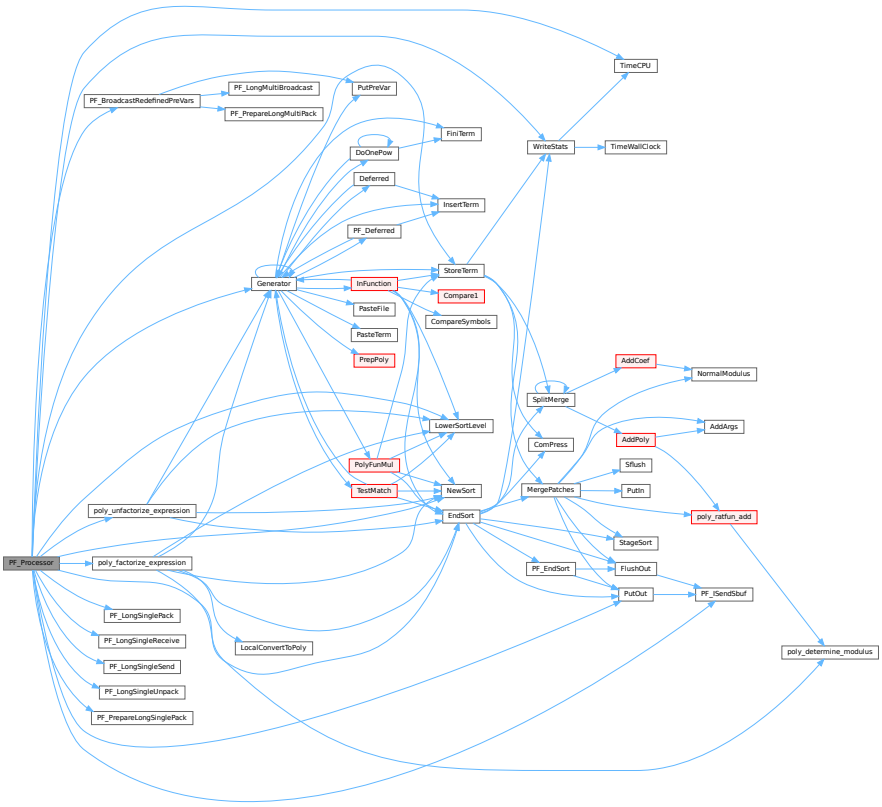
0 if OK, nonzero on error.

Definition at line 1533 of file parallel.c.

References EndSort(), Generator(), LowerSortLevel(), NewSort(), PACK_LONG, PF_BroadcastRedefinedPreVars(), PF_ISendSbuf(), PF_LongSinglePack(), PF_LongSingleReceive(), PF_LongSingleSend(), PF_LongSingleUnpack(), PF_PrepareLongSinglePack(), poly_factorize_expression(), poly_unfactorize_expression(), PutOut(), StoreTerm(), SWAP, TimeCPU(), and WriteStats().

Referenced by Processor().

Here is the call graph for this function:



4.97.4.17 PF_Init()

```
int PF_Init (
            int * argc,
            char *** argv )
```

All the library independent stuff. PF_LibInit() should do all library dependent initializations.

Parameters

argc	pointer to the number of arguments.
argv	pointer to the arguments.

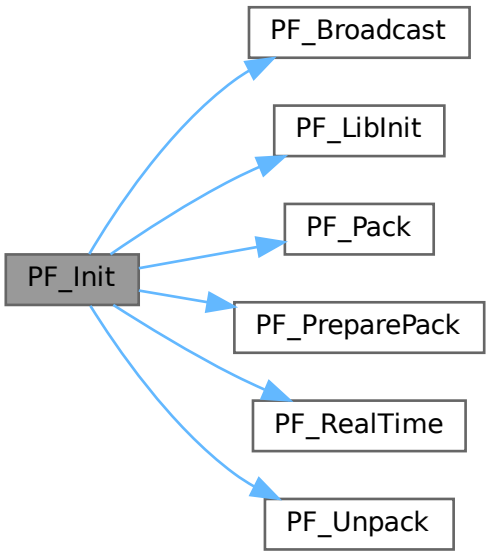
Returns

0 if OK, nonzero on error.

Definition at line 1947 of file parallel.c.

References PF_Broadcast(), PF_LibInit(), PF_Pack(), PF_PreparePack(), PF_RealTime(), and PF_Unpack().

Here is the call graph for this function:



4.97.4.18 PF_PreTerminate()

```
void PF_PreTerminate (
            int errorcode )
```

Prepares for termination. Called by Terminate().

Parameters

errorcode	an error code.

Definition at line 2041 of file parallel.c.

4.97.4.19 PF_Terminate()

```
int PF_Terminate (
            int errorcode )
```

Performs the finalization of ParFORM. To be called by Terminate().

Parameters

errorcode	an error code.

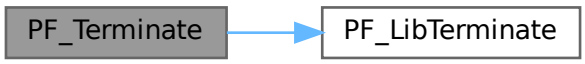
Returns

0 if OK, nonzero on error.

Definition at line 2061 of file parallel.c.

References PF_LibTerminate().

Here is the call graph for this function:



4.97.4.20 PF_GetSlaveTimes()

```
LONG PF_GetSlaveTimes (
            void  )
```

Returns the total CPU time of all slaves together. This function must be called on the master and all slaves.

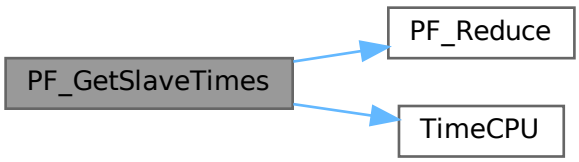
Returns

on the master, the sum of CPU times on all slaves.

Definition at line 2077 of file parallel.c.

References CHECK, PF_Reduce(), and TimeCPU().

Here is the call graph for this function:



4.97.4.21 PF_BroadcastNumber()

```
LONG PF_BroadcastNumber (
            LONG x )
```

Broadcasts a LONG value from the master to the all slaves.

Parameters

x	the number to be broadcast (set on the master).

Returns

the synchronised result.

Definition at line 2098 of file parallel.c.

References PF_Bcast().

Referenced by DoCheckpoint(), PF_BroadcastBuffer(), and StartVariables().

Here is the call graph for this function:

4.97.4.22 PF_BroadcastBuffer()

```
void PF_BroadcastBuffer (
            WORD ** buffer,
            LONG * length )
```

Broadcasts a buffer from the master to all the slaves.

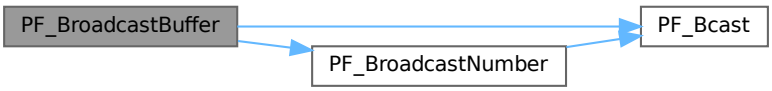
Parameters

in,out	*buffer*	on the master, the buffer to be broadcast. On the slaves, the buffer will be allocated if the length is greater than 0. The caller must free it.
in,out	*length*	on the master, the length of the buffer to be broadcast. On the slaves, it receives the length of transferred buffer. The actual transfer occurs only if the length is greater than 0.

Definition at line 2125 of file parallel.c.

References PF_Bcast(), and PF_BroadcastNumber().

Here is the call graph for this function:



4.97.4.23 PF_BroadcastString()

```
int PF_BroadcastString (
            UBYTE * str )
```

Broadcasts a string from the master to all slaves.

Parameters

in,out	*str*	The pointer to a null-terminated string.

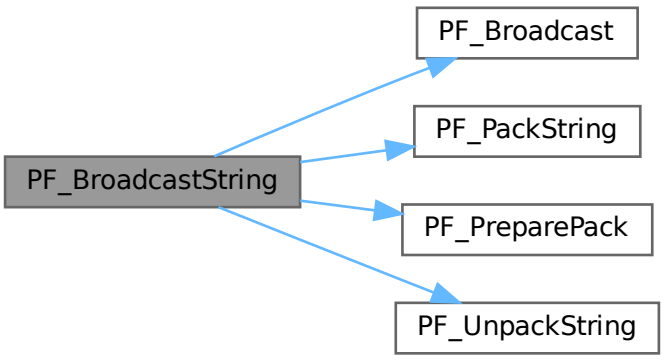
Returns

0 if OK, nonzero on error.

Definition at line 2167 of file parallel.c.

References PF_Broadcast(), PF_PackString(), PF_PreparePack(), and PF_UnpackString().

Here is the call graph for this function:



4.97.4.24 PF_BroadcastPreDollar()

```
int PF_BroadcastPreDollar (
            WORD ** dbuffer,
            LONG * newsize,
            int * numterms )
```

Broadcasts dollar variables set as a preprocessor variables. Only the master is able to make an assignment like #$a=g; where g is an expression: only the master has an access to the expression. So, the master broadcasts the result to slaves.

The result is in ∗dbuffer of the size is ∗newsize (in number of WORDs), +1 for trailing zero. For slave newsize and numterms are output parameters.

Parameters

in,out	*dbuffer*	the buffer for a dollar variable.
in,out	*newsize*	the size of the dollar variable in WORDs.
in,out	*numterms*	the number of terms in the dollar variable.

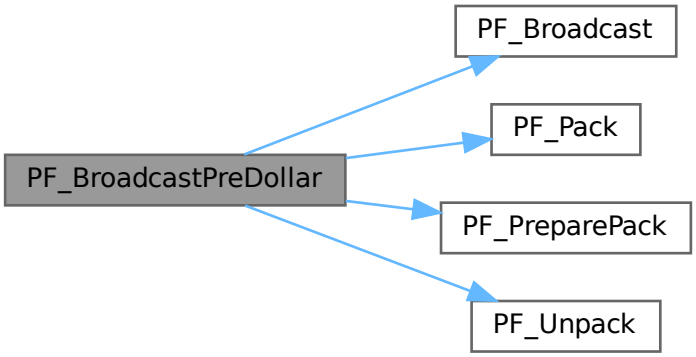
Returns

0 if OK, nonzero on error.

Definition at line 2222 of file parallel.c.

References PF_Broadcast(), PF_Pack(), PF_PreparePack(), and PF_Unpack().

Here is the call graph for this function:



4.97.4.25 PF_CollectModifiedDollars()

```
int PF_CollectModifiedDollars (
            void  )
```

Combines modified dollar variables on the all slaves, and store them into those on the master.

The potentially modified dollar variables are given in PotModdollars, and the number of them is given by NumPot↩ Moddollars.

The current module could be executed in parallel only if all potentially modified variables are listed in ModOptdollars, otherwise the module was switched to the sequential mode.

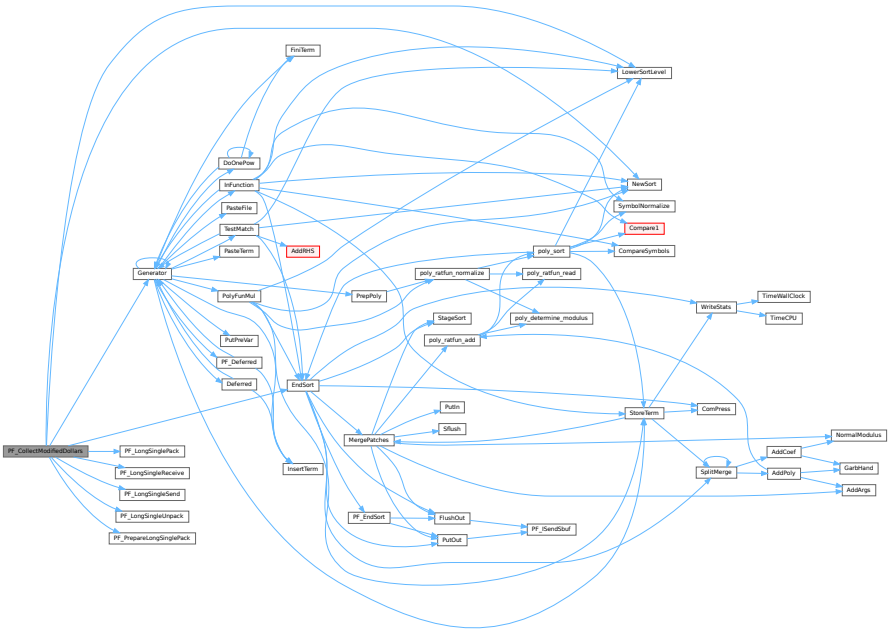
Returns

0 if OK, nonzero on error.

Definition at line 2509 of file parallel.c.

References EndSort(), Generator(), LowerSortLevel(), NewSort(), PF_LongSinglePack(), PF_LongSingleReceive(), PF_LongSingleSend(), PF_LongSingleUnpack(), PF_PrepareLongSinglePack(), VectorInit, VectorPtr, VectorReserve, and VectorSize.

Here is the call graph for this function:



4.97.4.26 PF_BroadcastModifiedDollars()

```
int PF_BroadcastModifiedDollars (
            void  )
```

Broadcasts modified dollar variables on the master to the all slaves.

The potentially modified dollar variables are given in PotModdollars, and the number of them is given by NumPot↩ Moddollars.

The current module could be executed in parallel only if all potentially modified variables are listed in ModOptdollars, otherwise the module was switched to the sequential mode. In either cases, we need to broadcast them.

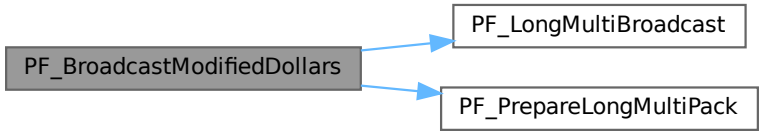
Returns

0 if OK, nonzero on error.

Definition at line 2788 of file parallel.c.

References PF_LongMultiBroadcast(), and PF_PrepareLongMultiPack().

Here is the call graph for this function:



4.97.4.27 PF_BroadcastRedefinedPreVars()

```
int PF_BroadcastRedefinedPreVars (
            void  )
```

Broadcasts preprocessor variables, which were changed by the Redefine statements in the current module, from the master to the all slaves.

The potentially redefined preprocessor variables are given in AC.pfirstnum, and the number of them is given by AC.numpfirstnum. For an actually redefined variable, the corresponding value in AC.inputnumbers is non-negative.

Returns

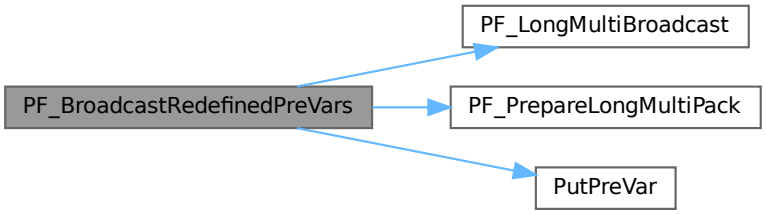
0 if OK, nonzero on error.

Definition at line 3005 of file parallel.c.

References PF_LongMultiBroadcast(), PF_PrepareLongMultiPack(), PutPreVar(), VectorPtr, and VectorReserve.

Referenced by PF_Processor().

Here is the call graph for this function:



4.97.4.28 PF_StoreInsideInfo()

```
int PF_StoreInsideInfo (
            void  )
```

Definition at line 3095 of file parallel.c.

4.97.4.29 PF_RestoreInsideInfo()

```
int PF_RestoreInsideInfo (
            void  )
```

Definition at line 3121 of file parallel.c.

4.97.4.30 PF_BroadcastCBuf()

```
int PF_BroadcastCBuf (
            int bufnum )
```

Broadcasts a compiler buffer specified by *bufnum* from the master to the all slaves.

Parameters

bufnum	The index of the compiler buffer to be broadcast.

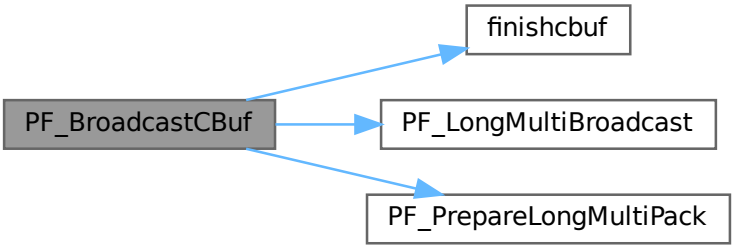
Returns

0 if OK, nonzero on error.

Definition at line 3147 of file parallel.c.

References CbUf::boomlijst, CbUf::Buffer, CbUf::BufferSize, CbUf::CanCommu, CbUf::dimension, finishcbuf(), CbUf::lhs, CbUf::numdum, CbUf::NumTerms, PF_LongMultiBroadcast(), PF_PrepareLongMultiPack(), CbUf::Pointer, CbUf::rhs, and CbUf::Top.

Here is the call graph for this function:



4.97.4.31 PF_BroadcastExpFlags()

```
int PF_BroadcastExpFlags (
            void  )
```

Broadcasts AR.expflags and several properties of each expression, e.g., e->vflags, from the master to all slaves.

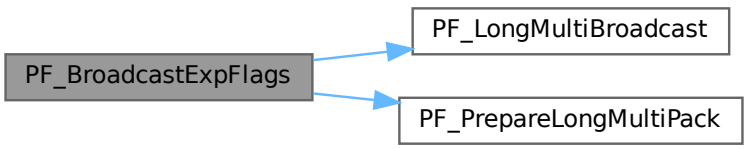
Returns

0 if OK, nonzero on error.

Definition at line 3258 of file parallel.c.

References PF_LongMultiBroadcast(), and PF_PrepareLongMultiPack().

Here is the call graph for this function:



4.97.4.32 PF_BroadcastExpr()

```
int PF_BroadcastExpr (
            EXPRESSIONS e,
            FILEHANDLE * file )
```

Broadcasts an expression from the master to the all slaves.

Parameters

e	The expression to be broadcast.
file	The file in which the expression is sitting.

Returns

0 if OK, nonzero on error.

Definition at line 3552 of file parallel.c.

Referenced by PF_BroadcastRHS().

4.97.4.33 PF_BroadcastRHS()

```
int PF_BroadcastRHS (
            void  )
```

Broadcasts expressions appearing in the right-hand side from the master to the all slaves.

Returns

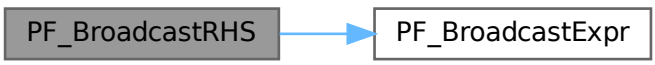
0 if OK, nonzero on error.

Definition at line 3580 of file parallel.c.

References PF_BroadcastExpr().

Referenced by Processor().

Here is the call graph for this function:



4.97.4.34 PF_InParallelProcessor()

```
int PF_InParallelProcessor (
            void  )
```

Processes expressions in the InParallel mode, i.e., dividing expressions marked by partodo over the slaves.

Returns

0 if OK, nonzero on error.

Definition at line 3627 of file parallel.c.

Referenced by Processor().

4.97.4.35 PF_SendFile()

```
int PF_SendFile (
            int to,
            FILE * fd )
```

Sends a file to the process specified by *to*.

Parameters

to	the destination process number.
fd	the file to be sent.

Returns

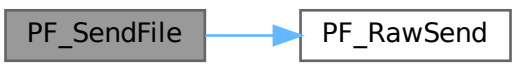
the size of sent data in bytes, or -1 on error.

Definition at line 4234 of file parallel.c.

References PF_RawSend().

Referenced by DoCheckpoint().

Here is the call graph for this function:



4.97.4.36 PF_RecvFile()

```
int PF_RecvFile (
            int from,
            FILE * fd )
```

Receives a file from the process specified by *from*.

Parameters

from	the source process number.
fd	the file to save the received data.

Returns

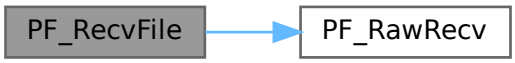
the size of received data in bytes, or -1 on error.

Definition at line 4272 of file parallel.c.

References PF_RawRecv().

Referenced by DoCheckpoint().

Here is the call graph for this function:



4.97.4.37 PF_MLock()

```
void PF_MLock (
            void  )
```

A function called by MLOCK(ErrorMessageLock) for slaves.

Definition at line 4353 of file parallel.c.

References VectorClear.

4.97.4.38 PF_MUnlock()

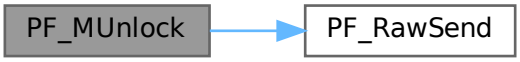
```
void PF_MUnlock (
             void  )
```

A function called by MUNLOCK(ErrorMessageLock) for slaves.

Definition at line 4369 of file parallel.c.

References PF_RawSend(), VectorEmpty, VectorPtr, and VectorSize.

Here is the call graph for this function:



4.97.4.39 PF_WriteFileToFile()

```
LONG PF_WriteFileToFile (
             int handle,
             UBYTE * buffer,
             LONG size )
```

Replaces WriteFileToFile() on the master and slaves.

It copies the given buffer into internal buffers if called between MLOCK(ErrorMessageLock) and MUNLOCK(Error↩ MessageLock) for slaves and handle is StdOut or LogHandle, otherwise calls WriteFileToFile().

Parameters

handle	a file handle that specifies the output.
buffer	a pointer to the source buffer containing the data to be written.
size	the size of data to be written in bytes.

Returns

the actual size of data written to the output in bytes.

Definition at line 4398 of file parallel.c.

References VectorClear, VectorPtr, VectorPushBacks, and VectorSize.

4.97.4.40 PF_FlushStdOutBuffer()

```
void PF_FlushStdOutBuffer (
            void  )
```

Explicitly Flushes the buffer for the standard output on the master, which is used if PF_ENABLE_STDOUT_↩ BUFFERING is defined.

Definition at line 4492 of file parallel.c.

References VectorClear, VectorPtr, and VectorSize.

4.97.4.41 PF_FreeErrorMessageBuffers()

```
void PF_FreeErrorMessageBuffers (
            void  )
```

Frees the buffers allocated for the synchronized output.

Currently, not used anywhere, but could be used in PF_Terminate().

Definition at line 4637 of file parallel.c.

References VectorFree.

4.97.4.42 PF_ReceiveRuntimeError()

```
void PF_ReceiveRuntimeError (
            void  )
```

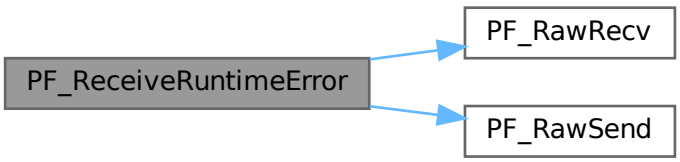
Handles a runtime error message received from another process. It ultimately calls Terminate(-1).

Definition at line 4820 of file parallel.c.

References CHECK, PF_RawRecv(), and PF_RawSend().

Referenced by PF_RecvWbuf().

Here is the call graph for this function:



4.97.5 Variable Documentation

4.97.5.1 PF

```
PARALLELVARS PF
```

Definition at line 99 of file parallel.c.

4.98 parallel.c

Go to the documentation of this file.

```
00001
00011 /* #[ License : */
00012 /*
00013  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00014  *   When using this file you are requested to refer to the publication
00015  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00016  *   This is considered a matter of courtesy as the development was paid
00017  *   for by FOM the Dutch physics granting agency and we would like to
00018  *   be able to track its scientific use to convince FOM of its value
00019  *   for the community.
00020  *
00021  *   This file is part of FORM.
00022  *
00023  *   FORM is free software: you can redistribute it and/or modify it under the
00024  *   terms of the GNU General Public License as published by the Free Software
00025  *   Foundation, either version 3 of the License, or (at your option) any later
00026  *   version.
00027  *
00028  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00029  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00030  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00031  *   details.
00032  *
00033  *   You should have received a copy of the GNU General Public License along
00034  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00035  */
00036 /* #] License : */
00037 /*
00038     #[ includes :
00039 */
00040 #include "form3.h"
00041 #include "vector.h"
00042
00043 #include <assert.h>  // must come after form3.h
00044
00045 /*
00046 #define PF_DEBUG_BCAST_LONG
00047 #define PF_DEBUG_BCAST_BUF
00048 #define PF_DEBUG_BCAST_PREDOLLAR
00049 #define PF_DEBUG_BCAST_RHSEXPR
00050 #define PF_DEBUG_BCAST_DOLLAR
00051 #define PF_DEBUG_BCAST_PREVAR
00052 #define PF_DEBUG_BCAST_CBUF
00053 #define PF_DEBUG_BCAST_EXPRFLAGS
00054 #define PF_DEBUG_REDUCE_DOLLAR
00055 */
00056
00057 /* mpi.c */
00058 LONG PF_RealTime(int);
00059 int PF_LibInit(int*, char***);
00060 int PF_LibTerminate(int);
00061 int PF_Probe(int*);
00062 int PF_RecvWbuf(WORD*,LONG*,int*);
00063 int PF_IRecvRbuf(PF_BUFFER*,int,int);
00064 int PF_WaitRbuf(PF_BUFFER *,int,LONG *);
00065 int PF_RawSend(int dest, void *buf, LONG l, int tag);
00066 LONG PF_RawRecv(int *src,void *buf,LONG thesize,int *tag);
00067 int PF_RawProbe(int *src, int *tag, int *bytesize);
00068 int PF_RawIsend(int dest, const void *buf, int count, MPI_Datatype type, int tag, MPI_Request
      *request);
00069 int PF_RawWaitAll(int count, MPI_Request *request, MPI_Status *status);
00070 int PF_Discard(int *src, int *tag);
00071
00072 /* Private functions */
00073
00074 static int PF_WaitAllSlaves(void);
```

```
00075
00076 static void PF_PackRedefinedPreVars(void);
00077 static void PF_UnpackRedefinedPreVars(void);
00078
00079 static int PF_Wait4MasterIP(int tag);
00080 static int PF_DoOneExpr(void);
00081 static int PF_ReadMaster(void);/*reads directly to its scratch!*/
00082 static int PF_Slave2MasterIP(int src);/*both master and slave*/
00083 static int PF_Master2SlaveIP(int dest, EXPRESSIONS e);
00084 static int PF_WalkThrough(WORD *t, LONG l, LONG chunk, LONG *count);
00085 static int PF_SendChunkIP(FILEHANDLE *curfile,  POSITION *position, int to, LONG thesize);
00086 static int PF_RecvChunkIP(FILEHANDLE *curfile, int from, LONG thesize);
00087
00088 static void PF_ReceiveErrorMessage(int src, int tag);
00089 static void PF_CatchErrorMessages(int *src, int *tag);
00090 static void PF_CatchErrorMessagesForAll(void);
00091 static int PF_ProbeWithCatchingErrorMessages(int *src);
00092
00093 static void PF_RaiseRuntimeError(void);
00094 static void PF_BroadcastRuntimeError(void);
00095 static void PF_PostEndSortBarrier(void);
00096
00097 /* Variables */
00098
00099 PARALLELVARS PF;
00100
00101 static int      PF_processing; /* Flag indicating that parallel processing of terms is in progress */
00102 static LONG     PF_goutterms;  /* (master) Total out terms at PF_EndSort(), used in PF_Statistics().
      */
00103 static POSITION PF_exprsize;   /* (master) The size of the expression at PF_EndSort(), used in
      PF_Processor(). */
00104
00105 /*
00106     This will work well only under Linux, see
00107         #ifdef PF_WITH_SCHED_YIELD
00108     below in PF_WaitAllSlaves().
00109 */
00110 #ifdef PF_WITH_SCHED_YIELD
00111  #include <sched.h>
00112 #endif
00113
00114 #ifdef PF_WITHLOG
00115  #define PRINTFBUF(TEXT,TERM,SIZE)  { UBYTE lbuf[24]; if(PF.log){ WORD iii;\
00116   NumToStr(lbuf,AC.CModule); \
00117   fprintf(stderr,"[%d|%s] %s : ",PF.me,lbuf,(char*)TEXT);\
00118   if(TERM){ fprintf(stderr,"[%d] ",(int)(*TERM));\
00119     if((SIZE)<500 && (SIZE)>0) for(iii=1;iii<(SIZE);iii++)\
00120       fprintf(stderr,"%d ",TERM[iii]); }\
00121   fprintf(stderr,"\n");\
00122   fflush(stderr); } }
00123 #else
00124  #define PRINTFBUF(TEXT,TERM,SIZE) {}
00125 #endif
00126
00131 #define SWAP(x, y) \
00132     do { \
00133         char swap_tmp__[sizeof(x) == sizeof(y) ? (int)sizeof(x) : -1]; \
00134         memcpy(swap_tmp__, &y, sizeof(x)); \
00135         memcpy(&y, &x, sizeof(x)); \
00136         memcpy(&x, swap_tmp__, sizeof(x)); \
00137     } while (0)
00138
00142 #define PACK_LONG(p, n) \
00143     do { \
00144         *(p)++ = (UWORD)((ULONG)(n) & (ULONG)WORDMASK); \
00145         *(p)++ = (UWORD)(((ULONG)(n) » BITSINWORD) & (ULONG)WORDMASK); \
00146     } while (0)
00147
00151 #define UNPACK_LONG(p, n) \
00152     do { \
00153         (n) = (LONG)((((ULONG)(p)[1] & (ULONG)WORDMASK) « BITSINWORD) | ((ULONG)(p)[0] &
      (ULONG)WORDMASK)); \
00154         (p) += 2; \
00155     } while (0)
00156
00160 #define CHECK(condition) _CHECK(condition, __FILE__, __LINE__)
00161 #define _CHECK(condition, file, line) __CHECK(condition, file, line)
00162 #define __CHECK(condition, file, line) \
00163     do { \
00164         if ( !(condition) ) { \
00165             Error0("Fatal error at " file ":" #line); \
00166             Terminate(-1); \
00167         } \
00168     } while (0)
00169
00170 /*
00171  * For debugging.
```

```
00172  */
00173 #define DBGOUT(lv1, lv2, a) do { if ( lv1 >= lv2 ) { printf a; fflush(stdout); } } while (0)
00174
00175 /* (AN.ninterms of master) == max(AN.ninterms of slaves) == sum(PF_linterms of slaves) at EndSort().
   */
00176 #define DBGOUT_NINTERMS(lv, a)
00177 /* #define DBGOUT_NINTERMS(lv, a) DBGOUT(1, lv, a) */
00178
00179 /*
00180     #] includes :
00181     #[ statistics :
00182         #[ variables : (should be part of a struct?)
00183 */
00184 static LONG PF_linterms;     /* local interms on this proces: PF_Proces */
00185 #define PF_STATS_SIZE 5
00186 static LONG **PF_stats = NULL;/* space for collecting statistics of all procs */
00187 static LONG PF_laststat;     /* last realtime when statistics were printed */
00188 static LONG PF_statsinterval;/* timeinterval for printing statistics */
00189 /*
00190         #] variables :
00191         #[ PF_Statistics :
00192 */
00193
00202 static int PF_Statistics(LONG **stats, int proc)
00203 {
00204     GETIDENTITY
00205     LONG real, cpu;
00206     WORD rpart, cpart;
00207     int i, j;
00208
00209     if ( AT.SS == AM.S0 && PF.me == MASTER ) {
00210         real = PF_RealTime(PF_TIME); rpart = (WORD)(real%100); real /= 100;
00211
00212         if ( PF_stats == NULL ) {
00213             PF_stats = (LONG**)Malloc1(PF.numtasks*sizeof(LONG*),"PF_stats 1");
00214             for ( i = 0; i < PF.numtasks; i++ ) {
00215                 PF_stats[i] = (LONG*)Malloc1(PF_STATS_SIZE*sizeof(LONG),"PF_stats 2");
00216                 for ( j = 0; j < PF_STATS_SIZE; j++ ) PF_stats[i][j] = 0;
00217             }
00218         }
00219         if ( proc > 0 ) for ( i = 0; i < PF_STATS_SIZE; i++ ) PF_stats[proc][i] = stats[0][i];
00220
00221         if ( real >= PF_laststat + PF_statsinterval || proc == 0 ) {
00222             LONG sum[PF_STATS_SIZE];
00223
00224             for ( i = 0; i < PF_STATS_SIZE; i++ ) sum[i] = 0;
00225             sum[0] = cpu = TimeCPU(1);
00226             cpart = (WORD)(cpu%1000);
00227             cpu /= 1000;
00228             cpart /= 10;
00229             if ( AC.OldParallelStats ) MesPrint("");
00230             if ( proc > 0 && AC.StatsFlag && AC.OldParallelStats ) {
00231                 MesPrint("proc          CPU          in         gen       left       byte");
00232                 MesPrint("%3d  : %7l.%2i %10l",0,cpu,cpart,AN.ninterms);
00233             }
00234             else if ( AC.StatsFlag && AC.OldParallelStats ) {
00235                 MesPrint("proc          CPU          in         gen       out        byte");
00236                 MesPrint("%3d  : %7l.%2i %10l %10l %10l",0,cpu,cpart,AN.ninterms,0,PF_goutterms);
00237             }
00238
00239             for ( i = 1; i < PF.numtasks; i++ ) {
00240                 cpart = (WORD)(PF_stats[i][0]%1000);
00241                 cpu = PF_stats[i][0] / 1000;
00242                 cpart /= 10;
00243                 if ( AC.StatsFlag && AC.OldParallelStats )
00244                     MesPrint("%3d  : %7l.%2i %10l %10l %10l",i,cpu,cpart,
00245                             PF_stats[i][2],PF_stats[i][3],PF_stats[i][4]);
00246                 for ( j = 0; j < PF_STATS_SIZE; j++ ) sum[j] += PF_stats[i][j];
00247             }
00248             cpart = (WORD)(sum[0]%1000);
00249             cpu = sum[0] / 1000;
00250             cpart /= 10;
00251             if ( AC.StatsFlag && AC.OldParallelStats ) {
00252                 MesPrint("Sum  = %7l.%2i %10l %10l %10l",cpu,cpart,sum[2],sum[3],sum[4]);
00253                 MesPrint("Real = %7l.%2i %20s (%l) %16s",
00254                         real,rpart,AC.Commercial,AC.CModule,EXPRNAME(AR.CurExpr));
00255                 MesPrint("");
00256             }
00257             PF_laststat = real;
00258         }
00259     }
00260     return(0);
00261 }
00262 /*
00263         #] PF_Statistics :
00264     #] statistics :
00265     #[ sort.c :
```

```
00266         #[ sort variables :
00267 */
00268
00272 typedef struct NoDe {
00273     struct NoDe *left;
00274     struct NoDe *rght;
00275     int lloser;
00276     int rloser;
00277     int lsrc;
00278     int rsrc;
00279 } NODE;
00280
00281 /*
00282     should/could be put in one struct
00283 */
00284 static  NODE *PF_root;          /* root of tree of losers */
00285 static  WORD PF_loser;          /* this is the last loser */
00286 static  WORD **PF_term;         /* these point to the active terms */
00287 static  WORD **PF_newcpos;      /* new coefficients of merged terms */
00288 static  WORD *PF_newclen;       /* length of new coefficients */
00289
00290 /*
00291     preliminary: could also write somewhere else?
00292 */
00293
00294 static  WORD *PF_WorkSpace;     /* used in PF_EndSort() */
00295 static  UWORD *PF_ScratchSpace; /* used in PF_GetLoser() */
00296
00297 /*
00298         #] sort variables :
00299         #[ PF_AllocBuf :
00300 */
00301
00318 static PF_BUFFER *PF_AllocBuf(int nbufs, LONG bsize, WORD free)
00319 {
00320     PF_BUFFER *buf;
00321     UBYTE *p, *stop;
00322     LONG allocsize;
00323     int i;
00324
00325     allocsize =
00326         (LONG)(sizeof(PF_BUFFER) + 4*nbufs*sizeof(WORD*) + (nbufs-free)*bsize);
00327
00328     allocsize +=
00329         (LONG)( nbufs * (  2 * sizeof(MPI_Status)
00330                          +     sizeof(MPI_Request)
00331                          +     sizeof(MPI_Datatype)
00332                        )  );
00333     allocsize += (LONG)( nbufs * 3 * sizeof(int) );
00334
00335     if ( ( buf = (PF_BUFFER*)Malloc1(allocsize,"PF_AllocBuf") ) == NULL ) return(NULL);
00336
00337     p = ((UBYTE *)buf) + sizeof(PF_BUFFER);
00338     stop = ((UBYTE *)buf) + allocsize;
00339
00340     buf->numbufs = nbufs;
00341     buf->active = 0;
00342
00343     buf->buff    = (WORD**)p;          p += buf->numbufs*sizeof(WORD*);
00344     buf->fill    = (WORD**)p;          p += buf->numbufs*sizeof(WORD*);
00345     buf->full    = (WORD**)p;          p += buf->numbufs*sizeof(WORD*);
00346     buf->stop    = (WORD**)p;          p += buf->numbufs*sizeof(WORD*);
00347     buf->status  = (MPI_Status *)p;    p += buf->numbufs*sizeof(MPI_Status);
00348     buf->retstat = (MPI_Status *)p;    p += buf->numbufs*sizeof(MPI_Status);
00349     buf->request = (MPI_Request *)p;   p += buf->numbufs*sizeof(MPI_Request);
00350     buf->type    = (MPI_Datatype *)p;  p += buf->numbufs*sizeof(MPI_Datatype);
00351     buf->index   = (int *)p;           p += buf->numbufs*sizeof(int);
00352
00353     for ( i = 0; i < buf->numbufs; i++ ) buf->request[i] = MPI_REQUEST_NULL;
00354     buf->tag     = (int *)p;           p += buf->numbufs*sizeof(int);
00355     buf->from    = (int *)p;           p += buf->numbufs*sizeof(int);
00356 /*
00357         and finally the real bufferspace
00358 */
00359     for ( i = free; i < buf->numbufs; i++ ) {
00360         buf->buff[i] = (WORD*)p; p += bsize;
00361         buf->stop[i] = (WORD*)p;
00362         buf->fill[i] = buf->full[i] = buf->buff[i];
00363     }
00364     if ( p != stop ) {
00365         MesPrint("Error in PF_AllocBuf p = %x stop = %x\n",p,stop);
00366         return(NULL);
00367     }
00368     return(buf);
00369 }
00370
00371 /*
```

```
00372         #] PF_AllocBuf :
00373         #[ PF_InitTree :
00374 */
00375
00387 static int PF_InitTree(void)
00388 {
00389     GETIDENTITY
00390     PF_BUFFER **rbuf = PF.rbufs;
00391     UBYTE *p, *stop;
00392     int numrbufs,numtasks = PF.numtasks;
00393     int i, j, src, numnodes;
00394     int numslaves = numtasks - 1;
00395     LONG size;
00396 /*
00397         #[ the buffers : for the new coefficients and the terms
00398            we need one for each slave
00399 */
00400     if ( PF_term == NULL ) {
00401         size =  2*numtasks*sizeof(WORD*) + sizeof(WORD)*
00402             ( numtasks*(1 + AM.MaxTal) + (AM.MaxTer/sizeof(WORD)+1) + 2*(AM.MaxTal+2));
00403
00404         PF_term = (WORD **)Malloc1(size,"PF_term");
00405         stop = ((UBYTE*)PF_term) + size;
00406         p = ((UBYTE*)PF_term) + numtasks*sizeof(WORD*);
00407
00408         PF_newcpos = (WORD **)p;  p += sizeof(WORD*) * numtasks;
00409         PF_newclen =  (WORD *)p;  p += sizeof(WORD)  * numtasks;
00410         for ( i = 0; i < numtasks; i++ ) {
00411             PF_newcpos[i] = (WORD *)p; p += sizeof(WORD)*AM.MaxTal;
00412             PF_newclen[i] = 0;
00413         }
00414         PF_WorkSpace = (WORD *)p;    p += AM.MaxTer+sizeof(WORD);
00415         PF_ScratchSpace = (UWORD*)p; p += 2*(AM.MaxTal+2)*sizeof(UWORD);
00416
00417         if ( p != stop ) { MesPrint("error in PF_InitTree"); return(-1); }
00418     }
00419 /*
00420         #] the buffers :
00421         #[ the receive buffers :
00422 */
00423     numrbufs = PF.numrbufs;
00424 /*
00425         this is the size we have in the combined sortbufs for one slave
00426 */
00427     size = (AT.SS->sTop2 - AT.SS->lBuffer - 1)/(PF.numtasks - 1);
00428
00429     if ( rbuf == NULL ) {
00430         if ( ( rbuf = (PF_BUFFER**)Malloc1(numtasks*sizeof(PF_BUFFER*), "Master: rbufs") ) == NULL )
    return(-1);
00431         if ( (rbuf[0] = PF_AllocBuf(1,0,1) ) == NULL ) return(-1);
00432         for ( i = 1; i < numtasks; i++ ) {
00433             if (!(rbuf[i] = PF_AllocBuf(numrbufs,sizeof(WORD)*size,1))) return(-1);
00434         }
00435     }
00436     rbuf[0]->buff[0] = AT.SS->lBuffer;
00437     rbuf[0]->full[0] = rbuf[0]->fill[0] = rbuf[0]->buff[0];
00438     rbuf[0]->stop[0] = rbuf[1]->buff[0] = rbuf[0]->buff[0] + 1;
00439     rbuf[1]->full[0] = rbuf[1]->fill[0] = rbuf[1]->buff[0];
00440     for ( i = 2; i < numtasks; i++ ) {
00441         rbuf[i-1]->stop[0] = rbuf[i]->buff[0] = rbuf[i-1]->buff[0] + size;
00442         rbuf[i]->full[0] = rbuf[i]->fill[0] = rbuf[i]->buff[0];
00443     }
00444     rbuf[numtasks-1]->stop[0] = rbuf[numtasks-1]->buff[0] + size;
00445
00446     for ( i = 1; i < numtasks; i++ ) {
00447         for ( j = 0; j < rbuf[i]->numbufs; j++ ) {
00448             rbuf[i]->full[j] = rbuf[i]->fill[j] = rbuf[i]->buff[j] + AM.MaxTer/sizeof(WORD) + 2;
00449         }
00450         PF_term[i] = rbuf[i]->fill[rbuf[i]->active];
00451         *PF_term[i] = 0;
00452         PF_IRecvRbuf(rbuf[i],rbuf[i]->active,i);
00453     }
00454     rbuf[0]->active = 0;
00455     PF_term[0] = rbuf[0]->buff[0];
00456     PF_term[0][0] = 0;  /* PF_term[0] is used for a zero term. */
00457     PF.rbufs = rbuf;
00458 /*
00459         #] the receive buffers :
00460         #[ the actual tree :
00461
00462      calculate number of nodes in mergetree and allocate space for them
00463 */
00464     if ( numslaves < 3 ) numnodes = 1;
00465     else {
00466         numnodes = 2;
00467         while ( numnodes < numslaves ) numnodes *= 2;
00468         numnodes -= 1;
```

```
00469     }
00470
00471     if ( PF_root == NULL )
00472     if ( ( PF_root = (NODE*)Malloc1(sizeof(NODE)*numnodes,"nodes in mergetree") ) == NULL )
00473         return(-1);
00474 /*
00475         then initialize all the nodes
00476 */
00477     src = 1;
00478     for ( i = 0; i < numnodes; i++ ) {
00479         if ( 2*(i+1) <= numnodes ) {
00480             PF_root[i].left = &(PF_root[2*(i+1)-1]);
00481             PF_root[i].lsrc = 0;
00482         }
00483         else {
00484             PF_root[i].left = 0;
00485             if ( src < numtasks ) PF_root[i].lsrc = src++;
00486             else                  PF_root[i].lsrc = 0;
00487         }
00488         PF_root[i].lloser = 0;
00489     }
00490     for ( i = 0; i < numnodes; i++ ) {
00491         if ( 2*(i+1)+1 <= numnodes ) {
00492             PF_root[i].rght = &(PF_root[2*(i+1)]);
00493             PF_root[i].rsrc = 0;
00494         }
00495         else {
00496             PF_root[i].rght = 0;
00497             if (src<numtasks) PF_root[i].rsrc = src++;
00498             else              PF_root[i].rsrc = 0;
00499         }
00500         PF_root[i].rloser = 0;
00501     }
00502 /*
00503         #] the actual tree :
00504 */
00505     return(numnodes);
00506 }
00507
00508 /*
00509         #] PF_InitTree :
00510         #[ PF_PutIn :
00511 */
00512
00531 static WORD *PF_PutIn(int src)
00532 {
00533     int tag;
00534     WORD im, r;
00535     WORD *m1, *m2;
00536     LONG size;
00537     PF_BUFFER *rbuf = PF.rbufs[src];
00538     int a = rbuf->active;
00539     int next = a+1 >= rbuf->numbufs ? 0 : a+1 ;
00540     WORD *lastterm = PF_term[src];
00541     WORD *term = rbuf->fill[a];
00542
00543     if ( src <= 0 ) return(PF_term[0]);
00544
00545     if ( rbuf->full[a] == rbuf->buff[a] + AM.MaxTer/sizeof(WORD) + 2 ) {
00546 /*
00547             very first term from this src
00548 */
00549         tag = PF_WaitRbuf(rbuf,a,&size);
00550         if ( tag == PF_RUNTIME_ERROR_MSGTAG ) {
00551             PF_ReceiveRuntimeError();
00552         }
00553         rbuf->full[a] += size;
00554         if ( tag == PF_ENDBUFFER_MSGTAG ) *rbuf->full[a]++ = 0;
00555         else if ( rbuf->numbufs > 1 ) {
00556 /*
00557                 post a nonblock. recv. for the next buffer
00558 */
00559             rbuf->full[next] = rbuf->buff[next] + AM.MaxTer/sizeof(WORD) + 2;
00560             size = (LONG)(rbuf->stop[next] - rbuf->full[next]);
00561             PF_IRecvRbuf(rbuf,next,src);
00562         }
00563     }
00564     if ( *term == 0 && term != rbuf->full[a] ) return(PF_term[0]);
00565 /*
00566         exception is for rare cases when the terms fitted exactly into buffer
00567 */
00568     if ( term + *term > rbuf->full[a] || term + 1 >= rbuf->full[a] ) {
00569 newterms:
00570         m1 = rbuf->buff[next] + AM.MaxTer/sizeof(WORD) + 1;
00571         if ( *term < 0 || term == rbuf->full[a] ) {
00572 /*
00573             copy term and lastterm to the new buffer, so that they end at m1
```

```
00574 */
00575             m2 = rbuf->full[a] - 1;
00576             while ( m2 >= term ) *m1-- = *m2--;
00577             rbuf->fill[next] = term = m1 + 1;
00578             m2 = lastterm + *lastterm - 1;
00579             while ( m2 >= lastterm ) *m1-- = *m2--;
00580             lastterm = m1 + 1;
00581         }
00582         else {
00583 /*
00584             copy beginning of term to the next buffer so that it ends at m1
00585 */
00586             m2 = rbuf->full[a] - 1;
00587             while ( m2 >= term ) *m1-- = *m2--;
00588             rbuf->fill[next] = term = m1 + 1;
00589         }
00590         if ( rbuf->numbufs == 1 ) {
00591             rbuf->full[a] = rbuf->buff[a] + AM.MaxTer/sizeof(WORD) + 2;
00592             size = (LONG)(rbuf->stop[a] - rbuf->full[a]);
00593             PF_IRecvRbuf(rbuf,a,src);
00594         }
00595 /*
00596             wait for new terms in the next buffer
00597 */
00598         rbuf->full[next] = rbuf->buff[next] + AM.MaxTer/sizeof(WORD) + 2;
00599         tag = PF_WaitRbuf(rbuf,next,&size);
00600         rbuf->full[next] += size;
00601         if ( tag == PF_ENDBUFFER_MSGTAG ) {
00602             *rbuf->full[next]++ = 0;
00603         }
00604         else if ( rbuf->numbufs > 1 ) {
00605 /*
00606             post a nonblock. recv. for active buffer, it is not needed anymore
00607 */
00608             rbuf->full[a] = rbuf->buff[a] + AM.MaxTer/sizeof(WORD) + 2;
00609             size = (LONG)(rbuf->stop[a] - rbuf->full[a]);
00610             PF_IRecvRbuf(rbuf,a,src);
00611         }
00612 /*
00613             now safely make next buffer active
00614 */
00615         a = rbuf->active = next;
00616     }
00617
00618     if ( *term < 0 ) {
00619 /*
00620             We need to decompress the term
00621 */
00622         im = *term;
00623         r = term[1] - im + 1;
00624         m1 = term + 2;
00625         m2 = lastterm - im + 1;
00626         while ( ++im <= 0 ) *--m1 = *--m2;
00627         *--m1 = r;
00628         rbuf->fill[a] = term = m1;
00629         if ( term + *term > rbuf->full[a] ) goto newterms;
00630     }
00631     rbuf->fill[a] += *term;
00632     return(term);
00633 }
00634
00635 /*
00636         #] PF_PutIn :
00637         #[ PF_GetLoser :
00638 */
00639
00658 static int PF_GetLoser(NODE *n)
00659 {
00660     GETIDENTITY
00661     WORD comp;
00662
00663     if ( PF_loser == 0 ) {
00664 /*
00665             this is for the right initialization of the tree only
00666 */
00667         if ( n->left ) n->lloser = PF_GetLoser(n->left);
00668         else {
00669             n->lloser = n->lsrc;
00670             if ( *(PF_term[n->lsrc] = PF_PutIn(n->lsrc)) == 0) n->lloser = 0;
00671         }
00672         PF_loser = 0;
00673         if ( n->rght ) n->rloser = PF_GetLoser(n->rght);
00674         else{
00675             n->rloser = n->rsrc;
00676             if ( *(PF_term[n->rsrc] = PF_PutIn(n->rsrc)) == 0 ) n->rloser = 0;
00677         }
00678         PF_loser = 0;
```

```
00679     }
00680     else if ( PF_loser == n->lloser ) {
00681         if ( n->left ) n->lloser = PF_GetLoser(n->left);
00682         else {
00683             n->lloser = n->lsrc;
00684             if ( *(PF_term[n->lsrc] = PF_PutIn(n->lsrc)) == 0 ) n->lloser = 0;
00685         }
00686     }
00687     else if ( PF_loser == n->rloser ) {
00688 newright:
00689         if ( n->rght ) n->rloser = PF_GetLoser(n->rght);
00690         else {
00691             n->rloser = n->rsrc;
00692             if ( *(PF_term[n->rsrc] = PF_PutIn(n->rsrc)) == 0 ) n->rloser = 0;
00693         }
00694     }
00695     if ( n->lloser > 0 && n->rloser > 0 ) {
00696         comp = CompareTerms(BHEAD PF_term[n->lloser],PF_term[n->rloser],(WORD)0);
00697         if ( comp > 0 )      return(n->lloser);
00698         else if (comp < 0 ) return(n->rloser);
00699         else {
00700 /*
00701         #[ terms are equal :
00702 */
00703             WORD *lcpos, *rcpos;
00704             UWORD *newcpos;
00705             WORD lclen, rclen, newclen, newnlen;
00706             SORTING *S = AT.SS;
00707
00708             if ( S->PolyWise ) {
00709 /*
00710             #[ Here we work with PolyFun :
00711 */
00712                 WORD *tt1, *w;
00713                 WORD r1,r2;
00714                 WORD *ml = PF_term[n->lloser];
00715                 WORD *mr = PF_term[n->rloser];
00716
00717                 if ( ( r1 = (int)*PF_term[n->lloser] ) <= 0 ) r1 = 20;
00718                 if ( ( r2 = (int)*PF_term[n->rloser] ) <= 0 ) r2 = 20;
00719                 tt1 = ml;
00720                 ml += S->PolyWise;
00721                 mr += S->PolyWise;
00722                 if ( S->PolyFlag == 2 ) {
00723                     w = poly_ratfun_add(BHEAD ml,mr);
00724                     if ( *tt1 + w[1] - ml[1] > AM.MaxTer/((LONG)sizeof(WORD)) ) {
00725                         MesPrint("Term too complex in PolyRatFun addition. MaxTermSize of %10l is too
      small",AM.MaxTer);
00726                         Terminate(-1);
00727                     }
00728                     AT.WorkPointer = w;
00729                 }
00730                 else {
00731                     w = AT.WorkPointer;
00732                     if ( w + ml[1] + mr[1] > AT.WorkTop ) {
00733                         MesPrint("A WorkSpace of %10l is too small",AM.WorkSize);
00734                         Terminate(-1);
00735                     }
00736                     AddArgs(BHEAD ml,mr,w);
00737                 }
00738                 r1 = w[1];
00739                 if ( r1 <= FUNHEAD || ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 ) ) {
00740                     goto cancelled;
00741                 }
00742                 if ( r1 == ml[1] ) {
00743                     NCOPY(ml,w,r1);
00744                 }
00745                 else if ( r1 < ml[1] ) {
00746                     r2 = ml[1] - r1;
00747                     mr = w + r1;
00748                     ml += ml[1];
00749                     while ( --r1 >= 0 ) *--ml = *--mr;
00750                     mr = ml - r2;
00751                     r1 = S->PolyWise;
00752                     while ( --r1 >= 0 ) *--ml = *--mr;
00753                     *ml -= r2;
00754                     PF_term[n->lloser] = ml;
00755                 }
00756                 else {
00757                     r2 = r1 - ml[1];
00758                     if ( r2 > 2*AM.MaxTal )
00759                         MesPrint("warning: new term in polyfun is large");
00760                     mr = tt1 - r2;
00761                     r1 = S->PolyWise;
00762                     ml = tt1;
00763                     *ml += r2;
00764                     PF_term[n->lloser] = mr;
```

```
00765                     NCOPY(mr,ml,r1);
00766                     r1 = w[1];
00767                     NCOPY(mr,w,r1);
00768                 }
00769                 PF_newclen[n->rloser] = 0;
00770                 PF_loser = n->rloser;
00771                 goto newright;
00772 /*
00773             #] Here we work with PolyFun :
00774 */
00775             }
00776             if ( ( lclen = PF_newclen[n->lloser] ) != 0 ) lcpos = PF_newcpos[n->lloser];
00777             else {
00778                 lcpos = PF_term[n->lloser];
00779                 lclen = *(lcpos += *lcpos - 1);
00780                 lcpos -= ABS(lclen) - 1;
00781             }
00782             if ( ( rclen = PF_newclen[n->rloser] ) != 0 ) rcpos = PF_newcpos[n->rloser];
00783             else {
00784                 rcpos = PF_term[n->rloser];
00785                 rclen = *(rcpos += *rcpos - 1);
00786                 rcpos -= ABS(rclen) -1;
00787             }
00788             lclen = ( (lclen > 0) ? (lclen-1) : (lclen+1) ) >> 1;
00789             rclen = ( (rclen > 0) ? (rclen-1) : (rclen+1) ) >> 1;
00790             newcpos = PF_ScratchSpace;
00791             if ( AddRat(BHEAD (UWORD *)lcpos,lclen,(UWORD *)rcpos,rclen,newcpos,&newnlen) )
    return(-1);
00792             if ( AN.ncmod != 0 ) {
00793                 if ( ( AC.modmode & POSNEG ) != 0 ) {
00794                     NormalModulus(newcpos,&newnlen);
00795                 }
00796                 if ( BigLong(newcpos,newnlen,(UWORD *)AC.cmod,ABS(AN.ncmod)) >=0 ) {
00797                     WORD ii;
00798                     SubPLon(newcpos,newnlen,(UWORD *)AC.cmod,ABS(AN.ncmod),newcpos,&newnlen);
00799                     newcpos[newnlen] = 1;
00800                     for ( ii = 1; ii < newnlen; ii++ ) newcpos[newnlen+ii] = 0;
00801                 }
00802             }
00803             if ( newnlen == 0 ) {
00804 /*
00805                     terms cancel, get loser of left subtree and then of right subtree
00806 */
00807 cancelled:
00808                 PF_loser = n->lloser;
00809                 PF_newclen[n->lloser] = 0;
00810                 if ( n->left ) n->lloser = PF_GetLoser(n->left);
00811                 else {
00812                     n->lloser = n->lsrc;
00813                     if ( *(PF_term[n->lsrc] = PF_PutIn(n->lsrc)) == 0 ) n->lloser = 0;
00814                 }
00815                 PF_loser = n->rloser;
00816                 PF_newclen[n->rloser] = 0;
00817                 goto newright;
00818             }
00819             else {
00820 /*
00821                     keep the left term and get the loser of right subtree
00822 */
00823                 newnlen *= 2;
00824                 newclen = ( newnlen > 0 ) ? ( newnlen + 1 ) : ( newnlen - 1 );
00825                 if ( newnlen < 0 ) newnlen = -newnlen;
00826                 PF_newclen[n->lloser] = newclen;
00827                 lcpos = PF_newcpos[n->lloser];
00828                 if ( newclen < 0 ) newclen = -newclen;
00829                 while ( newclen-- ) *lcpos++ = *newcpos++;
00830                 PF_loser = n->rloser;
00831                 PF_newclen[n->rloser] = 0;
00832                 goto newright;
00833             }
00834 /*
00835         #] terms are equal :
00836 */
00837         }
00838     }
00839     if (n->lloser > 0) return(n->lloser);
00840     if (n->rloser > 0) return(n->rloser);
00841     return(0);
00842 }
00843 /*
00844         #] PF_GetLoser :
00845         #[ PF_EndSort :
00846 */
00847
00874 int PF_EndSort(void)
00875 {
00876     GETIDENTITY
```

```
00877     FILEHANDLE *fout = AR.outfile;
00878     PF_BUFFER *sbuf=PF.sbuf;
00879     SORTING *S = AT.SS;
00880     WORD *outterm,*pp;
00881     LONG size, noutterms;
00882     POSITION position, oldposition;
00883     WORD i,cc;
00884     int oldgzipCompress;
00885 
00886     if ( AT.SS != AT.S0 || !PF.parallel ) return 0;
00887 
00888     if ( PF.me != MASTER ) {
00889 /*
00890         #[ the slaves have to initialize their sendbuffer :
00891 
00892         this is a slave and it's PObuffer should be the minimum of the
00893         sortiosize on the master and the POsize of our file.
00894         First save the original PObuffer and POstop of the outfile
00895 */
00896         size = (S->sTop2 - S->lBuffer - 1)/(PF.numtasks - 1);
00897         size -= (AM.MaxTer/sizeof(WORD) + 2);
00898         if ( fout->POsize < (LONG)(size*sizeof(WORD)) ) size = fout->POsize/sizeof(WORD);
00899         if ( sbuf == NULL ) {
00900             if ( (sbuf = PF_AllocBuf(PF.numsbufs, size*sizeof(WORD), 1)) == NULL ) return -1;
00901             sbuf->active = 0;
00902             PF.sbuf = sbuf;
00903         }
00904         sbuf->buff[0] = fout->PObuffer;
00905         sbuf->stop[0] = fout->PObuffer+size;
00906         if ( sbuf->stop[0] > fout->POstop ) return -1;
00907         for ( i = 0; i < PF.numsbufs; i++ )
00908             sbuf->fill[i] = sbuf->full[i] = sbuf->buff[i];
00909 
00910         fout->PObuffer = sbuf->buff[sbuf->active];
00911         fout->POstop = sbuf->stop[sbuf->active];
00912         fout->POsize = size*sizeof(WORD);
00913         fout->POfill = fout->POfull = fout->PObuffer;
00914 /*
00915         #] the slaves have to initialize their sendbuffer :
00916 */
00917         return(0);
00918     }
00919 /*
00920         this waits for all slaves to be ready to send terms back
00921 */
00922     PF_WaitAllSlaves(); /* Note, the returned value should be 0 on success. */
00923 /*
00924         Now collect the terms of all slaves and merge them.
00925         PF_GetLoser gives the position of the smallest term, which is the real
00926         work. The smallest term needs to be copied to the outbuf: use PutOut.
00927 */
00928     PF_InitTree();
00929     if ( AR.PolyFun == 0 ) { S->PolyFlag = 0; }
00930     else if ( AR.PolyFunType == 1 ) { S->PolyFlag = 1; }
00931     else if ( AR.PolyFunType == 2 ) {
00932         if ( AR.PolyFunExp == 2
00933           || AR.PolyFunExp == 3 ) S->PolyFlag = 1;
00934         else                      S->PolyFlag = 2;
00935     }
00936     *AR.CompressPointer = 0;
00937     SeekScratch(fout, &position);
00938     oldposition = position;
00939     oldgzipCompress = AR.gzipCompress;
00940     AR.gzipCompress = 0;
00941 
00942     noutterms = 0;
00943 
00944     while ( PF_loser >= 0 ) {
00945         if ( (PF_loser = PF_GetLoser(PF_root)) == 0 ) break;
00946         outterm = PF_term[PF_loser];
00947         noutterms++;
00948 
00949         if ( PF_newclen[PF_loser] != 0 ) {
00950 /*
00951             #[ this is only when new coeff was too long :
00952 */
00953             outterm = PF_WorkSpace;
00954             pp = PF_term[PF_loser];
00955             cc = *pp;
00956             while ( cc-- ) *outterm++ = *pp++;
00957             outterm = (outterm[-1] > 0) ? outterm-outterm[-1] : outterm+outterm[-1];
00958             if ( PF_newclen[PF_loser] > 0 ) cc =  (WORD)PF_newclen[PF_loser] - 1;
00959             else                            cc = -(WORD)PF_newclen[PF_loser] - 1;
00960             pp =  PF_newcpos[PF_loser];
00961             while ( cc-- ) *outterm++ = *pp++;
00962             *outterm++ = PF_newclen[PF_loser];
00963             *PF_WorkSpace = outterm - PF_WorkSpace;
```

```
00964             outterm = PF_WorkSpace;
00965             *PF_newcpos[PF_loser] = 0;
00966             PF_newclen[PF_loser] = 0;
00967 /*
00968             #] this is only when new coeff was too long :
00969 */
00970         }
00971         PRINTFBUF("PF_EndSort to PutOut: ",outterm,*outterm);
00972         PutOut(BHEAD outterm,&position,fout,1);
00973     }
00974     if ( FlushOut(&position,fout,0) ) {
00975         AR.gzipCompress = oldgzipCompress;
00976         return(-1);
00977     }
00978     S->TermsLeft = PF_goutterms = noutterms;
00979     DIFPOS(PF_exprsize, position, oldposition);
00980     AR.gzipCompress = oldgzipCompress;
00981     return(1);
00982 }
00983
00984 /*
00985         #] PF_EndSort :
00986     #] sort.c :
00987     #[ proces.c :
00988         #[ variables :
00989 */
00990
00991 static  WORD *PF_CurrentBracket;
00992
00993 /*
00994         #] variables :
00995         #[ PF_GetTerm :
00996 */
00997
01016 static WORD PF_GetTerm(WORD *term)
01017 {
01018     assert(PF.me != MASTER);
01019     GETIDENTITY
01020     FILEHANDLE *fi = AC.RhsExprInModuleFlag && PF.rhsInParallel ? &PF.slavebuf : AR.infile;
01021     WORD i;
01022     WORD *next, *np, *last, *lp = 0, *nextstop, *tp=term;
01023
01024     AN.deferskipped = 0;
01025     if ( fi->POfill >= fi->POfull || fi->POfull == fi->PObuffer ) {
01026 ReceiveNew:
01027       {
01028 /*
01029         #[ receive new terms from master :
01030 */
01031         int src = MASTER, tag;
01032         int follow = 0;
01033         LONG size,cpu,space = 0;
01034
01035         if ( PF.log ) {
01036             fprintf(stderr,"[%d] Starting to send to Master\n",PF.me);
01037             fflush(stderr);
01038         }
01039
01040         cpu = TimeCPU(1);
01041         PF_PreparePack();
01042         PF_Pack(&cpu               ,1,PF_LONG);
01043         PF_Pack(&space             ,1,PF_LONG);
01044         PF_Pack(&PF_linterms       ,1,PF_LONG);
01045         PF_Pack(&(AM.S0->GenTerms) ,1,PF_LONG);
01046         PF_Pack(&(AM.S0->TermsLeft),1,PF_LONG);
01047         PF_Pack(&follow            ,1,PF_INT );
01048
01049         if ( PF.log ) {
01050             fprintf(stderr,"[%d] Now sending with tag = %d\n",PF.me,PF_READY_MSGTAG);
01051             fflush(stderr);
01052         }
01053
01054         PF_Send(MASTER, PF_READY_MSGTAG);
01055
01056         if ( PF.log ) {
01057             fprintf(stderr,"[%d] returning from send\n",PF.me);
01058             fflush(stderr);
01059         }
01060
01061         size = fi->POstop - fi->PObuffer - 1;
01062         tag=PF_RecvWbuf(fi->PObuffer,&size,&src);
01063
01064         fi->POfill = fi->PObuffer;
01065         /* Get AN.ninterms which sits in the first 2 WORDs. */
01066         {
01067             LONG ninterms;
01068             UNPACK_LONG(fi->POfill, ninterms);
```

```
01069             if ( fi->POfill < fi->POfull ) {
01070                 DBGOUT_NINTERMS(2, ("PF.me=%d AN.ninterms=%d PF_linterms=%d ninterms=%d GET\n",
    (int)PF.me, (int)AN.ninterms, (int)PF_linterms, (int)ninterms));
01071                 AN.ninterms = ninterms - 1;
01072             } else {
01073                 DBGOUT_NINTERMS(2, ("PF.me=%d AN.ninterms=%d PF_linterms=%d ninterms=%d GETEND\n",
    (int)PF.me, (int)AN.ninterms, (int)PF_linterms, (int)ninterms));
01074             }
01075         }
01076         fi->POfull = fi->PObuffer + size;
01077         if ( tag == PF_ENDSORT_MSGTAG ) *fi->POfull++ = 0;
01078 /*
01079         #] receive new terms from master :
01080 */
01081       }
01082       if ( PF_CurrentBracket ) *PF_CurrentBracket = 0;
01083     }
01084     if ( *fi->POfill == 0 ) {
01085         fi->POfill = fi->POfull = fi->PObuffer;
01086         *term = 0;
01087         goto RegRet;
01088     }
01089     if ( AR.DeferFlag ) {
01090         if ( !PF_CurrentBracket ) {
01091 /*
01092         #[ alloc space :
01093 */
01094             PF_CurrentBracket =
01095                     (WORD*)Malloc1(AM.MaxTer,"PF_CurrentBracket");
01096             *PF_CurrentBracket = 0;
01097 /*
01098         #] alloc space :
01099 */
01100         }
01101         while ( *PF_CurrentBracket ) {  /* "for each term in the buffer" */
01102 /*
01103         #[ test : bracket & skip if it's equal to the last in PF_CurrentBracket
01104 */
01105             next = fi->POfill;
01106             nextstop = next + *next; nextstop -= ABS(nextstop[-1]);
01107             next++;
01108             last = PF_CurrentBracket+1;
01109             while ( next < nextstop ) {
01110 /*
01111                     scan the next term and PF_CurrentBracket
01112 */
01113                 if ( *last == HAAKJE && *next == HAAKJE ) {
01114 /*
01115                     the part outside brackets is equal => skip this term
01116 */
01117                     PRINTFBUF("PF_GetTerm skips",fi->POfill,*fi->POfill);
01118                     break;
01119                 }
01120 /*
01121                     check if the current subterms are equal
01122 */
01123                 np = next; next += next[1];
01124                 lp = last; last += last[1];
01125                 while ( np < next ) if ( *lp++ != *np++ ) goto strip;
01126             }
01127 /*
01128                 go on to next term
01129 */
01130             fi->POfill += *fi->POfill;
01131             AN.deferskipped++;
01132 /*
01133                 the usual checks
01134 */
01135             if ( fi->POfill >= fi->POfull || fi->POfull == fi->PObuffer )
01136                 goto ReceiveNew;
01137             if ( *fi->POfill == 0 ) {
01138                 fi->POfill = fi->POfull = fi->PObuffer;
01139                 *term = 0;
01140                 goto RegRet;
01141             }
01142 /*
01143         #] test :
01144 */
01145         }
01146 /*
01147         #[ copy :

01149         this term to CurrentBracket and the part outside of bracket
01150         to WorkSpace at term
01151 */
01152 strip:
01153         next = fi->POfill;
```

```
01154         nextstop = next + *next; nextstop -= ABS(nextstop[-1]);
01155         next++;
01156         tp++;
01157         lp = PF_CurrentBracket + 1;
01158         while ( next < nextstop ) {
01159             if ( *next == HAAKJE ) {
01160                 fi->POfill += *fi->POfill;
01161                 while ( next < fi->POfill ) *lp++ = *next++;
01162                 *PF_CurrentBracket = lp - PF_CurrentBracket;
01163                 *lp = 0;
01164                 *tp++ = 1;
01165                 *tp++ = 1;
01166                 *tp++ = 3;
01167                 *term = WORDDIF(tp,term);
01168                 PRINTFBUF("PF_GetTerm new brack",PF_CurrentBracket,*PF_CurrentBracket);
01169                 PRINTFBUF("PF_GetTerm POfill",fi->POfill,*fi->POfill);
01170                 goto RegRet;
01171             }
01172             np = next; next += next[1];
01173             while ( np < next ) *tp++ = *lp++ = *np++;
01174         }
01175         tp = term;
01176 /*
01177         #] copy :
01178 */
01179     }
01180
01181     i = *fi->POfill;
01182     while ( i-- ) *tp++ = *fi->POfill++;
01183 RegRet:
01184     PRINTFBUF("PF_GetTerm returns",term,*term);
01185     return(*term);
01186 }
01187
01188 /*
01189         #] PF_GetTerm :
01190         #[ PF_Deferred :
01191 */
01192
01201 WORD PF_Deferred(WORD *term, WORD level)
01202 {
01203     GETIDENTITY
01204     WORD *bra, *bstop;
01205     WORD *tstart;
01206     FILEHANDLE *fi = AC.RhsExprInModuleFlag && PF.rhsInParallel ? &PF.slavebuf : AR.infile;
01207     WORD *next = fi->POfill;
01208     WORD *termout = AT.WorkPointer;
01209     WORD *oldwork = AT.WorkPointer;
01210
01211     AT.WorkPointer = (WORD *)((UBYTE *)(AT.WorkPointer) + AM.MaxTer);
01212     AR.DeferFlag = 0;
01213
01214     PRINTFBUF("PF_Deferred (Term)    ",term,*term);
01215     PRINTFBUF("PF_Deferred (Bracket)",PF_CurrentBracket,*PF_CurrentBracket);
01216
01217     bra = bstop = PF_CurrentBracket;
01218     if ( *bstop > 0 ) {
01219         bstop += *bstop;
01220         bstop -= ABS(bstop[-1]);
01221     }
01222     bra++;
01223     while ( *bra != HAAKJE && bra < bstop ) bra += bra[1];
01224     if ( bra >= bstop ) {   /* No deferred action! */
01225         AT.WorkPointer = term + *term;
01226         if ( Generator(BHEAD term,level) ) goto DefCall;
01227         AR.DeferFlag = 1;
01228         AT.WorkPointer = oldwork;
01229         return(0);
01230     }
01231     bstop = bra;
01232     tstart = bra + bra[1];
01233     bra = PF_CurrentBracket;
01234     tstart--;
01235     *tstart = bra + *bra - tstart;
01236     bra++;
01237 /*
01238         Status of affairs:
01239         First bracket content starts at tstart.
01240         Next term starts at next.
01241         The outside of the bracket runs from bra = PF_CurrentBracket to bstop.
01242 */
01243     for(;;) {
01244         if ( InsertTerm(BHEAD term,0,AM.rbufnum,tstart,termout,0) < 0 ) {
01245             goto DefCall;
01246         }
01247 /*
01248             call Generator with new composed term
```

```
01249 */
01250         AT.WorkPointer = termout + *termout;
01251         if ( Generator(BHEAD termout,level) ) goto DefCall;
01252         AT.WorkPointer = termout;
01253         tstart = next + 1;
01254         if ( tstart >= fi->POfull ) goto ThatsIt;
01255         next += *next;
01256 /*
01257             compare with current bracket
01258 */
01259         while ( bra <= bstop ) {
01260             if ( *bra != *tstart ) goto ThatsIt;
01261             bra++; tstart++;
01262         }
01263 /*
01264             now bra and tstart should both be a HAAKJE
01265 */
01266         bra--; tstart--;
01267         if ( *bra != HAAKJE || *tstart != HAAKJE ) goto ThatsIt;
01268         tstart += tstart[1];
01269         tstart--;
01270         *tstart = next - tstart;
01271         bra = PF_CurrentBracket + 1;
01272     }
01273
01274 ThatsIt:
01275 /*
01276     AT.WorkPointer = oldwork;
01277 */
01278     AR.DeferFlag = 1;
01279     return(0);
01280 DefCall:
01281     MesCall("PF_Deferred");
01282     SETERROR(-1);
01283 }
01284
01285 /*
01286         #] PF_Deferred :
01287         #[ PF_Wait4Slave :
01288 */
01289
01290 static LONG **PF_W4Sstats = 0;
01291
01298 static int PF_Wait4Slave(int src)
01299 {
01300     int j, tag, next;
01301
01302     tag = PF_ANY_MSGTAG;
01303     PF_CatchErrorMessages(&src, &tag);
01304     PF_Receive(src, tag, &next, &tag);
01305
01306     if ( tag != PF_READY_MSGTAG ) {
01307         MesPrint("[%d] PF_Wait4Slave: received MSGTAG %d",(WORD)PF.me,(WORD)tag);
01308         return(-1);
01309     }
01310     if ( PF_W4Sstats == 0 ) {
01311         PF_W4Sstats = (LONG**)Malloc1(sizeof(LONG*),"");
01312         PF_W4Sstats[0] = (LONG*)Malloc1(PF_STATS_SIZE*sizeof(LONG),"");
01313     }
01314     PF_Unpack(PF_W4Sstats[0],PF_STATS_SIZE,PF_LONG);
01315     PF_Statistics(PF_W4Sstats,next);
01316
01317     PF_Unpack(&j,1,PF_INT);
01318
01319     if ( j ) {
01320 /*
01321         actions depending on rest of information in last message
01322 */
01323     }
01324     return(next);
01325 }
01326
01327 /*
01328         #] PF_Wait4Slave :
01329         #[ PF_Wait4SlaveIP :
01330 */
01331 /*
01332     array of expression numbers for PF_InParallel processor.
01333     Each time the master sends expression "i" to the slave
01334     "next" it sets partodoexr[next]=i:
01335 */
01336 static WORD *partodoexr=NULL;
01337
01345 static int PF_Wait4SlaveIP(int *src)
01346 {
01347     int j,tag,next;
01348
```

```
01349     tag = PF_ANY_MSGTAG;
01350     PF_CatchErrorMessages(src, &tag);
01351     PF_Receive(*src, tag, &next, &tag);
01352     *src=tag;
01353     if ( PF_W4Sstats == 0 ) {
01354         PF_W4Sstats = (LONG**)Malloc1(sizeof(LONG*),"");
01355         PF_W4Sstats[0] = (LONG*)Malloc1(PF_STATS_SIZE*sizeof(LONG),"");
01356     }
01357
01358     PF_Unpack(PF_W4Sstats[0],PF_STATS_SIZE,PF_LONG);
01359     if ( tag == PF_DATA_MSGTAG )
01360         AR.CurExpr = partodoexr[next];
01361     PF_Statistics(PF_W4Sstats,next);
01362
01363     PF_Unpack(&j,1,PF_INT);
01364
01365     if ( j ) {
01366     /* actions depending on rest of information in last message */
01367     }
01368
01369     return(next);
01370 }
01371 /*
01372         #] PF_Wait4SlaveIP :
01373         #[ PF_WaitAllSlaves :
01374 */
01375
01384 static int PF_WaitAllSlaves(void)
01385 {
01386     int i, readySlaves, tag, next = PF_ANY_SOURCE;
01387     UBYTE *has_sent = 0;
01388
01389     has_sent = (UBYTE*)Malloc1(sizeof(UBYTE)*(PF.numtasks + 1),"PF_WaitAllSlaves");
01390     for ( i = 0; i < PF.numtasks; i++ ) has_sent[i] = 0;
01391
01392     for ( readySlaves = 1; readySlaves < PF.numtasks; ) {
01393         if ( next != PF_ANY_SOURCE) { /*Go to the next slave:*/
01394             do{ /*Note, here readySlaves<PF.numtasks, so this loop can't be infinite*/
01395                 if ( ++next >= PF.numtasks ) next = 1;
01396             } while ( has_sent[next] == 1 );
01397         }
01398 /*
01399             Here PF_ProbeWithCatchingErrorMessages() is BLOCKING function if next = PF_ANY_SOURCE:
01400 */
01401         tag = PF_ProbeWithCatchingErrorMessages(&next);
01402 /*
01403             Here next != PF_ANY_SOURCE
01404 */
01405         switch ( tag ) {
01406             case PF_BUFFER_MSGTAG:
01407             case PF_ENDBUFFER_MSGTAG:
01408 /*
01409                     Slaves are ready to send their results back
01410 */
01411                 if ( has_sent[next] == 0 ) {
01412                     has_sent[next] = 1;
01413                     readySlaves++;
01414                 }
01415                 else {  /*error?*/
01416                     fprintf(stderr,"ERROR next=%d tag=%d\n",next,tag);
01417                 }
01418 /*
01419                     Note, we do NOT read results here! Messages from these slaves will be read
01420                     only after all slaves are ready, further in caller function
01421 */
01422                 break;
01423             case 0:
01424 /*
01425                     The slave is not ready. Just go to the next slave.
01426                     It may appear that there are no more ready slaves, and the master
01427                     will wait them in infinite loop. Stupid situation - the master can
01428                     receive buffers from ready slaves!
01429 */
01430 #ifdef PF_WITH_SCHED_YIELD
01431 /*
01432                         Relinquish the processor:
01433 */
01434                     sched_yield();
01435 #endif
01436                 break;
01437             case PF_DATA_MSGTAG:
01438                 tag=next;
01439                 next=PF_Wait4SlaveIP(&tag);
01440 /*
01441     tag must be == PF_DATA_MSGTAG!
01442 */
01443                 PF_Statistics(PF_stats,0);
```

```
01444                 PF_Slave2MasterIP(next);
01445                 PF_Master2SlaveIP(next,NULL);
01446                 if ( has_sent[next] == 0 ) {
01447                     has_sent[next]=1;
01448                     readySlaves++;
01449                 }else{
01450                     /*error?*/
01451                     fprintf(stderr,"ERROR next=%d tag=%d\n",next,tag);
01452                 }/*if ( has_sent[next] == 0 )*/
01453                 break;
01454             case PF_EMPTY_MSGTAG:
01455                 tag=next;
01456                 next=PF_Wait4SlaveIP(&tag);
01457 /*
01458     tag must be == PF_EMPTY_MSGTAG!
01459 */
01460                 PF_Master2SlaveIP(next,NULL);
01461                 if ( has_sent[next] == 0 ) {
01462                     has_sent[next]=1;
01463                     readySlaves++;
01464                 }else{
01465                     /*error?*/
01466                     fprintf(stderr,"ERROR next=%d tag=%d\n",next,tag);
01467                 }/*if ( has_sent[next] == 0 )*/
01468                 break;
01469             case PF_READY_MSGTAG:
01470 /*
01471                     idle slave
01472                     May be only PF_READY_MSGTAG:
01473 */
01474                 next = PF_Wait4Slave(next);
01475                 if ( next == -1 ) return(next); /*Cannot be!*/
01476                 if ( has_sent[0] == 0 ) {  /*Send the last chunk to the slave*/
01477                     PF.sbuf->active = 0;
01478                     has_sent[0] = 1;
01479                 }
01480                 else {
01481 /*
01482                         Last chunk was sent, so just send to slave ENDSORT
01483                         AN.ninterms must be sent because the slave expects it:
01484 */
01485                     PACK_LONG(PF.sbuf->fill[next], AN.ninterms);
01486 /*
01487                         This will tell to the slave that there are no more terms:
01488 */
01489                     *(PF.sbuf->fill[next])++ = 0;
01490                     PF.sbuf->active = next;
01491                 }
01492 /*
01493                     Send ENDSORT
01494 */
01495                 PF_ISendSbuf(next,PF_ENDSORT_MSGTAG);
01496                 break;
01497             default:
01498 /*
01499                     Error?
01500                     Indicates the error. This will force exit from the main loop:
01501 */
01502                 MesPrint("!!!Unexpected MPI message src=%d tag=%d.", next, tag);
01503                 readySlaves = PF.numtasks+1;
01504                 break;
01505         }
01506     }
01507
01508     if ( has_sent ) M_free(has_sent,"PF_WaitAllSlaves");
01509 /*
01510         0 on success (exit from the main loop by loop condition), or -1 if fails
01511         (exit from the main loop since readySlaves=PF.numtasks+1):
01512 */
01513     return(PF.numtasks-readySlaves);
01514 }
01515
01516 /*
01517         #] PF_WaitAllSlaves :
01518         #[ PF_Processor :
01519 */
01520
01533 int PF_Processor(EXPRESSIONS e, WORD i, WORD LastExpression)
01534 {
01535     GETIDENTITY
01536     WORD *term = AT.WorkPointer;
01537     LONG dd = 0;
01538     PF_BUFFER *sb = PF.sbuf;
01539     WORD j, *s, next;
01540     LONG size, cpu;
01541     POSITION position;
01542     int k, src, tag;
```

```
01543     FILEHANDLE *oldoutfile = AR.outfile;
01544
01545     if ( ( (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer ) ) > AT.WorkTop ) {
01546         MesWork();
01547     }
01548
01549     /* For redefine statements. */
01550     if ( AC.numpfirstnum > 0 ) {
01551         for ( j = 0; j < AC.numpfirstnum; j++ ) {
01552             AC.inputnumbers[j] = -1;
01553         }
01554     }
01555
01556     if ( AC.mparallelflag != PARALLELFLAG ) return(0);
01557
01558     PF_processing = 1;
01559
01560     if ( PF.me == MASTER ) {
01561 /*
01562         #[ Master:
01563             #[ write prototype to outfile:
01564 */
01565         WORD oldBracketOn = AR.BracketOn;
01566         WORD *oldBrackBuf = AT.BrackBuf;
01567         WORD oldbracketindexflag = AT.bracketindexflag;
01568
01569         LONG maxinterms;     /* the maximum number of terms in the bucket */
01570         int cmaxinterms;     /* a variable controling the transition of maxinterms */
01571         LONG termsinbucket;  /* the number of filled terms in the bucket */
01572         LONG ProcessBucketSize = AC.mProcessBucketSize;
01573
01574         if ( PF.log && AC.CModule >= PF.log )
01575             MesPrint("[%d] working on expression %s in module %l",PF.me,EXPRNAME(i),AC.CModule);
01576         if ( GetTerm(BHEAD term) <= 0 ) {
01577             MesPrint("[%d] Expression %d has problems in scratchfile",PF.me,i);
01578             return(-1);
01579         }
01580         term[3] = i;
01581         if ( AR.outtohide ) {
01582             SeekScratch(AR.hidefile,&position);
01583             e->onfile = position;
01584             if ( PutOut(BHEAD term,&position,AR.hidefile,0) < 0 ) return(-1);
01585         }
01586         else {
01587             SeekScratch(AR.outfile,&position);
01588             e->onfile = position;
01589             if ( PutOut(BHEAD term,&position,AR.outfile,0) < 0 ) return(-1);
01590         }
01591         AR.DeferFlag = 0;  /* The master leave the brackets!!! */
01592         AR.Eside = RHSIDE;
01593         if ( ( e->vflags & ISFACTORIZED ) != 0 ) {
01594             AR.BracketOn = 1;
01595             AT.BrackBuf = AM.BracketFactors;
01596             AT.bracketindexflag = 1;
01597         }
01598         if ( AT.bracketindexflag > 0 ) OpenBracketIndex(i);
01599 /*
01600             #] write prototype to outfile:
01601             #[ initialize sendbuffer if necessary:
01602
01603             the size of the sendbufs is:
01604             MIN(1/PF.numtasks*(AT.SS->sBufsize+AT.SS->lBufsize),AR.infile->POsize)
01605             No allocation for extra buffers necessary, just make sb->buf... point
01606             to the right places in the sortbuffers.
01607 */
01608         NewSort(BHEAD0);   /* we need AT.SS to be set for this!!! */
01609         if ( sb == 0 || sb->buff[0] != AT.SS->lBuffer ) {
01610             size = (LONG)((AT.SS->sTop2 - AT.SS->lBuffer)/(PF.numtasks));
01611             if ( size > (LONG)(AR.infile->POsize/sizeof(WORD) - 1) )
01612                 size = AR.infile->POsize/sizeof(WORD) - 1;
01613             if ( sb == 0 ) {
01614                 if ( ( sb = PF_AllocBuf(PF.numtasks,size*sizeof(WORD),PF.numtasks) ) == NULL )
01615                     return(-1);
01616             }
01617             sb->buff[0] = AT.SS->lBuffer;
01618             sb->full[0] = sb->fill[0] = sb->buff[0];
01619             for ( j = 1; j < PF.numtasks; j++ ) {
01620                 sb->stop[j-1] = sb->buff[j] = sb->buff[j-1] + size;
01621             }
01622             sb->stop[PF.numtasks-1] = sb->buff[PF.numtasks-1] + size;
01623             PF.sbuf = sb;
01624         }
01625         for ( j = 0; j < PF.numtasks; j++ ) {
01626             sb->full[j] = sb->fill[j] = sb->buff[j];
01627         }
01628 /*
01629             #] initialize sendbuffer if necessary:
```

```
01630             #[ loop for all terms in infile:
01631 */
01632         /*
01633          * The initial value of maxinterms is determined by the user given
01634          * ProcessBucketSize and the number of terms in the current expression.
01635          * We make the initial maxinterms smaller, so that we get the all
01636          * workers busy as soon as possible.
01637          */
01638         maxinterms = ProcessBucketSize / 100;
01639         if ( maxinterms > e->counter / (PF.numtasks - 1) / 4 )
01640             maxinterms = e->counter / (PF.numtasks - 1) / 4;
01641         if ( maxinterms < 1 ) maxinterms = 1;
01642         cmaxinterms = 0;
01643         /*
01644          * Copy them always to sb->buff[0]. When that is full, wait for
01645          * the next slave to accept terms, exchange sb->buff[0] and
01646          * sb->buff[next], send sb->buff[next] to next slave and go on
01647          * filling the now empty sb->buff[0].
01648          */
01649         AN.ninterms = 0;
01650         termsinbucket = 0;
01651         PACK_LONG(sb->fill[0], 1);
01652         while ( GetTerm(BHEAD term) ) {
01653             AN.ninterms++; dd = AN.deferskipped;
01654             if ( AC.CollectFun && *term <= (LONG)(AM.MaxTer/(2*sizeof(WORD))) ) {
01655                 if ( GetMoreTerms(term) < 0 ) {
01656                     LowerSortLevel(); return(-1);
01657                 }
01658             }
01659             PRINTFBUF("PF_Processor gets",term,*term);
01660             if ( termsinbucket >= maxinterms || sb->fill[0] + *term >= sb->stop[0] ) {
01661                 next = PF_Wait4Slave(PF_ANY_SOURCE);
01662 
01663                 sb->fill[next] = sb->fill[0];
01664                 sb->full[next] = sb->full[0];
01665                 SWAP(sb->stop[next], sb->stop[0]);
01666                 SWAP(sb->buff[next], sb->buff[0]);
01667                 sb->fill[0] = sb->full[0] = sb->buff[0];
01668                 sb->active = next;
01669 
01670                 PF_ISendSbuf(next,PF_TERM_MSGTAG);
01671 
01672                 /* Initialize the next bucket. */
01673                 termsinbucket = 0;
01674                 PACK_LONG(sb->fill[0], AN.ninterms);
01675                 /*
01676                  * For the "slow startup". We double maxinterms up to ProcessBucketSize
01677                  * after (hopefully) the all workers got some terms.
01678                  */
01679                 if ( cmaxinterms >= PF.numtasks - 2 ) {
01680                     maxinterms *= 2;
01681                     if ( maxinterms >= ProcessBucketSize ) {
01682                         cmaxinterms = -1;
01683                         maxinterms = ProcessBucketSize;
01684                     }
01685                 }
01686                 else if ( cmaxinterms >= 0 ) {
01687                     cmaxinterms++;
01688                 }
01689             }
01690             j = *(s = term);
01691             NCOPY(sb->fill[0], s, j);
01692             termsinbucket++;
01693         }
01694         /* NOTE: The last chunk will be sent to a slave at EndSort() => PF_EndSort()
01695          *       => PF_WaitAllSlaves(). */
01696         AN.ninterms += dd;
01697 /*
01698             #] loop for all terms in infile:
01699             #[ Clean up & EndSort:
01700 */
01701         if ( LastExpression ) {
01702             UpdateMaxSize();
01703             if ( AR.infile->handle >= 0 ) {
01704                 CloseFile(AR.infile->handle);
01705                 AR.infile->handle = -1;
01706                 remove(AR.infile->name);
01707                 PUTZERO(AR.infile->POposition);
01708             }
01709             AR.infile->POfill = AR.infile->POfull = AR.infile->PObuffer;
01710         }
01711         if ( AR.outtohide ) AR.outfile = AR.hidefile;
01712         PF.parallel = 1;
01713         if ( EndSort(BHEAD AM.S0->sBuffer,0) < 0 ) return(-1);
01714         PF.parallel = 0;
01715         if ( AR.outtohide ) {
01716             AR.outfile = oldoutfile;
```

```
01717             AR.hidefile->POfull = AR.hidefile->POfill;
01718         }
01719         UpdateMaxSize();
01720         AR.BracketOn = oldBracketOn;
01721         AT.BrackBuf = oldBrackBuf;
01722         if ( ( e->vflags & TOBEFACTORED ) != 0 )
01723             poly_factorize_expression(e);
01724         else if ( ( ( e->vflags & TOBEUNFACTORED ) != 0 )
01725          && ( ( e->vflags & ISFACTORIZED ) != 0 ) )
01726             poly_unfactorize_expression(e);
01727         AT.bracketindexflag = oldbracketindexflag;
01728         AR.GetFile = 0;
01729         AR.outtohide = 0;
01730         /*
01731          * NOTE: e->numdummies, e->vflags and AR.exprflags will be updated
01732          *       after gathering the information from all slaves.
01733          */
01734 /*
01735             #] Clean up & EndSort:
01736             #[ Collect (stats,prepro,...):
01737 */
01738         PF_PostEndSortBarrier();
01739
01740         DBGOUT_NINTERMS(1, ("PF.me=%d AN.ninterms=%d ENDSORT\n", (int)PF.me, (int)AN.ninterms));
01741         PF_CatchErrorMessagesForAll();
01742         e->numdummies = 0;
01743         for ( k = 1; k < PF.numtasks; k++ ) {
01744             PF_LongSingleReceive(PF_ANY_SOURCE, PF_ENDSORT_MSGTAG, &src, &tag);
01745             PF_LongSingleUnpack(PF_stats[src], PF_STATS_SIZE, PF_LONG);
01746             {
01747                 WORD numdummies, expchanged;
01748                 PF_LongSingleUnpack(&numdummies, 1, PF_WORD);
01749                 PF_LongSingleUnpack(&expchanged, 1, PF_WORD);
01750                 if ( e->numdummies < numdummies ) e->numdummies = numdummies;
01751                 AR.expchanged |= expchanged;
01752             }
01753             /* Now handle redefined preprocessor variables. */
01754             if ( AC.numpfirstnum > 0 ) PF_UnpackRedefinedPreVars();
01755         }
01756         /* Broadcast redefined preprocessor variables. */
01757         if ( AC.numpfirstnum > 0 ) {
01758             int RetCode = PF_BroadcastRedefinedPreVars();
01759             if ( RetCode ) return RetCode;
01760         }
01761         if ( ! AC.OldParallelStats ) {
01762             /* Now we can calculate AT.SS->GenTerms from the statistics of the slaves. */
01763             LONG genterms = 0;
01764             for ( k = 1; k < PF.numtasks; k++ ) {
01765                 genterms += PF_stats[k][3];
01766             }
01767             AT.SS->GenTerms = genterms;
01768             WriteStats(&PF_exprsize, STATSPOSTSORT, NOCHECKLOGTYPE);
01769             Expressions[AR.CurExpr].size = PF_exprsize;
01770         }
01771         PF_Statistics(PF_stats,0);
01772 /*
01773             #] Collect (stats,prepro,...):
01774             #[ Update flags :
01775 */
01776         if ( AM.S0->TermsLeft ) e->vflags &= ~ISZERO;
01777         else e->vflags |= ISZERO;
01778         if ( AR.expchanged == 0 ) e->vflags |= ISUNMODIFIED;
01779         if ( AM.S0->TermsLeft ) AR.expflags |= ISZERO;
01780         if ( AR.expchanged ) AR.expflags |= ISUNMODIFIED;
01781 /*
01782             #] Update flags :
01783         #] Master:
01784 */
01785     }
01786     else {
01787 /*
01788         #[ Slave :
01789 */
01790 /*
01791             #[ Generator Loop & EndSort :
01792
01793             loop for all terms to get from master, call Generator for each of them
01794             then call EndSort and do cleanup (to be implemented)
01795 */
01796         WORD oldBracketOn = AR.BracketOn;
01797         WORD *oldBrackBuf = AT.BrackBuf;
01798         WORD oldbracketindexflag = AT.bracketindexflag;
01799
01800         /* For redefine statements. */
01801         if ( AC.numpfirstnum > 0 ) {
01802             for ( j = 0; j < AC.numpfirstnum; j++ ) {
01803                 AC.inputnumbers[j] = -1;
```

```
01804             }
01805         }
01806 
01807         SeekScratch(AR.outfile,&position);
01808         e->onfile = position;
01809         AR.DeferFlag = AC.ComDefer;
01810         AR.Eside = RHSIDE;
01811         if ( ( e->vflags & ISFACTORIZED ) != 0 ) {
01812             AR.BracketOn = 1;
01813             AT.BrackBuf = AM.BracketFactors;
01814             AT.bracketindexflag = 1;
01815         }
01816         NewSort(BHEAD0);
01817         AR.MaxDum = AM.IndDum;
01818         AN.ninterms = 0;
01819         PF_linterms = 0;
01820         PF.parallel = 1;
01821         {
01822             FILEHANDLE *fi = AC.RhsExprInModuleFlag && PF.rhsInParallel ? &PF.slavebuf : AR.infile;
01823             fi->POfull = fi->POfill = fi->PObuffer;
01824         }
01825         /* FIXME: AN.ninterms is still broken when AN.deferskipped is non-zero.
01826          *        It still needs some work, also in PF_GetTerm(). (TU 30 Aug 2011) */
01827         while ( PF_GetTerm(term) ) {
01828             PF_linterms++; AN.ninterms++; dd = AN.deferskipped;
01829             AT.WorkPointer = term + *term;
01830             AN.RepPoint = AT.RepCount + 1;
01831             if ( ( e->vflags & ISFACTORIZED ) != 0 && term[1] == HAAKJE ) {
01832                 StoreTerm(BHEAD term);
01833                 continue;
01834             }
01835             if ( AR.DeferFlag ) {
01836                 AR.CurDum = AN.IndDum = Expressions[AR.CurExpr].numdummies + AM.IndDum;
01837             }
01838             else {
01839                 AN.IndDum = AM.IndDum;
01840                 AR.CurDum = ReNumber(BHEAD term);
01841             }
01842             if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
01843             if ( AN.ncmod ) {
01844                 if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
01845                 else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
01846             }
01847             else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
01848             if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
01849                 && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
01850                 PolyFunClean(BHEAD term);
01851             }
01852             if ( Generator(BHEAD term,0) ) {
01853                 LowerSortLevel();
01854                 Terminate(-1);
01855             }
01856             PF_linterms += dd; AN.ninterms += dd;
01857         }
01858         PF_linterms += dd; AN.ninterms += dd;
01859         {
01860             /*
01861              * EndSort() overrides AR.outfile->PObuffer etc. (See also PF_EndSort()),
01862              * but it causes a problem because
01863              *  (1) PF_EndSort() sets AR.outfile->PObuffer to a send-buffer.
01864              *  (2) RevertScratch() clears AR.infile, but then swaps buffers of AR.infile
01865              *      and AR.outfile.
01866              *  (3) RHS expressions are stored to AR.infile->PObuffer.
01867              *  (4) Again, PF_EndSort() sets AR.outfile->PObuffer, but now AR.outfile->PObuffer
01868              *      == AR.infile->PObuffer because of (1) and (2).
01869              *  (5) The result goes to AR.outfile. This breaks the RHS expressions,
01870              *      which may be needed for the next expression.
01871              * Solution: backup & restore AR.outfile->PObuffer etc. (TU 14 Sep 2011)
01872              */
01873             FILEHANDLE *fout = AR.outfile;
01874             WORD *oldbuff = fout->PObuffer;
01875             WORD *oldstop = fout->POstop;
01876             LONG  oldsize = fout->POsize;
01877             if ( EndSort(BHEAD AM.S0->sBuffer, 0) < 0 ) return -1;
01878             fout->PObuffer = oldbuff;
01879             fout->POstop   = oldstop;
01880             fout->POsize   = oldsize;
01881             fout->POfill = fout->POfull = fout->PObuffer;
01882         }
01883         AR.BracketOn = oldBracketOn;
01884         AT.BrackBuf = oldBrackBuf;
01885         AT.bracketindexflag = oldbracketindexflag;
01886 /*
01887             #] Generator Loop & EndSort :
01888             #[ Collect (stats,prepro...) :
01889 */
01890         PF_PostEndSortBarrier();
```

```
01891
01892         DBGOUT_NINTERMS(1, ("PF.me=%d AN.ninterms=%d PF_linterms=%d ENDSORT\n", (int)PF.me,
      (int)AN.ninterms, (int)PF_linterms));
01893         PF_PrepareLongSinglePack();
01894         cpu = TimeCPU(1);
01895         size = 0;
01896         PF_LongSinglePack(&cpu,              1, PF_LONG);
01897         PF_LongSinglePack(&size,             1, PF_LONG);
01898         PF_LongSinglePack(&PF_linterms,      1, PF_LONG);
01899         PF_LongSinglePack(&AM.S0->GenTerms,  1, PF_LONG);
01900         PF_LongSinglePack(&AM.S0->TermsLeft, 1, PF_LONG);
01901         {
01902             WORD numdummies = AR.MaxDum - AM.IndDum;
01903             PF_LongSinglePack(&numdummies,    1, PF_WORD);
01904             PF_LongSinglePack(&AR.expchanged, 1, PF_WORD);
01905         }
01906         /* Now handle redefined preprocessor variables. */
01907         if ( AC.numpfirstnum > 0 ) PF_PackRedefinedPreVars();
01908         PF_LongSingleSend(MASTER, PF_ENDSORT_MSGTAG);
01909         /* Broadcast redefined preprocessor variables. */
01910         if ( AC.numpfirstnum > 0 ) {
01911             int RetCode = PF_BroadcastRedefinedPreVars();
01912             if ( RetCode ) return RetCode;
01913         }
01914 /*
01915             #] Collect (stats,prepro...) :
01916
01917         This operation is moved to the beginning of each block, see PreProcessor
01918         in pre.c.
01919
01920         #] Slave :
01921 */
01922         if ( PF.log ) {
01923             UBYTE lbuf[24];
01924             NumToStr(lbuf,AC.CModule);
01925             fprintf(stderr,"[%d|%s] Endsort,Collect,Broadcast done\n",PF.me,lbuf);
01926             fflush(stderr);
01927         }
01928     }
01929     return(0);
01930 }
01931
01932 /*
01933         #] PF_Processor :
01934     #] proces.c :
01935     #[ startup :, prepro & compile
01936         #[ PF_Init :
01937 */
01938
01947 int PF_Init(int *argc, char ***argv)
01948 {
01949 /*
01950         this should definitely be somewhere else ...
01951 */
01952     PF_CurrentBracket = 0;
01953
01954     PF.numtasks = 0; /* number of tasks, is determined in PF_LibInit ! */
01955     PF.numsbufs = 2; /* might be changed by the environment variable on the master ! */
01956     PF.numrbufs = 2; /* might be changed by the environment variable on the master ! */
01957
01958     int ret = PF_LibInit(argc,argv);
01959     if (ret) { return ret; }
01960     PF_RealTime(PF_RESET);
01961
01962     PF.log = 0;
01963     PF.parallel = 0;
01964     PF_statsinterval = 10;
01965     PF.rhsInParallel=1;
01966     PF.exprbufsize=4096;/*in WORDs*/
01967
01968 #ifdef PF_WITHGETENV
01969     if ( PF.me == MASTER ) {
01970         char *c;
01971 /*
01972             get these from the environment at the moment should be in setfile/tail
01973 */
01974         if ( ( c = getenv("PF_LOG") ) != 0 ) {
01975             if ( *c ) PF.log = (int)atoi(c);
01976             else PF.log = 1;
01977             fprintf(stderr,"[%d] changing PF.log to %d\n",PF.me,PF.log);
01978             fflush(stderr);
01979         }
01980         if ( ( c = (char*)getenv("PF_RBUFS") ) != 0 ) {
01981             PF.numrbufs = (int)atoi(c);
01982             fprintf(stderr,"[%d] changing numrbufs to: %d\n",PF.me,PF.numrbufs);
01983             fflush(stderr);
01984         }
```

```
01985         if ( ( c = (char*)getenv("PF_SBUFS") ) != 0 ) {
01986             PF.numsbufs = (int)atoi(c);
01987             fprintf(stderr,"[%d] changing numsbufs to: %d\n",PF.me,PF.numsbufs);
01988             fflush(stderr);
01989         }
01990         if ( PF.numsbufs > 10 ) PF.numsbufs = 10;
01991         if ( PF.numsbufs <  1 ) PF.numsbufs = 1;
01992         if ( PF.numrbufs >  2 ) PF.numrbufs = 2;
01993         if ( PF.numrbufs <  1 ) PF.numrbufs = 1;
01994 
01995         if ( ( c = getenv("PF_STATS") ) ) {
01996             UBYTE lbuf[24];
01997             PF_statsinterval = (int)atoi(c);
01998             NumToStr(lbuf,PF_statsinterval);
01999             fprintf(stderr,"[%d] changing PF_statsinterval to %s\n",PF.me,lbuf);
02000             fflush(stderr);
02001             if ( PF_statsinterval < 1 ) PF_statsinterval = 10;
02002         }
02003     }
02004 #endif
02005 /*
02006     #[ Broadcast settings from getenv: could also be done in PF_DoSetup
02007 */
02008     if ( PF.me == MASTER ) {
02009         PF_PreparePack();
02010         PF_Pack(&PF.log,1,PF_INT);
02011         PF_Pack(&PF.numrbufs,1,PF_WORD);
02012         PF_Pack(&PF.numsbufs,1,PF_WORD);
02013     }
02014     PF_Broadcast();
02015     if ( PF.me != MASTER ) {
02016         PF_Unpack(&PF.log,1,PF_INT);
02017         PF_Unpack(&PF.numrbufs,1,PF_WORD);
02018         PF_Unpack(&PF.numsbufs,1,PF_WORD);
02019         if ( PF.log ) {
02020             fprintf(stderr, "[%d] log=%d rbufs=%d sbufs=%d\n",
02021                     PF.me, PF.log, PF.numrbufs, PF.numsbufs);
02022             fflush(stderr);
02023         }
02024     }
02025 /*
02026     #] Broadcast settings from getenv:
02027 */
02028     return(0);
02029 }
02030 /*
02031         #] PF_Init :
02032         #[ PF_PreTerminate :
02033 */
02034 
02041 void PF_PreTerminate(int errorcode)
02042 {
02043     if ( errorcode != 0 && PF_processing ) {
02044         PF_processing = 0;
02045         PF_RaiseRuntimeError();
02046     }
02047 }
02048 
02049 /*
02050         #] PF_PreTerminate :
02051         #[ PF_Terminate :
02052 */
02053 
02061 int PF_Terminate(int errorcode)
02062 {
02063     return PF_LibTerminate(errorcode);
02064 }
02065 
02066 /*
02067         #] PF_Terminate :
02068         #[ PF_GetSlaveTimes :
02069 */
02070 
02077 LONG PF_GetSlaveTimes(void)
02078 {
02079     LONG slavetimes = 0;
02080     LONG t = PF.me == MASTER ? 0 : AM.SumTime + TimeCPU(1);
02081     int ret = PF_Reduce(&t, &slavetimes, 1, PF_LONG, MPI_SUM, MASTER);
02082     CHECK(ret == 0);
02083     return slavetimes;
02084 }
02085 
02086 /*
02087         #] PF_GetSlaveTimes :
02088     #] startup :
02089     #[ PF_BroadcastNumber :
02090 */
```

```
02091
02098 LONG PF_BroadcastNumber(LONG x)
02099 {
02100 #ifdef PF_DEBUG_BCAST_LONG
02101     if ( PF.me == MASTER ) {
02102         MesPrint("» Broadcast LONG: %l", x);
02103     }
02104 #endif
02105     PF_Bcast(&x, sizeof(LONG));
02106     return x;
02107 }
02108
02109 /*
02110     #] PF_BroadcastNumber :
02111     #[ PF_BroadcastBuffer :
02112 */
02113
02125 void PF_BroadcastBuffer(WORD **buffer, LONG *length)
02126 {
02127     WORD *p;
02128     LONG rest;
02129 #ifdef PF_DEBUG_BCAST_BUF
02130     if ( PF.me == MASTER ) {
02131         MesPrint("» Broadcast Buffer: length=%l", *length);
02132     }
02133 #endif
02134     /* Initialize the buffer on the slaves. */
02135     if ( PF.me != MASTER ) {
02136         *buffer = NULL;
02137     }
02138     /* Broadcast the length of the buffer. */
02139     *length = PF_BroadcastNumber(*length);
02140     if ( *length <= 0 ) return;
02141     /* Allocate the buffer on the slaves. */
02142     if ( PF.me != MASTER ) {
02143         *buffer = (WORD *)Malloc1(*length * sizeof(WORD), "PF_BroadcastBuffer");
02144     }
02145     /* Broadcast the data in the buffer. */
02146     p = *buffer;
02147     rest = *length;
02148     while ( rest > 0 ) {
02149         int l = rest < (LONG)PF.exprbufsize ? (int)rest : PF.exprbufsize;
02150         PF_Bcast(p, l * sizeof(WORD));
02151         p += l;
02152         rest -= l;
02153     }
02154 }
02155
02156 /*
02157     #] PF_BroadcastBuffer :
02158     #[ PF_BroadcastString :
02159 */
02160
02167 int PF_BroadcastString(UBYTE *str)
02168 {
02169     int clength = 0;
02170 /*
02171         If string does not fit to the PF_buffer, it
02172         will be split into chunks. Next chunk is started at  str+clength
02173 */
02174         UBYTE *cstr=str;
02175 /*
02176         Note, compilation is performed INDEPENDENTLY on AC.mparallelflag!
02177         No if ( AC.mparallelflag == PARALLELFLAG ) !!
02178 */
02179     do {
02180         cstr += clength; /*at each step for all slaves and master */
02181
02182         if ( MASTER == PF.me ) { /*Pack str*/
02183 /*
02184                 initialize buffers
02185 */
02186             if ( PF_PreparePack() != 0 ) Terminate(-1);
02187             if ( ( clength = PF_PackString(cstr) ) <0  ) Terminate(-1);
02188         }
02189         PF_Broadcast();
02190
02191         if ( MASTER != PF.me ) {
02192 /*
02193                 Slave - unpack received string
02194                 For slaves buffers are initialised automatically.
02195 */
02196             if ( ( clength = PF_UnpackString(cstr) ) < 0 ) Terminate(-1);
02197         }
02198     } while ( cstr[clength-1] != '\0' );
02199     return (0);
02200 }
```

```
02201
02202 /*
02203     #] PF_BroadcastString :
02204     #[ PF_BroadcastPreDollar :
02205 */
02206
02222 int PF_BroadcastPreDollar(WORD **dbuffer, LONG *newsize, int *numterms)
02223 {
02224     int err = 0;
02225     LONG i;
02226 /*
02227         Note, compilation is performed INDEPENDENTLY on AC.mparallelflag!
02228         No if(AC.mparallelflag==PARALLELFLAG) !!
02229 */
02230     if ( MASTER == PF.me ) {
02231 /*
02232             The problem is that sometimes dollar variables are longer
02233             than PF_packbuf! So we split long expression into chunks.
02234             There are n filled chunks and one partially filled chunk:
02235 */
02236         LONG n = ((*newsize)+1)/PF_maxDollarChunkSize;
02237 /*
02238             ...and one more chunk for the rest; if the expression fits to
02239             the buffer without splitting, the latter will be the only one.
02240
02241             PF_maxDollarChunkSize is the maximal number of items fitted to
02242             the buffer. It is calculated in PF_LibInit() in mpi.c.
02243             PF_maxDollarChunkSize is calculated for the first step, when
02244             two fields (numterms and newsize, see below) are already packed.
02245             For simplicity, this value is used also for all steps, in
02246             despite  of it is   a bit less than maximally available space.
02247 */
02248         WORD *thechunk = *dbuffer;
02249
02250         err = PF_PreparePack();             /* initialize buffers */
02251         err |= PF_Pack(numterms,1,PF_INT);
02252         err |= PF_Pack(newsize,1,PF_LONG); /* pack the size */
02253 /*
02254             Pack and broadcast completely filled chunks.
02255             It may happen, this loop is not entered at all:
02256 */
02257         for ( i = 0; i < n; i++ ) {
02258             err |= PF_Pack(thechunk,PF_maxDollarChunkSize,PF_WORD);
02259             err |= PF_Broadcast();
02260             thechunk +=PF_maxDollarChunkSize;
02261             PF_PreparePack();
02262         }
02263 /*
02264             Pack and broadcast the rest:
02265 */
02266         if ( ( n = ( (*newsize)+1)%PF_maxDollarChunkSize ) != 0 ) {
02267             err |= PF_Pack(thechunk,n,PF_WORD);
02268             err |= PF_Broadcast();
02269         }
02270 #ifdef PF_DEBUG_BCAST_PREDOLLAR
02271         MesPrint("» Broadcast PreDollar: newsize=%d numterms=%d", (int)*newsize, *numterms);
02272 #endif
02273     }
02274     if ( MASTER != PF.me ) {  /* Slave - unpack received buffer */
02275         WORD *thechunk;
02276         LONG n, therest, thesize;
02277         err |= PF_Broadcast();
02278         err |=PF_Unpack(numterms,1,PF_INT);
02279         err |=PF_Unpack(newsize,1,PF_LONG);
02280 /*
02281             Now we know the buffer size.
02282 */
02283         thesize = (*newsize)+1;
02284 /*
02285             Evaluate the number of completely filled chunks. The last step must be
02286             treated separately, so -1:
02287 */
02288         n = (thesize/PF_maxDollarChunkSize) - 1;
02289 /*
02290             Note, here n can be <0, this is ok.
02291 */
02292         therest = thesize % PF_maxDollarChunkSize;
02293         thechunk = *dbuffer =
02294             (WORD*)Malloc1( thesize * sizeof(WORD),"$-buffer slave");
02295         if ( thechunk == NULL ) return(err|4);
02296 /*
02297             Unpack completely filled chunks and receive the next portion.
02298             It may happen, this loop is not entered at all:
02299 */
02300         for ( i = 0; i < n; i++ ) {
02301             err |= PF_Unpack(thechunk,PF_maxDollarChunkSize,PF_WORD);
02302             thechunk += PF_maxDollarChunkSize;
```

```
02303             err |= PF_Broadcast();
02304         }
02305 /*
02306             Now the last completely filled chunk:
02307 */
02308         if ( n >= 0 ) {
02309             err |= PF_Unpack(thechunk,PF_maxDollarChunkSize,PF_WORD);
02310             thechunk += PF_maxDollarChunkSize;
02311             if ( therest != 0 ) err |= PF_Broadcast();
02312         }
02313 /*
02314             Unpack the rest (it is already received!):
02315 */
02316         if ( therest != 0 ) err |= PF_Unpack(thechunk,therest,PF_WORD);
02317     }
02318     return (err);
02319 }
02320
02321 /*
02322     #] PF_BroadcastPreDollar :
02323     #[ Synchronization of modified dollar variables :
02324         #[ Helper functions :
02325             #[ dollarlen :
02326 */
02327
02331 static inline LONG dollarlen(const WORD *terms)
02332 {
02333     const WORD *p = terms;
02334     while ( *p ) p += *p;
02335     return p - terms;  /* Not including the null terminator. */
02336 }
02337
02338 /*
02339             #] dollarlen :
02340             #[ dollar_mod_type :
02341 */
02342
02347 static inline WORD dollar_mod_type(WORD index)
02348 {
02349     int i;
02350     for ( i = 0; i < NumModOptdollars; i++ )
02351         if ( ModOptdollars[i].number == index ) break;
02352     if ( i >= NumModOptdollars ) return -1;
02353     return ModOptdollars[i].type;
02354 }
02355
02356
02357 /*
02358             #] dollar_mod_type :
02359         #] Helper functions :
02360         #[ PF_CollectModifiedDollars :
02361 */
02362
02363 /*
02364             #[ dollar_to_be_collected :
02365 */
02366
02371 static inline int dollar_to_be_collected(WORD index)
02372 {
02373     switch ( dollar_mod_type(index) ) {
02374         case MODSUM:
02375         case MODMAX:
02376         case MODMIN:
02377             return 1;
02378         default:
02379             return 0;
02380     }
02381 }
02382
02383 /*
02384             #] dollar_to_be_collected :
02385             #[ copy_dollar :
02386 */
02387
02392 static inline void copy_dollar(WORD index, WORD type, const WORD *where, LONG size)
02393 {
02394     DOLLARS d = Dollars + index;
02395
02396     CleanDollarFactors(d);
02397
02398     if ( type != DOLZERO && where != NULL && where != &AM.dollarzero && where[0] != 0 && size > 0 ) {
02399         if ( size > d->size || size < d->size / 4 ) {  /* Reallocate if not enough or too much. */
02400             if ( d->where && d->where != &AM.dollarzero )
02401                 M_free(d->where, "old content of dollar");
02402             d->where = Malloc1(sizeof(WORD) * size, "copy buffer to dollar");
02403             d->size  = size;
02404         }
```

```
02405         d->type  = type;
02406         WCOPY(d->where, where, size);
02407     }
02408     else {
02409         if ( d->where && d->where != &AM.dollarzero )
02410             M_free(d->where, "old content of dollar");
02411         d->type  = DOLZERO;
02412         d->where = &AM.dollarzero;
02413         d->size  = 0;
02414     }
02415 }
02416
02417 /*
02418             #] copy_dollar :
02419             #[ compare_two_expressions :
02420 */
02421
02426 static inline int compare_two_expressions(const WORD *e1, const WORD *e2)
02427 {
02428     GETIDENTITY
02429     /*
02430      * We consider the cases that
02431      *   (1) the expression has no term,
02432      *   (2) the expression has only one term and it is a number,
02433      *   (3) otherwise.
02434      * Assume that the expressions are sorted and all terms are normalized.
02435      * The numerators of the coefficients must never be zero.
02436      *
02437      * Note that TwoExprCompare() is not adequate for our purpose
02438      * (as of 6 Aug. 2013), e.g., TwoExprCompare({0}, {4, 1, 1, -1}, LESS)
02439      * returns TRUE.
02440      */
02441     if ( e1[0] == 0 ) {
02442         if ( e2[0] == 0 ) {
02443             return(0);
02444         }
02445         else if ( e2[e2[0]] == 0 && e2[0] == ABS(e2[e2[0] - 1]) + 1 ) {
02446             if ( e2[e2[0] - 1] > 0 )
02447                 return(-1);
02448             else
02449                 return(+1);
02450         }
02451     }
02452     else if ( e1[e1[0]] == 0 && e1[0] == ABS(e1[e1[0] - 1]) + 1 ) {
02453         if ( e2[0] == 0 ) {
02454             if ( e1[e1[0] - 1] > 0 )
02455                 return(+1);
02456             else
02457                 return(-1);
02458         }
02459         else if ( e2[e2[0]] == 0 && e2[0] == ABS(e2[e2[0] - 1]) + 1 ) {
02460             return(CompCoef((WORD *)e1, (WORD *)e2));
02461         }
02462     }
02463     /* The expressions are not so simple. Define the order by each term. */
02464     while ( e1[0] && e2[0] ) {
02465         int c = CompareTerms(BHEAD (WORD *)e1, (WORD *)e2, 1);
02466         if ( c < 0 )
02467             return(-1);
02468         else if ( c > 0 )
02469             return(+1);
02470         e1 += e1[0];
02471         e2 += e2[0];
02472     }
02473     if ( e1[0] ) return(+1);
02474     if ( e2[0] ) return(-1);
02475     return(0);
02476 }
02477
02478 /*
02479             #] compare_two_expressions :
02480             #[ Variables :
02481 */
02482
02483 typedef struct {
02484     VectorStruct(WORD) buf;
02485     LONG size;
02486     WORD type;
02487 } dollar_buf;
02488
02489 /* Buffers used to store data for each variable from each slave. */
02490 static Vector(dollar_buf, dollar_slave_bufs);
02491
02492 /*
02493             #] Variables :
02494 */
02495
```

```
02509 int PF_CollectModifiedDollars(void)
02510 {
02511     int i, j, ndollars;
02512     /*
02513      * If the current module was executed in the sequential mode,
02514      * there are no modified module on the slaves.
02515      */
02516     if ( AC.mparallelflag != PARALLELFLAG && !AC.partodoflag ) return 0;
02517     /*
02518      * Count the number of (potentially) modified dollar variables, which we need to collect.
02519      * Here we need to collect all max/min/sum variables.
02520      */
02521     ndollars = 0;
02522     for ( i = 0; i < NumPotModdollars; i++ ) {
02523         WORD index = PotModdollars[i];
02524         if ( dollar_to_be_collected(index) ) ndollars++;
02525     }
02526     if ( ndollars == 0 ) return 0;  /* No dollars to be collected. */
02527
02528     if ( PF.me == MASTER ) {
02529 /*
02530             #[ Master :
02531 */
02532         int nslaves, nvars;
02533         /* Prepare receive buffers. We need ndollars*(PF.numtasks-1) buffers. */
02534         int nbufs = ndollars * (PF.numtasks - 1);
02535         VectorReserve(dollar_slave_bufs, nbufs);
02536         for ( i = VectorSize(dollar_slave_bufs); i < nbufs; i++ ) {
02537             VectorInit(VectorPtr(dollar_slave_bufs)[i].buf);
02538         }
02539         VectorSize(dollar_slave_bufs) = nbufs;
02540         /* Receive data from each slave. */
02541         for ( nslaves = 1; nslaves < PF.numtasks; nslaves++ ) {
02542             int src;
02543             PF_LongSingleReceive(PF_ANY_SOURCE, PF_DOLLAR_MSGTAG, &src, NULL);
02544             nvars = 0;
02545             for ( i = 0; i < NumPotModdollars; i++ ) {
02546                 WORD index = PotModdollars[i];
02547                 dollar_buf *b;
02548                 if ( !dollar_to_be_collected(index) ) continue;
02549                 b = &VectorPtr(dollar_slave_bufs)[(PF.numtasks - 1) * nvars + (src - 1)];
02550                 PF_LongSingleUnpack(&b->type, 1, PF_WORD);
02551                 if ( b->type != DOLZERO ) {
02552                     LONG size;
02553                     WORD *where;
02554                     PF_LongSingleUnpack(&size, 1, PF_LONG);
02555                     VectorReserve(b->buf, size + 1);
02556                     where = VectorPtr(b->buf);
02557                     PF_LongSingleUnpack(where, size, PF_WORD);
02558                     where[size] = 0;  /* The null terminator is needed. */
02559                     b->size = size + 1;  /* Including the null terminator. */
02560                     /* Note that we don't collect factored stuff for max/min/sum variables. */
02561                 }
02562                 else {
02563                     VectorReserve(b->buf, 1);
02564                     VectorPtr(b->buf)[0] = 0;
02565                     b->size = 0;
02566                 }
02567                 nvars++;
02568             }
02569         }
02570         /*
02571          * Combine received dollars. The FORM reference manual says maximum/minimum/sum
02572          * $-variables must have a numerical value, however, this routine should work also
02573          * for non-numerical cases, although the maximum/minimum value for non-numerical
02574          * terms has ambiguity.
02575          */
02576         nvars = 0;
02577         for ( i = 0; i < NumPotModdollars; i++ ) {
02578             WORD index = PotModdollars[i];
02579             WORD dtype;
02580             DOLLARS d;
02581             dollar_buf *b;
02582             if ( !dollar_to_be_collected(index) ) continue;
02583             d = Dollars + index;
02584             b = &VectorPtr(dollar_slave_bufs)[(PF.numtasks - 1) * nvars];
02585             dtype = dollar_mod_type(index);
02586             switch ( dtype ) {
02587                 case MODMAX:
02588                 case MODMIN: {
02589 /*
02590             #[ MODMAX & MODMIN :
02591 */
02592                     int selected = 0;
02593                     for ( j = 1; j < PF.numtasks - 1; j++ ) {
02594                         int c = compare_two_expressions(VectorPtr(b[j].buf),
      VectorPtr(b[selected].buf));
```

```
02595                       if ( (dtype == MODMAX && c > 0) || (dtype == MODMIN && c < 0) )
02596                           selected = j;
02597                   }
02598                   b = b + selected;
02599                   copy_dollar(index, b->type, VectorPtr(b->buf), b->size);
02600 /*
02601           #] MODMAX & MODMIN :
02602 */
02603                   break;
02604               }
02605               case MODSUM: {
02606 /*
02607           #[ MODSUM :
02608 */
02609                   GETIDENTITY
02610                   int err = 0;
02611
02612                   CBUF *C = cbuf + AM.rbufnum;
02613                   WORD *oldwork = AT.WorkPointer, *oldcterm = AN.cTerm;
02614                   WORD olddefer = AR.DeferFlag, oldnumlhs = AR.Cnumlhs, oldnumrhs = C->numrhs;
02615
02616                   LONG size;
02617                   WORD type, *dbuf;
02618
02619                   AN.cTerm = 0;
02620                   AR.DeferFlag = 0;
02621
02622                   if ( ((WORD *)((UBYTE *)AT.WorkPointer + AM.MaxTer)) > AT.WorkTop ) {
02623                       err = -1;
02624                       goto cleanup;
02625                       MesWork();
02626                   }
02627
02628                   if ( NewSort(BHEAD0) ) {
02629                       err = -1;
02630                       goto cleanup;
02631                   }
02632                   if ( NewSort(BHEAD0) ) {
02633                       LowerSortLevel();
02634                       err = -1;
02635                       goto cleanup;
02636                   }
02637
02638                   /*
02639                    * Sum up the original $-variable in the master and $-variables on all slaves.
02640                    * Note that $-variables on the slaves are set to zero at the beginning of
02641                    * the module (See also DoExecute()).
02642                    */
02643                   for ( j = 0; j < PF.numtasks; j++ ) {
02644                       const WORD *r;
02645                       for ( r = j == 0 ? Dollars[index].where : VectorPtr(b[j - 1].buf); *r; r += *r
      ) {
02646                           WCOPY(AT.WorkPointer, r, *r);
02647                           AT.WorkPointer += *r;
02648                           AR.Cnumlhs = 0;
02649                           if ( Generator(BHEAD oldwork, 0) ) {
02650                               LowerSortLevel(); LowerSortLevel();
02651                               err = -1;
02652                               goto cleanup;
02653                           }
02654                           AT.WorkPointer = oldwork;
02655                       }
02656                   }
02657
02658                   size = EndSort(BHEAD (WORD *)&dbuf, 2);
02659                   if ( size < 0 ) {
02660                       LowerSortLevel();
02661                       err = -1;
02662                       goto cleanup;
02663                   }
02664                   LowerSortLevel();
02665
02666                   /* Find special cases. */
02667                   type = DOLTERMS;
02668                   if ( dbuf[0] == 0 ) {
02669                       type = DOLZERO;
02670                   }
02671                   else if ( dbuf[dbuf[0]] == 0 ) {
02672                       const WORD *t = dbuf, *w;
02673                       WORD n, nsize;
02674                       n = *t;
02675                       nsize = t[n - 1];
02676                       if ( nsize < 0 ) nsize = -nsize;
02677                       if ( nsize == n - 1 ) {
02678                           nsize = (nsize - 1) / 2;
02679                           w = t + 1 + nsize;
02680                           if ( *w == 1 ) {
```

```
02681                                   w++; while ( w < t + n - 1 ) { if ( *w ) break; w++; }
02682                                   if ( w >= t + n - 1 ) type =  DOLNUMBER;
02683                               }
02684                               else if ( n == 7 && t[6] == 3 && t[5] == 1 && t[4] == 1 && t[1] == INDEX
      && t[2] == 3 ) {
02685                                   type = DOLINDEX;
02686                                   d->index = t[3];
02687                               }
02688                           }
02689                       }
02690                       copy_dollar(index, type, dbuf, dollarlen(dbuf) + 1);
02691                       M_free(dbuf, "temporary dollar buffer");
02692 cleanup:
02693                       AR.Cnumlhs = oldnumlhs;
02694                       C->numrhs = oldnumrhs;
02695                       AR.DeferFlag = olddefer;
02696                       AN.cTerm = oldcterm;
02697                       AT.WorkPointer = oldwork;
02698 
02699                       if ( err ) return err;
02700 /*
02701               #] MODSUM :
02702 */
02703                       break;
02704                   }
02705               }
02706               if ( d->type == DOLTERMS )
02707                   cbuf[AM.dbufnum].CanCommu[index] = numcommute(d->where,
      &cbuf[AM.dbufnum].NumTerms[index]);
02708               cbuf[AM.dbufnum].rhs[index] = d->where;
02709               nvars++;
02710 #ifdef PF_DEBUG_REDUCE_DOLLAR
02711               MesPrint("« Reduce $-var: %s", AC.dollarnames->namebuffer + d->name);
02712 #endif
02713           }
02714 /*
02715               #] Master :
02716 */
02717       }
02718       else {
02719 /*
02720               #[ Slave :
02721 */
02722           PF_PrepareLongSinglePack();
02723           /* Pack each variable. */
02724           for ( i = 0; i < NumPotModdollars; i++ ) {
02725               WORD index = PotModdollars[i];
02726               DOLLARS d;
02727               if ( !dollar_to_be_collected(index) ) continue;
02728               d = Dollars + index;
02729               PF_LongSinglePack(&d->type, 1, PF_WORD);
02730               if ( d->type != DOLZERO ) {
02731                   /*
02732                    * NOTE: d->size is the allocated buffer size for d->where in WORDs.
02733                    *       So dollarlen(d->where) can be < d->size-1. (TU 15 Dec 2011)
02734                    */
02735                   LONG size = dollarlen(d->where);
02736                   PF_LongSinglePack(&size, 1, PF_LONG);
02737                   PF_LongSinglePack(d->where, size, PF_WORD);
02738                   /* Note that we don't collect factored stuff for max/min/sum variables. */
02739               }
02740           }
02741           PF_LongSingleSend(MASTER, PF_DOLLAR_MSGTAG);
02742 /*
02743               #] Slave :
02744 */
02745       }
02746       return 0;
02747 }
02748 
02749 /*
02750           #] PF_CollectModifiedDollars :
02751           #[ PF_BroadcastModifiedDollars :
02752 */
02753 
02754 /*
02755               #[ dollar_to_be_broadcast :
02756 */
02757 
02762 static inline int dollar_to_be_broadcast(WORD index)
02763 {
02764       switch ( dollar_mod_type(index) ) {
02765           case MODLOCAL:
02766               return 0;
02767           default:
02768               return 1;
02769       }
```

```
02770 }
02771
02772 /*
02773             #] dollar_to_be_broadcast :
02774 */
02775
02788 int PF_BroadcastModifiedDollars(void)
02789 {
02790     int i, j, ndollars;
02791     /*
02792      * Count the number of (potentially) modified dollar variables, which we need to broadcast.
02793      * Here we need to broadcast all non-local variables.
02794      */
02795     ndollars = 0;
02796     for ( i = 0; i < NumPotModdollars; i++ ) {
02797         WORD index = PotModdollars[i];
02798         if ( dollar_to_be_broadcast(index) ) ndollars++;
02799     }
02800     if ( ndollars == 0 ) return 0;  /* No dollars to be broadcast. */
02801
02802     if ( PF.me == MASTER ) {
02803 /*
02804             #[ Master :
02805 */
02806         PF_PrepareLongMultiPack();
02807         /* Pack each variable. */
02808         for ( i = 0; i < NumPotModdollars; i++ ) {
02809             WORD index = PotModdollars[i];
02810             DOLLARS d;
02811             if ( !dollar_to_be_broadcast(index) ) continue;
02812             d = Dollars + index;
02813             PF_LongMultiPack(&d->type, 1, PF_WORD);
02814             if ( d->type != DOLZERO ) {
02815                 /*
02816                  * NOTE: d->size is the allocated buffer size for d->where in WORDs.
02817                  *       So dollarlen(d->where) can be < d->size-1. (TU 15 Dec 2011)
02818                  */
02819                 LONG size = dollarlen(d->where);
02820                 PF_LongMultiPack(&size, 1, PF_LONG);
02821                 PF_LongMultiPack(d->where, size, PF_WORD);
02822                 /* ...and the factored stuff. */
02823                 PF_LongMultiPack(&d->nfactors, 1, PF_WORD);
02824                 if ( d->nfactors > 1 ) {
02825                     for ( j = 0; j < d->nfactors; j++ ) {
02826                         FACDOLLAR *f = &d->factors[j];
02827                         PF_LongMultiPack(&f->type, 1, PF_WORD);
02828                         PF_LongMultiPack(&f->size, 1, PF_LONG);
02829                         if ( f->size > 0 )
02830                             PF_LongMultiPack(f->where, f->size, PF_WORD);
02831                         else
02832                             PF_LongMultiPack(&f->value, 1, PF_WORD);
02833                     }
02834                 }
02835             }
02836 #ifdef PF_DEBUG_BCAST_DOLLAR
02837             MesPrint("» Broadcast $-var: %s", AC.dollarnames->namebuffer + d->name);
02838 #endif
02839         }
02840 /*
02841             #] Master :
02842 */
02843     }
02844     if ( PF_LongMultiBroadcast() ) return -1;
02845     if ( PF.me != MASTER ) {
02846 /*
02847             #[ Slave :
02848 */
02849         for ( i = 0; i < NumPotModdollars; i++ ) {
02850             WORD index = PotModdollars[i];
02851             DOLLARS d;
02852             if ( !dollar_to_be_broadcast(index) ) continue;
02853             d = Dollars + index;
02854             /* Clear the contents of the dollar variable. */
02855             if ( d->where && d->where != &AM.dollarzero )
02856                 M_free(d->where, "old content of dollar");
02857             d->where = &AM.dollarzero;
02858             d->size = 0;
02859             CleanDollarFactors(d);
02860             /* Unpack and store the contents. */
02861             PF_LongMultiUnpack(&d->type, 1, PF_WORD);
02862             if ( d->type != DOLZERO ) {
02863                 LONG size;
02864                 PF_LongMultiUnpack(&size, 1, PF_LONG);
02865                 d->size = size + 1;
02866                 d->where = (WORD *)Malloc1(sizeof(WORD) * d->size, "dollar content");
02867                 PF_LongMultiUnpack(d->where, size, PF_WORD);
02868                 d->where[size] = 0;  /* The null terminator is needed. */
```

```
02869                 /* ...and the factored stuff. */
02870                 PF_LongMultiUnpack(&d->nfactors, 1, PF_WORD);
02871                 if ( d->nfactors > 1 ) {
02872                     d->factors = (FACDOLLAR *)Malloc1(sizeof(FACDOLLAR) * d->nfactors, "dollar
      factored stuff");
02873                     for ( j = 0; j < d->nfactors; j++ ) {
02874                         FACDOLLAR *f = &d->factors[j];
02875                         PF_LongMultiUnpack(&f->type, 1, PF_WORD);
02876                         PF_LongMultiUnpack(&f->size, 1, PF_LONG);
02877                         if ( f->size > 0 ) {
02878                             f->where = (WORD *)Malloc1(sizeof(WORD) * (f->size + 1), "dollar factor
      content");
02879                             PF_LongMultiUnpack(f->where, f->size, PF_WORD);
02880                             f->where[f->size] = 0;  /* The null terminator is needed. */
02881                             f->value = 0;
02882                         }
02883                         else {
02884                             f->where = NULL;
02885                             PF_LongMultiUnpack(&f->value, 1, PF_WORD);
02886                         }
02887                     }
02888                 }
02889             }
02890             if ( d->type == DOLTERMS )
02891                 cbuf[AM.dbufnum].CanCommu[index] = numcommute(d->where,
      &cbuf[AM.dbufnum].NumTerms[index]);
02892             cbuf[AM.dbufnum].rhs[index] = d->where;
02893         }
02894 /*
02895             #] Slave :
02896 */
02897     }
02898     return 0;
02899 }
02900
02901 /*
02902         #] PF_BroadcastModifiedDollars :
02903     #] Synchronization of modified dollar variables :
02904     #[ Synchronization of redefined preprocessor variables :
02905         #[ Variables :
02906 */
02907
02908 /* A buffer used in receivers. */
02909 static Vector(UBYTE, prevarbuf);
02910
02911 /*
02912         #] Variables :
02913         #[ PF_PackRedefinedPreVars :
02914 */
02915
02924 static void PF_PackRedefinedPreVars(void)
02925 {
02926     int i;
02927     /* First, pack the number of redefined preprocessor variables. */
02928     int nredefs = 0;
02929     for ( i = 0; i < AC.numpfirstnum; i++ )
02930         if ( AC.inputnumbers[i] >= 0 ) nredefs++;
02931     PF_LongSinglePack(&nredefs, 1, PF_INT);
02932     /* Then, pack each variable. */
02933     for ( i = 0; i < AC.numpfirstnum; i++ )
02934         if ( AC.inputnumbers[i] >= 0) {
02935             WORD index = AC.pfirstnum[i];
02936             UBYTE *value = PreVar[index].value;
02937             int bytes = strlen((char *)value);
02938             PF_LongSinglePack(&index, 1, PF_WORD);
02939             PF_LongSinglePack(&bytes, 1, PF_INT);
02940             PF_LongSinglePack(value, bytes, PF_BYTE);
02941             PF_LongSinglePack(&AC.inputnumbers[i], 1, PF_LONG);
02942     }
02943 }
02944
02945 /*
02946         #] PF_PackRedefinedPreVars :
02947         #[ PF_UnpackRedefinedPreVars :
02948 */
02949
02959 static void PF_UnpackRedefinedPreVars(void)
02960 {
02961     int i, j;
02962     /* Unpack the number of redefined preprocessor variables. */
02963     int nredefs;
02964     PF_LongSingleUnpack(&nredefs, 1, PF_INT);
02965     if ( nredefs > 0 ) {
02966         /* Then unpack each variable. */
02967         for ( i = 0; i < nredefs; i++ ) {
02968             WORD index;
02969             int bytes;
```

```
02970             UBYTE *value;
02971             LONG inputnumber;
02972             PF_LongSingleUnpack(&index, 1, PF_WORD);
02973             PF_LongSingleUnpack(&bytes, 1, PF_INT);
02974             VectorReserve(prevarbuf, bytes + 1);
02975             value = VectorPtr(prevarbuf);
02976             PF_LongSingleUnpack(value, bytes, PF_BYTE);
02977             value[bytes] = '\0';  /* The null terminator is needed. */
02978             PF_LongSingleUnpack(&inputnumber, 1, PF_LONG);
02979             /* Put this variable if it must be updated. */
02980             for ( j = 0; j < AC.numpfirstnum; j++ )
02981                 if ( AC.pfirstnum[j] == index ) break;
02982             if ( AC.inputnumbers[j] < inputnumber ) {
02983                 AC.inputnumbers[j] = inputnumber;
02984                 PutPreVar(PreVar[index].name, value, NULL, 1);
02985             }
02986         }
02987     }
02988 }
02989
02990 /*
02991         #] PF_UnpackRedefinedPreVars :
02992         #[ PF_BroadcastRedefinedPreVars :
02993 */
02994
03005 int PF_BroadcastRedefinedPreVars(void)
03006 {
03007     /*
03008      * NOTE: Because the compilation is performed on the all processes
03009      * independently on AC.mparallelflag, we always have to broadcast redefined
03010      * preprocessor variables from the master to the all slaves.
03011      */
03012     if ( PF.me == MASTER ) {
03013 /*
03014             #[ Master :
03015 */
03016         int i, nredefs;
03017         PF_PrepareLongMultiPack();
03018         /* First, pack the number of redefined preprocessor variables. */
03019         nredefs = 0;
03020         for ( i = 0; i < AC.numpfirstnum; i++ )
03021             if ( AC.inputnumbers[i] >= 0 ) nredefs++;
03022         PF_LongMultiPack(&nredefs, 1, PF_INT);
03023         /* Then, pack each variable. */
03024         for ( i = 0; i < AC.numpfirstnum; i++ )
03025             if ( AC.inputnumbers[i] >= 0) {
03026                 WORD index = AC.pfirstnum[i];
03027                 UBYTE *value = PreVar[index].value;
03028                 int bytes = strlen((char *)value);
03029                 PF_LongMultiPack(&index, 1, PF_WORD);
03030                 PF_LongMultiPack(&bytes, 1, PF_INT);
03031                 PF_LongMultiPack(value, bytes, PF_BYTE);
03032 #ifdef PF_DEBUG_BCAST_PREVAR
03033                 MesPrint("» Broadcast PreVar: %s = \"%s\"", PreVar[index].name, value);
03034 #endif
03035             }
03036 /*
03037             #] Master :
03038 */
03039     }
03040     if ( PF_LongMultiBroadcast() ) return -1;
03041     if ( PF.me != MASTER ) {
03042 /*
03043             #[ Slave :
03044 */
03045         int i, nredefs;
03046         /* Unpack the number of redefined preprocessor variables. */
03047         PF_LongMultiUnpack(&nredefs, 1, PF_INT);
03048         if ( nredefs > 0 ) {
03049             /* Then unpack each variable and put it. */
03050             for ( i = 0; i < nredefs; i++ ) {
03051                 WORD index;
03052                 int bytes;
03053                 UBYTE *value;
03054                 PF_LongMultiUnpack(&index, 1, PF_WORD);
03055                 PF_LongMultiUnpack(&bytes, 1, PF_INT);
03056                 VectorReserve(prevarbuf, bytes + 1);
03057                 value = VectorPtr(prevarbuf);
03058                 PF_LongMultiUnpack(value, bytes, PF_BYTE);
03059                 value[bytes] = '\0';  /* The null terminator is needed. */
03060                 PutPreVar(PreVar[index].name, value, NULL, 1);
03061             }
03062         }
03063 /*
03064             #] Slave :
03065 */
03066     }
```

```
03067     return 0;
03068 }
03069
03070 /*
03071         #] PF_BroadcastRedefinedPreVars :
03072     #] Synchronization of redefined preprocessor variables :
03073     #[ Preprocessor Inside instruction :
03074         #[ Variables :
03075 */
03076
03077 /* Saved values of AC.RhsExprInModuleFlag, PotModdollars and AC.pfirstnum. */
03078 static WORD oldRhsExprInModuleFlag;
03079 static Vector(WORD, oldPotModdollars);
03080 static Vector(WORD, oldpfirstnum);
03081
03082 /*
03083         #] Variables :
03084         #[ PF_StoreInsideInfo :
03085 */
03086
03087 /*
03088  * Saves the current values of AC.RhsExprInModuleFlag, PotModdollars
03089  * and AC.pfirstnum.
03090  *
03091  * Called by DoInside().
03092  *
03093  * @return  0  if OK, nonzero on error.
03094  */
03095 int PF_StoreInsideInfo(void)
03096 {
03097     int i;
03098     oldRhsExprInModuleFlag = AC.RhsExprInModuleFlag;
03099     VectorClear(oldPotModdollars);
03100     for ( i = 0; i < NumPotModdollars; i++ )
03101         VectorPushBack(oldPotModdollars, PotModdollars[i]);
03102     VectorClear(oldpfirstnum);
03103     for ( i = 0; i < AC.numpfirstnum; i++ )
03104         VectorPushBack(oldpfirstnum, AC.pfirstnum[i]);
03105     return 0;
03106 }
03107
03108 /*
03109         #] PF_StoreInsideInfo :
03110         #[ PF_RestoreInsideInfo :
03111 */
03112
03113 /*
03114  * Restores the saved values of AC.RhsExprInModuleFlag, PotModdollars
03115  * and AC.pfirstnum.
03116  *
03117  * Called by DoEndInside().
03118  *
03119  * @return  0  if OK, nonzero on error.
03120  */
03121 int PF_RestoreInsideInfo(void)
03122 {
03123     int i;
03124     AC.RhsExprInModuleFlag = oldRhsExprInModuleFlag;
03125     NumPotModdollars = VectorSize(oldPotModdollars);
03126     for ( i = 0; i < NumPotModdollars; i++ )
03127         PotModdollars[i] = VectorPtr(oldPotModdollars)[i];
03128     AC.numpfirstnum = VectorSize(oldpfirstnum);
03129     for ( i = 0; i < AC.numpfirstnum; i++ )
03130         AC.pfirstnum[i] = VectorPtr(oldpfirstnum)[i];
03131     return 0;
03132 }
03133
03134 /*
03135         #] PF_RestoreInsideInfo :
03136     #] Preprocessor Inside instruction :
03137     #[ PF_BroadcastCBuf :
03138 */
03139
03147 int PF_BroadcastCBuf(int bufnum)
03148 {
03149     CBUF *C = cbuf + bufnum;
03150     int i;
03151     LONG l;
03152     if ( PF.me == MASTER ) {
03153 /*
03154         #[ Master :
03155 */
03156         PF_PrepareLongMultiPack();
03157         /* Pack CBUF struct except pointers. */
03158         PF_LongMultiPack(&C->BufferSize, 1, PF_LONG);
03159         PF_LongMultiPack(&C->numlhs, 1, PF_INT);
03160         PF_LongMultiPack(&C->numrhs, 1, PF_INT);
```

```
03161         PF_LongMultiPack(&C->maxlhs, 1, PF_INT);
03162         PF_LongMultiPack(&C->maxrhs, 1, PF_INT);
03163         PF_LongMultiPack(&C->mnumlhs, 1, PF_INT);
03164         PF_LongMultiPack(&C->mnumrhs, 1, PF_INT);
03165         PF_LongMultiPack(&C->numtree, 1, PF_INT);
03166         PF_LongMultiPack(&C->rootnum, 1, PF_INT);
03167         PF_LongMultiPack(&C->MaxTreeSize, 1, PF_INT);
03168         /* Now pointers. Pointer, lhs and rhs are packed as offsets. We don't pack Top. */
03169         l = C->Pointer - C->Buffer;
03170         PF_LongMultiPack(&l, 1, PF_LONG);
03171         PF_LongMultiPack(C->Buffer, l, PF_WORD);
03172         for ( i = 0; i < C->numlhs + 1; i++ ) {
03173             l = C->lhs[i] - C->Buffer;
03174             PF_LongMultiPack(&l, 1, PF_LONG);
03175         }
03176         for ( i = 0; i < C->numrhs + 1; i++ ) {
03177             l = C->rhs[i] - C->Buffer;
03178             PF_LongMultiPack(&l, 1, PF_LONG);
03179         }
03180         PF_LongMultiPack(C->CanCommu, C->numrhs + 1, PF_LONG);
03181         PF_LongMultiPack(C->NumTerms, C->numrhs + 1, PF_LONG);
03182         PF_LongMultiPack(C->numdum, C->numrhs + 1, PF_WORD);
03183         PF_LongMultiPack(C->dimension, C->numrhs + 1, PF_WORD);
03184         if ( C->MaxTreeSize > 0 )
03185             PF_LongMultiPack(C->boomlijst, (C->numtree + 1) * (sizeof(COMPTREE) / sizeof(int)),
      PF_INT);
03186 #ifdef PF_DEBUG_BCAST_CBUF
03187         MesPrint("» Broadcast CBuf %d", bufnum);
03188 #endif
03189 /*
03190         #] Master :
03191 */
03192     }
03193     if ( PF_LongMultiBroadcast() ) return -1;
03194     if ( PF.me != MASTER ) {
03195 /*
03196         #[ Slave :
03197 */
03198         /* First, free already allocated buffers. */
03199         finishcbuf(bufnum);
03200         /* Unpack CBUF struct except pointers. */
03201         PF_LongMultiUnpack(&C->BufferSize, 1, PF_LONG);
03202         PF_LongMultiUnpack(&C->numlhs, 1, PF_INT);
03203         PF_LongMultiUnpack(&C->numrhs, 1, PF_INT);
03204         PF_LongMultiUnpack(&C->maxlhs, 1, PF_INT);
03205         PF_LongMultiUnpack(&C->maxrhs, 1, PF_INT);
03206         PF_LongMultiUnpack(&C->mnumlhs, 1, PF_INT);
03207         PF_LongMultiUnpack(&C->mnumrhs, 1, PF_INT);
03208         PF_LongMultiUnpack(&C->numtree, 1, PF_INT);
03209         PF_LongMultiUnpack(&C->rootnum, 1, PF_INT);
03210         PF_LongMultiUnpack(&C->MaxTreeSize, 1, PF_INT);
03211         /* Allocate new buffers. */
03212         C->Buffer = (WORD *)Malloc1(C->BufferSize * sizeof(WORD), "compiler buffer");
03213         C->Top = C->Buffer + C->BufferSize;
03214         C->lhs = (WORD **)Malloc1(C->maxlhs * sizeof(WORD *), "compiler buffer");
03215         C->rhs = (WORD **)Malloc1(C->maxrhs * (sizeof(WORD *) + 2 * sizeof(LONG) + 2 * sizeof(WORD)),
      "compiler buffer");
03216         C->CanCommu = (LONG *)(C->rhs + C->maxrhs);
03217         C->NumTerms = C->CanCommu + C->maxrhs;
03218         C->numdum = (WORD *)(C->NumTerms + C->maxrhs);
03219         C->dimension = C->numdum + C->maxrhs;
03220         if ( C->MaxTreeSize > 0 )
03221             C->boomlijst = (COMPTREE *)Malloc1(C->MaxTreeSize * sizeof(COMPTREE), "compiler buffer");
03222         /* Unpack buffers. */
03223         PF_LongMultiUnpack(&l, 1, PF_LONG);
03224         PF_LongMultiUnpack(C->Buffer, l, PF_WORD);
03225         C->Pointer = C->Buffer + l;
03226         for ( i = 0; i < C->numlhs + 1; i++ ) {
03227             PF_LongMultiUnpack(&l, 1, PF_LONG);
03228             C->lhs[i] = C->Buffer + l;
03229         }
03230         for ( i = 0; i < C->numrhs + 1; i++ ) {
03231             PF_LongMultiUnpack(&l, 1, PF_LONG);
03232             C->rhs[i] = C->Buffer + l;
03233         }
03234         PF_LongMultiUnpack(C->CanCommu, C->numrhs + 1, PF_LONG);
03235         PF_LongMultiUnpack(C->NumTerms, C->numrhs + 1, PF_LONG);
03236         PF_LongMultiUnpack(C->numdum, C->numrhs + 1, PF_WORD);
03237         PF_LongMultiUnpack(C->dimension, C->numrhs + 1, PF_WORD);
03238         if ( C->MaxTreeSize > 0 )
03239             PF_LongMultiUnpack(C->boomlijst, (C->numtree + 1) * (sizeof(COMPTREE) / sizeof(int)),
      PF_INT);
03240 /*
03241         #] Slave :
03242 */
03243     }
03244     return 0;
```

```
03245 }
03246
03247 /*
03248     #] PF_BroadcastCBuf :
03249     #[ PF_BroadcastExpFlags :
03250 */
03251
03258 int PF_BroadcastExpFlags(void)
03259 {
03260     WORD i;
03261     EXPRESSIONS e;
03262     if ( PF.me == MASTER ) {
03263 /*
03264         #[ Master :
03265 */
03266         PF_PrepareLongMultiPack();
03267         PF_LongMultiPack(&AR.expflags, 1, PF_WORD);
03268         for ( i = 0; i < NumExpressions; i++ ) {
03269             e = &Expressions[i];
03270             PF_LongMultiPack(&e->counter,    1, PF_WORD);
03271             PF_LongMultiPack(&e->vflags,     1, PF_WORD);
03272             PF_LongMultiPack(&e->uflags,     1, PF_WORD);
03273             PF_LongMultiPack(&e->numdummies, 1, PF_WORD);
03274             PF_LongMultiPack(&e->numfactors, 1, PF_WORD);
03275 #ifdef PF_DEBUG_BCAST_EXPRFLAGS
03276             MesPrint("» Broadcast ExprFlags: %s", AC.exprnames->namebuffer + e->name);
03277 #endif
03278         }
03279 /*
03280         #] Master :
03281 */
03282     }
03283     if ( PF_LongMultiBroadcast() ) return -1;
03284     if ( PF.me != MASTER ) {
03285 /*
03286         #[ Slave :
03287 */
03288         PF_LongMultiUnpack(&AR.expflags, 1, PF_WORD);
03289         for ( i = 0; i < NumExpressions; i++ ) {
03290             e = &Expressions[i];
03291             PF_LongMultiUnpack(&e->counter,    1, PF_WORD);
03292             PF_LongMultiUnpack(&e->vflags,     1, PF_WORD);
03293             PF_LongMultiUnpack(&e->uflags,     1, PF_WORD);
03294             PF_LongMultiUnpack(&e->numdummies, 1, PF_WORD);
03295             PF_LongMultiUnpack(&e->numfactors, 1, PF_WORD);
03296         }
03297 /*
03298         #] Slave :
03299 */
03300     }
03301     return 0;
03302 }
03303
03304 /*
03305     #] PF_BroadcastExpFlags :
03306     #[ PF_SetScratch :
03307 */
03308
03315 static void PF_SetScratch(FILEHANDLE *f,POSITION *position)
03316 {
03317     if(
03318             ( f->handle >= 0) && ISGEPOS(*position,f->POposition) &&
03319             ( ISGEPOSINC(*position,f->POposition,(f->POfull-f->PObuffer)*sizeof(WORD)) ==0 )
03320         )/*position is inside the buffer! SetScratch() will do nothing.*/
03321             f->POfull=f->PObuffer;/*force SetScratch() to re-read the position from the beginning:*/
03322     SetScratch(f,position);
03323 }
03324
03325 /*
03326     #] PF_SetScratch :
03327     #[ PF_pushScratch :
03328 */
03329
03336 static int PF_pushScratch(FILEHANDLE *f)
03337 {
03338     LONG size,RetCode;
03339     if ( f->handle < 0){
03340         /*Create the file*/
03341         if ( ( RetCode = CreateFile(f->name) ) >= 0 ) {
03342             f->handle = (WORD)RetCode;
03343             PUTZERO(f->filesize);
03344             PUTZERO(f->POposition);
03345         }
03346         else{
03347             MesPrint("Cannot create scratch file %s",f->name);
03348             return(-1);
03349         }
```

```
03350     }/*if ( f->handle < 0)*/
03351     size = (f->POfill-f->PObuffer)*sizeof(WORD);
03352     if( size > 0 ){
03353         SeekFile(f->handle,&(f->POposition),SEEK_SET);
03354         if ( WriteFile(f->handle,(UBYTE *)(f->PObuffer),size) != size ){
03355             MesPrint("Error while writing to disk. Disk full?");
03356             return(-1);
03357         }
03358         ADDPOS(f->filesize,size);
03359         ADDPOS(f->POposition,size);
03360         f->POfill = f->POfull=f->PObuffer;
03361     }/*if( size > 0 )*/
03362     return(0);
03363 }
03364
03365 /*
03366     #] PF_pushScratch :
03367     #[ Broadcasting RHS expressions :
03368         #[ PF_WalkThroughExprMaster :
03369     Returns <=0 if the expression is ready, or dl+1;
03370 */
03371
03372 static int PF_WalkThroughExprMaster(FILEHANDLE *curfile, int dl)
03373 {
03374     LONG l=0;
03375     for(;;){
03376         if(curfile->POfull-curfile->POfill < dl){
03377             POSITION pos;
03378             SeekScratch(curfile,&pos);
03379             PF_SetScratch(curfile,&pos);
03380         }/*if(curfile->POfull-curfile->POfill < dl)*/
03381         curfile->POfill+=dl;
03382         l+=dl;
03383         if( l >= PF.exprbufsize){
03384             if( l == PF.exprbufsize){
03385                 if( *(curfile->POfill) == 0)/*expression is ready*/
03386                     return(0);
03387                 }
03388             l-=PF.exprbufsize;
03389             curfile->POfill-=l;
03390             return l+1;
03391         }
03392
03393         dl=*(curfile->POfill);
03394         if(dl == 0)
03395             return l-PF.exprbufsize;
03396
03397         if(dl<0){/*compressed term*/
03398             if(curfile->POfull-curfile->POfill < 1){
03399                 POSITION pos;
03400                 SeekScratch(curfile,&pos);
03401                 PF_SetScratch(curfile,&pos);
03402             }/*if(curfile->POfull-curfile->POfill < 1)*/
03403             dl=*(curfile->POfill+1)+2;
03404         }/*if(*(curfile->POfill)<0)*/
03405     }/*for(;;)*/
03406 }
03407
03408 /*
03409         #] PF_WalkThroughExprMaster :
03410         #[ PF_WalkThroughExprSlave :
03411     Returns <=0 if the expression is ready, or dl+1;
03412 */
03413
03414 static int PF_WalkThroughExprSlave(FILEHANDLE *curfile, LONG *counter, int dl)
03415 {
03416     LONG l=0;
03417     for(;;){
03418         if(curfile->POstop-curfile->POfill < dl){
03419             if(PF_pushScratch(curfile))
03420                 return(-PF.exprbufsize-1);
03421         }
03422         curfile->POfill+=dl;
03423         curfile->POfull=curfile->POfill;
03424         l+=dl;
03425         if( l >= PF.exprbufsize){
03426             if( l == PF.exprbufsize){
03427                 /*
03428                  * This access is valid because PF.exprbufsize+1 WORDs are
03429                  * broadcasted, this shortcut is not mandatory though. (TU 15 Sep 2011)
03430                  */
03431                 if( *(curfile->POfill) == 0)/*expression is ready*/
03432                     return(0);
03433                 }
03434             l-=PF.exprbufsize;
03435             curfile->POfill-=l;
03436             curfile->POfull=curfile->POfill;
```

```
03437             return l+1;
03438         }
03439
03440         dl=*(curfile->POfill);
03441         if(dl == 0)
03442             return l-PF.exprbufsize;
03443         (*counter)++;
03444         if(dl<0){/*compressed term*/
03445             if(curfile->POstop-curfile->POfill < 1){
03446                 if(PF_pushScratch(curfile))
03447                     return(-PF.exprbufsize-1);
03448             }
03449             /*
03450              * This access is always valid because PF.exprbufsize+1 WORDs are
03451              * broadcasted. (TU 15 Sep 2011)
03452              */
03453             dl=*(curfile->POfill+1)+2;
03454         }/*if(*(curfile->POfill)<0)*/
03455     }/*for(;;)*/
03456 }
03457
03458 /*
03459         #] PF_WalkThroughExprSlave :
03460         #[ PF_rhsBCastMaster :
03461 */
03462
03470 static int PF_rhsBCastMaster(FILEHANDLE *curfile, EXPRESSIONS e)
03471 {
03472     LONG l=1;/*PF_WalkThroughExpr returns length + 1*/
03473     SetScratch(curfile,&(e->onfile));
03474     do{
03475         /*
03476          * We need to broadcast PF.exprbufsize+1 WORDs because PF_WalkThroughExprSlave
03477          * may access to an additional 1 WORD. It is better to rewrite the routines
03478          * in such a way as to broadcast only PF.exprbufsize WORDs. (TU 15 Sep 2011)
03479          */
03480         if ( curfile->POfull - curfile->POfill < PF.exprbufsize + 1 ) {
03481             POSITION pos;
03482             SeekScratch(curfile,&pos);
03483             PF_SetScratch(curfile,&pos);
03484         }
03485         if ( PF_Bcast(curfile->POfill, (PF.exprbufsize + 1) * sizeof(WORD)) )
03486             return -1;
03487         l=PF_WalkThroughExprMaster(curfile,l-1);
03488     }while(l>0);
03489     if(l<0)/*The tail is extra, decrease POfill*/
03490         curfile->POfill-=l;
03491     return(0);
03492 }
03493
03494 /*
03495         #] PF_rhsBCastMaster :
03496         #[ PF_rhsBCastSlave :
03497 */
03498
03507 static int PF_rhsBCastSlave(FILEHANDLE *curfile, EXPRESSIONS e)
03508 {
03509     LONG l=1;/*PF_WalkThroughExpr returns length + 1*/
03510     LONG counter = 0;
03511     do{
03512         /*
03513          * We need to broadcast PF.exprbufsize+1 WORDs because PF_WalkThroughExprSlave
03514          * may access to an additional 1 WORD. It is better to rewrite the routines
03515          * in such a way as to broadcast only PF.exprbufsize WORDs. (TU 15 Sep 2011)
03516          */
03517         if ( curfile->POstop - curfile->POfill < PF.exprbufsize + 1 ) {
03518             if(PF_pushScratch(curfile))
03519                 return(-1);
03520         }
03521         if ( PF_Bcast(curfile->POfill, (PF.exprbufsize + 1) * sizeof(WORD)) )
03522             return(-1);
03523         l = PF_WalkThroughExprSlave(curfile, &counter, l - 1);
03524     }while(l>0);
03525     if(l<0){/*The tail is extra, decrease POfill*/
03526         if(l<-PF.exprbufsize)/*error due to a PF_pushScratch() failure */
03527             return(-1);
03528         curfile->POfill-=l;
03529     }
03530     if ( curfile->handle >= 0 ) {
03531         if ( PF_pushScratch(curfile) ) return -1;
03532     }
03533     curfile->POfull=curfile->POfill;
03534     if ( curfile != AR.hidefile ) AR.InInBuf = curfile->POfull-curfile->PObuffer;
03535     else                          AR.InHiBuf = curfile->POfull-curfile->PObuffer;
03536     CHECK(counter == e->counter + 1);  /* The first term is the prototype. */
03537     return(0);
03538 }
```

```
03539
03540 /*
03541         #] PF_rhsBCastSlave :
03542         #[ PF_BroadcastExpr :
03543 */
03544
03552 int PF_BroadcastExpr(EXPRESSIONS e, FILEHANDLE *file)
03553 {
03554     if ( PF.me == MASTER ) {
03555         if ( PF_rhsBCastMaster(file, e) ) return -1;
03556 #ifdef PF_DEBUG_BCAST_RHSEXPR
03557         MesPrint("» Broadcast RhsExpr: %s", AC.exprnames->namebuffer + e->name);
03558 #endif
03559     }
03560     else {
03561         POSITION pos;
03562         SetEndHScratch(file, &pos);
03563         e->onfile = pos;
03564         if ( PF_rhsBCastSlave(file, e) ) return -1;
03565     }
03566     return 0;
03567 }
03568
03569 /*
03570         #] PF_BroadcastExpr :
03571         #[ PF_BroadcastRHS :
03572 */
03573
03580 int PF_BroadcastRHS(void)
03581 {
03582     int i;
03583     for ( i = 0; i < NumExpressions; i++ ) {
03584         EXPRESSIONS e = &Expressions[i];
03585         if ( !(e->vflags & ISINRHS) ) continue;
03586         switch ( e->status ) {
03587             case LOCALEXPRESSION:
03588             case SKIPLEXPRESSION:
03589             case DROPLEXPRESSION:
03590             case GLOBALEXPRESSION:
03591             case SKIPGEXPRESSION:
03592             case DROPGEXPRESSION:
03593             case HIDELEXPRESSION:
03594             case HIDEGEXPRESSION:
03595             case INTOHIDELEXPRESSION:
03596             case INTOHIDEGEXPRESSION:
03597                 if ( PF_BroadcastExpr(e, AR.infile) ) return -1;
03598                 break;
03599             case HIDDENLEXPRESSION:
03600             case HIDDENGEXPRESSION:
03601             case DROPHLEXPRESSION:
03602             case DROPHGEXPRESSION:
03603             case UNHIDELEXPRESSION:
03604             case UNHIDEGEXPRESSION:
03605                 if ( PF_BroadcastExpr(e, AR.hidefile) ) return -1;
03606                 break;
03607         }
03608     }
03609     if ( PF.me != MASTER )
03610         UpdatePositions();
03611     return 0;
03612 }
03613
03614 /*
03615         #] PF_BroadcastRHS :
03616     #] Broadcasting RHS expressions :
03617     #[ InParallel mode :
03618         #[ PF_InParallelProcessor :
03619 */
03620
03627 int PF_InParallelProcessor(void)
03628 {
03629     GETIDENTITY
03630     int i, next,tag;
03631     EXPRESSIONS e;
03632     /*
03633      * Skip expressions with zero terms. All the master and slaves need to
03634      * change the "partodo" flag.
03635      */
03636     if ( PF.numtasks >= 3 ) {
03637         for ( i = 0; i < NumExpressions; i++ ) {
03638             e = Expressions + i;
03639             if ( e->partodo > 0 && e->counter == 0 ) {
03640                 e->partodo = 0;
03641             }
03642         }
03643     }
03644     PF_processing = 1;
```

```
03645     if(PF.me == MASTER){
03646         if ( PF.numtasks >= 3 ) {
03647             partodoexr = (WORD*)Malloc1(sizeof(WORD)*(PF.numtasks+1),"PF_InParallelProcessor");
03648             for ( i = 0; i < NumExpressions; i++ ) {
03649                 e = Expressions+i;
03650                 if ( e->partodo <= 0 ) continue;
03651                 switch(e->status){
03652                     case LOCALEXPRESSION:
03653                     case GLOBALEXPRESSION:
03654                     case UNHIDELEXPRESSION:
03655                     case UNHIDEGEXPRESSION:
03656                     case INTOHIDELEXPRESSION:
03657                     case INTOHIDEGEXPRESSION:
03658                         tag=PF_ANY_SOURCE;
03659                         next=PF_Wait4SlaveIP(&tag);
03660                         if(next<0)
03661                             return(-1);
03662                         if(tag == PF_DATA_MSGTAG){
03663                             PF_Statistics(PF_stats,0);
03664                             if(PF_Slave2MasterIP(next))
03665                                 return(-1);
03666                         }
03667                         if(PF_Master2SlaveIP(next,e))
03668                             return(-1);
03669                         partodoexr[next]=i;
03670                         break;
03671                     default:
03672                         e->partodo = 0;
03673                         continue;
03674                 }/*switch(e->status)*/
03675             }/*for ( i = 0; i < NumExpressions; i++ )*/
03676             /*Here some slaves are working, other are waiting on PF_Send.
03677                 Wait all of them.*/
03678             /*At this point no new slaves may be launched so PF_WaitAllSlaves()
03679                 does not modify partodoexr[].*/
03680             if(PF_WaitAllSlaves())
03681                 return(-1);
03682 
03683             if ( AC.CollectFun ) AR.DeferFlag = 0;
03684             if(partodoexr){
03685                 M_free(partodoexr,"PF_InParallelProcessor");
03686                 partodoexr=NULL;
03687             }/*if(partodoexr)*/
03688         }/*if ( PF.numtasks >= 3 ) */
03689         else {
03690             for ( i = 0; i < NumExpressions; i++ ) {
03691                 Expressions[i].partodo = 0;
03692             }
03693         }
03694         PF_PostEndSortBarrier();
03695         return(0);
03696     }/*if(PF.me == MASTER)*/
03697     /*Slave:*/
03698     if(PF_Wait4MasterIP(PF_EMPTY_MSGTAG))
03699         return(-1);
03700     /*master is ready to listen to me*/
03701     do{
03702         WORD *oldwork= AT.WorkPointer;
03703         tag=PF_ReadMaster();/*reads directly to its scratch!*/
03704         if(tag<0)
03705             return(-1);
03706         if(tag == PF_DATA_MSGTAG){
03707             oldwork = AT.WorkPointer;
03708 
03709             /* For redefine statements. */
03710             if ( AC.numpfirstnum > 0 ) {
03711                 int j;
03712                 for ( j = 0; j < AC.numpfirstnum; j++ ) {
03713                     AC.inputnumbers[j] = -1;
03714                 }
03715             }
03716 
03717             if(PF_DoOneExpr())/*the processor*/
03718                 return(-1);
03719             if(PF_Wait4MasterIP(PF_DATA_MSGTAG))
03720                 return(-1);
03721             if(PF_Slave2MasterIP(PF.me))/*both master and slave*/
03722                 return(-1);
03723             AT.WorkPointer=oldwork;
03724         }/*if(tag == PF_DATA_MSGTAG)*/
03725     }while(tag!=PF_EMPTY_MSGTAG);
03726     PF.exprtodo=-1;
03727     PF_PostEndSortBarrier();
03728     return(0);
03729 }/*PF_InParallelProcessor*/
03730 
03731 /*
```

```
03732         #] PF_InParallelProcessor :
03733         #[ PF_Wait4MasterIP :
03734 */
03735
03736 static int PF_Wait4MasterIP(int tag)
03737 {
03738     int follow = 0;
03739     LONG cpu,space = 0;
03740
03741     if(PF.log){
03742         fprintf(stderr,"[%d] Starting to send to Master\n",PF.me);
03743         fflush(stderr);
03744     }
03745
03746     PF_PreparePack();
03747     cpu = TimeCPU(1);
03748     PF_Pack(&cpu               ,1,PF_LONG);
03749     PF_Pack(&space             ,1,PF_LONG);
03750     PF_Pack(&PF_linterms       ,1,PF_LONG);
03751     PF_Pack(&(AM.S0->GenTerms) ,1,PF_LONG);
03752     PF_Pack(&(AM.S0->TermsLeft),1,PF_LONG);
03753     PF_Pack(&follow            ,1,PF_INT );
03754
03755     if(PF.log){
03756         fprintf(stderr,"[%d] Now sending with tag = %d\n",PF.me,tag);
03757         fflush(stderr);
03758     }
03759
03760     PF_Send(MASTER, tag);
03761
03762     if(PF.log){
03763         fprintf(stderr,"[%d] returning from send\n",PF.me);
03764         fflush(stderr);
03765     }
03766     return(0);
03767 }
03768 /*
03769         #] PF_Wait4MasterIP :
03770         #[ PF_DoOneExpr :
03771 */
03772
03780 static int PF_DoOneExpr(void)/*the processor*/
03781 {
03782                 GETIDENTITY
03783                 EXPRESSIONS e;
03784                 int i;
03785                 WORD *term;
03786                 POSITION position, outposition;
03787                 FILEHANDLE *fi, *fout;
03788                 LONG dd = 0;
03789                 WORD oldBracketOn = AR.BracketOn;
03790                 WORD *oldBrackBuf = AT.BrackBuf;
03791                 WORD oldbracketindexflag = AT.bracketindexflag;
03792                 e = Expressions + PF.exprtodo;
03793                 i = PF.exprtodo;
03794                 AR.CurExpr = i;
03795                 AR.SortType = AC.SortType;
03796                 AR.expchanged = 0;
03797                 if ( ( e->vflags & ISFACTORIZED ) != 0 ) {
03798                     AR.BracketOn = 1;
03799                     AT.BrackBuf = AM.BracketFactors;
03800                     AT.bracketindexflag = 1;
03801                 }
03802
03803                 position = AS.OldOnFile[i];
03804                 if ( e->status == HIDDENLEXPRESSION || e->status == HIDDENGEXPRESSION
03805                     || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION ) {
03806                     AR.GetFile = 2; fi = AR.hidefile;
03807                 }
03808                 else {
03809                     AR.GetFile = 0; fi = AR.infile;
03810                 }
03811 /*
03812                 PUTZERO(fi->POposition);
03813                 if ( fi->handle >= 0 ) {
03814                     fi->POfill = fi->POfull = fi->PObuffer;
03815                 }
03816 */
03817                 SetScratch(fi,&position);
03818                 term = AT.WorkPointer;
03819                 AR.CompressPointer = AR.CompressBuffer;
03820                 AR.CompressPointer[0] = 0;
03821                 AR.KeptInHold = 0;
03822                 if ( GetTerm(BHEAD term) <= 0 ) {
03823                     MesPrint("Expression %d has problems in scratchfile",i);
03824                     Terminate(-1);
03825                 }
```

```
03826                 if ( AT.bracketindexflag > 0 ) OpenBracketIndex(i);
03827                 term[3] = i;
03828                 PUTZERO(outposition);
03829                 fout = AR.outfile;
03830                 fout->POfill = fout->POfull = fout->PObuffer;
03831                 fout->POposition = outposition;
03832                 if ( fout->handle >= 0 ) {
03833                     fout->POposition = outposition;
03834                 }
03835 /*
03836                 The next statement is needed because we need the system
03837                 to believe that the expression is at position zero for
03838                 the moment. In this worker, with no memory of other expressions,
03839                 it is. This is needed for when a bracket index is made
03840                 because there e->onfile is an offset. Afterwards, when the
03841                 expression is written to its final location in the masters
03842                 output e->onfile will get its real value.
03843 */
03844                 PUTZERO(e->onfile);
03845                 if ( PutOut(BHEAD term,&outposition,fout,0) < 0 ) return -1;
03846
03847                 AR.DeferFlag = AC.ComDefer;
03848
03849 /*              AR.sLevel = AB[0]->R.sLevel;*/
03850                 term = AT.WorkPointer;
03851                 NewSort(BHEAD0);
03852                 AR.MaxDum = AM.IndDum;
03853                 AN.ninterms = 0;
03854                 while ( GetTerm(BHEAD term) ) {
03855                   SeekScratch(fi,&position);
03856                   AN.ninterms++; dd = AN.deferskipped;
03857                   if ( ( e->vflags & ISFACTORIZED ) != 0 && term[1] == HAAKJE ) {
03858                       StoreTerm(BHEAD term);
03859                   }
03860                   else {
03861                   if ( AC.CollectFun && *term <= (AM.MaxTer/(2*(LONG)sizeof(WORD))) ) {
03862                     if ( GetMoreTerms(term) < 0 ) {
03863                       LowerSortLevel(); return(-1);
03864                     }
03865                     SeekScratch(fi,&position);
03866                   }
03867                   AT.WorkPointer = term + *term;
03868                   AN.RepPoint = AT.RepCount + 1;
03869                   if ( AR.DeferFlag ) {
03870                     AR.CurDum = AN.IndDum = Expressions[PF.exprtodo].numdummies;
03871                   }
03872                   else {
03873                     AN.IndDum = AM.IndDum;
03874                     AR.CurDum = ReNumber(BHEAD term);
03875                   }
03876                   if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
03877                   if ( AN.ncmod ) {
03878                     if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
03879                     else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
03880                   }
03881                   else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
03882                   if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
03883                         && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
03884                         PolyFunClean(BHEAD term);
03885                   }
03886                   if ( Generator(BHEAD term,0) ) {
03887                     LowerSortLevel(); return(-1);
03888                   }
03889                   AN.ninterms += dd;
03890                   }
03891                   SetScratch(fi,&position);
03892                   if ( fi == AR.hidefile ) {
03893                     AR.InHiBuf = (fi->POfull-fi->PObuffer)
03894                         -DIFBASE(position,fi->POposition)/sizeof(WORD);
03895                   }
03896                   else {
03897                     AR.InInBuf = (fi->POfull-fi->PObuffer)
03898                         -DIFBASE(position,fi->POposition)/sizeof(WORD);
03899                   }
03900                 }
03901                 AN.ninterms += dd;
03902                 if ( EndSort(BHEAD AM.S0->sBuffer,0) < 0 ) return(-1);
03903                 e->numdummies = AR.MaxDum - AM.IndDum;
03904                 AR.BracketOn = oldBracketOn;
03905                 AT.BrackBuf = oldBrackBuf;
03906                 if ( ( e->vflags & TOBEFACTORED ) != 0 )
03907                         poly_factorize_expression(e);
03908                 else if ( ( ( e->vflags & TOBEUNFACTORED ) != 0 )
03909                  && ( ( e->vflags & ISFACTORIZED ) != 0 ) )
03910                         poly_unfactorize_expression(e);
03911                 if ( AM.S0->TermsLeft )   e->vflags &= ~ISZERO;
03912                 else                      e->vflags |= ISZERO;
```

```
03913                 if ( AR.expchanged == 0 ) e->vflags |= ISUNMODIFIED;
03914 /*              if ( AM.S0->TermsLeft ) AR.expflags |= ISZERO;
03915                 if ( AR.expchanged )    AR.expflags |= ISUNMODIFIED;*/
03916                 AR.GetFile = 0;
03917                 AT.bracketindexflag = oldbracketindexflag;
03918
03919                 fout->POfull = fout->POfill;
03920     return(0);
03921 }
03922
03923 /*
03924         #] PF_DoOneExpr :
03925         #[ PF_Slave2MasterIP :
03926 */
03927
03928 typedef struct bufIPstruct {
03929     LONG i;
03930     struct ExPrEsSiOn e;
03931 } bufIPstruct_t;
03932
03933 static int PF_Slave2MasterIP(int src)/*both master and slave*/
03934 {
03935     EXPRESSIONS e;
03936     bufIPstruct_t exprData;
03937     int i,l;
03938     FILEHANDLE *fout=AR.outfile;
03939     POSITION pos;
03940     /*Here we know the length of data to send in advance:
03941         slave has the only one expression in its scratch file, and it sends
03942         this information to the master.*/
03943     if(PF.me != MASTER){/*slave*/
03944         e = Expressions + PF.exprtodo;
03945         /*Fill in the expression data:*/
03946         memcpy(&(exprData.e), e, sizeof(struct ExPrEsSiOn));
03947         SeekScratch(fout,&pos);
03948         exprData.i=BASEPOSITION(pos);
03949         /*Send the metadata:*/
03950         if(PF_RawSend(MASTER,&exprData,sizeof(bufIPstruct_t),0))
03951             return(-1);
03952         i=exprData.i;
03953         SETBASEPOSITION(pos,0);
03954         do{
03955             int blen=PF.exprbufsize*sizeof(WORD);
03956             if(i<blen)
03957                 blen=i;
03958             l=PF_SendChunkIP(fout,&pos, MASTER, blen);
03959             /*Here always l == blen!*/
03960             if(l<0)
03961                 return(-1);
03962             ADDPOS(pos,l);
03963             i-=l;
03964         }while(i>0);
03965         if ( fout->handle >= 0 ) { /* Now get rid of the file */
03966             CloseFile(fout->handle);
03967             fout->handle = -1;
03968             remove(fout->name);
03969             PUTZERO(fout->POposition);
03970             PUTZERO(fout->filesize);
03971             fout->POfill = fout->POfull = fout->PObuffer;
03972         }
03973         /* Now handle redefined preprocessor variables. */
03974         if ( AC.numpfirstnum > 0 ) {
03975             PF_PrepareLongSinglePack();
03976             PF_PackRedefinedPreVars();
03977             PF_LongSingleSend(MASTER, PF_MISC_MSGTAG);
03978         }
03979         return(0);
03980     }/*if(PF.me != MASTER)*/
03981     /*Master*/
03982     /*partodoexr[src] is the number of expression.*/
03983     e = Expressions +partodoexr[src];
03984     /*Get metadata:*/
03985     i = PF_ANY_MSGTAG;
03986     PF_CatchErrorMessages(&src, &i);
03987     if (PF_RawRecv(&src, &exprData,sizeof(bufIPstruct_t),&i)!= sizeof(bufIPstruct_t))
03988         return(-1);
03989     /*Fill in the expression data:*/
03990 /*  memcpy(e, &(exprData.e), sizeof(struct ExPrEsSiOn)); */
03991     e->counter    = exprData.e.counter;
03992     e->vflags     = exprData.e.vflags;
03993     e->uflags     = exprData.e.uflags;
03994     e->numdummies = exprData.e.numdummies;
03995     e->numfactors = exprData.e.numfactors;
03996     if ( !(e->vflags & ISZERO) )       AR.expflags |= ISZERO;
03997     if ( !(e->vflags & ISUNMODIFIED) ) AR.expflags |= ISUNMODIFIED;
03998     SeekScratch(fout,&pos);
03999     e->onfile = pos;
```

```
04000     i=exprData.i;
04001     while(i>0){
04002         int blen=PF.exprbufsize*sizeof(WORD);
04003         if(i<blen)
04004             blen=i;
04005         l=PF_RecvChunkIP(fout,src,blen);
04006         /*Here always l == blen!*/
04007         if(l<0)
04008             return(-1);
04009         i-=l;
04010     }
04011     /* Now handle redefined preprocessor variables. */
04012     if ( AC.numpfirstnum > 0 ) {
04013         PF_LongSingleReceive(src, PF_MISC_MSGTAG, NULL, NULL);
04014         PF_UnpackRedefinedPreVars();
04015     }
04016     return(0);
04017 }
04018
04019 /*
04020         #] PF_Slave2MasterIP :
04021         #[ PF_Master2SlaveIP :
04022 */
04023
04024 static int PF_Master2SlaveIP(int dest, EXPRESSIONS e)
04025 {
04026     bufIPstruct_t exprData;
04027     FILEHANDLE *fi;
04028     POSITION pos;
04029     int l;
04030     LONG ll=0,count=0;
04031     WORD *t;
04032     if(e==NULL){/*Say to the slave that no more job:*/
04033         if(PF_RawSend(dest,&exprData,sizeof(bufIPstruct_t),PF_EMPTY_MSGTAG))
04034             return(-1);
04035         return(0);
04036     }
04037     memcpy(&(exprData.e), e, sizeof(struct ExPrEsSiOn));
04038     exprData.i=e-Expressions;
04039     if ( AC.StatsFlag && AC.OldParallelStats ) {
04040         MesPrint("");
04041         MesPrint(" Sending expression %s to slave %d",EXPRNAME(exprData.i),dest);
04042     }
04043     if(PF_RawSend(dest,&exprData,sizeof(bufIPstruct_t),PF_DATA_MSGTAG))
04044         return(-1);
04045     if ( e->status == HIDDENLEXPRESSION || e->status == HIDDENGEXPRESSION
04046         || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION )
04047         fi = AR.hidefile;
04048     else
04049         fi = AR.infile;
04050     pos=e->onfile;
04051     SetScratch(fi,&pos);
04052     do{
04053         l=PF_SendChunkIP(fi, &pos, dest, PF.exprbufsize*sizeof(WORD));
04054         if(l<0)
04055             return(-1);
04056         t=fi->PObuffer+ (DIFBASE(pos,fi->POposition))/sizeof(WORD);
04057         ll=PF_WalkThrough(t,ll,l/sizeof(WORD),&count);
04058         ADDPOS(pos,l);
04059     }while(ll>-2);
04060     return(0);
04061 }
04062
04063 /*
04064         #] PF_Master2SlaveIP :
04065         #[ PF_ReadMaster :
04066 */
04067
04068 static int PF_ReadMaster(void)/*reads directly to its scratch!*/
04069 {
04070     bufIPstruct_t exprData;
04071     int tag,m=MASTER;
04072     EXPRESSIONS e;
04073     FILEHANDLE *fi;
04074     POSITION pos;
04075     LONG count=0;
04076     WORD *t;
04077     LONG ll=0;
04078     int l;
04079     /*Get metadata:*/
04080     tag = PF_ANY_MSGTAG;
04081     PF_CatchErrorMessages(&m, &tag);
04082     if (PF_RawRecv(&m, &exprData,sizeof(bufIPstruct_t),&tag)!= sizeof(bufIPstruct_t))
04083         return(-1);
04084
04085     if(tag == PF_EMPTY_MSGTAG)/*No data, no job*/
04086         return(tag);
```

```
04087
04088     /*data expected, tag must be == PF_DATA_MSGTAG!*/
04089     PF.exprtodo=exprData.i;
04090     e=Expressions + PF.exprtodo;
04091     /*Fill in the expression data:*/
04092 /*  memcpy(e, &(exprData.e), sizeof(struct ExPrEsSiOn)); */
04093     if ( e->status == HIDDENLEXPRESSION || e->status == HIDDENGEXPRESSION
04094         || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION )
04095         fi = AR.hidefile;
04096     else
04097         fi = AR.infile;
04098     SetEndHScratch(fi,&pos);
04099     e->onfile=AS.OldOnFile[PF.exprtodo]=pos;
04100
04101     do{
04102         l=PF_RecvChunkIP(fi,MASTER,PF.exprbufsize*sizeof(WORD));
04103         if(l<0)
04104             return(-1);
04105         t=fi->POfull-l/sizeof(WORD);
04106         ll=PF_WalkThrough(t,ll,l/sizeof(WORD),&count);
04107     }while(ll>-2);
04108     /*Now -ll-2 is the number of "extra" elements transferred from the master.*/
04109     fi->POfull-=-ll-2;
04110     fi->POfill=fi->POfull;
04111     return(PF_DATA_MSGTAG);
04112 }
04113
04114 /*
04115         #] PF_ReadMaster :
04116         #[ PF_SendChunkIP :
04117     thesize is in bytes. Returns the number of sent bytes or <0 on error:
04118 */
04119
04120 static int PF_SendChunkIP(FILEHANDLE *curfile, POSITION *position, int to, LONG thesize)
04121 {
04122     LONG l=thesize;
04123     if(
04124         ISLESSPOS(*position,curfile->POposition) ||
04125         ISGEPOSINC(*position,curfile->POposition,
04126         ((curfile->POfull-curfile->PObuffer)*sizeof(WORD)-thesize) )
04127     ){
04128         if(curfile->handle< 0)
04129             l=(curfile->POfull-curfile->PObuffer)*sizeof(WORD) - (LONG)(position->p1);
04130         else{
04131             PF_SetScratch(curfile,position);
04132             if(
04133                 ISGEPOSINC(*position,curfile->POposition,
04134                 ((curfile->POfull-curfile->PObuffer)*sizeof(WORD)-thesize) )
04135                 )
04136             l=(curfile->POfull-curfile->PObuffer)*sizeof(WORD) - (LONG)position->p1;
04137         }
04138     }
04139     /*Now we are able to sent l bytes from the
04140         curfile->PObuffer[position-curfile->POposition]*/
04141     if(PF_RawSend(to,curfile->PObuffer+ (DIFBASE(*position,curfile->POposition))/sizeof(WORD),l,0))
04142         return(-1);
04143     return(l);
04144 }
04145
04146 /*
04147         #] PF_SendChunkIP :
04148         #[ PF_RecvChunkIP :
04149     thesize is in bytes. Returns the number of sent bytes or <0 on error:
04150 */
04151
04152 static int PF_RecvChunkIP(FILEHANDLE *curfile, int from, LONG thesize)
04153 {
04154     LONG receivedBytes;
04155
04156     if( (LONG)((curfile->POstop - curfile->POfull)*sizeof(WORD)) < thesize )
04157         if(PF_pushScratch(curfile))
04158             return(-1);
04159     /*Now there is enough space from curfile->POfill to curfile->POstop*/
04160     {/*Block:*/
04161         int tag=0;
04162         receivedBytes=PF_RawRecv(&from,curfile->POfull,thesize,&tag);
04163     }/*:Block*/
04164     if(receivedBytes >= 0 ){
04165         curfile->POfull+=receivedBytes/sizeof(WORD);
04166         curfile->POfill=curfile->POfull;
04167     }/*if(receivedBytes >= 0 )*/
04168     return(receivedBytes);
04169 }
04170
04171 /*
04172         #] PF_RecvChunkIP :
04173         #[ PF_WalkThrough :
```

```
04174     Returns:
04175     >=  0 -- initial offset,
04176         -1 -- the first element of t contains the length of the tail of compressed term,
04177     <= -2 -- -(d+2), where d is the number of extra transferred elements.
04178     Expects:
04179     l -- initial offset or -1,
04180     chunk -- number of transferred elements (not bytes!)
04181     *count -- incremented each time a new term is found
04182 */
04183
04184 static int PF_WalkThrough(WORD *t, LONG l, LONG chunk, LONG *count)
04185 {
04186     if(l<0) /*==-1!*/
04187         l=(*t)+1;/*the first element of t contains the length of
04188                           the tail of compressed term*/
04189     else{
04190         if(l>=chunk)/*next term is out of the chunk*/
04191             return(l-chunk);
04192         t+=l;
04193         chunk-=l;/*note, l was less than chunk so chunk >0!*/
04194         l=*t;
04195     }
04196     /*Main loop:*/
04197     while(l!=0){
04198         if(l>0){/*an offset to the next term*/
04199             if(l<chunk){
04200                 t+=l;
04201                 chunk-=l;/*note, l was less than chunk so chunk >0!*/
04202                 l=*t;
04203                 (*count)++;
04204             }/*if(l<chunk)*/
04205             else
04206                 return(l-chunk);
04207         }/*if(l>0)*/
04208         else{ /* l<0 */
04209             if(chunk < 2)/*i.e., chunk == 1*/
04210                 return(-1);/*the first WORD in the next chunk is length of the tail of the compressed
      term*/
04211             l=*(t+1)+2;/*+2 since
04212                     1. t points to the length field -1,
04213                     2. the size of a tail of compressed term is equal to the number of WORDs in this
      tail*/
04214         }
04215     }/*while(l!=0)*/
04216     return(-1-chunk);/* -(2+(chunk-1)), chunk>0 ! */
04217 }
04218
04219 /*
04220         #] PF_WalkThrough :
04221     #] InParallel mode :
04222     #[ PF_SendFile :
04223 */
04224
04225 #define PF_SNDFILEBUFSIZE 4096
04226
04234 int PF_SendFile(int to, FILE *fd)
04235 {
04236     size_t len=0;
04237     if(fd == NULL){
04238         if(PF_RawSend(to,&to,sizeof(int),PF_EMPTY_MSGTAG))
04239             return(-1);
04240         return(0);
04241     }
04242     for(;;){
04243         char buf[PF_SNDFILEBUFSIZE];
04244         size_t l;
04245         l=fread(buf, 1, PF_SNDFILEBUFSIZE, fd);
04246         len+=l;
04247         if(l==PF_SNDFILEBUFSIZE){
04248             if(PF_RawSend(to,buf,PF_SNDFILEBUFSIZE,PF_BUFFER_MSGTAG))
04249                 return(-1);
04250         }
04251         else{
04252             if(PF_RawSend(to,buf,l,PF_ENDBUFFER_MSGTAG))
04253                 return(-1);
04254             break;
04255         }
04256     }/*for(;;)*/
04257     return(len);
04258 }
04259
04260 /*
04261     #] PF_SendFile :
04262     #[ PF_RecvFile :
04263 */
04264
04272 int PF_RecvFile(int from, FILE *fd)
```

```
04273 {
04274     size_t len=0;
04275     int tag;
04276     do{
04277         char buf[PF_SNDFILEBUFSIZE];
04278         int l;
04279             l=PF_RawRecv(&from,buf,PF_SNDFILEBUFSIZE,&tag);
04280             if(l<0)
04281                 return(-1);
04282             if(tag == PF_EMPTY_MSGTAG)
04283                 return(-1);
04284 
04285             if( fwrite(buf,1,l,fd)!=l )
04286                 return(-1);
04287             len+=l;
04288     }while(tag!=PF_ENDBUFFER_MSGTAG);
04289     return(len);
04290 }
04291 
04292 /*
04293     #] PF_RecvFile :
04294     #[ Synchronised output :
04295         #[ Explanations :
04296 */
04297 
04298 /*
04299  * If the master and slaves output statistics or error messages to the same stream
04300  * or file (e.g., the standard output or the log file) simultaneously, then
04301  * a mixing of their outputs can occur. To avoid this, TFORM uses a lock of
04302  * ErrorMessageLock, but there is no locking functionality in the original MPI
04303  * specification. We need to synchronise the output from the master and slaves.
04304  *
04305  * The idea of the synchronised output (by, e.g., MesPrint()) implemented here is
04306  *   Slaves:
04307  *     1. Save the output by WriteFile() (set to PF_WriteFileToFile())
04308  *        into some buffers between MLOCK(ErrorMessageLock) and
04309  *        MUNLOCK(ErrorMessageLock), which call PF_MLock() and PF_MUnlock(),
04310  *        respectively. The output for AM.StdOut and AC.LogHandle are saved to
04311  *        the buffers.
04312  *     2. At MUNLOCK(ErrorMessageLock), send the output in the buffer to the master,
04313  *        with PF_STDOUT_MSGTAG or PF_LOG_MSGTAG.
04314  *   Master:
04315  *     1. Receive the buffered output from slaves, and write them by
04316  *        WriteFileToFile().
04317  *   The main problem is how and where the master receives messages from
04318  *   the slaves (PF_ReceiveErrorMessage()). For this purpose there are three
04319  *   helper functions: PF_CatchErrorMessages() and PF_CatchErrorMessagesForAll()
04320  *   which remove messages with PF_STDOUT_MSGTAG or PF_LOG_MSGTAG from the top
04321  *   of the message queue, and PF_ProbeWithCatchingErrorMessages() which is same as
04322  *   PF_Probe() except removing these messages.
04323  */
04324 
04325 /*
04326         #] Explanations :
04327         #[ Variables :
04328 */
04329 
04330 static int errorMessageLock = 0;     /* (slaves) The lock count. See PF_MLock() and PF_MUnlock(). */
04331 static Vector(UBYTE, stdoutBuffer);  /* (slaves) The buffer for AM.StdOut. */
04332 static Vector(UBYTE, logBuffer);     /* (slaves) The buffer for AC.LogHandle. */
04333 #define recvBuffer logBuffer         /* (master) The buffer for receiving messages. */
04334 
04335 /*
04336  * If PF_ENABLE_STDOUT_BUFFERING is defined, the master performs the line buffering
04337  * (using stdoutBuffer) at PF_WriteFileToFile().
04338  */
04339 #ifndef PF_ENABLE_STDOUT_BUFFERING
04340 #ifdef UNIX
04341 #define PF_ENABLE_STDOUT_BUFFERING
04342 #endif
04343 #endif
04344 
04345 /*
04346         #] Variables :
04347         #[ PF_MLock :
04348 */
04349 
04353 void PF_MLock(void)
04354 {
04355     assert(PF.me != MASTER);
04356     if ( errorMessageLock++ > 0 ) return;
04357     VectorClear(stdoutBuffer);
04358     VectorClear(logBuffer);
04359 }
04360 
04361 /*
04362         #] PF_MLock :
```

```
04363         #[ PF_MUnlock :
04364 */
04365
04369 void PF_MUnlock(void)
04370 {
04371     assert(PF.me != MASTER);
04372     if ( --errorMessageLock > 0 ) return;
04373     if ( !VectorEmpty(stdoutBuffer) ) {
04374         PF_RawSend(MASTER, VectorPtr(stdoutBuffer), VectorSize(stdoutBuffer), PF_STDOUT_MSGTAG);
04375     }
04376     if ( !VectorEmpty(logBuffer) ) {
04377         PF_RawSend(MASTER, VectorPtr(logBuffer), VectorSize(logBuffer), PF_LOG_MSGTAG);
04378     }
04379 }
04380
04381 /*
04382         #] PF_MUnlock :
04383         #[ PF_WriteFileToFile :
04384 */
04385
04398 LONG PF_WriteFileToFile(int handle, UBYTE *buffer, LONG size)
04399 {
04400     if ( PF.me != MASTER && errorMessageLock > 0 ) {
04401         if ( handle == AM.StdOut ) {
04402             VectorPushBacks(stdoutBuffer, buffer, size);
04403             return size;
04404         }
04405         else if ( handle == AC.LogHandle ) {
04406             VectorPushBacks(logBuffer, buffer, size);
04407             return size;
04408         }
04409     }
04410 #ifdef PF_ENABLE_STDOUT_BUFFERING
04411     /*
04412      * On my computer, sometimes a single linefeed "\n" sent to the standard
04413      * output is ignored on the execution of mpiexec. A typical example is:
04414      *   $ cat foo.c
04415      *     #include <unistd.h>
04416      *     int main() {
04417      *       write(1, "    ", 4);
04418      *       write(1, "\n", 1);
04419      *       write(1, "    ", 4);
04420      *       write(1, "123\n", 4);
04421      *       return 0;
04422      *     }
04423      * or even as a shell script:
04424      *   $ cat foo.sh
04425      *     #! bin/sh
04426      *     printf "    "
04427      *     printf "\n"
04428      *     printf "    "
04429      *     printf "123\n"
04430      * When I ran it on mpiexec
04431      *   $ while :; do mpiexec -np 1 ./foo.sh; done
04432      * I observed the single linefeed (printf "\n") was sometimes ignored. Even
04433      * though this phenomenon might be specific to my environment, I added this
04434      * code because someone may encounter a similar phenomenon and feel it
04435      * frustrating. (TU 16 Jun 2011)
04436      *
04437      * Phenomenon:
04438      *   A single linefeed sent to the standard output occasionally ignored
04439      *   on mpiexec.
04440      *
04441      * Environment:
04442      *   openSUSE 11.4 (x86_64)
04443      *   kernel: 2.6.37.6-0.5-desktop
04444      *   gcc: 4.5.1 20101208
04445      *   mpich2-1.3.2p1 configured with '--enable-shared --with-pm=smpd'
04446      *
04447      * Solution:
04448      *   In Unix (in which Uwrite() calls write() system call without any buffering),
04449      *   we perform the line buffering here. A single linefeed is also buffered.
04450      *
04451      * XXX:
04452      *   At the end of the program the buffered output (text without LF) will not be flushed,
04453      *   i.e., will not be written to the standard output. This is not problematic at a normal run.
04454      *   The buffer can be explicitly flushed by PF_FlushStdOutBuffer().
04455      */
04456     if ( PF.me == MASTER && handle == AM.StdOut ) {
04457         size_t oldsize;
04458         /* Assume the newline character is LF (when UNIX is defined). */
04459         if ( (size > 0 && buffer[size - 1] != LINEFEED) || (size == 1 && buffer[0] == LINEFEED) ) {
04460             VectorPushBacks(stdoutBuffer, buffer, size);
04461             return size;
04462         }
04463         if ( (oldsize = VectorSize(stdoutBuffer)) > 0 ) {
04464             LONG ret;
```

```
04465             VectorPushBacks(stdoutBuffer, buffer, size);
04466             ret = WriteFileToFile(handle, VectorPtr(stdoutBuffer), VectorSize(stdoutBuffer));
04467             VectorClear(stdoutBuffer);
04468             if ( ret < 0 ) {
04469                 return ret;
04470             }
04471             else if ( ret < (LONG)oldsize ) {
04472                 return 0;  /* This means the buffered output in previous calls is lost. */
04473             }
04474             else {
04475                 return ret - (LONG)oldsize;
04476             }
04477         }
04478     }
04479 #endif
04480     return WriteFileToFile(handle, buffer, size);
04481 }
04482
04483 /*
04484         #] PF_WriteFileToFile :
04485         #[ PF_FlushStdOutBuffer :
04486 */
04487
04492 void PF_FlushStdOutBuffer(void)
04493 {
04494 #ifdef PF_ENABLE_STDOUT_BUFFERING
04495     if ( PF.me == MASTER && VectorSize(stdoutBuffer) > 0 ) {
04496         WriteFileToFile(AM.StdOut, VectorPtr(stdoutBuffer), VectorSize(stdoutBuffer));
04497         VectorClear(stdoutBuffer);
04498     }
04499 #endif
04500 }
04501
04502 /*
04503         #] PF_FlushStdOutBuffer :
04504         #[ PF_ReceiveErrorMessage :
04505 */
04506
04515 static void PF_ReceiveErrorMessage(int src, int tag)
04516 {
04517     assert(PF.me == MASTER);
04518     int size;
04519     int ret = PF_RawProbe(&src, &tag, &size);
04520     CHECK(ret == 0);
04521     switch ( tag ) {
04522         case PF_STDOUT_MSGTAG:
04523         case PF_LOG_MSGTAG:
04524             VectorReserve(recvBuffer, size);
04525             ret = PF_RawRecv(&src, VectorPtr(recvBuffer), size, &tag);
04526             CHECK(ret == size);
04527             if ( size > 0 ) {
04528                 int handle = (tag == PF_STDOUT_MSGTAG) ? AM.StdOut : AC.LogHandle;
04529 #ifdef PF_ENABLE_STDOUT_BUFFERING
04530                 if ( handle == AM.StdOut ) PF_WriteFileToFile(handle, VectorPtr(recvBuffer), size);
04531                 else
04532 #endif
04533                     WriteFileToFile(handle, VectorPtr(recvBuffer), size);
04534             }
04535             break;
04536     }
04537 }
04538
04539 /*
04540         #] PF_ReceiveErrorMessage :
04541         #[ PF_CatchErrorMessages :
04542 */
04543
04553 static void PF_CatchErrorMessages(int *src, int *tag)
04554 {
04555     for (;;) {
04556         int asrc = *src;
04557         int atag = *tag;
04558         int ret = PF_RawProbe(&asrc, &atag, NULL);
04559         CHECK(ret == 0);
04560         if ( atag == PF_STDOUT_MSGTAG || atag == PF_LOG_MSGTAG ) {
04561             assert(PF.me == MASTER);
04562             PF_ReceiveErrorMessage(asrc, atag);
04563             continue;
04564         }
04565         if ( atag == PF_RUNTIME_ERROR_MSGTAG ) {
04566             PF_ReceiveRuntimeError();
04567         }
04568         *src = asrc;
04569         *tag = atag;
04570         break;
04571     }
04572 }
```

```
04573
04574 /*
04575         #] PF_CatchErrorMessages :
04576         #[ PF_CatchErrorMessagesForAll :
04577 */
04578
04583 static void PF_CatchErrorMessagesForAll(void)
04584 {
04585     assert(PF.me == MASTER);
04586     int i;
04587     for ( i = 1; i < PF.numtasks; i++ ) {
04588         int src = i;
04589         int tag = PF_ANY_MSGTAG;
04590         PF_CatchErrorMessages(&src, &tag);
04591     }
04592 }
04593
04594 /*
04595         #] PF_CatchErrorMessagesForAll :
04596         #[ PF_ProbeWithCatchingErrorMessages :
04597 */
04598
04609 static int PF_ProbeWithCatchingErrorMessages(int *src)
04610 {
04611     for (;;) {
04612         int newsrc = *src;
04613         int tag = PF_Probe(&newsrc);
04614         if ( tag == PF_STDOUT_MSGTAG || tag == PF_LOG_MSGTAG ) {
04615             assert(PF.me == MASTER);
04616             PF_ReceiveErrorMessage(newsrc, tag);
04617             continue;
04618         }
04619         if ( tag == PF_RUNTIME_ERROR_MSGTAG ) {
04620             PF_ReceiveRuntimeError();
04621         }
04622         *src = newsrc;
04623         return tag;
04624     }
04625 }
04626
04627 /*
04628         #] PF_ProbeWithCatchingErrorMessages :
04629         #[ PF_FreeErrorMessageBuffers :
04630 */
04631
04637 void PF_FreeErrorMessageBuffers(void)
04638 {
04639     VectorFree(stdoutBuffer);
04640     VectorFree(logBuffer);
04641 }
04642
04643 /*
04644         #] PF_FreeErrorMessageBuffers :
04645     #] Synchronised output :
04646     #[ Handling runtime errors :
04647         #[ PF_RaiseRuntimeError :
04648 */
04649
04655 static void PF_RaiseRuntimeError(void)
04656 {
04657     if ( PF.me == MASTER ) {
04658         PF_BroadcastRuntimeError();
04659     }
04660     else {
04661         int ret, dummy;
04662         ret = PF_RawSend(MASTER, &dummy, 0, PF_RUNTIME_ERROR_MSGTAG);
04663         CHECK(ret == 0);
04664         int src = MASTER;
04665         int tag = PF_RUNTIME_ERROR_MSGTAG;
04666         ret = PF_RawRecv(&src, &dummy, 0, &tag);
04667         CHECK(ret == 0);
04668     }
04669 }
04670
04671 /*
04672         #] PF_RaiseRuntimeError :
04673         #[ PF_BroadcastRuntimeError :
04674 */
04675
04680 static void PF_BroadcastRuntimeError(void)
04681 {
04682     assert(PF.me == MASTER);
04683
04684     int ret, dummy = 0;
04685     MPI_Request requests[PF.numtasks - 1];
04686
04687     /*
```

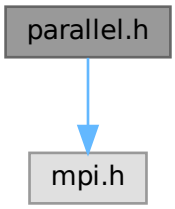
```
04688      * Notify all slaves of program termination by sending PF_RUNTIME_ERROR_MSGTAG.
04689      * This must be non-blocking to avoid deadlock if some slaves have already
04690      * performed a blocking send.
04691      */
04692     for ( int i = 1; i < PF.numtasks; i++ ) {
04693         ret = PF_RawIsend(i, &dummy, 0, PF_BYTE, PF_RUNTIME_ERROR_MSGTAG, &requests[i - 1]);
04694         CHECK(ret == 0);
04695     }
04696
04697     /*
04698      * Receive exactly one PF_RUNTIME_SYNC_MSGTAG or PF_RUNTIME_ERROR_MSGTAG
04699      * message from each slave.
04700      */
04701     for ( int i = 1; i < PF.numtasks; i++ ) {
04702 retry:
04703         int asrc = PF_ANY_SOURCE;  // blocking probe
04704         int tag = PF_Probe(&asrc);
04705         CHECK(tag >= 0);
04706         assert(1 <= asrc && asrc < PF.numtasks);
04707         switch ( tag ) {
04708             case PF_STDOUT_MSGTAG:
04709             case PF_LOG_MSGTAG:
04710                 PF_ReceiveErrorMessage(asrc, tag);
04711                 goto retry;
04712             case PF_RUNTIME_ERROR_MSGTAG:
04713             case PF_RUNTIME_SYNC_MSGTAG:
04714                 ret = PF_RawRecv(&asrc, &dummy, 0, &tag);
04715                 CHECK(ret == 0);
04716                 break;
04717             default:
04718                 ret = PF_Discard(&asrc, &tag);
04719                 CHECK(ret == 0);
04720                 goto retry;
04721         }
04722     }
04723
04724     ret = PF_RawWaitAll(PF.numtasks - 1, requests, MPI_STATUSES_IGNORE);
04725     CHECK(ret == 0);
04726 }
04727
04728 /*
04729         #] PF_BroadcastRuntimeError :
04730         #[ PF_PostEndSortBarrier :
04731 */
04732
04737 void PF_PostEndSortBarrier(void)
04738 {
04739     assert(PF_processing);
04740     PF_processing = 0;
04741
04742     int ret, dummy;
04743
04744     /*
04745      * Each slave reports either PF_RUNTIME_SYNC_MSGTAG (completed without errors)
04746      * or PF_RUNTIME_ERROR_MSGTAG (see PF_RaiseRuntimeError())
04747      * to the master. After collecting one message from each slave, the master sends
04748      * PF_RUNTIME_SYNC_MSGTAG to all slaves on success,
04749      * or PF_RUNTIME_ERROR_MSGTAG otherwise.
04750      *
04751      * This matches the behaviour of PF_RaiseRuntimeError() and
04752      * PF_ReceiveRuntimeError(). In all cases, each slave sends exactly one
04753      * PF_RUNTIME_SYNC_MSGTAG or PF_RUNTIME_ERROR_MSGTAG message to the master,
04754      * and the master sends exactly one such message to each slave.
04755      */
04756     if ( PF.me == MASTER ) {
04757         int error = 0;
04758         for ( int i = 1; i < PF.numtasks; i++ ) {
04759 retry:
04760             int asrc = PF_ANY_SOURCE;  // blocking probe
04761             int tag = PF_Probe(&asrc);
04762             CHECK(tag >= 0);
04763             assert(1 <= asrc && asrc < PF.numtasks);
04764             switch ( tag ) {
04765                 case PF_STDOUT_MSGTAG:
04766                 case PF_LOG_MSGTAG:
04767                     PF_ReceiveErrorMessage(asrc, tag);
04768                     goto retry;
04769                 case PF_RUNTIME_ERROR_MSGTAG:
04770                     error = 1;
04771                     ret = PF_RawRecv(&asrc, &dummy, 0, &tag);
04772                     CHECK(ret == 0);
04773                     break;
04774                 case PF_RUNTIME_SYNC_MSGTAG:
04775                     ret = PF_RawRecv(&asrc, &dummy, 0, &tag);
04776                     CHECK(ret == 0);
04777                     break;
04778                 default:
```

```
04779                     MesPrint("!!!Unexpected MPI message src=%d tag=%d.", asrc, tag);
04780                     ret = PF_Discard(&asrc, &tag);
04781                     CHECK(ret == 0);
04782                     goto retry;
04783                 }
04784             }
04785             for ( int i = 1; i < PF.numtasks; i++ ) {
04786                 ret = PF_RawSend(i, &dummy, 0, error ? PF_RUNTIME_ERROR_MSGTAG : PF_RUNTIME_SYNC_MSGTAG);
04787                 CHECK(ret == 0);
04788             }
04789         }
04790         else {
04791             int tag;
04792             ret = PF_RawSend(MASTER, &dummy, 0, PF_RUNTIME_SYNC_MSGTAG);
04793             CHECK(ret == 0);
04794             int src = MASTER;
04795             ret = PF_RawRecv(&src, &dummy, 0, &tag);
04796             CHECK(ret == 0);
04797             switch ( tag ) {
04798                 case PF_RUNTIME_SYNC_MSGTAG:
04799                     break;
04800                 case PF_RUNTIME_ERROR_MSGTAG:
04801                     PF.notMyFault = 1;
04802                     Terminate(-1);
04803                     break;
04804                 default:
04805                     MesPrint("!!!Unexpected MPI message src=%d tag=%d.", src, tag);
04806                     break;
04807             }
04808         }
04809 }
04810
04811 /*
04812         #] PF_PostEndSortBarrier :
04813         #[ PF_ReceiveRuntimeError :
04814 */
04815
04820 void PF_ReceiveRuntimeError(void)
04821 {
04822     PF_processing = 0;
04823     PF.notMyFault = 1;
04824     if ( PF.me == MASTER ) {
04825         PF_BroadcastRuntimeError();
04826     }
04827     else {
04828         int ret, dummy;
04829         int src = MASTER;
04830         int tag = PF_RUNTIME_ERROR_MSGTAG;
04831         ret = PF_RawRecv(&src, &dummy, 0, &tag);
04832         CHECK(ret == 0);
04833         ret = PF_RawSend(MASTER, &dummy, 0, PF_RUNTIME_ERROR_MSGTAG);
04834         CHECK(ret == 0);
04835     }
04836     Terminate(-1);
04837 }
04838
04839 /*
04840         #] PF_ReceiveRuntimeError :
04841     #] Handling runtime errors :
04842 */
```

4.99 parallel.h File Reference

```
#include <mpi.h>
```

Include dependency graph for parallel.h:



Data Structures

- struct PF_BUFFER
- struct ParallelVars

Macros

- #define MASTER 0
- #define PF_RESET 0 /* reset the timer */
- #define PF_TIME 1 /* get the elapsed time */
- #define PF_TERM_MSGTAG 10 /* master -> slave: sending terms */
- #define PF_ENDSORT_MSGTAG 11 /* master -> slave: no more terms to be distributed, slave -> master: after EndSort() */
- #define PF_DOLLAR_MSGTAG 12 /* slave -> master: sending $-variables */
- #define PF_BUFFER_MSGTAG 20 /* slave -> master: sending sorted terms, or in PF_SendFile()/PF_RecvFile() */
- #define PF_ENDBUFFER_MSGTAG 21 /* same as PF_BUFFER_MSGTAG, but indicates the end of operation */
- #define PF_READY_MSGTAG 30 /* slave -> master: slave is idle and can accept terms */
- #define PF_DATA_MSGTAG 50 /* InParallel, DoCheckpoint() */
- #define PF_EMPTY_MSGTAG 52 /* InParallel, DoCheckpoint(), PF_SendFile(), PF_RecvFile() */
- #define PF_STDOUT_MSGTAG 60 /* slave -> master: sending text to the stdout */
- #define PF_LOG_MSGTAG 61 /* slave -> master: sending text to the log file */
- #define PF_OPT_MCTS_MSGTAG 70 /* master <-> slave: optimization */
- #define PF_OPT_HORNER_MSGTAG 71 /* master <-> slave: optimization */
- #define PF_OPT_COLLECT_MSGTAG 72 /* slave -> master: optimization */
- #define PF_RUNTIME_ERROR_MSGTAG 80 /* slave <-> master: runtime error */
- #define PF_RUNTIME_SYNC_MSGTAG 81 /* master <-> slave: sync after EndSort() */
- #define PF_MISC_MSGTAG 100
- #define GNUC_PREREQ(major, minor, patchlevel) 0
- #define OMPI_SKIP_MPICXX 1
- #define indices ((INDICES)(AC.IndexList.lijst))
- #define PF_ANY_SOURCE MPI_ANY_SOURCE

- #define PF_ANY_MSGTAG MPI_ANY_TAG
- #define PF_COMM MPI_COMM_WORLD
- #define PF_BYTE MPI_BYTE
- #define PF_INT MPI_INT
- #define PF_LongMultiPack(buffer, count, type) PF_LongMultiPackImpl(buffer, count, sizeof_datatype(type), type)
- #define PF_LongMultiUnpack(buffer, count, type) PF_LongMultiUnpackImpl(buffer, count, sizeof_↩ datatype(type), type)

Typedefs

- typedef struct ParallelVars **PARALLELVARS**

Functions

- int PF_ISendSbuf (int, int)
- int PF_Bcast (void *buffer, int count)
- int PF_Reduce (const void *sendbuf, void *recvbuf, int count, MPI_Datatype type, MPI_Op op, int root)
- int PF_RawSend (int, void *, LONG, int)
- LONG PF_RawRecv (int *, void *, LONG, int *)
- int PF_PreparePack (void)
- int PF_Pack (const void *buffer, size_t count, MPI_Datatype type)
- int PF_Unpack (void *buffer, size_t count, MPI_Datatype type)
- int PF_PackString (const UBYTE *str)
- int PF_UnpackString (UBYTE *str)
- int PF_Send (int to, int tag)
- int PF_Receive (int src, int tag, int *psrc, int *ptag)
- int PF_Broadcast (void)
- int PF_PrepareLongSinglePack (void)
- int PF_LongSinglePack (const void *buffer, size_t count, MPI_Datatype type)
- int PF_LongSingleUnpack (void *buffer, size_t count, MPI_Datatype type)
- int PF_LongSingleSend (int to, int tag)
- int PF_LongSingleReceive (int src, int tag, int *psrc, int *ptag)
- int PF_PrepareLongMultiPack (void)
- int PF_LongMultiPackImpl (const void *buffer, size_t count, size_t eSize, MPI_Datatype type)
- int PF_LongMultiUnpackImpl (void *buffer, size_t count, size_t eSize, MPI_Datatype type)
- int PF_LongMultiBroadcast (void)
- int PF_EndSort (void)
- WORD PF_Deferred (WORD *, WORD)
- int PF_Processor (EXPRESSIONS, WORD, WORD)
- int PF_Init (int *, char ***)
- void PF_PreTerminate (int errorcode)
- int PF_Terminate (int)
- LONG PF_GetSlaveTimes (void)
- LONG PF_BroadcastNumber (LONG)
- void PF_BroadcastBuffer (WORD **buffer, LONG *length)
- int PF_BroadcastString (UBYTE *)
- int PF_BroadcastPreDollar (WORD **, LONG *, int *)
- int PF_CollectModifiedDollars (void)
- int PF_BroadcastModifiedDollars (void)
- int PF_BroadcastRedefinedPreVars (void)
- int PF_BroadcastCBuf (int bufnum)
- int PF_BroadcastExpFlags (void)

- int PF_StoreInsideInfo (void)
- int PF_RestoreInsideInfo (void)
- int PF_BroadcastExpr (EXPRESSIONS e, FILEHANDLE *file)
- int PF_BroadcastRHS (void)
- int PF_InParallelProcessor (void)
- int PF_SendFile (int to, FILE *fd)
- int PF_RecvFile (int from, FILE *fd)
- void PF_MLock (void)
- void PF_MUnlock (void)
- LONG PF_WriteFileToFile (int, UBYTE *, LONG)
- void PF_FlushStdOutBuffer (void)
- void PF_ReceiveRuntimeError (void) NORETURN

Variables

- PARALLELVARS PF
- LONG PF_maxDollarChunkSize

4.99.1 Detailed Description

Header file with things relevant to ParForm.

Definition in file parallel.h.

4.99.2 Macro Definition Documentation

4.99.2.1 MASTER

```
#define MASTER 0
```

Definition at line 43 of file parallel.h.

4.99.2.2 PF_RESET

```
#define PF_RESET 0 /* reset the timer */
```

Definition at line 48 of file parallel.h.

4.99.2.3 PF_TIME

```
#define PF_TIME 1 /* get the elapsed time */
```

Definition at line 49 of file parallel.h.

4.99.2.4 PF_TERM_MSGTAG

```
#define PF_TERM_MSGTAG 10 /* master -> slave:  sending terms */
```

Definition at line 54 of file parallel.h.

4.99.2.5 PF_ENDSORT_MSGTAG

```
#define PF_ENDSORT_MSGTAG 11 /* master -> slave:  no more terms to be distributed, slave ->
master:  after EndSort() */
```

Definition at line 55 of file parallel.h.

4.99.2.6 PF_DOLLAR_MSGTAG

```
#define PF_DOLLAR_MSGTAG 12 /* slave -> master:  sending $-variables */
```

Definition at line 56 of file parallel.h.

4.99.2.7 PF_BUFFER_MSGTAG

```
#define PF_BUFFER_MSGTAG 20 /* slave -> master:  sending sorted terms, or in PF_SendFile()/PF_RecvFile()
*/
```

Definition at line 57 of file parallel.h.

4.99.2.8 PF_ENDBUFFER_MSGTAG

```
#define PF_ENDBUFFER_MSGTAG 21 /* same as PF_BUFFER_MSGTAG, but indicates the end of operation
*/
```

Definition at line 58 of file parallel.h.

4.99.2.9 PF_READY_MSGTAG

```
#define PF_READY_MSGTAG 30 /* slave -> master:  slave is idle and can accept terms */
```

Definition at line 59 of file parallel.h.

4.99.2.10 PF_DATA_MSGTAG

```
#define PF_DATA_MSGTAG 50 /* InParallel, DoCheckpoint() */
```

Definition at line 60 of file parallel.h.

4.99.2.11 PF_EMPTY_MSGTAG

```
#define PF_EMPTY_MSGTAG 52 /* InParallel, DoCheckpoint(), PF_SendFile(), PF_RecvFile() */
```

Definition at line 61 of file parallel.h.

4.99.2.12 PF_STDOUT_MSGTAG

```
#define PF_STDOUT_MSGTAG 60 /* slave -> master:  sending text to the stdout */
```

Definition at line 62 of file parallel.h.

4.99.2.13 PF_LOG_MSGTAG

```
#define PF_LOG_MSGTAG 61 /* slave -> master:  sending text to the log file */
```

Definition at line 63 of file parallel.h.

4.99.2.14 PF_OPT_MCTS_MSGTAG

```
#define PF_OPT_MCTS_MSGTAG 70 /* master <-> slave:  optimization */
```

Definition at line 64 of file parallel.h.

4.99.2.15 PF_OPT_HORNER_MSGTAG

```
#define PF_OPT_HORNER_MSGTAG 71 /* master <-> slave:  optimization */
```

Definition at line 65 of file parallel.h.

4.99.2.16 PF_OPT_COLLECT_MSGTAG

```
#define PF_OPT_COLLECT_MSGTAG 72 /* slave -> master:  optimization */
```

Definition at line 66 of file parallel.h.

4.99.2.17 PF_RUNTIME_ERROR_MSGTAG

```
#define PF_RUNTIME_ERROR_MSGTAG 80 /* slave <-> master:  runtime error */
```

Definition at line 67 of file parallel.h.

4.99.2.18 PF_RUNTIME_SYNC_MSGTAG

```
#define PF_RUNTIME_SYNC_MSGTAG 81 /* master <-> slave:  sync after EndSort() */
```

Definition at line 68 of file parallel.h.

4.99.2.19 PF_MISC_MSGTAG

```
#define PF_MISC_MSGTAG 100
```

Definition at line 69 of file parallel.h.

4.99.2.20 GNUC_PREREQ

```
#define GNUC_PREREQ(
              major,
              minor,
              patchlevel ) 0
```

Definition at line 79 of file parallel.h.

4.99.2.21 OMPI_SKIP_MPICXX

```
#define OMPI_SKIP_MPICXX 1
```

Definition at line 120 of file parallel.h.

4.99.2.22 indices

```
#define indices ((INDICES)(AC.IndexList.lijst))
```

Definition at line 125 of file parallel.h.

4.99.2.23 PF_ANY_SOURCE

```
#define PF_ANY_SOURCE MPI_ANY_SOURCE
```

Definition at line 135 of file parallel.h.

4.99.2.24 PF_ANY_MSGTAG

```
#define PF_ANY_MSGTAG MPI_ANY_TAG
```

Definition at line 136 of file parallel.h.

4.99.2.25 PF_COMM

```
#define PF_COMM MPI_COMM_WORLD
```

Definition at line 137 of file parallel.h.

4.99.2.26 PF_BYTE

```
#define PF_BYTE MPI_BYTE
```

Definition at line 138 of file parallel.h.

4.99.2.27 PF_INT

```
#define PF_INT MPI_INT
```

Definition at line 139 of file parallel.h.

4.99.2.28 PF_LongMultiPack

```
#define PF_LongMultiPack(
              buffer,
              count,
              type ) PF_LongMultiPackImpl(buffer, count, sizeof_datatype(type), type)
```

Definition at line 246 of file parallel.h.

4.99.2.29 PF_LongMultiUnpack

```
#define PF_LongMultiUnpack(
              buffer,
              count,
              type ) PF_LongMultiUnpackImpl(buffer, count, sizeof_datatype(type), type)
```

Definition at line 247 of file parallel.h.

4.99.3 Function Documentation

4.99.3.1 PF_ISendSbuf()

```
int PF_ISendSbuf (
             int to,
             int tag )  [extern]
```

Nonblocking send operation. It sends everything from `buff` to `fill` of the active buffer. Depending on *tag* it also can do waiting for other sends to finish or set the active buffer to the next one.

Parameters

to	the destination process number.
tag	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 266 of file mpi.c.

Referenced by FlushOut(), PF_Processor(), and PutOut().

4.99.3.2 PF_Bcast()

```
int PF_Bcast (
            void * buffer,
            int count )  [extern]
```

Broadcasts a message from the master to slaves.

Parameters

in,out	*buffer*	the starting address of buffer. The contents in this buffer on the master will be transferred to those on the slaves.
	count	the length of the buffer in bytes.

Returns

0 if OK, nonzero on error.

Definition at line 452 of file mpi.c.

Referenced by PF_BroadcastBuffer(), and PF_BroadcastNumber().

4.99.3.3 PF_Reduce()

```
int PF_Reduce (
            const void * sendbuf,
            void * recvbuf,
            int count,
            MPI_Datatype type,
            MPI_Op op,
            int root )  [extern]
```

Performs a reduce operation across all processes.

Parameters

in	*sendbuf*	the buffer containing the data to be reduced.
out	*recvbuf*	the buffer to store the reduced result (only for the root process).
	count	the number of elements in the buffers.
	type	the datatype of the elements.
	op	the operation to apply.
	root	the root process number.

Returns

0 if OK, nonzero on error.

Definition at line 475 of file mpi.c.

Referenced by PF_GetSlaveTimes().

4.99.3.4 PF_RawSend()

```
int PF_RawSend (
            int dest,
            void * buf,
            LONG l,
            int tag )  [extern]
```

Sends *l* bytes from *buf* to *dest*. Returns 0 on success, or -1.

Parameters

dest	the destination process number.
buf	the send buffer.
l	the size of data in the send buffer in bytes.
tag	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 497 of file mpi.c.

4.99.3.5 PF_RawRecv()

```
LONG PF_RawRecv (
            int * src,
            void * buf,
            LONG thesize,
            int * tag )  [extern]
```

Receives not more than *thesize* bytes from *src*, returns the actual number of received bytes, or -1 on failure.

Parameters

in,out	*src*	the source process number. In output, that of the actual received message.
out	*buf*	the receive buffer.
	thesize	the size of the receive buffer in bytes.
out	*tag*	the message tag of the actual received message.

Returns

the actual sizeof received data in bytes, or -1 on failure.

Definition at line 518 of file mpi.c.

4.99.3.6 PF_PreparePack()

```
int PF_PreparePack (
            void  )  [extern]
```

Prepares for the next pack operations on the sender.

Returns

0 if OK, nonzero on error.

Definition at line 736 of file mpi.c.

Referenced by Optimize(), PF_BroadcastPreDollar(), PF_BroadcastString(), and PF_Init().

4.99.3.7 PF_Pack()

```
int PF_Pack (
            const void * buffer,
            size_t count,
            MPI_Datatype type )  [extern]
```

Adds data into the pack buffer.

Parameters

buffer	the pointer to the buffer storing the data to be packed.
count	the number of elements in the buffer.
type	the data type of elements in the buffer.

Returns

0 if OK, nonzero on error.

Definition at line 754 of file mpi.c.

References MPI_ERRCODE_CHECK.

Referenced by Optimize(), PF_BroadcastPreDollar(), and PF_Init().

4.99.3.8 PF_Unpack()

```
int PF_Unpack (
            void * buffer,
            size_t count,
            MPI_Datatype type )  [extern]
```

Retrieves the next data in the pack buffer.

Parameters

out	*buffer*	the pointer to the buffer to store the unpacked data.
	count	the number of elements of data to be received.
	type	the data type of elements of data to be received.

Returns

0 if OK, nonzero on error.

Definition at line 783 of file mpi.c.

References MPI_ERRCODE_CHECK.

Referenced by Optimize(), PF_BroadcastPreDollar(), and PF_Init().

4.99.3.9 PF_PackString()

```
int PF_PackString (
            const UBYTE * str )  [extern]
```

Packs a string *str* into the packed buffer PF_packbuf, including the trailing zero.

The first element (PF_INT) is the length of the packed portion of the string. If the string does not fit to the buffer PF↩ _packbuf, the function packs only the initial portion. It returns the number of packed bytes, so if (str[length-1]=='\0') then the whole string fits to the buffer, if not, then the rest (str+length) must be packed and send again. On error, the function returns the negative error code.

One exception: the string "\0!\0" is used as an image of the NULL, so all 3 characters will be packed.

Parameters

str	a string to be packed.

Returns

the number of packed bytes, or a negative value on failure.

Definition at line 818 of file mpi.c.

Referenced by PF_BroadcastString().

4.99.3.10 PF_UnpackString()

```
int PF_UnpackString (
            UBYTE * str )  [extern]
```

Unpacks a string to *str* from the packed buffer PF_packbuf, including the trailing zero.

It returns the number of unpacked bytes, so if (str[length-1]=='\0') then the whole string was unpacked, if not, then the rest must be appended to (str+length). On error, the function returns the negative error code.

Parameters

out	*str*	the buffer to store the unpacked string

Returns

the number of unpacked bytes, or a negative value on failure.

Definition at line 886 of file mpi.c.

Referenced by PF_BroadcastString().

4.99.3.11 PF_Send()

```
int PF_Send (
              int to,
              int tag )  [extern]
```

Sends the contents in the pack buffer to the process specified by *to*.

Example:

```
if ( PF.me == SRC ) {
  PF_PreparePack();
  // Packing operations here...
  PF_Send(DEST, TAG);
}
else if ( PF.me == DEST ) {
  PF_Receive(SRC, TAG, &actual_src, &actual_tag);
  // Unpacking operations here...
}
```

Parameters

to	the destination process number.
tag	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 933 of file mpi.c.

References MPI_ERRCODE_CHECK.

4.99.3.12 PF_Receive()

```
int PF_Receive (
              int src,
              int tag,
              int * psrc,
              int * ptag )  [extern]
```

Receives data into the pack buffer from the process specified by *src*. This function allows &src == psrc or &tag == ptag. Either *psrc* or *ptag* can be NULL.

See the example of PF_Send().

Parameters

	src	the source process number (can be PF_ANY_SOURCE).
	tag	the source message tag (can be PF_ANY_TAG).
out	*psrc*	the actual source process number of received message.
out	*ptag*	the received message tag.

Returns

0 if OK, nonzero on error.

Definition at line 959 of file mpi.c.

References MPI_ERRCODE_CHECK.

4.99.3.13 PF_Broadcast()

```
int PF_Broadcast (
            void  )  [extern]
```

Broadcasts the contents in the pack buffer on the master to those on the slaves.

Example:

```
if ( PF.me == MASTER ) {
  PF_PreparePack();
  // Packing operations here...
}
PF_Broadcast();
if ( PF.me != MASTER ) {
  // Unpacking operations here...
}
```

Returns

0 if OK, nonzero on error.

Definition at line 994 of file mpi.c.

References MPI_ERRCODE_CHECK.

Referenced by Optimize(), PF_BroadcastPreDollar(), PF_BroadcastString(), and PF_Init().

4.99.3.14 PF_PrepareLongSinglePack()

```
int PF_PrepareLongSinglePack (
            void  )  [extern]
```

Prepares for the next long-single-pack operations on the sender.

Returns

0 if OK, nonzero on error.

Definition at line 1561 of file mpi.c.

Referenced by PF_CollectModifiedDollars(), and PF_Processor().

4.99.3.15 PF_LongSinglePack()

```
int PF_LongSinglePack (
            const void * buffer,
            size_t count,
            MPI_Datatype type )  [extern]
```

Adds data into the "long single" pack buffer.

Parameters

buffer	the pointer to the buffer storing the data to be packed.
count	the number of elements in the buffer.
type	the data type of elements in the buffer.

Returns

0 if OK, nonzero on error.

Definition at line 1579 of file mpi.c.

Referenced by PF_CollectModifiedDollars(), and PF_Processor().

4.99.3.16 PF_LongSingleUnpack()

```
int PF_LongSingleUnpack (
            void * buffer,
            size_t count,
            MPI_Datatype type )  [extern]
```

Retrieves the next data in the "long single" pack buffer.

Parameters

out	*buffer*	the pointer to the buffer to store the unpacked data.
	count	the number of elements of data to be received.
	type	the data type of elements of data to be received.

Returns

0 if OK, nonzero on error.

Definition at line 1613 of file mpi.c.

Referenced by PF_CollectModifiedDollars(), and PF_Processor().

4.99.3.17 PF_LongSingleSend()

```
int PF_LongSingleSend (
            int to,
            int tag )  [extern]
```

Sends the contents in the "long single" pack buffer to the process specified by *to*.

Example:

```
if ( PF.me == SRC ) {
  PF_PrepareLongSinglePack();
  // Packing operations here...
  PF_LongSingleSend(DEST, TAG);
}
else if ( PF.me == DEST ) {
  PF_LongSingleReceive(SRC, TAG, &actual_src, &actual_tag);
  // Unpacking operations here...
}
```

Parameters

to	the destination process number.
tag	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 1650 of file mpi.c.

Referenced by PF_CollectModifiedDollars(), and PF_Processor().

4.99.3.18 PF_LongSingleReceive()

```
int PF_LongSingleReceive (
            int src,
            int tag,
            int * psrc,
            int * ptag )  [extern]
```

Receives data into the "long single" pack buffer from the process specified by *src*. This function allows &src == psrc or &tag == ptag. Either *psrc* or *ptag* can be NULL.

See the example of PF_LongSingleSend().

Parameters

	src	the source process number (can be PF_ANY_SOURCE).
	tag	the source message tag (can be PF_ANY_TAG).
out	*psrc*	the actual source process number of received message.
out	*ptag*	the received message tag.

Returns

0 if OK, nonzero on error.

Definition at line 1693 of file mpi.c.

Referenced by PF_CollectModifiedDollars(), and PF_Processor().

4.99.3.19 PF_PrepareLongMultiPack()

```
int PF_PrepareLongMultiPack (
            void  )  [extern]
```

Prepares for the next long-multi-pack operations on the sender.

Returns

0 if OK, nonzero on error.

Definition at line 1752 of file mpi.c.

Referenced by Optimize(), PF_BroadcastCBuf(), PF_BroadcastExpFlags(), PF_BroadcastModifiedDollars(), and PF_BroadcastRedefinedPreVars().

4.99.3.20 PF_LongMultiPackImpl()

```
int PF_LongMultiPackImpl (
            const void * buffer,
            size_t count,
            size_t eSize,
            MPI_Datatype type )  [extern]
```

Adds data into the "long multi" pack buffer.

Parameters

buffer	the pointer to the buffer storing the data to be packed.
count	the number of elements in the buffer.
eSize	the byte size of each element of data.
type	the data type of elements in the buffer.

Returns

0 if OK, nonzero on error.

Definition at line 1771 of file mpi.c.

References PF_LongMultiPackImpl().

Referenced by PF_LongMultiPackImpl().

Here is the call graph for this function:

4.99.3.21 PF_LongMultiUnpackImpl()

```
int PF_LongMultiUnpackImpl (
            void * buffer,
            size_t count,
            size_t eSize,
            MPI_Datatype type )  [extern]
```

Retrieves the next data in the "long multi" pack buffer.

Parameters

out	*buffer*	the pointer to the buffer to store the unpacked data.
	count	the number of elements of data to be received.
	eSize	the byte size of each element of data.
	type	the data type of elements of data to be received.

Returns

0 if OK, nonzero on error.

Definition at line 1830 of file mpi.c.

References PF_LongMultiUnpackImpl().

Referenced by PF_LongMultiUnpackImpl().

Here is the call graph for this function:

4.99.3.22 PF_LongMultiBroadcast()

```
int PF_LongMultiBroadcast (
              void  )  [extern]
```

Broadcasts the contents in the "long multi" pack buffer on the master to those on the slaves.

Example:

```
if ( PF.me == MASTER ) {
  PF_PrepareLongMultiPack();
  // Packing operations here...
}
PF_LongMultiBroadcast();
if ( PF.me != MASTER ) {
  // Unpacking operations here...
}
```

Returns

0 if OK, nonzero on error.

Definition at line 1916 of file mpi.c.

Referenced by Optimize(), PF_BroadcastCBuf(), PF_BroadcastExpFlags(), PF_BroadcastModifiedDollars(), and PF_BroadcastRedefinedPreVars().

4.99.3.23 PF_EndSort()

```
int PF_EndSort (
            void  )  [extern]
```

Finishes a master sorting with collecting terms from slaves. Called by EndSort().

If this is not the masterprocess, just initialize the sendbuffers and return 0, else PF_EndSort() sends the rest of the terms in the sendbuffer to the next slave and a dummy message to all slaves with tag PF_ENDSORT_MSGTAG. Then it receives the sorted terms, sorts them using a recursive 'tree of losers' (PF_GetLoser()) and writes them to the outputfile.

Returns

1 if the sorting on the master was done. 0 if EndSort() still must perform a regular sorting because it is not at the ground level or not on the master or in the sequential mode or in the InParallel mode. -1 if an error occurred.

Remarks

The slaves will send the sorted terms back to the master in the regular sorting (after the initialization of the send buffer in PF_EndSort()). See PutOut() and FlushOut().

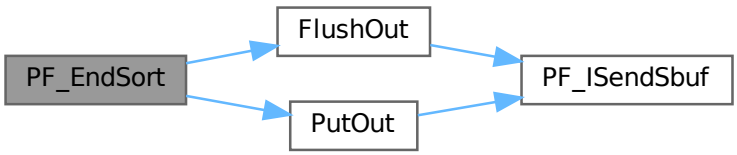
This function has been changed such that when it returns 1, AM.S0->TermsLeft is set correctly. But AM.S0->GenTerms is not set: it will be set after collecting the statistics from the slaves at the end of PF_Processor(). (TU 30 Jun 2011)

Definition at line 874 of file parallel.c.

References FlushOut(), and PutOut().

Referenced by EndSort().

Here is the call graph for this function:



4.99.3.24 PF_Deferred()

```
WORD PF_Deferred (
            WORD * term,
            WORD level )  [extern]
```

Replaces Deferred() on the slaves.

Parameters

term	the term that must be multiplied by the contents of the current bracket.
level	the compiler level.

Returns

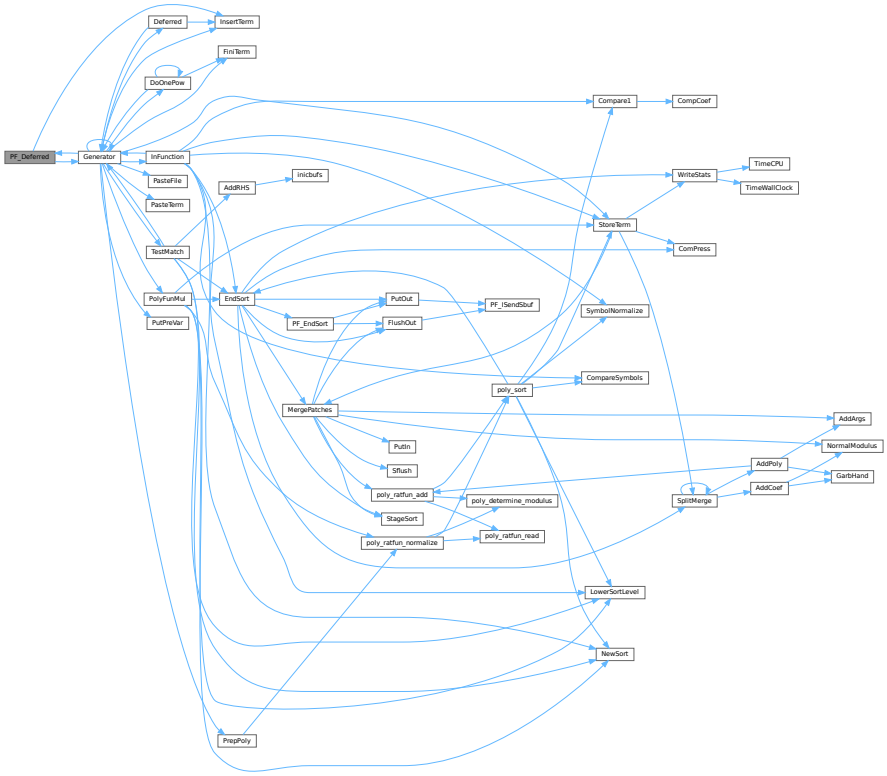
0 if OK, nonzero on error.

Definition at line 1201 of file parallel.c.

References Generator(), and InsertTerm().

Referenced by Generator().

Here is the call graph for this function:



4.99.3.25 PF_Processor()

```
int PF_Processor (
            EXPRESSIONS e,
            WORD i,
            WORD LastExpression )  [extern]
```

Replaces parts of Processor() on the masters and slaves. On the master PF_Processor() is responsible for proper distribution of terms from the input file to the slaves. On the slaves it calls Generator() for all the terms that this process gets, but PF_GetTerm() gets terms from the master (not directly from infile).

Parameters

e	The pointer to the current expression.
i	The index for the current expression.
LastExpression	The flag indicating whether it is the last expression.

Returns

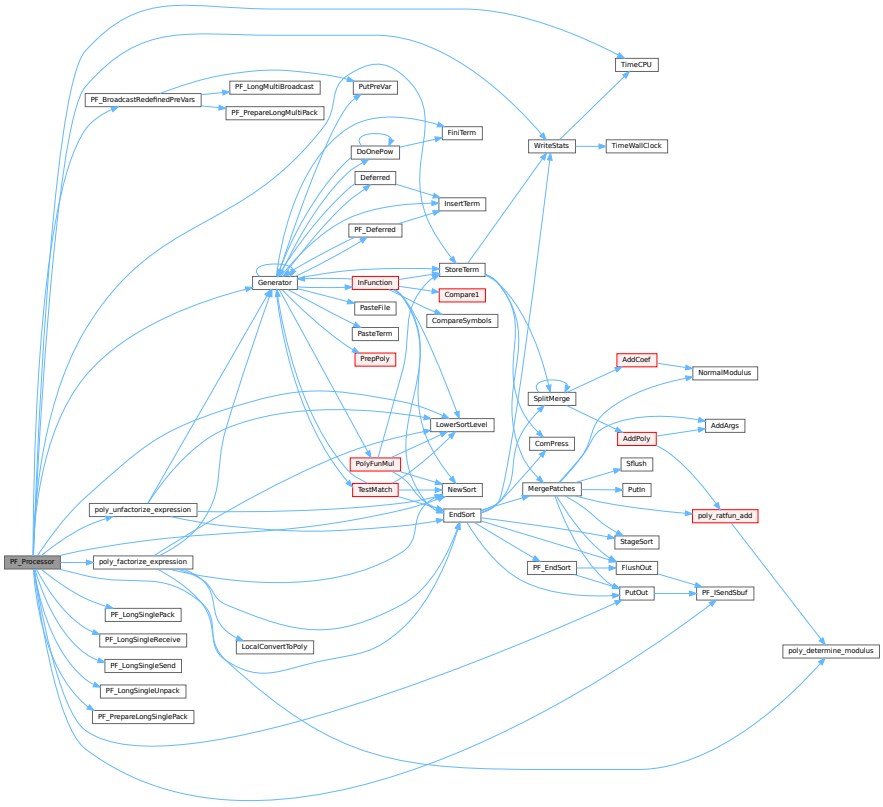
0 if OK, nonzero on error.

Definition at line 1533 of file parallel.c.

References EndSort(), Generator(), LowerSortLevel(), NewSort(), PACK_LONG, PF_BroadcastRedefinedPreVars(), PF_ISendSbuf(), PF_LongSinglePack(), PF_LongSingleReceive(), PF_LongSingleSend(), PF_LongSingleUnpack(), PF_PrepareLongSinglePack(), poly_factorize_expression(), poly_unfactorize_expression(), PutOut(), StoreTerm(), SWAP, TimeCPU(), and WriteStats().

Referenced by Processor().

Here is the call graph for this function:



4.99.3.26 PF_Init()

```
int PF_Init (
            int * argc,
            char *** argv )  [extern]
```

All the library independent stuff. PF_LibInit() should do all library dependent initializations.

Parameters

argc	pointer to the number of arguments.
argv	pointer to the arguments.

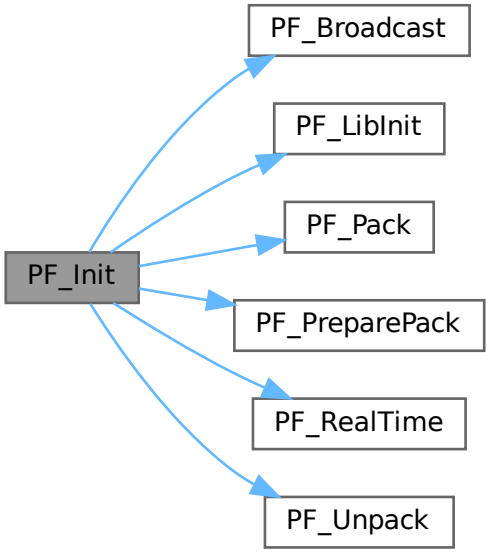
Returns

0 if OK, nonzero on error.

Definition at line 1947 of file parallel.c.

References PF_Broadcast(), PF_LibInit(), PF_Pack(), PF_PreparePack(), PF_RealTime(), and PF_Unpack().

Here is the call graph for this function:



4.99.3.27 PF_PreTerminate()

```
void PF_PreTerminate (
            int errorcode )  [extern]
```

Prepares for termination. Called by Terminate().

Parameters

errorcode	an error code.

Definition at line 2041 of file parallel.c.

4.99.3.28 PF_Terminate()

```
int PF_Terminate (
            int errorcode )  [extern]
```

Performs the finalization of ParFORM. To be called by Terminate().

Parameters

errorcode	an error code.

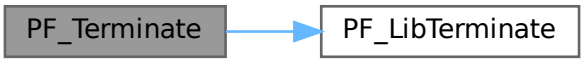
Returns

0 if OK, nonzero on error.

Definition at line 2061 of file parallel.c.

References PF_LibTerminate().

Here is the call graph for this function:



4.99.3.29 PF_GetSlaveTimes()

```
LONG PF_GetSlaveTimes (
            void  )  [extern]
```

Returns the total CPU time of all slaves together. This function must be called on the master and all slaves.

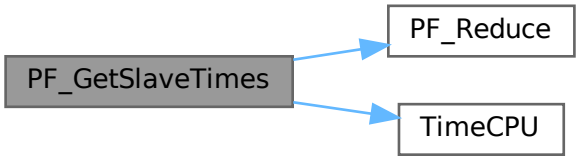
Returns

on the master, the sum of CPU times on all slaves.

Definition at line 2077 of file parallel.c.

References CHECK, PF_Reduce(), and TimeCPU().

Here is the call graph for this function:



4.99.3.30 PF_BroadcastNumber()

```
LONG PF_BroadcastNumber (
            LONG x )  [extern]
```

Broadcasts a LONG value from the master to the all slaves.

Parameters

x	the number to be broadcast (set on the master).

Returns

the synchronised result.

Definition at line 2098 of file parallel.c.

References PF_Bcast().

Referenced by DoCheckpoint(), PF_BroadcastBuffer(), and StartVariables().

Here is the call graph for this function:

4.99.3.31 PF_BroadcastBuffer()

```
void PF_BroadcastBuffer (
            WORD ** buffer,
            LONG * length )  [extern]
```

Broadcasts a buffer from the master to all the slaves.

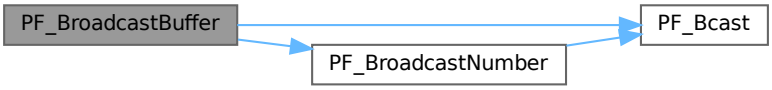
Parameters

in,out	*buffer*	on the master, the buffer to be broadcast. On the slaves, the buffer will be allocated if the length is greater than 0. The caller must free it.
in,out	*length*	on the master, the length of the buffer to be broadcast. On the slaves, it receives the length of transferred buffer. The actual transfer occurs only if the length is greater than 0.

Definition at line 2125 of file parallel.c.

References PF_Bcast(), and PF_BroadcastNumber().

Here is the call graph for this function:



4.99.3.32 PF_BroadcastString()

```
int PF_BroadcastString (
            UBYTE * str )  [extern]
```

Broadcasts a string from the master to all slaves.

Parameters

in,out	*str*	The pointer to a null-terminated string.

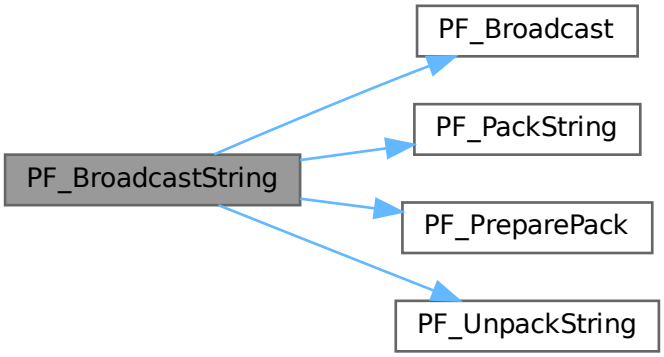
Returns

0 if OK, nonzero on error.

Definition at line 2167 of file parallel.c.

References PF_Broadcast(), PF_PackString(), PF_PreparePack(), and PF_UnpackString().

Here is the call graph for this function:



4.99.3.33 PF_BroadcastPreDollar()

```
int PF_BroadcastPreDollar (
            WORD ** dbuffer,
            LONG * newsize,
            int * numterms )  [extern]
```

Broadcasts dollar variables set as a preprocessor variables. Only the master is able to make an assignment like #$a=g; where g is an expression: only the master has an access to the expression. So, the master broadcasts the result to slaves.

The result is in ∗dbuffer of the size is ∗newsize (in number of WORDs), +1 for trailing zero. For slave newsize and numterms are output parameters.

Parameters

in,out	*dbuffer*	the buffer for a dollar variable.
in,out	*newsize*	the size of the dollar variable in WORDs.
in,out	*numterms*	the number of terms in the dollar variable.

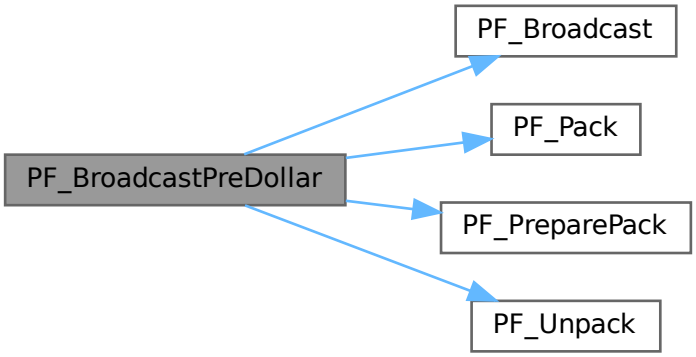
Returns

0 if OK, nonzero on error.

Definition at line 2222 of file parallel.c.

References PF_Broadcast(), PF_Pack(), PF_PreparePack(), and PF_Unpack().

Here is the call graph for this function:



4.99.3.34 PF_CollectModifiedDollars()

```
int PF_CollectModifiedDollars (
            void  )  [extern]
```

Combines modified dollar variables on the all slaves, and store them into those on the master.

The potentially modified dollar variables are given in PotModdollars, and the number of them is given by NumPot↩ Moddollars.

The current module could be executed in parallel only if all potentially modified variables are listed in ModOptdollars, otherwise the module was switched to the sequential mode.

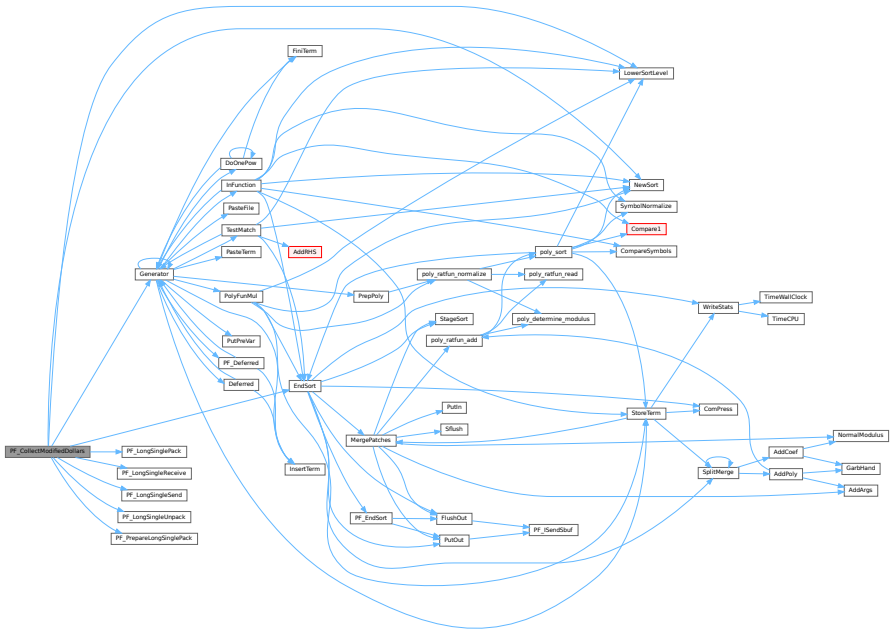
Returns

0 if OK, nonzero on error.

Definition at line 2509 of file parallel.c.

References EndSort(), Generator(), LowerSortLevel(), NewSort(), PF_LongSinglePack(), PF_LongSingleReceive(), PF_LongSingleSend(), PF_LongSingleUnpack(), PF_PrepareLongSinglePack(), VectorInit, VectorPtr, VectorReserve, and VectorSize.

Here is the call graph for this function:



4.99.3.35 PF_BroadcastModifiedDollars()

```
int PF_BroadcastModifiedDollars (
            void  )  [extern]
```

Broadcasts modified dollar variables on the master to the all slaves.

The potentially modified dollar variables are given in PotModdollars, and the number of them is given by NumPot↩Moddollars.

The current module could be executed in parallel only if all potentially modified variables are listed in ModOptdollars, otherwise the module was switched to the sequential mode. In either cases, we need to broadcast them.

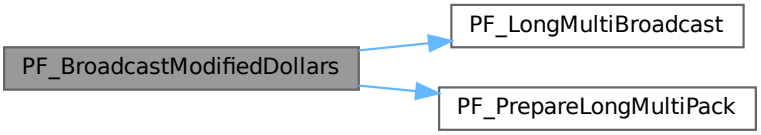
Returns

0 if OK, nonzero on error.

Definition at line 2788 of file parallel.c.

References PF_LongMultiBroadcast(), and PF_PrepareLongMultiPack().

Here is the call graph for this function:



4.99.3.36 PF_BroadcastRedefinedPreVars()

```
int PF_BroadcastRedefinedPreVars (
            void  )  [extern]
```

Broadcasts preprocessor variables, which were changed by the Redefine statements in the current module, from the master to the all slaves.

The potentially redefined preprocessor variables are given in AC.pfirstnum, and the number of them is given by AC.numpfirstnum. For an actually redefined variable, the corresponding value in AC.inputnumbers is non-negative.

Returns

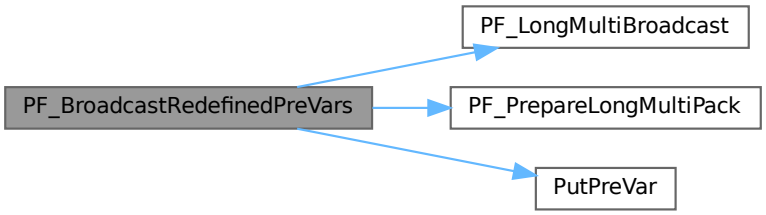
0 if OK, nonzero on error.

Definition at line 3005 of file parallel.c.

References PF_LongMultiBroadcast(), PF_PrepareLongMultiPack(), PutPreVar(), VectorPtr, and VectorReserve.

Referenced by PF_Processor().

Here is the call graph for this function:



4.99.3.37 PF_BroadcastCBuf()

```
int PF_BroadcastCBuf (
            int bufnum )  [extern]
```

Broadcasts a compiler buffer specified by *bufnum* from the master to the all slaves.

Parameters

bufnum	The index of the compiler buffer to be broadcast.

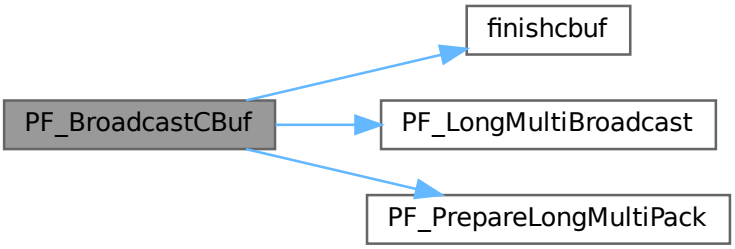
Returns

0 if OK, nonzero on error.

Definition at line 3147 of file parallel.c.

References CbUf::boomlijst, CbUf::Buffer, CbUf::BufferSize, CbUf::CanCommu, CbUf::dimension, finishcbuf(), CbUf::lhs, CbUf::numdum, CbUf::NumTerms, PF_LongMultiBroadcast(), PF_PrepareLongMultiPack(), CbUf::Pointer, CbUf::rhs, and CbUf::Top.

Here is the call graph for this function:



4.99.3.38 PF_BroadcastExpFlags()

```
int PF_BroadcastExpFlags (
            void  )  [extern]
```

Broadcasts AR.expflags and several properties of each expression, e.g., e->vflags, from the master to all slaves.

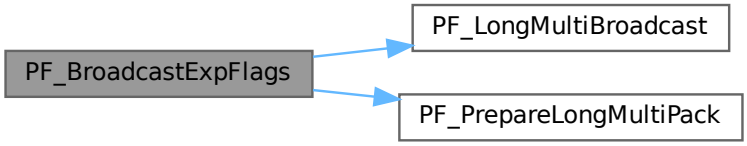
Returns

0 if OK, nonzero on error.

Definition at line 3258 of file parallel.c.

References PF_LongMultiBroadcast(), and PF_PrepareLongMultiPack().

Here is the call graph for this function:



4.99.3.39 PF_StoreInsideInfo()

```
int PF_StoreInsideInfo (
            void  )  [extern]
```

Definition at line 3095 of file parallel.c.

4.99.3.40 PF_RestoreInsideInfo()

```
int PF_RestoreInsideInfo (
            void  )  [extern]
```

Definition at line 3121 of file parallel.c.

4.99.3.41 PF_BroadcastExpr()

```
int PF_BroadcastExpr (
            EXPRESSIONS e,
            FILEHANDLE * file )  [extern]
```

Broadcasts an expression from the master to the all slaves.

Parameters

e	The expression to be broadcast.
file	The file in which the expression is sitting.

Returns

0 if OK, nonzero on error.

Definition at line 3552 of file parallel.c.

Referenced by PF_BroadcastRHS().

4.99.3.42 PF_BroadcastRHS()

```
int PF_BroadcastRHS (
            void  )  [extern]
```

Broadcasts expressions appearing in the right-hand side from the master to the all slaves.

Returns

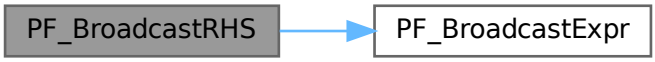
0 if OK, nonzero on error.

Definition at line 3580 of file parallel.c.

References PF_BroadcastExpr().

Referenced by Processor().

Here is the call graph for this function:



4.99.3.43 PF_InParallelProcessor()

```
int PF_InParallelProcessor (
            void  )  [extern]
```

Processes expressions in the InParallel mode, i.e., dividing expressions marked by partodo over the slaves.

Returns

0 if OK, nonzero on error.

Definition at line 3627 of file parallel.c.

Referenced by Processor().

4.99.3.44 PF_SendFile()

```
int PF_SendFile (
            int to,
            FILE * fd )  [extern]
```

Sends a file to the process specified by *to*.

Parameters

to	the destination process number.
fd	the file to be sent.

Returns

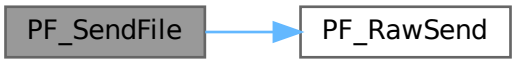
the size of sent data in bytes, or -1 on error.

Definition at line 4234 of file parallel.c.

References PF_RawSend().

Referenced by DoCheckpoint().

Here is the call graph for this function:



4.99.3.45 PF_RecvFile()

```
int PF_RecvFile (
            int from,
            FILE * fd )  [extern]
```

Receives a file from the process specified by *from*.

Parameters

from	the source process number.
fd	the file to save the received data.

Returns

the size of received data in bytes, or -1 on error.

Definition at line 4272 of file parallel.c.

References PF_RawRecv().

Referenced by DoCheckpoint().

Here is the call graph for this function:

4.99.3.46 PF_MLock()

```
void PF_MLock (
            void  )  [extern]
```

A function called by MLOCK(ErrorMessageLock) for slaves.

Definition at line 4353 of file parallel.c.

References VectorClear.

4.99.3.47 PF_MUnlock()

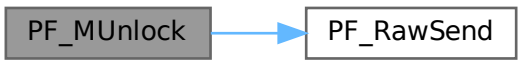
```
void PF_MUnlock (
            void  )  [extern]
```

A function called by MUNLOCK(ErrorMessageLock) for slaves.

Definition at line 4369 of file parallel.c.

References PF_RawSend(), VectorEmpty, VectorPtr, and VectorSize.

Here is the call graph for this function:



4.99.3.48 PF_WriteFileToFile()

```
LONG PF_WriteFileToFile (
            int handle,
            UBYTE * buffer,
            LONG size )  [extern]
```

Replaces WriteFileToFile() on the master and slaves.

It copies the given buffer into internal buffers if called between MLOCK(ErrorMessageLock) and MUNLOCK(Error↩MessageLock) for slaves and handle is StdOut or LogHandle, otherwise calls WriteFileToFile().

Parameters

handle	a file handle that specifies the output.
buffer	a pointer to the source buffer containing the data to be written.
size	the size of data to be written in bytes.

Returns

the actual size of data written to the output in bytes.

Definition at line 4398 of file parallel.c.

References VectorClear, VectorPtr, VectorPushBacks, and VectorSize.

4.99.3.49 PF_FlushStdOutBuffer()

```
void PF_FlushStdOutBuffer (
            void  )  [extern]
```

Explicitly Flushes the buffer for the standard output on the master, which is used if PF_ENABLE_STDOUT_↩ BUFFERING is defined.

Definition at line 4492 of file parallel.c.

References VectorClear, VectorPtr, and VectorSize.

4.99.3.50 PF_ReceiveRuntimeError()

```
void PF_ReceiveRuntimeError (
            void  )  [extern]
```

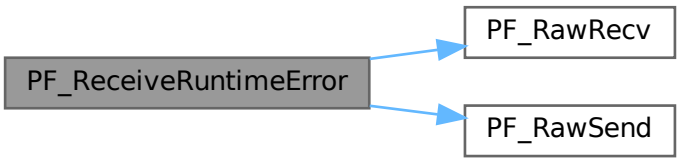
Handles a runtime error message received from another process. It ultimately calls Terminate(-1).

Definition at line 4820 of file parallel.c.

References CHECK, PF_RawRecv(), and PF_RawSend().

Referenced by PF_RecvWbuf().

Here is the call graph for this function:



4.99.4 Variable Documentation

4.99.4.1 PF

```
PARALLELVARS PF  [extern]
```

Definition at line 99 of file parallel.c.

4.99.4.2 PF_maxDollarChunkSize

```
LONG PF_maxDollarChunkSize  [extern]
```

Definition at line 74 of file mpi.c.

4.100 parallel.h

Go to the documentation of this file.

```
00001 #ifndef __PARALLEL__
00002 #define __PARALLEL__
00003
00009 /* #[ License : */
00010 /*
00011  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00012  *   When using this file you are requested to refer to the publication
00013  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00014  *   This is considered a matter of courtesy as the development was paid
00015  *   for by FOM the Dutch physics granting agency and we would like to
00016  *   be able to track its scientific use to convince FOM of its value
00017  *   for the community.
00018  *
00019  *   This file is part of FORM.
00020  *
00021  *   FORM is free software: you can redistribute it and/or modify it under the
00022  *   terms of the GNU General Public License as published by the Free Software
00023  *   Foundation, either version 3 of the License, or (at your option) any later
00024  *   version.
00025  *
00026  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00027  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00028  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00029  *   details.
00030  *
00031  *   You should have received a copy of the GNU General Public License along
00032  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00033  */
00034 /* #] License : */
00035
00036 /*
00037     #[ macros & definitions :
00038 */
00039
00040 /*
00041  * Rank of the master process.
00042  */
00043 #define MASTER 0
00044
00045 /*
00046  * Selector constants for PF_RealTime().
00047  */
00048 #define PF_RESET 0  /* reset the timer */
00049 #define PF_TIME  1  /* get the elapsed time */
00050
00051 /*
00052  * Message tags for communication during parallel execution.
00053  */
00054 #define PF_TERM_MSGTAG          10  /* master -> slave: sending terms */
00055 #define PF_ENDSORT_MSGTAG       11  /* master -> slave: no more terms to be distributed, slave ->
      master: after EndSort() */
00056 #define PF_DOLLAR_MSGTAG        12  /* slave -> master: sending $-variables */
00057 #define PF_BUFFER_MSGTAG        20  /* slave -> master: sending sorted terms, or in
      PF_SendFile()/PF_RecvFile() */
00058 #define PF_ENDBUFFER_MSGTAG     21  /* same as PF_BUFFER_MSGTAG, but indicates the end of operation */
00059 #define PF_READY_MSGTAG         30  /* slave -> master: slave is idle and can accept terms */
00060 #define PF_DATA_MSGTAG          50  /* InParallel, DoCheckpoint() */
00061 #define PF_EMPTY_MSGTAG         52  /* InParallel, DoCheckpoint(), PF_SendFile(), PF_RecvFile() */
00062 #define PF_STDOUT_MSGTAG        60  /* slave -> master: sending text to the stdout */
00063 #define PF_LOG_MSGTAG           61  /* slave -> master: sending text to the log file */
00064 #define PF_OPT_MCTS_MSGTAG      70  /* master <-> slave: optimization */
00065 #define PF_OPT_HORNER_MSGTAG    71  /* master <-> slave: optimization */
00066 #define PF_OPT_COLLECT_MSGTAG   72  /* slave -> master: optimization */
00067 #define PF_RUNTIME_ERROR_MSGTAG 80  /* slave <-> master: runtime error */
00068 #define PF_RUNTIME_SYNC_MSGTAG  81  /* master <-> slave: sync after EndSort() */
00069 #define PF_MISC_MSGTAG         100
00070
00071 /*
00072  * A macro for checking the version of gcc.
```

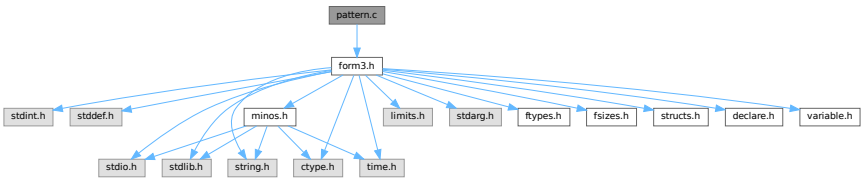
```
00073  */
00074 #if defined(__GNUC__) && defined(__GNUC_MINOR__) && defined(__GNUC_PATCHLEVEL__)
00075 #  define GNUC_PREREQ(major, minor, patchlevel) \
00076      ((__GNUC__ << 16) + (__GNUC_MINOR__ << 8) + __GNUC_PATCHLEVEL__ >= \
00077      ((major) << 16) + ((minor) << 8) + (patchlevel))
00078 #else
00079 #  define GNUC_PREREQ(major, minor, patchlevel) 0
00080 #endif
00081
00082 /*
00083  * The macro "indices" defined in variable.h collides with some function
00084  * argument names in the MPI-3.0 standard.
00085  */
00086 #undef indices
00087
00088 /* Avoid messy padding warnings which may appear in mpi.h. */
00089 #if GNUC_PREREQ(4, 6, 0)
00090 #  pragma GCC diagnostic push
00091 #  pragma GCC diagnostic ignored "-Wpadded"
00092 #  pragma GCC diagnostic ignored "-Wunused-parameter"
00093 #endif
00094 #if defined(__clang__) && defined(__has_warning)
00095 #  pragma clang diagnostic push
00096 #  if __has_warning("-Wpadded")
00097 #    pragma clang diagnostic ignored "-Wpadded"
00098 #  endif
00099 #  if __has_warning("-Wunused-parameter")
00100 #    pragma clang diagnostic ignored "-Wunused-parameter"
00101 #  endif
00102 #endif
00103
00104 #  ifdef __cplusplus
00105      /*
00106       * form3.h (which includes parallel.h) is included from newpoly.h as
00107       *   extern "C" {
00108       *   #include "form3.h"
00109       *   }
00110       * On the other hand, C++ interfaces to MPI are defined in mpi.h if it is
00111       * included from C++ sources. We first leave from the C-linkage, include
00112       * mpi.h, and then go back to the C-linkage.
00113       * (TU 7 Jun 2011)
00114       */
00115 }
00116 #    define OMPI_SKIP_MPICXX 1
00117 #    include <mpi.h>
00118 extern "C" {
00119 #  else
00120 #    define OMPI_SKIP_MPICXX 1
00121 #    include <mpi.h>
00122 #  endif
00123
00124 /* Now redefine "indices" in the same way as in variable.h. */
00125 #define indices ((INDICES)(AC.IndexList.lijst))
00126
00127 /* Restore the warning settings. */
00128 #if GNUC_PREREQ(4, 6, 0)
00129 #  pragma GCC diagnostic pop
00130 #endif
00131 #if defined(__clang__) && defined(__has_warning)
00132 #  pragma clang diagnostic pop
00133 #endif
00134
00135 #  define PF_ANY_SOURCE MPI_ANY_SOURCE
00136 #  define PF_ANY_MSGTAG MPI_ANY_TAG
00137 #  define PF_COMM MPI_COMM_WORLD
00138 #  define PF_BYTE MPI_BYTE
00139 #  define PF_INT  MPI_INT
00140 #if BITSINWORD == 32
00141 #  define PF_WORD MPI_INT32_T
00142 #  define PF_LONG MPI_INT64_T
00143 #elif BITSINWORD == 16
00144 #  define PF_WORD MPI_INT16_T
00145 #  define PF_LONG MPI_INT32_T
00146 #else
00147 #  error Can not detect if this is a 32-bit or 64-bit platform.
00148 #endif
00149
00150 /*
00151     #] macros & definitions :
00152     #[ s/r-bufs :
00153 */
00154
00159 typedef struct {
00160     WORD **buff;
00161     WORD **fill;
00162     WORD **full;
00163     WORD **stop;
```

```
00164     MPI_Status *status;
00165     MPI_Status *retstat;
00166     MPI_Request *request;
00167     MPI_Datatype *type;   /* this is needed in PF_Wait for Get_count */
00168     int *index;           /* dummies for returnvalues */
00169     int *tag;             /* for the version with blocking send/receives */
00170     int *from;
00171     int numbufs;          /* number of cyclic buffers */
00172     int active;           /* flag telling which buffer is active */
00173 } PF_BUFFER;
00174
00175 /*
00176     #] s/r-bufs :
00177     #[ global variables used by the PF_functions : need to be known everywhere
00178 */
00179
00180 typedef struct ParallelVars {
00181     FILEHANDLE  slavebuf;       /* (slave) allocated if there are RHS expressions */
00182     /* special buffers for nonblocking, unbuffered send/receives */
00183     PF_BUFFER  *sbuf;           /* set of cyclic send buffers for master _and_ slave */
00184     PF_BUFFER **rbufs;          /* array of sets of cyclic receive buffers for master */
00185     int         me;             /* Internal number of task: master is 0 */
00186     int         numtasks;       /* total number of tasks */
00187     int         parallel;       /* flags telling the master and slaves to do the sorting parallel */
00188                                 /* [05nov2003 mt] This flag must be set to 0 in iniModule! */
00189     int         rhsInParallel;  /* flag for parallel executing even if there are RHS expressions */
00190     int         mkSlaveInfile;  /* flag tells that slavebuf is used on the slaves */
00191     int         exprbufsize;    /* buffer size in WORDs to be used for transferring expressions */
00192     int         exprtodo;       /* >= 0: the expression to do in InParallel, -1: otherwise */
00193     int         log;            /* flag for logging mode */
00194     int         notMyFault;     /* flag for termination due to another process's runtime error */
00195     WORD        numsbufs;       /* number of cyclic send buffers (PF.sbuf->numbufs) */
00196     WORD        numrbufs;       /* number of cyclic receive buffers (PF.rbufs[i]->numbufs,
      i=1,...numtasks-1) */
00197 } PARALLELVARS;
00198
00199 extern PARALLELVARS PF;
00200 /*[04oct2005 mt]:*/
00201 /*for broadcasting dollarvars, see parallel.c:PF_BroadcastPreDollar():*/
00202 extern LONG PF_maxDollarChunkSize;
00203 /*:[04oct2005 mt]*/
00204
00205 /*
00206     #] global variables used by the PF_functions :
00207     #[ Function prototypes :
00208 */
00209
00210 /* mpi.c */
00211 extern int    PF_ISendSbuf(int,int);
00212 extern int    PF_Bcast(void *buffer, int count);
00213 extern int    PF_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype type, MPI_Op op,
      int root);
00214 extern int    PF_RawSend(int,void *,LONG,int);
00215 extern LONG   PF_RawRecv(int *,void *,LONG,int *);
00216
00217 extern int    PF_PreparePack(void);
00218 extern int    PF_Pack(const void *buffer, size_t count, MPI_Datatype type);
00219 extern int    PF_Unpack(void *buffer, size_t count, MPI_Datatype type);
00220 extern int    PF_PackString(const UBYTE *str);
00221 extern int    PF_UnpackString(UBYTE *str);
00222 extern int    PF_Send(int to, int tag);
00223 extern int    PF_Receive(int src, int tag, int *psrc, int *ptag);
00224 extern int    PF_Broadcast(void);
00225
00226 extern int    PF_PrepareLongSinglePack(void);
00227 extern int    PF_LongSinglePack(const void *buffer, size_t count, MPI_Datatype type);
00228 extern int    PF_LongSingleUnpack(void *buffer, size_t count, MPI_Datatype type);
00229 extern int    PF_LongSingleSend(int to, int tag);
00230 extern int    PF_LongSingleReceive(int src, int tag, int *psrc, int *ptag);
00231
00232 extern int    PF_PrepareLongMultiPack(void);
00233 extern int    PF_LongMultiPackImpl(const void *buffer, size_t count, size_t eSize, MPI_Datatype type);
00234 extern int    PF_LongMultiUnpackImpl(void *buffer, size_t count, size_t eSize, MPI_Datatype type);
00235 extern int    PF_LongMultiBroadcast(void);
00236
00237 static inline size_t sizeof_datatype(MPI_Datatype type)
00238 {
00239     if ( type == PF_BYTE ) return sizeof(char);
00240     if ( type == PF_INT  ) return sizeof(int);
00241     if ( type == PF_WORD ) return sizeof(WORD);
00242     if ( type == PF_LONG ) return sizeof(LONG);
00243     return(0);
00244 }
00245
00246 #define PF_LongMultiPack(buffer, count, type) PF_LongMultiPackImpl(buffer, count,
      sizeof_datatype(type), type)
00247 #define PF_LongMultiUnpack(buffer, count, type) PF_LongMultiUnpackImpl(buffer, count,
```

```
      sizeof_datatype(type), type)
00248
00249 /* parallel.c */
00250 extern int    PF_EndSort(void);
00251 extern WORD   PF_Deferred(WORD *,WORD);
00252 extern int    PF_Processor(EXPRESSIONS,WORD,WORD);
00253 extern int    PF_Init(int*,char ***);
00254 extern void   PF_PreTerminate(int errorcode);
00255 extern int    PF_Terminate(int);
00256 extern LONG   PF_GetSlaveTimes(void);
00257 extern LONG   PF_BroadcastNumber(LONG);
00258 extern void   PF_BroadcastBuffer(WORD **buffer, LONG *length);
00259 extern int    PF_BroadcastString(UBYTE *);
00260 extern int    PF_BroadcastPreDollar(WORD **, LONG *,int *);
00261 extern int    PF_CollectModifiedDollars(void);
00262 extern int    PF_BroadcastModifiedDollars(void);
00263 extern int    PF_BroadcastRedefinedPreVars(void);
00264 extern int    PF_BroadcastCBuf(int bufnum);
00265 extern int    PF_BroadcastExpFlags(void);
00266 extern int    PF_StoreInsideInfo(void);
00267 extern int    PF_RestoreInsideInfo(void);
00268 extern int    PF_BroadcastExpr(EXPRESSIONS e, FILEHANDLE *file);
00269 extern int    PF_BroadcastRHS(void);
00270 extern int    PF_InParallelProcessor(void);
00271 extern int    PF_SendFile(int to, FILE *fd);
00272 extern int    PF_RecvFile(int from, FILE *fd);
00273 extern void   PF_MLock(void);
00274 extern void   PF_MUnlock(void);
00275 extern LONG   PF_WriteFileToFile(int,UBYTE *,LONG);
00276 extern void   PF_FlushStdOutBuffer(void);
00277 extern void   PF_ReceiveRuntimeError(void) NORETURN;
00278
00279 /*
00280     #] Function prototypes :
00281 */
00282
00283 #endif
```

4.101 pattern.c File Reference

`#include "form3.h"`
Include dependency graph for pattern.c:



Macros

- #define PutInBuffers(pow)

Functions

- int TestMatch (PHEAD WORD *term, WORD *level)
- void Substitute (PHEAD WORD *term, WORD *pattern, WORD power)
- int FindAll (PHEAD WORD *term, WORD *pattern, WORD level, WORD *par)
- int TestSelect (WORD *term, WORD *setp)
- void SubsInAll (PHEAD0)
- void TransferBuffer (int from, int to, int spectator)
- int TakeIDfunction (PHEAD WORD *term)

4.101.1 Detailed Description

Top level pattern matching routines. More pattern matching is found in findpat.c, function.c, symmetr.c and smart.c. The last three files contain the matching inside functions. The file pattern.c contains also the very important routine Substitute. All regular pattern matching is just the finding of the pattern and indicating what are the wildcards etc. The routine Substitute does the actual removal of the pattern and replaces it by a subterm of the type SUBEXPRESSION.

Definition in file pattern.c.

4.101.2 Macro Definition Documentation

4.101.2.1 PutInBuffers

```
#define PutInBuffers(
              pow )
```

Value:

```
    AddRHS(AT.ebufnum,1); \
    *out++ = SUBEXPRESSION; \
    *out++ = SUBEXPSIZE; \
    *out++ = C->numrhs; \
    *out++ = pow; \
    *out++ = AT.ebufnum; \
    FILLSUB(out) \
    r = AT.pWorkSpace[rhs+i]; \
    if ( *r > 0 ) { \
        oldinr = r[*r]; r[*r] = 0; \
        AddNtoC(AT.ebufnum,(*r+1-ARGHEAD),(r+ARGHEAD),14); \
        r[*r] = oldinr; \
    } \
    else { \
        ToGeneral(r,buffer,1); \
        buffer[buffer[0]] = 0; \
        AddNtoC(AT.ebufnum,buffer[0]+1,buffer,15); \
    }
```

Definition at line 2195 of file pattern.c.

4.101.3 Function Documentation

4.101.3.1 TestMatch()

```
int TestMatch (
             PHEAD WORD * term,
             WORD * level )
```

This routine governs the pattern matching. If it decides that a substitution should be made, this can be either the insertion of a right hand side (C->rhs) or the automatic generation of terms as a result of an operation (like trace). The object to be replaced is removed from term and a subexpression pointer is inserted. If the substitution is made more than once there can be more subexpression pointers. Its number is positive as it corresponds to the level at which the C->rhs can be found in the compiler output. The subexpression pointer contains the wildcard substitution information. The power is found in *AT.TMout. For operations the subexpression pointer is negative and corresponds to an address in the array AT.TMout. In this array are then the instructions for the routine to be called and its number in the array 'Operations' The format is here: length,functionnumber,length-2 parameters

There is a certain complexity wrt repeat levels. Another complication is the poking of the wildcard values in the subexpression prototype in the compiler buffer. This was how things were done in the past with sequential FORM, but with the advent of TFORM this cannot be maintained. Now, for TFORM we make a copy of it. 7-may-2008 (JV):

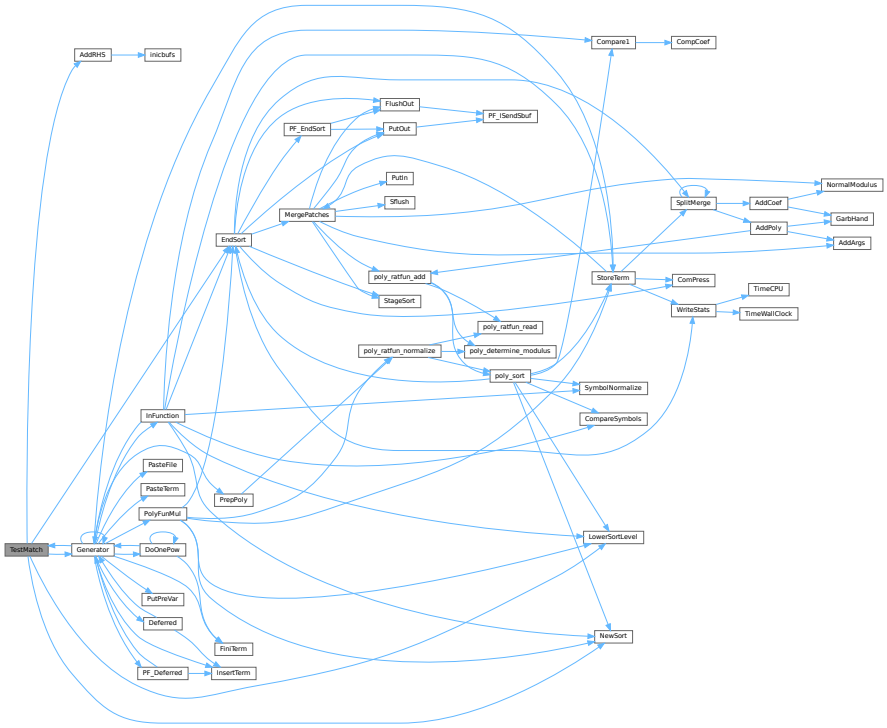
We cannot yet guarantee that this has been done 100% correctly. There are errors that occur in TFORM only and that may indicate problems.

Definition at line 97 of file pattern.c.

References AddRHS(), EndSort(), Generator(), CbUf::lhs, LowerSortLevel(), and NewSort().

Referenced by Generator().

Here is the call graph for this function:



4.101.3.2 Substitute()

```
void Substitute (
            PHEAD WORD * term,
            WORD * pattern,
            WORD power )
```

Definition at line 643 of file pattern.c.

4.101.3.3 FindAll()

```
int FindAll (
            PHEAD WORD * term,
            WORD * pattern,
            WORD level,
            WORD * par )
```

Definition at line 1199 of file pattern.c.

4.101.3.4 TestSelect()

```
int TestSelect (
            WORD * term,
            WORD * setp )
```

Definition at line 1750 of file pattern.c.

4.101.3.5 SubsInAll()

```
void SubsInAll (
            PHEAD0  )
```

Definition at line 1984 of file pattern.c.

4.101.3.6 TransferBuffer()

```
void TransferBuffer (
            int from,
            int to,
            int spectator )
```

Definition at line 2145 of file pattern.c.

4.101.3.7 TakeIDfunction()

```
int TakeIDfunction (
            PHEAD WORD * term )
```

Definition at line 2215 of file pattern.c.

4.102 pattern.c

Go to the documentation of this file.

```
00001
00012 /* #[ License : */
00013 /*
00014  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00015  *   When using this file you are requested to refer to the publication
00016  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00017  *   This is considered a matter of courtesy as the development was paid
00018  *   for by FOM the Dutch physics granting agency and we would like to
00019  *   be able to track its scientific use to convince FOM of its value
00020  *   for the community.
00021  *
00022  *   This file is part of FORM.
00023  *
00024  *   FORM is free software: you can redistribute it and/or modify it under the
00025  *   terms of the GNU General Public License as published by the Free Software
00026  *   Foundation, either version 3 of the License, or (at your option) any later
00027  *   version.
00028  *
00029  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00030  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00031  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00032  *   details.
00033  *
00034  *   You should have received a copy of the GNU General Public License along
```

```
00035  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00036  */
00037 /* #] License : */
00038 /*
00039 !!! Notice the change in OnePV in FindAll (7-may-2008 JV).
00040
00041     #[ Includes : pattern.c
00042 */
00043
00044 #include "form3.h"
00045
00046 /*
00047     #] Includes :
00048     #[ Patterns :
00049         #[ Rules :
00050
00051         There are several rules governing the allowable replacements.
00052         1:  Multi with anything but symbols or dotproducts reverts
00053             to many.
00054         2:  Each symbol can have only one (wildcard) power, so
00055             x^2*x^n? is illegal.
00056         3:  when a single vector is used it replaces all occurrences
00057             of the vector. Therefore q*q(mu) or q*q(mu) cannot occur.
00058             Also q*q cannot be done.
00059         4:  Loose vector elements are replaced with p(mu), dotproducts
00060             with p?.q.
00061         5:  p?.q? is allowed.
00062         6:  x^n? can revert to n = 0 if there is no power of x.
00063         7:  x?^n? must match some x. There could be an ambiguity otherwise.
00064
00065         #] Rules :
00066         #[ TestMatch :          WORD TestMatch(term,level)
00067 */
00068
00097 int TestMatch(PHEAD WORD *term, WORD *level)
00098 {
00099     GETBIDENTITY
00100     WORD *ll, *m, *w, *llf, *OldWork, *StartWork, *ww, *mm, *t, *OldTermBuffer = 0;
00101     WORD power = 0, i, msign = 0, ll2;
00102     int match = 0;
00103     int numdollars = 0, protosize, oldallnumrhs;
00104     CBUF *C = cbuf+AM.rbufnum, *CC;
00105     AT.idallflag = 0;
00106     do {
00107 /*
00108         #[ Preliminaries :
00109 */
00110     ll = C->lhs[*level];
00111     if ( *ll == TYPEEXPRESSION ) {
00112 /*
00113         Expressions are not subject to anything.
00114 */
00115         return(0);
00116     }
00117     else if ( *ll == TYPEREPEAT ) {
00118         *++AN.RepPoint = 0;
00119         return(0);          /* Will force the next level */
00120     }
00121     else if ( *ll == TYPEENDREPEAT ) {
00122         if ( *AN.RepPoint ) {
00123             AN.RepPoint[-1] = 1;        /* Mark the higher level as dirty */
00124             *AN.RepPoint = 0;
00125             *level = ll[2];         /* Level to jump back to */
00126         }
00127         else {
00128             AN.RepPoint--;
00129             if ( AN.RepPoint < AT.RepCount ) {
00130 /* INTERNAL_ERROR_EXCL_START */
00131                 MLOCK(ErrorMessageLock);
00132                 MesPrint("!>Internal problems with REPEAT count");
00133                 MUNLOCK(ErrorMessageLock);
00134                 Terminate(-1);
00135 /* INTERNAL_ERROR_EXCL_STOP */
00136             }
00137         }
00138         return(0);          /* Force the next level */
00139     }
00140     else if ( *ll == TYPEOPERATION ) {
00141 /*
00142         Operations have always their own level.
00143 */
00144         if ( (*(FG.OperaFind[ll[2]]))(BHEAD term,ll) ) return(-1);
00145         else return(0);
00146     }
00147 /*
00148         #] Preliminaries :
00149 */
```

```
00150     OldWork = AT.WorkPointer;
00151     if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
00152     ww = AT.WorkPointer;
00153 /*
00154         Here we need to make a copy of the subexpression object because we
00155         will be writing the values of the wildcards in it.
00156         Originally we copied it into the private version of the compiler buffer
00157         that is used for scratch space (ebufnum). This caused errors in the
00158         routines like ScanFunctions when the ebufnum Buffer was expanded
00159         and inpat was still pointing at the old Buffer. This expansion
00160         could be done in AddWild and hence cannot be fixed at > 100 places.
00161         The solution is to use AN.patternbuffer (JV 16-mar-2009).
00162 */
00163     {
00164         WORD *ta = ll, *ma;
00165         int ja = ta[1];
00166 /*
00167         New code (16-mar-2009) JV
00168 */
00169         if ( ( ja + 2 ) > AN.patternbuffersize ) {
00170             if ( AN.patternbuffer ) M_free(AN.patternbuffer,"AN.patternbuffer");
00171             AN.patternbuffersize = 2 * ja + 2;
00172             AN.patternbuffer = (WORD *)Malloc1(AN.patternbuffersize * sizeof(WORD),
00173                     "AN.patternbuffer");
00174         }
00175         ma = AN.patternbuffer;
00176         m = ma + IDHEAD;
00177         NCOPY(ma,ta,ja);
00178         *ma = 0;
00179     }
00180     AN.FullProto = m;
00181     AN.WildValue = w = m + SUBEXPSIZE;
00182     protosize = IDHEAD + m[1];
00183     m += m[1];
00184     AN.WildStop = m;
00185     StartWork = ww;
00186     ll2 = ll[2];
00187 /*
00188         #[ Expand dollars :
00189 */
00190     if ( ( ll[4] & DOLLARFLAG ) != 0 ) {    /* We have at least one dollar in the pattern */
00191         WORD oldRepPoint = *AN.RepPoint, olddefer = AR.DeferFlag;
00192         AR.Eside = LHSIDEX;
00193 /*
00194         Copy into WorkSpace. This means that AN.patternbuffer will be free.
00195 */
00196         ww = AT.WorkPointer; i = m[0]; mm = m;
00197         NCOPY(ww,mm,i);
00198         *StartWork += 3;
00199         *ww++ = 1; *ww++ = 1; *ww++ = 3;
00200         AT.WorkPointer = ww;
00201         AR.DeferFlag = 0;
00202         NewSort(BHEAD0);
00203         if ( Generator(BHEAD StartWork,AR.Cnumlhs) ) {
00204             LowerSortLevel();
00205             AT.WorkPointer = OldWork;
00206             AR.DeferFlag = olddefer;
00207             return(-1);
00208         }
00209         AT.WorkPointer = ww;
00210         if ( EndSort(BHEAD ww,0) < 0 ) {}
00211         AR.DeferFlag = olddefer;
00212         if ( *ww == 0 || *(ww+*ww) != 0 ) {
00213             if ( AP.lhdollarerror == 0 ) {
00214 /*
00215                 If race condition we just get more error messages
00216 */
00217                 MLOCK(ErrorMessageLock);
00218                 MesPrint("&LHS must be one term");
00219                 MUNLOCK(ErrorMessageLock);
00220                 AP.lhdollarerror = 1;
00221             }
00222             AT.WorkPointer = OldWork;
00223             return(-1);
00224         }
00225         m = ww; ww = m + *m;
00226         if ( m[*m-1] < 0 ) { msign = 1; m[*m-1] = -m[*m-1]; }
00227         if ( *ww || m[*m-1] != 3 || m[*m-2] != 1 || m[*m-3] != 1 ) {
00228             MLOCK(ErrorMessageLock);
00229             MesPrint("Dollar variable develops into an illegal pattern in id-statement");
00230             MUNLOCK(ErrorMessageLock);
00231             return(-1);
00232         }
00233         *m -= m[*m-1];
00234         if ( ( *m + 1 + protosize ) > AN.patternbuffersize ) {
00235             if ( AN.patternbuffer ) M_free(AN.patternbuffer,"AN.patternbuffer");
00236             AN.patternbuffersize = 2 * (*m) + 2 + protosize;
```

```
00237             AN.patternbuffer = (WORD *)Malloc1(AN.patternbuffersize * sizeof(WORD),
00238                     "AN.patternbuffer");
00239             mm = ll; ww = AN.patternbuffer; i = protosize;
00240             NCOPY(ww,mm,i);
00241             AN.FullProto = AN.patternbuffer + IDHEAD;
00242             AN.WildValue = w = AN.FullProto + SUBEXPSIZE;
00243             AN.WildStop = AN.patternbuffer + protosize;
00244         }
00245         mm = AN.patternbuffer + protosize;
00246         i = *m;
00247         NCOPY(mm,m,i);
00248         m = AN.patternbuffer + protosize;
00249         AR.Eside = RHSIDE;
00250         *mm = 0;
00251 /*
00252         Test the pattern. If only wildcard powers -> SUBONCE
00253 */
00254         {
00255             WORD *mmm = m + *m, *m1 = m+1, jm, noveto = 0;
00256             while ( m1 < mmm ) {
00257                 if ( *m1 == SYMBOL ) {
00258                     for ( jm = 2; jm < m1[1]; jm+=2 ) {
00259                         if ( m1[jm+1] < MAXPOWER && m1[jm+1] > -MAXPOWER ) break;
00260                     }
00261                     if ( jm < m1[1] ) { noveto = 1; break; }
00262                 }
00263                 else if ( *m1 == DOTPRODUCT ) {
00264                     for ( jm = 2; jm < m1[1]; jm+=3 ) {
00265                         if ( m1[jm+2] < MAXPOWER && m1[jm+2] > -MAXPOWER ) break;
00266                     }
00267                     if ( jm < m1[1] ) { noveto = 1; break; }
00268                 }
00269                 else { noveto = 1; break; }
00270                 m1 += m1[1];
00271             }
00272             if ( noveto == 0 ) {
00273                 ll2 = ll2 & ~SUBMASK;
00274                 ll2 |= SUBONCE;
00275             }
00276         }
00277         AT.WorkPointer = ww = StartWork;
00278         *AN.RepPoint = oldRepPoint;
00279     }
00280 /*
00281         #] Expand dollars :
00282 
00283     In case of id,all we have to check at this point that there are only
00284     functions in the pattern.
00285 */
00286     if ( ( ll2 & SUBMASK ) == SUBALL ) {
00287         WORD *t = AN.patternbuffer+IDHEAD, *tt;
00288         WORD *tstop, *ttstop, ii;
00289         t += t[1]; tstop = t + *t; t++;
00290         while ( t < tstop ) {
00291             if ( *t < FUNCTION ) break;
00292             t += t[1];
00293         }
00294         if ( t < tstop ) {
00295             MLOCK(ErrorMessageLock);
00296             MesPrint("Error: id,all can only be used with (products of) functions and/or tensors.");
00297             MUNLOCK(ErrorMessageLock);
00298             return(-1);
00299         }
00300         OldTermBuffer = AN.termbuffer;
00301         AN.termbuffer = TermMalloc("id,all");
00302 /*
00303         Now make sure that only regular functions and tensors can take part.
00304 */
00305         tt = term; ttstop = tt+*tt; ttstop -= ABS(ttstop[-1]); tt++;
00306         t = AN.termbuffer+1;
00307         while ( tt < ttstop ) {
00308             if ( *tt >= FUNCTION && *tt != AR.PolyFun && *tt != AR.PolyFunInv ) {
00309                 ii = tt[1]; NCOPY(t,tt,ii);
00310             }
00311             else tt += tt[1];
00312         }
00313         *t++ = 1; *t++ = 1; *t++ = 3; AN.termbuffer[0] = t-AN.termbuffer;
00314     }
00315 /*
00316     To be puristic, we need to check that all wildcards in the prototype
00317     are actually present. If the LHS contained a replace_ this may not be
00318     the case.
00319 */
00320     ClearWild(BHEAD0);
00321     while ( w < AN.WildStop ) {
00322         if ( *w == LOADDOLLAR ) numdollars++;
00323         w += w[1];
```

```
00324     }
00325     AN.RepFunNum = 0;
00326     /* rep = */ AN.RepFunList = AT.WorkPointer;
00327     AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer/2);
00328     if ( AT.WorkPointer >= AT.WorkTop ) {
00329         MLOCK(ErrorMessageLock);
00330         MesWork();
00331         MUNLOCK(ErrorMessageLock);
00332         return(-1);
00333     }
00334     AN.DisOrderFlag = ll2 & SUBDISORDER;
00335     AN.nogroundlevel = 0;
00336     switch ( ll2 & SUBMASK ) {
00337         case SUBONLY :
00338             /* Must be an exact match */
00339             AN.UseFindOnly = 1; AN.ForFindOnly = 0;
00340             if ( FindRest(BHEAD term,m) && ( AN.UsedOtherFind ||
00341                 FindOnly(BHEAD term,m) ) ) {
00342                 power = 1;
00343                 if ( msign ) term[term[0]-1] = -term[term[0]-1];
00344             }
00345             else power = 0;
00346             break;
00347         case SUBMANY :
00348             AN.UseFindOnly = -1;
00349             if ( ( power = FindRest(BHEAD term,m) ) > 0 ) {
00350                 if ( ( power = FindOnce(BHEAD term,m) ) > 0 ) {
00351                     AN.UseFindOnly = 0;
00352                     do {
00353                         if ( msign ) term[term[0]-1] = -term[term[0]-1];
00354                         Substitute(BHEAD term,m,1);
00355                         if ( numdollars ) {
00356                             WildDollars(BHEAD (WORD *)0);
00357                             numdollars = 0;
00358                         }
00359                         if ( ww < term+term[0] ) ww = term+term[0];
00360                         ClearWild(BHEAD0);
00361                         AT.WorkPointer = ww;
00362 /*                      if ( rep < ww ) {*/
00363                             AN.RepFunNum = 0;
00364                             /* rep = */ AN.RepFunList = ww;
00365                             AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer/2);
00366                             if ( AT.WorkPointer >= AT.WorkTop ) {
00367                                 MLOCK(ErrorMessageLock);
00368                                 MesWork();
00369                                 MUNLOCK(ErrorMessageLock);
00370                                 return(-1);
00371                             }
00372 /*
00373                         }
00374                         else {
00375                             AN.RepFunList = rep;
00376                             AN.RepFunNum = 0;
00377                         }
00378 */
00379                         AN.nogroundlevel = 0;
00380                     } while ( FindRest(BHEAD term,m) && ( AN.UsedOtherFind ||
00381                             FindOnce(BHEAD term,m) ) );
00382                     match = 1;
00383                 }
00384                 else if ( power < 0 ) {
00385                     do {
00386                         if ( msign ) term[term[0]-1] = -term[term[0]-1];
00387                         Substitute(BHEAD term,m,1);
00388                         if ( numdollars ) {
00389                             WildDollars(BHEAD (WORD *)0);
00390                             numdollars = 0;
00391                         }
00392                         if ( ww < term+term[0] ) ww = term+term[0];
00393                         ClearWild(BHEAD0);
00394                         AT.WorkPointer = ww;
00395 /*                      if ( rep < ww ) { */
00396                             AN.RepFunNum = 0;
00397                             /* rep = */ AN.RepFunList = ww;
00398                             AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer/2);
00399                             if ( AT.WorkPointer >= AT.WorkTop ) {
00400                                 MLOCK(ErrorMessageLock);
00401                                 MesWork();
00402                                 MUNLOCK(ErrorMessageLock);
00403                                 return(-1);
00404                             }
00405 /*
00406                         }
00407                         else {
00408                             AN.RepFunList = rep;
00409                             AN.RepFunNum = 0;
00410                         }
```

```
00411 */
00412                     } while ( FindRest(BHEAD term,m) );
00413                     match = 1;
00414                 }
00415             }
00416             else if ( power < 0 ) {
00417                 if ( FindOnce(BHEAD term,m) ) {
00418                     do {
00419                         if ( msign ) term[term[0]-1] = -term[term[0]-1];
00420                         Substitute(BHEAD term,m,1);
00421                         if ( numdollars ) {
00422                             WildDollars(BHEAD (WORD *)0);
00423                             numdollars = 0;
00424                         }
00425                         if ( ww < term+term[0] ) ww = term+term[0];
00426                         ClearWild(BHEAD0);
00427                         AT.WorkPointer = ww;
00428 /*                      if ( rep < ww ) { */
00429                             AN.RepFunNum = 0;
00430                             /* rep = */ AN.RepFunList = ww;
00431                             AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer/2);
00432                             if ( AT.WorkPointer >= AT.WorkTop ) {
00433                                 MLOCK(ErrorMessageLock);
00434                                 MesWork();
00435                                 MUNLOCK(ErrorMessageLock);
00436                                 return(-1);
00437                             }
00438 /*
00439                         }
00440                         else {
00441                             AN.RepFunList = rep;
00442                             AN.RepFunNum = 0;
00443                         }
00444 */
00445                     } while ( FindOnce(BHEAD term,m) );
00446                     match = 1;
00447                 }
00448             }
00449             if ( match ) {
00450                 if ( ( ll2 & SUBAFTER ) != 0 ) *level = AC.Labels[ll[3]];
00451             }
00452             else {
00453                 if ( ( ll2 & SUBAFTERNOT ) != 0 ) *level = AC.Labels[ll[3]];
00454             }
00455             goto nextlevel;
00456         case SUBONCE :
00457             AN.UseFindOnly = 0;
00458             if ( FindRest(BHEAD term,m) && ( AN.UsedOtherFind || FindOnce(BHEAD term,m) ) ) {
00459                 power = 1;
00460                 if ( msign ) term[term[0]-1] = -term[term[0]-1];
00461             }
00462             else power = 0;
00463             break;
00464         case SUBMULTI :
00465             power = FindMulti(BHEAD term,m);
00466             if ( ( power & 1 ) != 0 && msign ) term[term[0]-1] = -term[term[0]-1];
00467             break;
00468         case SUBVECTOR :
00469             while ( ( power = FindAll(BHEAD term,m,*level,(WORD *)0) ) != 0 ) {
00470                 if ( ( power & 1 ) != 0 && msign ) term[term[0]-1] = -term[term[0]-1];
00471                 match = 1;
00472             }
00473             break;
00474         case SUBSELECT :
00475             llf = ll + IDHEAD;  llf += llf[1];  llf += *llf;
00476             AN.UseFindOnly = 1; AN.ForFindOnly = llf;
00477             if ( FindRest(BHEAD term,m) && ( AN.UsedOtherFind || FindOnly(BHEAD term,m) ) ) {
00478                 if ( msign ) term[term[0]-1] = -term[term[0]-1];
00479 /*
00480                 The following code needs to be hacked a bit to allow for
00481                 all types of sets and for occurrence anywhere in the term
00482                 The code at the end of FindOnly is a bit mysterious.
00483 */
00484                 if ( llf[1] > 2 ) {
00485                     WORD *t1, *t2;
00486                     if ( *term > AN.sizeselecttermundo ) {
00487                         if ( AN.selecttermundo ) M_free(AN.selecttermundo,"AN.selecttermundo");
00488                         AN.sizeselecttermundo = *term +10;
00489                         AN.selecttermundo = (WORD *)Malloc1(
00490                             AN.sizeselecttermundo*sizeof(WORD),"AN.selecttermundo");
00491                     }
00492                     t1 = term; t2 = AN.selecttermundo; i = *term;
00493                     NCOPY(t2,t1,i);
00494                 }
00495                 power = 1;
00496                 Substitute(BHEAD term,m,power);
00497                 if ( llf[1] > 2 ) {
```

```
00498                     if ( TestSelect(term,llf) ) {
00499                         WORD *t1, *t2;
00500                         power = 0;
00501                         t1 = term; t2 = AN.selecttermundo; i = *t2;
00502                         NCOPY(t1,t2,i);
00503 #if IDHEAD > 3
00504                         if ( ( ll2 & SUBAFTERNOT ) != 0 ) {
00505                             *level = AC.Labels[ll[3]];
00506                         }
00507 #endif
00508                         goto nextlevel;
00509                     }
00510                 }
00511                 if ( numdollars ) {
00512                     WildDollars(BHEAD (WORD *)0);
00513                     numdollars = 0;
00514                 }
00515                 match = 1;
00516                 if ( ( ll2 & SUBAFTER ) != 0 ) {
00517                     *level = AC.Labels[ll[3]];
00518                 }
00519             }
00520             else {
00521                 if ( ( ll2 & SUBAFTERNOT ) != 0 ) {
00522                     *level = AC.Labels[ll[3]];
00523                 }
00524                 power = 0;
00525             }
00526             goto nextlevel;
00527         case SUBALL:
00528             AN.UseFindOnly = 0;
00529             CC = cbuf+AT.allbufnum;
00530             oldallnumrhs = CC->numrhs;
00531             t = AddRHS(AT.allbufnum,1);
00532             *t = 0;
00533             AT.idallflag = 1;
00534             AT.idallmaxnum = ll[5];
00535             AT.idallnum = 0;
00536             if ( FindRest(BHEAD AN.termbuffer,m) || AT.idallflag > 1 ) {
00537                 WORD *t, *tstop, *tt, first = 1, ii;
00538                 power = 1;
00539                 *CC->Pointer++ = 0;
00540                 if ( msign ) term[term[0]-1] = -term[term[0]-1];
00541 /*
00542                 If we come here the matches are all already in the
00543                 compiler buffer. All we need to do is take out all
00544                 functions and replace them by a SUBEXPRESSION that
00545                 points to this buffer.
00546                 Note: the PolyFun/PolyRatFun should be excluded from this.
00547                 This works because each match writes incrementally to
00548                 the buffer using the routine SubsInAll.
00549
00550                 The call to WildDollars should be made in Generator.....
00551 */
00552                 t = term; tstop = t + *t; ii = ABS(tstop[-1]); tstop -= ii;
00553                 tt = AT.WorkPointer+1;
00554                 t++;
00555                 while ( t < tstop ) {
00556                     if ( *t >= FUNCTION && *t != AR.PolyFun && *t != AR.PolyFunInv ) {
00557                         if ( first ) { /* SUBEXPRESSION */
00558                             *tt++ = SUBEXPRESSION;
00559                             *tt++ = SUBEXPSIZE;
00560                             *tt++ = CC->numrhs;
00561                             *tt++ = 1;
00562                             *tt++ = AT.allbufnum;
00563                             FILLSUB(tt)
00564                             first = 0;
00565                         }
00566                         t += t[1];
00567                     }
00568                     else {
00569                         i = t[1]; NCOPY(tt,t,i);
00570                     }
00571                 }
00572                 if ( ( ll[4] & NORMALIZEFLAG ) != 0 ) {
00573 /*
00574                     In case of the normalization option, we have to divide
00575                     by AT.idallnum;
00576 */
00577                     WORD na = t[ii-1];
00578                     na = REDLENG(na);
00579                     for ( i = 0; i < ii; i++ ) tt[i] = t[i];
00580                     Divvy(BHEAD (UWORD *)tt,&na,(UWORD *)(&(AT.idallnum)),1);
00581                     na = INCLENG(na);
00582                     ii = ABS(na);
00583                     tt[ii-1] = na;
00584                     tt += ii;
```

```
00585                 }
00586                 else {
00587                     NCOPY(tt,t,ii);
00588                 }
00589                 ii = tt-AT.WorkPointer;
00590                 *(AT.WorkPointer) = ii;
00591                 tt = AT.WorkPointer; t = term;
00592                 NCOPY(t,tt,ii);
00593 
00594                 if ( ( ll2 & SUBAFTER ) != 0 ) { /* ifmatch -> */
00595                     *level = AC.Labels[ll[3]];
00596                 }
00597                 TermFree(AN.termbuffer,"id,all");
00598                 AN.termbuffer = OldTermBuffer;
00599                 AT.WorkPointer = AN.RepFunList;
00600                 AT.idallflag = 0;
00601                 CC->Pointer[0] = 0;
00602                 TransferBuffer(AT.aebufnum,AT.ebufnum,AT.allbufnum);
00603                 return(1);
00604             }
00605             AT.idallflag = 0;
00606             power = 0;
00607             CC->numrhs = oldallnumrhs;
00608             TermFree(AN.termbuffer,"id,all");
00609             AN.termbuffer = OldTermBuffer;
00610             break;
00611         default :
00612             break;
00613     }
00614     if ( power ) {
00615         Substitute(BHEAD term,m,power);
00616         if ( numdollars ) {
00617             WildDollars(BHEAD (WORD *)0);
00618             numdollars = 0;
00619         }
00620         match = 1;
00621         if ( ( ll2 & SUBAFTER ) != 0 ) { /* ifmatch -> */
00622             *level = AC.Labels[ll[3]];
00623         }
00624     }
00625     else {
00626         AT.WorkPointer = AN.RepFunList;
00627         if ( ( ll2 & SUBAFTERNOT ) != 0 ) { /* ifnomatch -> */
00628             *level = AC.Labels[ll[3]];
00629         }
00630     }
00631 nextlevel:;
00632     } while ( (*level)++ < AR.Cnumlhs && C->lhs[*level][0] == TYPEIDOLD );
00633     (*level)--;
00634     AT.WorkPointer = AN.RepFunList;
00635     return(match);
00636 }
00637 
00638 /*
00639         #] TestMatch :
00640         #[ Substitute :         void Substitute(term,pattern,power)
00641 */
00642 
00643 void Substitute(PHEAD WORD *term, WORD *pattern, WORD power)
00644 {
00645     GETBIDENTITY
00646     WORD *TemTerm;
00647     WORD *t, *m;
00648     WORD *tstop, *mstop;
00649     WORD *xstop, *ystop;
00650     WORD nt, *fill, nq, mt;
00651     WORD *q, *subterm, *tcoef, oldval1 = 0, newval3, i = 0;
00652     WORD PutExpr = 0, sign = 0;
00653     TemTerm = AT.WorkPointer;
00654     if ( ( (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer*2) ) > AT.WorkTop ) {
00655         MLOCK(ErrorMessageLock);
00656         MesWork();
00657         MUNLOCK(ErrorMessageLock);
00658         Terminate(-1);
00659     }
00660     m = pattern;
00661     mstop = m + *m;
00662     m++;
00663     t = term;
00664     t += *term - 1;
00665     tcoef = t;
00666     tstop = t - ABS(*t) + 1;
00667     t = term;
00668     t++;
00669     fill = TemTerm;
00670     fill++;
00671     if ( m < mstop ) { do {
```

```
00672 /*
00673             #[ SYMBOLS :
00674 */
00675         if ( *m == SYMBOL ) {
00676             ystop = m + m[1];
00677             m += 2;
00678             while ( *t != SYMBOL && t < tstop ) {
00679                 nq = t[1];
00680                 NCOPY(fill,t,nq);
00681             }
00682             if ( t >= tstop ) goto SubCoef;
00683             *fill++ = SYMBOL;
00684             fill++;
00685             subterm = fill;
00686             xstop = t + t[1];
00687             t += 2;
00688             do {
00689                 if ( *m == *t && t < xstop ) {
00690                     nt = t[1];
00691                     mt = m[1];
00692                     if ( mt >= 2*MAXPOWER ) {
00693                         if ( CheckWild(BHEAD mt-2*MAXPOWER,SYMTONUM,-MAXPOWER,&newval3) ) {
00694                             nt -= AN.oldvalue;
00695                             goto SubsL1;
00696                         }
00697                     }
00698                     else if ( mt <= -2*MAXPOWER ) {
00699                         if ( CheckWild(BHEAD -mt-2*MAXPOWER,SYMTONUM,-MAXPOWER,&newval3) ) {
00700                             nt += AN.oldvalue;
00701                             goto SubsL1;
00702                         }
00703                     }
00704                     else {
00705                         nt -= mt * power;
00706 SubsL1:                  if ( nt ) {
00707                             *fill++ = *t;
00708                             *fill++ = nt;
00709                         }
00710                     }
00711                     m += 2; t+= 2;
00712                 }
00713                 else if ( *m >= 2*MAXPOWER ) {
00714                     while ( t < xstop ) { *fill++ = *t++; *fill++ = *t++; }
00715                     nq = WORDDIF(fill,subterm);
00716                     fill = subterm;
00717                     while ( nq > 0 ) {
00718                         if ( !CheckWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,*fill,&newval3) ) {
00719                             mt = m[1];
00720                             if ( mt >= 2*MAXPOWER ) {
00721                                 if ( CheckWild(BHEAD mt-2*MAXPOWER,SYMTONUM,-MAXPOWER,&newval3) ) {
00722                                     if ( fill[1] -= AN.oldvalue ) goto SubsL2;
00723                                 }
00724                             }
00725                             else if ( mt <= -2*MAXPOWER ) {
00726                                 if ( CheckWild(BHEAD -mt-2*MAXPOWER,SYMTONUM,-MAXPOWER,&newval3) ) {
00727                                     if ( fill[1] += AN.oldvalue ) goto SubsL2;
00728                                 }
00729                             }
00730                             else {
00731                                 if ( fill[1] -= mt * power ) {
00732 SubsL2:                              fill += nq;
00733                                     nq = 0;
00734                                 }
00735                             }
00736                             break;
00737                         }
00738                         nq -= 2;
00739                         fill += 2;
00740                     }
00741                     if ( nq ) {
00742                         nq -= 2;
00743                         q = fill + 2;
00744                         while ( --nq >= 0 ) *fill++ = *q++;
00745                     }
00746                     m += 2;
00747                 }
00748                 else if ( *m < *t || t >= xstop ) { m += 2; }
00749                 else { *fill++ = *t++; *fill++ = *t++; }
00750             } while ( m < ystop );
00751             while ( t < xstop ) *fill++ = *t++;
00752             nq = WORDDIF(fill,subterm);
00753             if ( nq > 0 ) {
00754                 nq += 2;
00755                 subterm[-1] = nq;
00756             }
00757             else { fill = subterm; fill -= 2; }
00758         }
```

```
00759 /*
00760             #] SYMBOLS :
00761             #[ DOTPRODUCTS :
00762 */
00763         else if ( *m == DOTPRODUCT ) {
00764             ystop = m + m[1];
00765             m += 2;
00766             while ( *t > DOTPRODUCT && t < tstop ) {
00767                 nq = t[1];
00768                 NCOPY(fill,t,nq);
00769             }
00770             if ( t >= tstop ) goto SubCoef;
00771             if ( *t != DOTPRODUCT ) {
00772                 m = ystop;
00773                 goto EndLoop;
00774             }
00775             *fill++ = DOTPRODUCT;
00776             fill++;
00777             subterm = fill;
00778             xstop = t + t[1];
00779             t += 2;
00780             do {
00781                 if ( *m == *t && m[1] == t[1] && t < xstop ) {
00782                     nt = t[2];
00783                     mt = m[2];
00784                     if ( mt >= 2*MAXPOWER ) {
00785                         if ( CheckWild(BHEAD mt-2*MAXPOWER,SYMTONUM,-MAXPOWER,&newval3) ) {
00786                             nt -= AN.oldvalue;
00787                             goto SubsL3;
00788                         }
00789                     }
00790                     else if ( mt <= -2*MAXPOWER ) {
00791                         if ( CheckWild(BHEAD -mt-2*MAXPOWER,SYMTONUM,-MAXPOWER,&newval3) ) {
00792                             nt += AN.oldvalue;
00793                             goto SubsL3;
00794                         }
00795                     }
00796                     else {
00797                         nt -= mt * power;
00798 SubsL3:                  if ( nt ) {
00799                             *fill++ = *t++;
00800                             *fill++ = *t;
00801                             *fill++ = nt;
00802                             t += 2;
00803                         }
00804                         else t += 3;
00805                     }
00806                     m += 3;
00807                 }
00808                 else if ( *m >= (AM.OffsetVector+WILDOFFSET) ) {
00809                     while ( t < xstop ) {
00810                         *fill++ = *t++; *fill++ = *t++; *fill++ = *t++;
00811                     }
00812                     oldval1 = 1;
00813                     goto SubsL4;
00814                 }
00815                 else if ( m[1] >= (AM.OffsetVector+WILDOFFSET) ) {
00816                     while ( *m >= *t && t < xstop ) {
00817                         *fill++ = *t++; *fill++ = *t++; *fill++ = *t++;
00818                     }
00819                     oldval1 = 0;
00820 SubsL4:             nq = WORDDIF(fill,subterm);
00821                     fill = subterm;
00822                     while ( nq > 0 ) {
00823                         if ( ( oldval1 && ( (
00824                            !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*fill,&newval3)
00825                         && !CheckWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,fill[1],&newval3)
00826                         ) || (
00827                            !CheckWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,*fill,&newval3)
00828                         && !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,fill[1],&newval3)
00829                         ) ) ) || ( !oldval1 && ( (
00830                            *m == *fill
00831                         && !CheckWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,fill[1],&newval3)
00832                         ) || (
00833                            !CheckWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,*fill,&newval3)
00834                         && *m == fill[1] ) ) ) ) {
00835                             mt = m[2];
00836                             if ( mt >= 2*MAXPOWER ) {
00837                                 if ( CheckWild(BHEAD mt-2*MAXPOWER,SYMTONUM,-MAXPOWER,&newval3) ) {
00838                                     if ( fill[2] -= AN.oldvalue )
00839                                             goto SubsL5;
00840                                 }
00841                             }
00842                             else if ( mt <= -2*MAXPOWER ) {
00843                                 if ( CheckWild(BHEAD -mt-2*MAXPOWER,SYMTONUM,-MAXPOWER,&newval3) ) {
00844                                     if ( fill[2] += AN.oldvalue )
00845                                             goto SubsL5;
```

```
00846                                   }
00847                               }
00848                               else {
00849                                   if ( fill[2] -= mt * power ) {
00850 SubsL5:                              fill += nq;
00851                                       nq = 0;
00852                                   }
00853                               }
00854                               m += 3;
00855                               break;
00856                           }
00857                           fill += 3; nq -= 3;
00858                       }
00859                       if ( nq ) {
00860                           nq -= 3;
00861                           q = fill + 3;
00862                           while ( --nq >= 0 ) *fill++ = *q++;
00863                       }
00864                   }
00865                   else if ( t >= xstop || *m < *t || ( *m == *t && m[1] < t[1] ) )
00866                       { m += 3; }
00867                   else {
00868                       *fill++ = *t++; *fill++ = *t++; *fill++ = *t++;
00869                   }
00870               } while ( m < ystop );
00871               while ( t < xstop ) *fill++ = *t++;
00872               nq = WORDDIF(fill,subterm);
00873               if ( nq > 0 ) {
00874                   nq += 2;
00875                   subterm[-1] = nq;
00876               }
00877               else { fill = subterm; fill -= 2; }
00878           }
00879 /*
00880               #] DOTPRODUCTS :
00881               #[ FUNCTIONS :
00882 */
00883           else if ( *m >= FUNCTION ) {
00884               while ( *t >= FUNCTION || *t == SUBEXPRESSION ) {
00885                   nt = WORDDIF(t,term);
00886                   for ( mt = 0; mt < AN.RepFunNum; mt += 2 ) {
00887                       if ( nt == AN.RepFunList[mt] ) break;
00888                   }
00889                   if ( mt >= AN.RepFunNum ) {
00890                       nq = t[1];
00891                       NCOPY(fill,t,nq);
00892                   }
00893                   else {
00894                       WORD *oldt = 0;
00895                       if ( *m == GAMMA && m[1] != FUNHEAD+1 ) {
00896                           oldt = t;
00897                           if ( ( i = AN.RepFunList[mt+1] ) > 0 ) {
00898                               *fill++ = GAMMA;
00899                               *fill++ = i + FUNHEAD+1;
00900                               FILLFUN(fill)
00901                               nq = i + 1;
00902                               t += FUNHEAD;
00903                               NCOPY(fill,t,nq);
00904                           }
00905                           t = oldt;
00906                       }
00907                       else if ( ( *t == LEVICIVITA ) || ( *t >= FUNCTION
00908                       && (functions[*t-FUNCTION].symmetric & ~REVERSEORDER) == ANTISYMMETRIC )
00909                                   ) sign += AN.RepFunList[mt+1];
00910                       else if ( *m >= FUNCTION+WILDOFFSET
00911                       && (functions[*m-FUNCTION-WILDOFFSET].symmetric & ~REVERSEORDER) == ANTISYMMETRIC
00912                                   ) sign += AN.RepFunList[mt+1];
00913                       if ( !PutExpr ) {
00914                           xstop = t + t[1];
00915                           t = AN.FullProto;
00916                           nq = t[1];
00917                           t[3] = power;
00918                           NCOPY(fill,t,nq);
00919                           t = xstop;
00920                           PutExpr = 1;
00921                       }
00922                       else t += t[1];
00923                       if ( *m == GAMMA && m[1] != FUNHEAD+1 ) {
00924                           i = oldt[1] - m[1] - i;
00925                           if ( i > 0 ) {
00926                               *fill++ = GAMMA;
00927                               *fill++ = i + FUNHEAD+1;
00928                               FILLFUN(fill)
00929                               *fill++ = oldt[FUNHEAD];
00930                               t = t - i;
00931                               NCOPY(fill,t,i);
00932                           }
```

```
00933                     }
00934                     break;
00935                 }
00936             }
00937             m += m[1];
00938         }
00939 /*
00940             #] FUNCTIONS :
00941             #[ VECTORS :
00942 */
00943         else if ( *m == VECTOR ) {
00944             while ( *t > VECTOR ) {
00945                 nq = t[1];
00946                 NCOPY(fill,t,nq);
00947             }
00948             xstop = t + t[1];
00949             ystop = m + m[1];
00950             t += 2;
00951             m += 2;
00952             *fill++ = VECTOR;
00953             fill++;
00954             subterm = fill;
00955             do {
00956                 if ( *m == *t && m[1] == t[1] ) {
00957                     m += 2; t += 2;
00958                 }
00959                 else if ( *m >= (AM.OffsetVector+WILDOFFSET) ) {
00960                     while ( t < xstop ) *fill++ = *t++;
00961                     nq = WORDDIF(fill,subterm);
00962                     fill = subterm;
00963                     if ( m[1] < (AM.OffsetIndex+WILDOFFSET) ) {
00964                         do {
00965                             if ( m[1] == fill[1] &&
00966                             !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*fill,&newval3) )
00967                                 break;
00968                             fill += 2;
00969                             nq -= 2;
00970                         } while ( nq > 0 );
00971                     }
00972                     else {      /* Double wildcard */
00973                         do {
00974                             if ( !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,fill[1],&newval3)
00975                             && !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*fill,&newval3) )
00976                                 break;
00977                             if ( *fill == oldval1 && fill[1] == AN.oldvalue ) break;
00978                             fill += 2;
00979                             nq -= 2;
00980                         } while ( nq > 0 );
00981                     }
00982                     nq -= 2;
00983                     q = fill + 2;
00984                     if ( nq > 0 ) { NCOPY(fill,q,nq); }
00985                     m += 2;
00986                 }
00987                 else if ( *m <= *t &&
00988                 m[1] >= (AM.OffsetIndex + WILDOFFSET) ) {
00989                     while ( *m == *t && t < xstop )
00990                         { *fill++ = *t++; *fill++ = *t++; }
00991                     nq = WORDDIF(fill,subterm);
00992                     fill = subterm;
00993                     do {
00994                         if ( *m == *fill &&
00995                         !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,fill[1],&newval3) )
00996                             break;
00997                         nq -= 2;
00998                         fill += 2;
00999                     } while ( nq > 0 );
01000                     nq -= 2;
01001                     q = fill + 2;
01002                     if ( nq > 0 ) { NCOPY(fill,q,nq); }
01003                     m += 2;
01004                 }
01005                 else { *fill++ = *t++; *fill++ = *t++; }
01006             } while ( m < ystop );
01007             while ( t < xstop ) *fill++ = *t++;
01008             nq = WORDDIF(fill,subterm);
01009             if ( nq > 0 ) {
01010                 nq += 2;
01011                 subterm[-1] = nq;
01012             }
01013             else { fill = subterm; fill -= 2; }
01014         }
01015 /*
01016             #] VECTORS :
01017             #[ INDICES :
01018
01019             Currently without wildcards
```

```
01020 */
01021         else if ( *m == INDEX ) {
01022             while ( *t > INDEX ) {
01023                 nq = t[1];
01024                 NCOPY(fill,t,nq);
01025             }
01026             xstop = t + t[1];
01027             ystop = m + m[1];
01028             t += 2;
01029             m += 2;
01030             *fill++ = INDEX;
01031             fill++;
01032             subterm = fill;
01033             do {
01034                 if ( *m == *t ) {
01035                     m += 1; t += 1;
01036                 }
01037                 else if ( *m >= (AM.OffsetIndex+WILDOFFSET) ) {
01038                     while ( t < xstop ) *fill++ = *t++;
01039                     nq = WORDDIF(fill, subterm);
01040                     fill = subterm;
01041                     do {
01042                         if ( !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,*fill,&newval3) ) {
01043                             break;
01044                         }
01045                         fill += 1;
01046                         nq -= 1;
01047                     } while ( nq > 0 );
01048                     nq -= 1;
01049                     if ( nq > 0 ) {
01050                         q = fill + 1;
01051                         NCOPY(fill,q,nq);
01052                     }
01053                     m += 1;
01054                 }
01055                 else {
01056                     *fill++ = *t++;
01057                 }
01058             } while ( m < ystop );
01059             while ( t < xstop ) *fill++ = *t++;
01060             nq = WORDDIF(fill,subterm);
01061             if ( nq > 0 ) {
01062                 nq += 2;
01063                 subterm[-1] = nq;
01064             }
01065             else { fill = subterm; fill -= 2; }
01066         }
01067 /*
01068             #] INDICES :
01069             #[ DELTAS :
01070 */
01071         else if ( *m == DELTA ) {
01072             while ( *t > DELTA ) {
01073                 nq = t[1];
01074                 NCOPY(fill,t,nq);
01075             }
01076             xstop = t + t[1];
01077             ystop = m + m[1];
01078             t += 2;
01079             m += 2;
01080             *fill++ = DELTA;
01081             fill++;
01082             subterm = fill;
01083             do {
01084                 if ( *t == *m && t[1] == m[1] ) { m += 2; t += 2; }
01085                 else if ( *m >= (AM.OffsetIndex+WILDOFFSET) ) { /* Two dummies */
01086                     while ( t < xstop ) *fill++ = *t++;
01087 /*                  fill = subterm; */
01088                     oldval1 = 1;
01089                     goto SubsL6;
01090                 }
01091                 else if ( m[1] >= (AM.OffsetIndex+WILDOFFSET) ) {
01092                     while ( (*m == *t || *m == t[1] ) && ( t < xstop ) ) {
01093                         *fill++ = *t++; *fill++ = *t++;
01094                     }
01095                     oldval1 = 0;
01096 SubsL6:             nq = WORDDIF(fill,subterm);
01097                     fill = subterm;
01098                     do {
01099                         if ( ( oldval1 && ( (
01100                            !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,*fill,&newval3)
01101                         && !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,fill[1],&newval3)
01102                         ) || (
01103                            !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,*fill,&newval3)
01104                         && !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,fill[1],&newval3)
01105                         ) ) ) || ( !oldval1 && ( (
01106                            *m == *fill
```

```
01107                         && !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,fill[1],&newval3)
01108                         ) || (
01109                            *m == fill[1]
01110                         && !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,*fill,&newval3)
01111                         ) ) ) ) break;
01112                         fill += 2;
01113                         nq -= 2;
01114                     } while ( nq > 0 );
01115                     nq -= 2;
01116                     if ( nq > 0 ) {
01117                         q = fill + 2;
01118                         NCOPY(fill,q,nq);
01119                     }
01120                     m += 2;
01121                 }
01122                 else {
01123                     *fill++ = *t++; *fill++ = *t++;
01124                 }
01125             } while ( m < ystop );
01126             while ( t < xstop ) *fill++ = *t++;
01127             nq = WORDDIF(fill,subterm);
01128             if ( nq > 0 ) {
01129                 nq += 2;
01130                 subterm[-1] = nq;
01131             }
01132             else { fill = subterm; fill -= 2; }
01133         }
01134 /*
01135             #] DELTAS :
01136 */
01137 EndLoop:;
01138     } while ( m < mstop ); }
01139     while ( t < tstop ) *fill++ = *t++;
01140 SubCoef:
01141     if ( !PutExpr ) {
01142         t = AN.FullProto;
01143         nq = t[1];
01144         t[3] = power;
01145         NCOPY(fill,t,nq);
01146     }
01147     t = tcoef;
01148     nq = ABS(*t);
01149     t = tstop;
01150     NCOPY(fill,t,nq);
01151     nq = WORDDIF(fill,TemTerm);
01152     fill = term;
01153     t = TemTerm;
01154     *fill++ = nq--;
01155     t++;
01156     NCOPY(fill,t,nq);
01157     if ( sign ) {
01158         if ( ( sign & 1 ) != 0 ) fill[-1] = -fill[-1];
01159     }
01160     if ( AT.WorkPointer < fill ) AT.WorkPointer = fill;
01161     AN.RepFunNum = 0;
01162 }
01163
01164 /*
01165         #] Substitute :
01166         #[ FindSpecial :        WORD FindSpecial(term)
01167
01168     Routine to detect simplifications regarding the special functions
01169     exponent, denominator.
01170
01171
01172 void FindSpecial(WORD *term)
01173 {
01174     WORD *t;
01175     WORD *tstop;
01176     t = term; t += *t - 1; tstop = t - ABS(*t) + 1; t = term;
01177     t++;
01178     if ( t < tstop ) { do {
01179         if ( *t == EXPONENT ) {
01180             Exponents can become simpler when:
01181             a: the exponent of an expression becomes an integer.
01182             b: The expression becomes zero.
01183         }
01184         else if ( *t == DENOMINATOR ) {
01185             Denominators can become simpler when:
01186             a: The denominator is a single term without functions.
01187             b: An overall coefficient can be removed.
01188             c: An overall object can be removed.
01189             The task is here to bring the denominator in an unique form.
01190         }
01191         t += *t;
01192     } while ( t < tstop ); }
01193 }
```

```
01194
01195         #] FindSpecial :
01196         #[ FindAll :            WORD FindAll(term,pattern,level,par)
01197 */
01198
01199 int FindAll(PHEAD WORD *term, WORD *pattern, WORD level, WORD *par)
01200 {
01201     GETBIDENTITY
01202     WORD *t, *m, *r, *mm, rnum;
01203     WORD *tstop, *mstop, *TwoProto, *vwhere = 0, oldv, oldvv, vv, level2;
01204     WORD v, nq, OffNum = AM.OffsetVector + WILDOFFSET, i, ii = 0, jj;
01205     WORD fromindex, *intens, notflag1 = 0, notflag2 = 0;
01206     CBUF *C;
01207     C = cbuf+AM.rbufnum;
01208     v = pattern[3];     /* The vector to be found */
01209     m = t = term;
01210     m += *m;
01211     m -= ABS(m[-1]);
01212     t++;
01213     if ( t < m ) do {
01214         tstop = t + t[1];
01215         fromindex = 2;
01216 /*
01217             #[ VECTOR :
01218 */
01219         if ( *t == VECTOR ) {
01220             r = t;
01221             r += 2;
01222 InVect:
01223             while ( r < tstop ) {
01224                 oldv = *r;
01225                 if ( v >= OffNum ) {
01226                     vwhere = AN.FullProto + 3 + SUBEXPSIZE;
01227                     if ( vwhere[1] == FROMSET || vwhere[1] == SETTONUM ) {
01228                         WORD *afirst, *alast, j;
01229                         j = vwhere[3];
01230                         if ( j > WILDOFFSET ) { j -= 2*WILDOFFSET; notflag1 = 1; }
01231                         else { notflag1 = 0; }
01232                         afirst = SetElements + Sets[j].first;
01233                         alast  = SetElements + Sets[j].last;
01234                         ii = 1;
01235                         if ( notflag1 == 0 ) {
01236                           do {
01237                             if ( *afirst == *r ) {
01238                                 if ( vwhere[1] == SETTONUM ) {
01239                                     AN.FullProto[8+SUBEXPSIZE] = SYMTONUM;
01240                                     AN.FullProto[11+SUBEXPSIZE] = ii;
01241                                 }
01242                                 else if ( vwhere[4] >= 0 ) {
01243                                     oldv = *(afirst - Sets[j].first
01244                                     + Sets[vwhere[4]].first);
01245                                 }
01246                                 goto DoVect;
01247                             }
01248                             ii++;
01249                           } while ( ++afirst < alast );
01250                         }
01251                         else {
01252                           do {
01253                             if ( *afirst == *r ) break;
01254                           } while ( ++afirst < alast );
01255                           if ( afirst >= alast ) goto DoVect;
01256                         }
01257                     }
01258                     else goto DoVect;
01259                 }
01260                 else if ( v == *r ) {
01261 DoVect:             m = AT.WorkPointer;
01262                     tstop = t;
01263                     t = term;
01264                     mstop = t + *t;
01265                     do { *m++ = *t++; } while ( t < tstop );
01266                     vwhere = m;
01267                     t = AN.FullProto;
01268                     nq = t[1];
01269                     t[3] = 1;
01270                     NCOPY(m,t,nq);
01271                     t = tstop;
01272                     if ( fromindex == 1 ) m[-1] = FUNNYVEC;
01273                     else m[-1] = r[1];      /* The index is always here! */
01274                     if ( v >= OffNum ) vwhere[3+SUBEXPSIZE] = oldv;
01275                     if ( vwhere[1] > 12+SUBEXPSIZE ) {
01276                         vwhere[11+SUBEXPSIZE] = ii;
01277                         vwhere[8+SUBEXPSIZE] = SYMTONUM;
01278                     }
01279                     if ( t[1] > fromindex+2 ) {
01280                         *m++ = *t++;
```

```
01281                           *m++ = *t++ - fromindex;
01282                           while ( t < r ) *m++ = *t++;
01283                           t += fromindex;
01284                       }
01285                       else t += t[1];
01286                       do { *m++ = *t++; } while ( t < mstop );
01287                       *AT.WorkPointer = nq = WORDDIF(m,AT.WorkPointer);
01288                       m = AT.WorkPointer;
01289                       t = term;
01290                       NCOPY(t,m,nq);
01291                       AT.WorkPointer = t;
01292                       return(1);
01293                   }
01294                   r += fromindex;
01295               }
01296           }
01297 /*
01298               #] VECTOR :
01299               #[ DOTPRODUCT :
01300 */
01301           else if ( *t == DOTPRODUCT ) {
01302               r = t;
01303               r += 2;
01304               do {
01305                   if ( ( i = r[2] ) < 0 ) goto NextDot;
01306                   if ( *r == r[1] ) {     /* p.p */
01307                       oldv = *r;
01308                       if ( v == *r ) {    /* v.v */
01309 TwoVec:                    m = AT.WorkPointer;
01310                           tstop = t;
01311                           t = term;
01312                           mstop = t + *t;
01313                           do { *m++ = *t++; } while ( t < tstop );
01314                           do {
01315                               vwhere = m;
01316                               t = AN.FullProto;
01317                               nq = t[1];
01318                               t[3] = 2;
01319                               NCOPY(m,t,nq);
01320                               m[-1] = ++AR.CurDum;
01321                               if ( v >= OffNum ) vwhere[3+SUBEXPSIZE] = oldv;
01322                           } while ( --i > 0 );
01323 CopRest:                   t = tstop;
01324                           if ( t[1] > 5 ) {
01325                               *m++ = *t++;
01326                               *m++ = *t++ - 3;
01327                               while ( t < r ) *m++ = *t++;
01328                               t += 3;
01329                           }
01330                           else t += t[1];
01331                           do { *m++ = *t++; } while ( t < mstop );
01332                           *AT.WorkPointer = nq = WORDDIF(m,AT.WorkPointer);
01333                           m = AT.WorkPointer;
01334                           t = term;
01335                           NCOPY(t,m,nq);
01336                           AT.WorkPointer = t;
01337                           return(1);
01338                       }
01339                       else if ( v >= OffNum ) {   /* v?.v? */
01340                           vwhere = AN.FullProto + 3+SUBEXPSIZE;
01341                           if ( vwhere[1] == FROMSET || vwhere[1] == SETTONUM ) {
01342                               WORD *afirst, *alast, j;
01343                               j = vwhere[3];
01344                               if ( j > WILDOFFSET ) { j -= 2*WILDOFFSET; notflag1 = 1; }
01345                               else { notflag1 = 0; }
01346                               afirst = SetElements + Sets[j].first;
01347                               alast  = SetElements + Sets[j].last;
01348                               ii = 1;
01349                               if ( notflag1 == 0 ) {
01350                                 do {
01351                                   if ( *afirst == *r ) {
01352                                       if ( vwhere[1] == SETTONUM ) {
01353                                           AN.FullProto[8+SUBEXPSIZE] = SYMTONUM;
01354                                           AN.FullProto[11+SUBEXPSIZE] = ii;
01355                                       }
01356                                       else if ( vwhere[4] >= 0 ) {
01357                                           oldv = *(afirst - Sets[j].first
01358                                           + Sets[vwhere[4]].first);
01359                                       }
01360                                       goto TwoVec;
01361                                   }
01362                                   ii++;
01363                                 } while ( ++afirst < alast );
01364                               }
01365                               else {
01366                                 do {
01367                                   if ( *afirst == *r ) break;
```

```
01368                               } while ( ++afirst < alast );
01369                               if ( afirst >= alast ) goto TwoVec;
01370                             }
01371                         }
01372                         else goto TwoVec;
01373                     }
01374                 }
01375                 else {
01376                     if ( v == r[1] ) { r[1] = *r; *r = v; }
01377                     oldv = *r;
01378                     oldvv = r[1];
01379                     if ( v == *r ) {
01380                         if ( !par ) { while ( ++level <= AR.Cnumlhs
01381                         && C->lhs[level][0] == TYPEIDOLD ) {
01382                             m = C->lhs[level];
01383                             m += IDHEAD;
01384                             if ( m[-IDHEAD+2] == SUBVECTOR ) {
01385                             if ( ( vv = m[m[1]+3] ) == r[1] ) {
01386 OnePV:                          TwoProto = AN.FullProto;
01387 TwoPV:                          m = AT.WorkPointer;
01388                                 tstop = t;
01389                                 t = term;
01390                                 mstop = t + *t;
01391                                 do { *m++ = *t++; } while ( t < tstop );
01392                                 do {
01393                                     t = AN.FullProto;
01394                                     vwhere = m + 3 +SUBEXPSIZE;
01395                                     nq = t[1];
01396                                     t[3] = 1;
01397                                     NCOPY(m,t,nq);
01398                                     m[-1] = ++AR.CurDum;
01399                                     if ( v >= OffNum ) *vwhere = oldv;
01400                                     if ( vwhere[-2-SUBEXPSIZE] > 12+SUBEXPSIZE ) {
01401                                         vwhere[8] = ii;
01402                                         vwhere[5] = SYMTONUM;
01403                                     }
01404                                     t = TwoProto;
01405                                     vwhere = m + 3+SUBEXPSIZE;
01406                                     mm = m;
01407                                     nq = t[1];
01408                                     t[3] = 1;
01409                                     NCOPY(m,t,nq);
01410 /*
01411         The next two lines repair a bug. without them it takes twice
01412         the rhs of the first vector.
01413 */
01414                                     mm[2] = C->lhs[level][IDHEAD+2];
01415                                     mm[4] = C->lhs[level][IDHEAD+4];
01416                                     m[-1] = AR.CurDum;
01417                                     if ( vv >= OffNum ) *vwhere = oldvv;
01418                                 } while ( --i > 0 );
01419                                 goto CopRest;
01420                             }
01421                             else if ( vv > OffNum ) {
01422                                 vwhere = AN.FullProto + 3+SUBEXPSIZE;
01423                                 if ( vwhere[1] == FROMSET || vwhere[1] == SETTONUM ) {
01424                                     WORD *afirst, *alast, j;
01425                                     j = vwhere[3];
01426                                     if ( j > WILDOFFSET ) { j -= 2*WILDOFFSET; notflag1 = 1; }
01427                                     else { notflag1 = 0; }
01428                                     afirst = SetElements + Sets[j].first;
01429                                     alast  = SetElements + Sets[j].last;
01430                                     if ( notflag1 == 0 ) {
01431                                       ii = 1;
01432                                       do {
01433                                         if ( *afirst == r[1] ) {
01434                                             if ( vwhere[1] == SETTONUM ) {
01435                                                 AN.FullProto[8+SUBEXPSIZE] = SYMTONUM;
01436                                                 AN.FullProto[11+SUBEXPSIZE] = ii;
01437                                             }
01438                                             else if ( vwhere[4] >= 0 ) {
01439                                                 oldvv = *(afirst - Sets[j].first
01440                                                 + Sets[vwhere[4]].first);
01441                                             }
01442                                             goto OnePV;
01443                                         }
01444                                         ii++;
01445                                       } while ( ++afirst < alast );
01446                                     }
01447                                     else {
01448                                       do {
01449                                         if ( *afirst == *r ) break;
01450                                       } while ( ++afirst < alast );
01451                                       if ( afirst >= alast ) goto OnePV;
01452                                     }
01453                                 }
01454                                 else goto OnePV;
```

```
01455                               }
01456                               }
01457                           }}
01458 /*
01459             v.q with v matching and no match for the q, also
01460             not in following idold statements.
01461             Notice that a following q.p? cannot match.
01462 */
01463                           rnum = r[1];
01464 OneOnly:                  m = AT.WorkPointer;
01465                           tstop = t;
01466                           t = term;
01467                           mstop = t + *t;
01468                           do { *m++ = *t++; } while ( t < tstop );
01469                           vwhere = m;
01470                           t = AN.FullProto;
01471                           nq = t[1];
01472                           t[3] = i;
01473                           NCOPY(m,t,nq);
01474                           m[-4] = INDTOIND;
01475                           m[-1] = rnum;
01476                           if ( v >= OffNum ) vwhere[3+SUBEXPSIZE] = oldv;
01477                           goto CopRest;
01478                       }
01479                       else if ( v >= OffNum ) {
01480                           vwhere = AN.FullProto + 3+SUBEXPSIZE;
01481                           if ( vwhere[1] == FROMSET || vwhere[1] == SETTONUM ) {
01482                               WORD *afirst, *alast, *bfirst, *blast, j;
01483                               j = vwhere[3];
01484                               if ( j > WILDOFFSET ) { j -= 2*WILDOFFSET; notflag1 = 1; }
01485                               else { notflag1 = 0; }
01486                               afirst = SetElements + Sets[j].first;
01487                               alast  = SetElements + Sets[j].last;
01488                               ii = 1;
01489                               if ( notflag1 == 0 ) {
01490                                 do {
01491                                   if ( *afirst == *r ) {
01492                                       if ( vwhere[1] == SETTONUM ) {
01493                                           AN.FullProto[8+SUBEXPSIZE] = SYMTONUM;
01494                                           AN.FullProto[11+SUBEXPSIZE] = ii;
01495                                       }
01496                                       else if ( vwhere[4] >= 0 ) {
01497                                           oldv = *(afirst - Sets[j].first
01498                                           + Sets[vwhere[4]].first);
01499                                       }
01500 Hitlevel1:                           level2 = level;
01501                                       do {
01502                                           if ( !par ) m = C->lhs[level2];
01503                                           else m = par;
01504                                           m += IDHEAD;
01505                                           if ( m[-IDHEAD+2] == SUBVECTOR ) {
01506                                           if ( ( vv = m[m[1]+3] ) == r[1] )
01507                                               goto OnePV;
01508                                           else if ( vv >= OffNum ) {
01509                                               if ( m[SUBEXPSIZE+4] != FROMSET &&
01510                                               m[SUBEXPSIZE+4] != SETTONUM ) goto OnePV;
01511                                               j = m[SUBEXPSIZE+6];
01512                                               if ( j > WILDOFFSET ) { j -= 2*WILDOFFSET; notflag2 = 1; }
01513                                               else { notflag2 = 0; }
01514                                               bfirst = SetElements + Sets[j].first;
01515                                               blast  = SetElements + Sets[j].last;
01516                                               jj = 1;
01517                                               if ( notflag2 == 0 ) {
01518                                                 do {
01519                                                   if ( *bfirst == r[1] ) {
01520                                                       if ( m[SUBEXPSIZE+4] == SETTONUM ) {
01521                                                           m[SUBEXPSIZE+8] = SYMTONUM;
01522                                                           m[SUBEXPSIZE+11] = jj;
01523                                                       }
01524                                                       else if ( m[SUBEXPSIZE+7] >= 0 ) {
01525                                                           oldvv = *(bfirst - Sets[j].first
01526                                                           + Sets[m[SUBEXPSIZE+7]].first);
01527                                                       }
01528                                                       goto OnePV;
01529                                                   }
01530                                                   jj++;
01531                                                 } while ( ++bfirst < blast );
01532                                               }
01533                                               else {
01534                                                 do {
01535                                                   if ( *bfirst == r[1] ) break;
01536                                                 } while ( ++bfirst < blast );
01537                                                 if ( bfirst >= blast ) goto OnePV;
01538                                               }
01539                                           }
01540                                           }
01541                                       } while ( ++level2 < AR.Cnumlhs &&
```

```
01542                                       C->lhs[level2][0] == TYPEIDOLD );
01543                                       rnum = r[1];
01544                                       goto OneOnly;
01545                                   }
01546                                   else if ( *afirst == r[1] ) {
01547                                       if ( vwhere[1] == SETTONUM ) {
01548                                           AN.FullProto[8+SUBEXPSIZE] = SYMTONUM;
01549                                           AN.FullProto[11+SUBEXPSIZE] = ii;
01550                                       }
01551                                       else if ( vwhere[4] >= 0 ) {
01552                                           oldv = *(afirst - Sets[j].first
01553                                           + Sets[vwhere[4]].first);
01554                                       }
01555 Hitlevel2:                            level2 = level;
01556                                       while ( ++level2 < AR.Cnumlhs &&
01557                                       C->lhs[level2][0] == TYPEIDOLD ) {
01558                                           if ( !par ) m = C->lhs[level2];
01559                                           else m = par;
01560                                           m += IDHEAD;
01561                                           if ( m[-IDHEAD+2] == SUBVECTOR ) {
01562                                           if ( ( vv = m[6] ) == *r )
01563                                               goto OnePV;
01564                                           else if ( vv >= OffNum ) {
01565                                               if ( m[SUBEXPSIZE+4] != FROMSET && m[SUBEXPSIZE+4]
01566                                               != SETTONUM ) {
01567                                                   j = *r;
01568                                                   *r = r[1];
01569                                                   r[1] = j;
01570                                                   goto OnePV;
01571                                               }
01572                                               j = m[SUBEXPSIZE+6];
01573                                               bfirst = SetElements + Sets[j].first;
01574                                               blast  = SetElements + Sets[j].last;
01575                                               jj = 1;
01576                                               do {
01577                                                   if ( *bfirst == *r ) {
01578                                                       if ( m[SUBEXPSIZE+4] == SETTONUM ) {
01579                                                           m[SUBEXPSIZE+8] = SYMTONUM;
01580                                                           m[SUBEXPSIZE+11] = jj;
01581                                                       }
01582                                                       else if ( m[SUBEXPSIZE+7] >= 0 ) {
01583                                                           oldvv = *(bfirst - Sets[j].first
01584                                                           + Sets[m[SUBEXPSIZE+7]].first);
01585                                                       }
01586                                                       j = *r;
01587                                                       *r = r[1];
01588                                                       r[1] = j;
01589                                                       j = oldv; oldv = oldvv; oldvv = j;
01590                                                       goto OnePV;
01591                                                   }
01592                                                   jj++;
01593                                               } while ( ++bfirst < blast );
01594                                           }
01595                                               }
01596                                       }
01597                                       jj = *r; *r = r[1]; r[1] = jj;
01598                                       jj = oldv; oldv = oldvv; oldvv = j;
01599                                       rnum = r[1];
01600                                       goto OneOnly;
01601                                   }
01602                                   ii++;
01603                                 } while ( ++afirst < alast );
01604                               }
01605                               else {
01606                                 do {
01607                                   if ( *afirst == *r ) break;
01608                                 } while ( ++afirst < alast );
01609                                 if ( afirst >= alast ) goto Hitlevel1;
01610                                 do {
01611                                   if ( *afirst == r[1] ) break;
01612                                 } while ( ++afirst < alast );
01613                                 if ( afirst >= alast ) goto Hitlevel2;
01614                               }
01615                           }
01616                           else { /* Matches twice */
01617                               vv = v;
01618                               TwoProto = AN.FullProto;
01619                               goto TwoPV;
01620                           }
01621                       }
01622                   }
01623 NextDot:              r += 3;
01624               } while ( r < tstop );
01625           }
01626 /*
01627               #] DOTPRODUCT :
01628               #[ LEVICIVITA :
```

```
01629 */
01630         else if ( *t == LEVICIVITA ) {
01631             intens = 0;
01632             r = t;
01633             r += FUNHEAD;
01634 OneVect:;
01635             while ( r < tstop ) {
01636                 oldv = *r;
01637                 if ( v >= OffNum && *r < -10 ) {
01638                     vwhere = AN.FullProto + 3+SUBEXPSIZE;
01639                     if ( vwhere[1] == FROMSET || vwhere[1] == SETTONUM ) {
01640                         WORD *afirst, *alast, j;
01641                         j = vwhere[3];
01642                         if ( j > WILDOFFSET ) { j -= 2*WILDOFFSET; notflag1 = 1; }
01643                         else { notflag1 = 0; }
01644                         afirst = SetElements + Sets[j].first;
01645                         alast  = SetElements + Sets[j].last;
01646                         ii = 1;
01647                         if ( notflag1 == 0 ) {
01648                           do {
01649                             if ( *afirst == *r ) {
01650                                 if ( vwhere[1] == SETTONUM ) {
01651                                     AN.FullProto[8+SUBEXPSIZE] = SYMTONUM;
01652                                     AN.FullProto[11+SUBEXPSIZE] = ii;
01653                                 }
01654                                 else if ( vwhere[4] >= 0 ) {
01655                                     oldv = *(afirst - Sets[j].first
01656                                     + Sets[vwhere[4]].first);
01657                                 }
01658                                 goto DoVect;
01659                             }
01660                             ii++;
01661                           } while ( ++afirst < alast );
01662                         }
01663                         else {
01664                           do {
01665                             if ( *afirst == *r ) break;
01666                           } while ( ++afirst < alast );
01667                           if ( afirst >= alast ) goto DoVect;
01668                         }
01669                     }
01670                     else goto LeVect;
01671                 }
01672                 else if ( v == *r ) {
01673 LeVect:             m = AT.WorkPointer;
01674                     mstop = term + *term;
01675                     t = term;
01676                     *r = ++AR.CurDum;
01677                     if ( intens ) *intens = DIRTYSYMFLAG;
01678                     do { *m++ = *t++; } while ( t < tstop );
01679                     t = AN.FullProto;
01680                     nq = t[1];
01681                     t[3] = 1;
01682                     if ( v >= OffNum ) *vwhere = oldv;
01683                     NCOPY(m,t,nq);
01684                     m[-1] = AR.CurDum;
01685                     t = tstop;
01686                     do { *m++ = *t++; } while ( t < mstop );
01687                     *AT.WorkPointer = nq = WORDDIF(m,AT.WorkPointer);
01688                     m = AT.WorkPointer;
01689                     t = term;
01690                     NCOPY(t,m,nq);
01691                     AT.WorkPointer = t;
01692                     return(1);
01693                 }
01694                 r++;
01695             }
01696         }
01697 /*
01698             #] LEVICIVITA :
01699             #[ GAMMA :
01700 */
01701         else if ( *t == GAMMA ) {
01702             intens = 0;
01703             r = t;
01704             r += FUNHEAD+1;
01705             if ( r < tstop ) goto OneVect;
01706         }
01707 /*
01708             #] GAMMA :
01709             #[ INDEX :
01710 */
01711         else if ( *t == INDEX ) {   /* The 'forgotten' part */
01712             r = t;
01713             r += 2;
01714             fromindex = 1;
01715             goto InVect;
```

```
01716         }
01717 /*
01718             #] INDEX :
01719             #[ FUNCTION :
01720 */
01721         else if ( *t >= FUNCTION ) {
01722             if ( *t >= FUNCTION
01723              && functions[*t-FUNCTION].spec >= TENSORFUNCTION
01724              && t[1] > FUNHEAD ) {
01725 /*
01726                 Tensors are linear in their vectors!
01727 */
01728                 r = t;
01729                 r += FUNHEAD;
01730                 intens = t+2;
01731                 goto OneVect;
01732             }
01733         }
01734 /*
01735             #] FUNCTION :
01736 */
01737         t += t[1];
01738     } while ( t < m );
01739     return(0);
01740 }
01741
01742 /*
01743         #] FindAll :
01744         #[ TestSelect :
01745
01746         Returns 1 if any of the objects in any of the sets in setp
01747         occur anywhere in the term
01748 */
01749
01750 int TestSelect(WORD *term, WORD *setp)
01751 {
01752     WORD *tstop, *t, *s, *el, *elstop, *termstop, *tt, n, ns;
01753     GETSTOP(term,tstop);
01754     term += 1;
01755     while ( term < tstop ) {
01756     switch ( *term ) {
01757         case SYMBOL:
01758             n = term[1] - 2;
01759             t = term + 2;
01760             while ( n > 0 ) {
01761                 ns = setp[1] - 2;
01762                 s = setp + 2;
01763                 while ( --ns >= 0 ) {
01764                     if ( Sets[*s].type != CSYMBOL ) { s++; continue; }
01765                     el = SetElements + Sets[*s].first;
01766                     elstop = SetElements + Sets[*s].last;
01767                     while ( el < elstop ) {
01768                         if ( *el++ == *t ) return(1);
01769                     }
01770                     s++;
01771                 }
01772                 n -= 2;
01773                 t += 2;
01774             }
01775             break;
01776         case VECTOR:
01777             n = term[1] - 2;
01778             t = term + 2;
01779             while ( n > 0 ) {
01780                 ns = setp[1] - 2;
01781                 s = setp + 2;
01782                 while ( --ns >= 0 ) {
01783                     if ( Sets[*s].type != CVECTOR ) { s++; continue; }
01784                     el = SetElements + Sets[*s].first;
01785                     elstop = SetElements + Sets[*s].last;
01786                     while ( el < elstop ) {
01787                         if ( *el++ == *t ) return(1);
01788                     }
01789                     s++;
01790                 }
01791                 t++;
01792                 ns = setp[1] - 2;
01793                 s = setp + 2;
01794                 while ( --ns >= 0 ) {
01795                     if ( Sets[*s].type != CINDEX
01796                     && Sets[*s].type != CNUMBER ) { s++; continue; }
01797                     el = SetElements + Sets[*s].first;
01798                     elstop = SetElements + Sets[*s].last;
01799                     while ( el < elstop ) {
01800                         if ( *el++ == *t ) return(1);
01801                     }
01802                     s++;
```

```
01803                 }
01804                 n -= 2;
01805                 t++;
01806             }
01807             break;
01808         case INDEX:
01809             n = term[1] - 2;
01810             t = term + 2;
01811             goto dotensor;
01812         case DOTPRODUCT:
01813             n = term[1] - 2;
01814             t = term + 2;
01815             while ( n > 0 ) {
01816                 ns = setp[1] - 2;
01817                 s = setp + 2;
01818                 while ( --ns >= 0 ) {
01819                     if ( Sets[*s].type != CVECTOR ) { s++; continue; }
01820                     el = SetElements + Sets[*s].first;
01821                     elstop = SetElements + Sets[*s].last;
01822                     while ( el < elstop ) {
01823                         if ( *el++ == *t ) return(1);
01824                     }
01825                     s++;
01826                 }
01827                 t++;
01828                 ns = setp[1] - 2;
01829                 s = setp + 2;
01830                 while ( --ns >= 0 ) {
01831                     if ( Sets[*s].type != CVECTOR ) { s++; continue; }
01832                     el = SetElements + Sets[*s].first;
01833                     elstop = SetElements + Sets[*s].last;
01834                     while ( el < elstop ) {
01835                         if ( *el++ == *t ) return(1);
01836                     }
01837                     s++;
01838                 }
01839                 n -= 3;
01840                 t += 2;
01841             }
01842             break;
01843         case DELTA:
01844             n = term[1] - 2;
01845             t = term + 2;
01846             goto dotensor;
01847         default:
01848             if ( *term < FUNCTION ) break;
01849             ns = setp[1] - 2;
01850             s = setp + 2;
01851             while ( --ns >= 0 ) {
01852                 if ( Sets[*s].type != CFUNCTION ) { s++; continue; }
01853                 el = SetElements + Sets[*s].first;
01854                 elstop = SetElements + Sets[*s].last;
01855                 while ( el < elstop ) {
01856                     if ( *el++ == *term ) return(1);
01857                 }
01858                 s++;
01859             }
01860             if ( functions[*term-FUNCTION].spec > 0 ) {
01861                 n = term[1] - FUNHEAD;
01862                 t = term + FUNHEAD;
01863 dotensor:
01864                 while ( n > 0 ) {
01865                     ns = setp[1] - 2;
01866                     s = setp + 2;
01867                     while ( --ns >= 0 ) {
01868                         if ( *t < MINSPEC ) {
01869                             if ( Sets[*s].type != CVECTOR ) { s++; continue; }
01870                         }
01871                         else if ( *t >= 0 ) {
01872                             if ( Sets[*s].type != CINDEX
01873                             && Sets[*s].type != CNUMBER ) { s++; continue; }
01874                         }
01875                         else { s++; continue; }
01876                         el = SetElements + Sets[*s].first;
01877                         elstop = SetElements + Sets[*s].last;
01878                         while ( el < elstop ) {
01879                             if ( *el++ == *t ) return(1);
01880                         }
01881                         s++;
01882                     }
01883                     t++;
01884                     n--;
01885                 }
01886             }
01887             else {
01888                 termstop = term + term[1];
01889                 tt = term + FUNHEAD;
```

```
01890                 while ( tt < termstop ) {
01891                     if ( *tt < 0 ) {
01892                         if ( *tt == -SYMBOL ) {
01893                             ns = setp[1] - 2;
01894                             s = setp + 2;
01895                             while ( --ns >= 0 ) {
01896                                 if ( Sets[*s].type != CSYMBOL ) { s++; continue; }
01897                                 el = SetElements + Sets[*s].first;
01898                                 elstop = SetElements + Sets[*s].last;
01899                                 while ( el < elstop ) {
01900                                     if ( *el++ == tt[1] ) return(1);
01901                                 }
01902                                 s++;
01903                             }
01904                             tt += 2;
01905                         }
01906                         else if ( *tt == -VECTOR || *tt == -MINVECTOR ) {
01907                             ns = setp[1] - 2;
01908                             s = setp + 2;
01909                             while ( --ns >= 0 ) {
01910                                 if ( Sets[*s].type != CVECTOR ) { s++; continue; }
01911                                 el = SetElements + Sets[*s].first;
01912                                 elstop = SetElements + Sets[*s].last;
01913                                 while ( el < elstop ) {
01914                                     if ( *el++ == tt[1] ) return(1);
01915                                 }
01916                                 s++;
01917                             }
01918                             tt += 2;
01919                         }
01920                         else if ( *tt == -INDEX ) {
01921                             ns = setp[1] - 2;
01922                             s = setp + 2;
01923                             while ( --ns >= 0 ) {
01924                                 if ( Sets[*s].type != CINDEX
01925                                 && Sets[*s].type != CNUMBER ) { s++; continue; }
01926                                 el = SetElements + Sets[*s].first;
01927                                 elstop = SetElements + Sets[*s].last;
01928                                 while ( el < elstop ) {
01929                                     if ( *el++ == tt[1] ) return(1);
01930                                 }
01931                                 s++;
01932                             }
01933                             tt += 2;
01934                         }
01935                         else if ( *tt <= -FUNCTION ) {
01936                             ns = setp[1] - 2;
01937                             s = setp + 2;
01938                             while ( --ns >= 0 ) {
01939                                 if ( Sets[*s].type != CFUNCTION ) { s++; continue; }
01940                                 el = SetElements + Sets[*s].first;
01941                                 elstop = SetElements + Sets[*s].last;
01942                                 while ( el < elstop ) {
01943                                     if ( *el++ == -(*tt) ) return(1);
01944                                 }
01945                                 s++;
01946                             }
01947                             tt++;
01948                         }
01949                         else tt += 2;
01950                     }
01951                     else {
01952                         t = tt + ARGHEAD;
01953                         tt += *tt;
01954                         while ( t < tt ) {
01955                             if ( TestSelect(t,setp) ) return(1);
01956                             t += *t;
01957                         }
01958                     }
01959                 }
01960             }
01961             break;
01962     }
01963     term += term[1];
01964     }
01965     return(0);
01966 }
01967
01968 /*
01969         #] TestSelect :
01970         #[ SubsInAll :          void SubsInAll()
01971
01972         This routine takes a match in id,all and stores it away in
01973         the AT.allbufnum 'compiler' buffer, after taking out the pattern.
01974         The main problem here is that id,all usually has (lots of) wildcards
01975         and their assignments are on stack and the difficult ones are in
01976         AT.ebufnum. Popping the stack while looking for more matches would
```

```
01977         loose those. Hence we have to copy them into yet another compiler
01978         buffer: AT.aebufnum. Because this may involve many matches and
01979         because the original term has only a limited number of arguments,
01980         it will pay to look for already existing ones in this buffer.
01981         (to be done later).
01982 */
01983
01984 void SubsInAll(PHEAD0)
01985 {
01986     GETBIDENTITY
01987     WORD *TemTerm;
01988     WORD *t, *m, *term;
01989     WORD *tstop, *mstop, *xstop;
01990     WORD nt, *fill, nq, mt;
01991     WORD *tcoef, i = 0;
01992     WORD PutExpr = 0, sign = 0;
01993 /*
01994     We start with building the term in the WorkSpace.
01995     Afterwards we will transfer it to AT.allbufnum.
01996     We have to make sure there is room in the WorkSpace.
01997 */
01998     AT.idallflag = 2;
01999     TemTerm = AT.WorkPointer;
02000     if ( ( (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer*2) ) > AT.WorkTop ) {
02001         MLOCK(ErrorMessageLock);
02002         MesWork();
02003         MUNLOCK(ErrorMessageLock);
02004         Terminate(-1);
02005     }
02006     m = AN.patternbuffer + IDHEAD; m += m[1];
02007     mstop = m + *m;
02008     m++;
02009     term = AN.termbuffer;
02010     tstop = term + *term; tcoef = tstop-1; tstop -= ABS(tstop[-1]);
02011     t = term;
02012     t++;
02013     fill = TemTerm;
02014     fill++;
02015     while ( m < mstop ) {
02016         while ( t < tstop ) {
02017             nt = WORDDIF(t,term);
02018             for ( mt = 0; mt < AN.RepFunNum; mt += 2 ) {
02019                 if ( nt == AN.RepFunList[mt] ) break;
02020             }
02021             if ( mt >= AN.RepFunNum ) {
02022                 nq = t[1];
02023                 NCOPY(fill,t,nq);
02024             }
02025             else {
02026                 WORD *oldt = 0;
02027                 if ( *m == GAMMA && m[1] != FUNHEAD+1 ) {
02028                     oldt = t;
02029                     if ( ( i = AN.RepFunList[mt+1] ) > 0 ) {
02030                         *fill++ = GAMMA;
02031                         *fill++ = i + FUNHEAD+1;
02032                         FILLFUN(fill)
02033                         nq = i + 1;
02034                         t += FUNHEAD;
02035                         NCOPY(fill,t,nq);
02036                     }
02037                     t = oldt;
02038                 }
02039                 else if ( ( *t == LEVICIVITA ) || ( *t >= FUNCTION
02040                 && (functions[*t-FUNCTION].symmetric & ~REVERSEORDER) == ANTISYMMETRIC )
02041                                 ) sign += AN.RepFunList[mt+1];
02042                 else if ( *m >= FUNCTION+WILDOFFSET
02043                 && (functions[*m-FUNCTION-WILDOFFSET].symmetric & ~REVERSEORDER) == ANTISYMMETRIC
02044                                 ) sign += AN.RepFunList[mt+1];
02045                 if ( !PutExpr ) {
02046                     WORD *pstart = fill, *p, *w, *ww;
02047                     xstop = t + t[1];
02048                     t = AN.FullProto;
02049                     nq = t[1];
02050                     t[3] = 1;
02051                     NCOPY(fill,t,nq);
02052                     t = xstop;
02053                     PutExpr = 1;
02054 /*
02055                     Here we need provisions for keeping wildcard matches
02056                     that reside in AT.ebufnum. We will move them to
02057                     AT.aebufnum.
02058                     Problem: the SUBEXPRESSION assumes automatically
02059                     that the compiler buffer is AT.ebufnum. We have to
02060                     correct that in TransferBuffer.
02061 */
02062                     p = pstart + SUBEXPSIZE;
02063                     while ( p < fill ) {
```

```
02064                           switch ( *p ) {
02065                               case SYMTOSUB:
02066                               case VECTOSUB:
02067                               case INDTOSUB:
02068                               case ARGTOARG:
02069                               case ARLTOARL:
02070                                   w = cbuf[AT.ebufnum].rhs[p[3]];
02071                                   ww = cbuf[AT.ebufnum].rhs[p[3]+1];
02072 /*
02073                                   Here we could search for whether this
02074                                   object sits in the buffer already.
02075                                   To be done later.
02076                                   By the way: ww-w fits inside a WORD.
02077 */
02078                                   AddRHS(AT.aebufnum,1);
02079                                   AddNtoC(AT.aebufnum,ww-w,w,11);
02080                                   p[3] = cbuf[AT.aebufnum].numrhs;
02081                                   cbuf[AT.aebufnum].rhs[p[3]+1] = cbuf[AT.aebufnum].Pointer;
02082                                   p += p[1];
02083                                   break;
02084                               case FROMSET:
02085                               case SETTONUM:
02086                               case LOADDOLLAR:
02087                                   p += p[1];
02088                                   break;
02089                               default:
02090                                   p += p[1];
02091                                   break;
02092                           }
02093
02094                       }
02095                   }
02096                   else t += t[1];
02097                   if ( *m == GAMMA && m[1] != FUNHEAD+1 ) {
02098                       i = oldt[1] - m[1] - i;
02099                       if ( i > 0 ) {
02100                           *fill++ = GAMMA;
02101                           *fill++ = i + FUNHEAD+1;
02102                           FILLFUN(fill)
02103                           *fill++ = oldt[FUNHEAD];
02104                           t = t - i;
02105                           NCOPY(fill,t,i);
02106                       }
02107                   }
02108                   break;
02109               }
02110           }
02111           m += m[1];
02112       }
02113       while ( t < tstop ) *fill++ = *t++;
02114       if ( !PutExpr ) {
02115           t = AN.FullProto;
02116           nq = t[1];
02117           t[3] = 1;
02118           NCOPY(fill,t,nq);
02119       }
02120       t = tcoef;
02121       nq = ABS(*t);
02122       t = tstop;
02123       NCOPY(fill,t,nq);
02124       if ( sign ) {
02125           if ( ( sign & 1 ) != 0 ) fill[-1] = -fill[-1];
02126       }
02127       *TemTerm = fill-TemTerm;
02128 /*
02129       And now we copy this to AT.allbufnum
02130 */
02131       AddNtoC(AT.allbufnum,TemTerm[0],TemTerm,12);
02132       cbuf[AT.allbufnum].Pointer[0] = 0;
02133       AN.RepFunNum = 0;
02134 }
02135
02136 /*
02137           #] SubsInAll :
02138           #[ TransferBuffer :
02139
02140           Adds the whole content of a (compiler)buffer to another buffer.
02141           In spectator we have an expression in the RHS that needs the
02142           wildcard resolutions adapted by an offset.
02143 */
02144
02145 void TransferBuffer(int from,int to,int spectator)
02146 {
02147       CBUF *C  = cbuf + spectator;
02148       CBUF *Cf = cbuf + from;
02149       CBUF *Ct = cbuf + to;
02150       int offset = Ct->numrhs;
```

```
02151     LONG i;
02152     WORD *t, *tt, *ttt, *tstop, size;
02153     for ( i = 1; i <= Cf->numrhs; i++ ) {
02154         size = Cf->rhs[i+1]-Cf->rhs[i];
02155         AddRHS(to,1);
02156         AddNtoC(to,size,Cf->rhs[i],13);
02157     }
02158     Ct->rhs[Ct->numrhs+1] = Ct->Pointer;
02159     Cf->numrhs = 0;
02160 /*
02161     Now we have to update the 'pointers' in the spectator.
02162 */
02163     t = C->rhs[C->numrhs];
02164     while ( *t ) {
02165         tt = t+1; t += *t;
02166         tstop = t-ABS(t[-1]);
02167         while ( tt < tstop ) {
02168             if ( *tt == SUBEXPRESSION ) {
02169                 ttt = tt+SUBEXPSIZE; tt += tt[1];
02170                 while ( ttt < tt ) {
02171                     switch ( *ttt ) {
02172                         case SYMTOSUB:
02173                         case VECTOSUB:
02174                         case INDTOSUB:
02175                         case ARGTOARG:
02176                         case ARLTOARL:
02177                             ttt[3] += offset;
02178                             break;
02179                         default:
02180                             break;
02181                     }
02182                     ttt += 4;
02183                 }
02184             }
02185             else tt += tt[1];
02186         }
02187     }
02188 }
02189
02190 /*
02191         #] TransferBuffer :
02192         #[ TakeIDfunction :
02193 */
02194
02195 #define PutInBuffers(pow) \
02196     AddRHS(AT.ebufnum,1); \
02197     *out++ = SUBEXPRESSION; \
02198     *out++ = SUBEXPSIZE; \
02199     *out++ = C->numrhs; \
02200     *out++ = pow; \
02201     *out++ = AT.ebufnum; \
02202     FILLSUB(out) \
02203     r = AT.pWorkSpace[rhs+i]; \
02204     if ( *r > 0 ) { \
02205         oldinr = r[*r]; r[*r] = 0; \
02206         AddNtoC(AT.ebufnum,(*r+1-ARGHEAD),(r+ARGHEAD),14); \
02207         r[*r] = oldinr; \
02208     } \
02209     else { \
02210         ToGeneral(r,buffer,1); \
02211         buffer[buffer[0]] = 0; \
02212         AddNtoC(AT.ebufnum,buffer[0]+1,buffer,15); \
02213     }
02214
02215 int TakeIDfunction(PHEAD WORD *term)
02216 {
02217     WORD *tstop, *t, *r, *m, *f, *nextf, *funstop, *left, *l, *newterm;
02218     WORD *out, oldinr, pow;
02219     WORD buffer[20];
02220     int i, ii, j, numsub, numfound = 0, first;
02221     LONG lhs,rhs;
02222     CBUF *C;
02223     GETSTOP(term,tstop);
02224     for ( t = term+1; t < tstop; t += t[1] ) { if ( *t == IDFUNCTION ) break; }
02225     if ( t >= tstop ) return(0);
02226 /*
02227     Step 1: test validity
02228 */
02229     funstop = t + t[1]; f = t + FUNHEAD;
02230     left = term + *term;
02231     l = left+1; numsub = 0;
02232     while ( f < funstop ) {
02233         nextf = f; NEXTARG(nextf)
02234         if ( nextf >= funstop ) { return(0); } /* odd number of arguments */
02235         if ( *f == -SYMBOL ) { *l++ = SYMBOL; *l++ = 4; *l++ = f[1]; *l++ = 1; }
02236         else if ( *f < -FUNCTION ) { *l++ = *f; *l++ = FUNHEAD; FILLFUN(l) }
02237         else if ( *f > 0 ) {
```

```
02238             if ( *f != f[ARGHEAD]+ARGHEAD ) goto noaction;
02239             if ( nextf[-1] != 3 || nextf[-2] != 1 || nextf[-3] != 1 ) goto noaction;
02240             if ( f[ARGHEAD] <= 4 ) goto noaction;
02241             if ( f[ARGHEAD] != f[ARGHEAD+2]+4 ) goto noaction;
02242             if ( f[ARGHEAD] == 8 && f[ARGHEAD+1] == SYMBOL ) {
02243                 for ( i = 0; i < 4; i++ ) *l++ = f[ARGHEAD+1+i];
02244             }
02245             else if ( f[ARGHEAD] == 9 && f[ARGHEAD+1] == DOTPRODUCT ) {
02246                 for ( i = 0; i < 5; i++ ) *l++ = f[ARGHEAD+1+i];
02247             }
02248             else if ( f[ARGHEAD+1] >= FUNCTION ) {
02249                 for ( i = 0; i < f[ARGHEAD+1]-4; i++ ) *l++ = f[ARGHEAD+1+i];
02250             }
02251             else goto noaction;
02252         }
02253         else goto noaction;
02254         numsub++;
02255         f = nextf;
02256         NEXTARG(f)
02257     }
02258     C = cbuf+AT.ebufnum;
02259     AT.WorkPointer = l;
02260     *left = l-left;
02261 /*
02262     Put the pointers to the lhs and the rhs in the pointer workspace
02263 */
02264     WantAddPointers(2*numsub);
02265     lhs = AT.pWorkPointer;
02266     rhs = lhs+numsub;
02267     AT.pWorkPointer = rhs+numsub;
02268     f = t + FUNHEAD; l = left+1;
02269     for ( i = 0; i < numsub; i++ ) {
02270         AT.pWorkSpace[lhs+i] = l; l += l[1];
02271         NEXTARG(f);
02272         AT.pWorkSpace[rhs+i] = f;
02273         NEXTARG(f);
02274     }
02275 /*
02276     Take out the patterns and replace them by SUBEXPRESSIONs pointing at
02277     the e buffer. We put the resulting term above the left sides.
02278     Note that we take out only the first id_ if there is more than one!
02279 */
02280     first = 1;
02281     t = term+1; newterm = AT.WorkPointer; out = newterm+1;
02282     while ( t < tstop ) {
02283         if ( *t == IDFUNCTION && first ) { first = 0; t += t[1]; continue; }
02284         if ( *t >= FUNCTION ) {
02285             for ( i = 0; i < numsub; i++ ) {
02286                 m = AT.pWorkSpace[lhs+i];
02287                 if ( *m != *t ) continue;
02288                 for ( j = 1; j < t[1]; j++ ) {
02289                     if ( m[j] != t[j] ) break;
02290                 }
02291                 if ( j != t[1] ) continue;
02292                 numfound++;
02293 /*
02294                 We have a match! Set up a SUBEXPRESSION subterm and put the
02295                 corresponding rhs in the eBuffer.
02296 */
02297                 PutInBuffers(1)
02298                 t += t[1];
02299             }
02300             if ( i == numsub ) {    /* no match. Just copy to output. */
02301                 j = t[1]; NCOPY(out,t,j)
02302             }
02303         }
02304         else if ( *t == SYMBOL ) {
02305             for ( i = 0; i < numsub; i++ ) {
02306                 m = AT.pWorkSpace[lhs+i];
02307                 if ( *m != SYMBOL ) continue;
02308                 for ( ii = 2; ii < t[1]; ii += 2 ) {
02309                     if ( m[2] != t[ii] ) continue;
02310                     pow = t[ii+1]/m[3];
02311                     if ( pow <= 0 ) continue;
02312                     t[ii+1] = t[ii+1]%m[3];
02313                     numfound++;
02314 /*
02315                     Create the proper rhs in the eBuffer and set up a
02316                     SUBEXPRESSION subterm.
02317 */
02318                     PutInBuffers(pow)
02319                 }
02320             }
02321 /*
02322             Now we copy whatever remains of the SYMBOL subterm to the output
02323 */
02324             m = out; *out++ = t[0]; *out++ = t[1];
```

```
02325             for ( ii = 2; ii < t[1]; ii += 2 ) {
02326                 if ( t[ii+1] ) { *out++ = t[ii]; *out++ = t[ii+1]; }
02327             }
02328             m[1] = out-m;
02329             if ( m[1] == 2 ) out = m;
02330             t += t[1];
02331         }
02332         else if ( *t == DOTPRODUCT ) {
02333             for ( i = 0; i < numsub; i++ ) {
02334                 m = AT.pWorkSpace[lhs+i];
02335                 if ( *m != DOTPRODUCT ) continue;
02336                 for ( ii = 2; ii < t[1]; ii += 3 ) {
02337                     if ( m[2] != t[ii] || m[3] != t[ii+1] ) continue;
02338                     pow = t[ii+2]/m[4];
02339                     if ( pow <= 0 ) continue;
02340                     t[ii+2] = t[ii+2]%m[4];
02341                     numfound++;
02342 /*
02343                     Create the proper rhs in the eBuffer and set up a
02344                     SUBEXPRESSION subterm.
02345 */
02346                     PutInBuffers(pow)
02347                 }
02348             }
02349 /*
02350             Now we copy whatever remains of the DOTPRODUCT subterm to the output
02351 */
02352             m = out; *out++ = t[0]; *out++ = t[1];
02353             for ( ii = 2; ii < t[1]; ii += 3 ) {
02354                 if ( t[ii+2] ) { *out++ = t[ii]; *out++ = t[ii+1]; *out++ = t[ii+2]; }
02355             }
02356             m[1] = out-m;
02357             if ( m[1] == 2 ) out = m;
02358             t += t[1];
02359         }
02360         else {
02361             j = t[1]; NCOPY(out,t,j)
02362         }
02363     }
02364 /*
02365     Copy the coefficient and set the size.
02366 */
02367     t = tstop; r = term+*term; while ( t < r ) *out++ = *t++;
02368     *newterm = out-newterm;
02369 /*
02370     Finally we move the new term over the original term.
02371 */
02372     i = *newterm;
02373     t = term; r = newterm; NCOPY(t,r,i)
02374 /*
02375     At this point we can return and if the calling Generator jumps back to
02376     its start, TestSub can take care of the expansions of SUBEXPRESSIONs.
02377 */
02378     AT.pWorkPointer = lhs;
02379     AT.WorkPointer = t;
02380     return(numfound);
02381 noaction:
02382     return(0);
02383 }
02384
02385 /*
02386         #] TakeIDfunction :
02387     #] Patterns :
02388 */
02389
```

4.103 poly.cc

```
00001 /* @file poly.cc
00002  *
00003  *  Contains the class for representing sparse multivariate
00004  *  polynomials with integer coefficients
00005  */
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
```

```
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032 /*
00033     #[ includes :
00034 */
00035
00036 #include "poly.h"
00037 #include "polygcd.h"
00038
00039 #include <cassert>
00040 #include <cctype>
00041 #include <cmath>
00042 #include <algorithm>
00043 #include <map>
00044 #include <iostream>
00045
00046 using namespace std;
00047
00048 /*
00049     #] includes :
00050     #[ constructor (small constant polynomial) :
00051 */
00052
00053 // constructor for a small constant polynomial
00054 poly::poly (PHEAD int a, WORD modp, WORD modn):
00055     size_of_terms(AM.MaxTer/(LONG)sizeof(WORD)),
00056     modp(0),
00057     modn(1)
00058 {
00059     POLY_STOREIDENTITY;
00060
00061     terms = TermMalloc("poly::poly(int)");
00062
00063     if (a == 0) {
00064         terms[0] = 1; // length
00065     }
00066     else {
00067         terms[0] = 4 + AN.poly_num_vars;                        // length
00068         terms[1] = 3 + AN.poly_num_vars;                        // length
00069         for (int i=0; i<AN.poly_num_vars; i++) terms[2+i] = 0; // powers
00070         terms[2+AN.poly_num_vars] = ABS(a);                     // coefficient
00071         terms[3+AN.poly_num_vars] = a>0 ? 1 : -1;               // length coefficient
00072     }
00073
00074     if (modp!=-1) setmod(modp,modn);
00075 }
00076
00077 /*
00078     #] constructor (small constant polynomial) :
00079     #[ constructor (large constant polynomial) :
00080 */
00081
00082 // constructor for a large constant polynomial
00083 poly::poly (PHEAD const UWORD *a, WORD na, WORD modp, WORD modn):
00084     size_of_terms(AM.MaxTer/(LONG)sizeof(WORD)),
00085     modp(0),
00086     modn(1)
00087 {
00088     POLY_STOREIDENTITY;
00089
00090     terms = TermMalloc("poly::poly(UWORD*,WORD)");
00091
00092     // remove leading zeros
00093     while (*(a+ABS(na)-1)==0)
00094         na -= SGN(na);
00095
00096     terms[0] = 3 + AN.poly_num_vars + ABS(na);              // length
00097     terms[1] = terms[0] - 1;                                // length
00098     for (int i=0; i<AN.poly_num_vars; i++) terms[2+i] = 0;  // powers
00099     termscopy((WORD *)a, 2+AN.poly_num_vars, ABS(na));      // coefficient
00100     terms[2+AN.poly_num_vars+ABS(na)] = na;                 // length coefficient
00101
00102     if (modp!=-1) setmod(modp,modn);
```

```
00103 }
00104
00105 /*
00106     #] constructor (large constant polynomial) :
00107     #[ constructor (deep copy polynomial) :
00108 */
00109
00110 // copy constructor: makes a deep copy of a polynomial
00111 poly::poly (const poly &a, WORD modp, WORD modn):
00112     size_of_terms(AM.MaxTer/(LONG)sizeof(WORD))
00113 {
00114 #ifdef POLY_MOVE_DEBUG
00115     std::cout « "poly copy ctor" « std::endl;
00116 #endif
00117     POLY_GETIDENTITY(a);
00118     POLY_STOREIDENTITY;
00119
00120     terms = TermMalloc("poly::poly(poly&)");
00121
00122     *this = a;
00123
00124     if (modp!=-1) setmod(modp,modn);
00125 }
00126
00127 /*
00128     #] constructor (deep copy polynomial) :
00129     #[ destructor :
00130 */
00131
00132 // destructor
00133 poly::~poly () {
00134
00135     POLY_GETIDENTITY(*this);
00136
00137     if (size_of_terms == AM.MaxTer/(LONG)sizeof(WORD))
00138         TermFree(terms, "poly::~poly");
00139     else
00140         M_free(terms, "poly::~poly");
00141 }
00142
00143 /*
00144     #] destructor :
00145     #[ expand_memory :
00146 */
00147
00148 // expands the memory allocated for terms to at least twice its size
00149 void poly::expand_memory (int i) {
00150
00151     POLY_GETIDENTITY(*this);
00152
00153     LONG new_size_of_terms = MaX(2*size_of_terms, i);
00154     if (new_size_of_terms > MAXPOSITIVE) {
00155         MLOCK(ErrorMessageLock);
00156         MesPrint ((char*)"ERROR: polynomials too large (> WORDSIZE)");
00157         MUNLOCK(ErrorMessageLock);
00158         Terminate(1);
00159     }
00160
00161     WORD *newterms = (WORD *)Malloc1(new_size_of_terms * sizeof(WORD), "poly::expand_memory");
00162     WCOPY(newterms, terms, size_of_terms);
00163
00164     if (size_of_terms == AM.MaxTer/(LONG)sizeof(WORD))
00165         TermFree(terms, "poly::expand_memory");
00166     else
00167         M_free(terms, "poly::expand_memory");
00168
00169     terms = newterms;
00170     size_of_terms = new_size_of_terms;
00171 }
00172
00173 /*
00174     #] expand_memory :
00175     #[ setmod :
00176 */
00177
00178 // sets the coefficient space to ZZ/p^n
00179 void poly::setmod(WORD _modp, WORD _modn) {
00180
00181     POLY_GETIDENTITY(*this);
00182
00183     if (_modp>0 && (_modp!=modp || _modn<modn)) {
00184         modp = _modp;
00185         modn = _modn;
00186
00187         WORD nmodq=0;
00188         UWORD *modq=NULL;
00189         bool smallq;
```

```
00190
00191         if (modn == 1) {
00192             modq = (UWORD *)&modp;
00193             nmodq = 1;
00194             smallq = true;
00195         }
00196         else {
00197             RaisPowCached(BHEAD modp,modn,&modq,&nmodq);
00198             smallq = false;
00199         }
00200
00201         coefficients_modulo(modq,nmodq,smallq);
00202     }
00203     else {
00204         modp = _modp;
00205         modn = _modn;
00206     }
00207 }
00208
00209 /*
00210     #] setmod :
00211     #[ coefficients_modulo :
00212 */
00213
00214 // reduces all coefficients of the polynomial modulo a
00215 void poly::coefficients_modulo (UWORD *a, WORD na, bool small) {
00216
00217     POLY_GETIDENTITY(*this);
00218
00219     int j=1;
00220     for (int i=1, di; i<terms[0]; i+=di) {
00221         di = terms[i];
00222
00223         if (i!=j)
00224             for (int k=0; k<di; k++)
00225                 terms[j+k] = terms[i+k];
00226
00227         WORD n = terms[j+terms[j]-1];
00228
00229         if (ABS(n)==1 && small) {
00230             // small number modulo small number
00231             terms[j+1+AN.poly_num_vars] = n*((UWORD)terms[j+1+AN.poly_num_vars] % a[0]);
00232             if (terms[j+1+AN.poly_num_vars] == 0)
00233                 n=0;
00234             else {
00235                 if (terms[j+1+AN.poly_num_vars] > +(WORD)a[0]/2) terms[j+1+AN.poly_num_vars]-=a[0];
00236                 if (terms[j+1+AN.poly_num_vars] < -(WORD)a[0]/2) terms[j+1+AN.poly_num_vars]+=a[0];
00237                 n = SGN(terms[j+1+AN.poly_num_vars]);
00238                 terms[j+1+AN.poly_num_vars] = ABS(terms[j+1+AN.poly_num_vars]);
00239             }
00240         }
00241         else
00242             TakeNormalModulus((UWORD *)&terms[j+1+AN.poly_num_vars], &n, a, na, NOUNPACK);
00243
00244         if (n!=0) {
00245             terms[j] = 2+AN.poly_num_vars+ABS(n);
00246             terms[j+terms[j]-1] = n;
00247             j += terms[j];
00248         }
00249     }
00250
00251     terms[0] = j;
00252 }
00253
00254 /*
00255     #] coefficients_modulo :
00256     #[ to_string :
00257 */
00258
00259 // converts an integer to a string
00260 const string int_to_string (WORD x) {
00261     char res[41];
00262     snprintf (res, 41, "%i",x);
00263     return res;
00264 }
00265
00266 // converts a polynomial to a string
00267 const string poly::to_string() const {
00268
00269     POLY_GETIDENTITY(*this);
00270
00271     string res;
00272
00273     int printtimes;
00274     UBYTE *scoeff = (UBYTE *)NumberMalloc("poly::to_string");
00275
00276     if (terms[0]==1)
```

```
00277         // zero
00278         res = "0";
00279     else {
00280         for (int i=1; i<terms[0]; i+=terms[i]) {
00281
00282             // sign
00283             WORD ncoeff = terms[i+terms[i]-1];
00284             if (ncoeff < 0) {
00285                 ncoeff*=-1;
00286                 res += "-";
00287             }
00288             else {
00289                 if (i>1) res += "+";
00290             }
00291
00292             if (ncoeff==1 && terms[i+terms[i]-1-ncoeff]==1) {
00293                 // coeff=1, so don't print coefficient and '*'
00294                 printtimes = 0;
00295             }
00296             else {
00297                 // print coefficient
00298                 PrtLong((UWORD*)&terms[i+terms[i]-1-ncoeff], ncoeff, scoeff);
00299                 res += string((char *)scoeff);
00300                 printtimes=1;
00301             }
00302
00303             // print variables
00304             for (int j=0; j<AN.poly_num_vars; j++) {
00305                 if (terms[i+1+j] > 0) {
00306                     if (printtimes) res += "*";
00307                     res += string(1,'a'+j);
00308                     if (terms[i+1+j] > 1) res += "^" + int_to_string(terms[i+1+j]);
00309                     printtimes = 1;
00310                 }
00311             }
00312
00313             // iff coeff=1 and all power=0, print '1' after all
00314             if (!printtimes) res += "1";
00315         }
00316     }
00317
00318     // eventual modulo
00319     if (modp>0) {
00320         res += " (mod ";
00321         res += int_to_string(modp);
00322         if (modn>1) {
00323             res += "^";
00324             res += int_to_string(modn);
00325         }
00326         res += ")";
00327     }
00328
00329     NumberFree(scoeff,"poly::to_string");
00330
00331     return res;
00332 }
00333
00334 /*
00335     #] to_string :
00336     #[ ostream operator :
00337 */
00338
00339 // output stream operator
00340 ostream& operator« (ostream &out, const poly &a) {
00341     return out « a.to_string();
00342 }
00343
00344 /*
00345     #] ostream operator :
00346     #[ monomial_compare :
00347 */
00348
00349 // compares two monomials (0:equal, <0:a smaller, >0:b smaller)
00350 int poly::monomial_compare (PHEAD const WORD *a, const WORD *b) {
00351     for (int i=0; i<AN.poly_num_vars; i++)
00352         if (a[i+1]!=b[i+1]) return a[i+1]-b[i+1];
00353     return 0;
00354 }
00355
00356 /*
00357     #] monomial_compare :
00358     #[ normalize :
00359 */
00360
00361 // brings a polynomial to normal form
00362 const poly & poly::normalize() {
00363
```

```
00364     POLY_GETIDENTITY(*this);
00365
00366     WORD nmodq=0;
00367     UWORD *modq=NULL;
00368
00369     if (modp!=0) {
00370         if (modn == 1) {
00371             modq = (UWORD *)&modp;
00372             nmodq = 1;
00373         }
00374         else {
00375             RaisPowCached(BHEAD modp,modn,&modq,&nmodq);
00376         }
00377     }
00378
00379     // find and sort all monomials
00380     // terms[0]/num_vars+3 is an upper bound for number of terms in a
00381
00382     LONG poffset = AT.pWorkPointer;
00383     WantAddPointers((terms[0]/(AN.poly_num_vars+3)));
00384     AT.pWorkPointer += terms[0]/(AN.poly_num_vars+3);
00385     #define p (&(AT.pWorkSpace[poffset]))
00386
00387 //  WORD **p = (WORD **)Malloc1(terms[0]/(AN.poly_num_vars+3) * sizeof(WORD*), "poly::normalize");
00388
00389     int nterms = 0;
00390     for (int i=1; i<terms[0]; i+=terms[i])
00391         p[nterms++] = terms + i;
00392
00393     sort(p, p + nterms, monomial_larger(BHEAD0));
00394
00395     WORD *tmp;
00396     if (size_of_terms == AM.MaxTer/(LONG)sizeof(WORD)) {
00397         tmp = (WORD *)TermMalloc("poly::normalize");
00398     }
00399     else {
00400         tmp = (WORD *)Malloc1(size_of_terms * sizeof(WORD), "poly::normalize");
00401     }
00402     int j=1;
00403     int prevj=0;
00404     tmp[0]=0;
00405     tmp[1]=0;
00406
00407     for (int i=0; i<nterms; i++) {
00408         if (i>0 && monomial_compare(BHEAD &tmp[j], p[i])==0) {
00409             // duplicate term, so add coefficients
00410             WORD ncoeff = tmp[j+tmp[j]-1];
00411             AddLong((UWORD *)&tmp[j+1+AN.poly_num_vars], ncoeff,
00412                             (UWORD *)&p[i][1+AN.poly_num_vars], p[i][p[i][0]-1],
00413                             (UWORD *)&tmp[j+1+AN.poly_num_vars], &ncoeff);
00414
00415             tmp[j+1+AN.poly_num_vars+ABS(ncoeff)] = ncoeff;
00416             tmp[j] = 2+AN.poly_num_vars+ABS(ncoeff);
00417         }
00418         else {
00419             // new term
00420             prevj = j;
00421             j += tmp[j];
00422             WCOPY(&tmp[j],p[i],p[i][0]);
00423         }
00424
00425         if (modp!=0) {
00426             // bring coefficient to normal form mod p^n
00427             WORD ntmp = tmp[j+tmp[j]-1];
00428             TakeNormalModulus((UWORD *)&tmp[j+1+AN.poly_num_vars], &ntmp,
00429                                                     modq,nmodq, NOUNPACK);
00430             tmp[j] = 2+AN.poly_num_vars+ABS(ntmp);
00431             tmp[j+tmp[j]-1] = ntmp;
00432         }
00433
00434         // add terms to polynomial
00435         if (tmp[j+tmp[j]-1]==0) {
00436             tmp[j]=0;
00437             j=prevj;
00438         }
00439     }
00440
00441     j+=tmp[j];
00442
00443     tmp[0] = j;
00444     WCOPY(terms,tmp,tmp[0]);
00445
00446 //  M_free(p, "poly::normalize");
00447
00448     if (size_of_terms == AM.MaxTer/(LONG)sizeof(WORD))
00449         TermFree(tmp, "poly::normalize");
00450     else
```

```
00451         M_free(tmp, "poly::normalize");
00452
00453     AT.pWorkPointer = poffset;
00454     #undef p
00455
00456     return *this;
00457 }
00458
00459 /*
00460     #] normalize :
00461     #[ last_monomial_index :
00462 */
00463
00464 // index of the last monomial, i.e., the constant term
00465 WORD poly::last_monomial_index () const {
00466     POLY_GETIDENTITY(*this);
00467     return terms[0] - ABS(terms[terms[0]-1]) - AN.poly_num_vars - 2;
00468 }
00469
00470
00471 // pointer to the last monomial (the constant term)
00472 WORD * poly::last_monomial () const {
00473     return &terms[last_monomial_index()];
00474 }
00475
00476 /*
00477     #] last_monomial_index :
00478     #[ compare_degree_vector :
00479 */
00480
00481 int poly::compare_degree_vector(const poly & b) const {
00482     POLY_GETIDENTITY(*this);
00483
00484     // special cases if one or both are 0
00485     if (terms[0] == 1 &&  b[0] == 1) return 0;
00486     if (terms[0] == 1) return -1;
00487     if (b[0] == 1) return 1;
00488
00489     for (int i = 0; i < AN.poly_num_vars; i++) {
00490         int d1 = degree(i);
00491         int d2 = b.degree(i);
00492         if (d1 != d2) return d1 - d2;
00493     }
00494
00495     return 0;
00496 }
00497
00498 /*
00499     #] compare_degree_vector :
00500     #[ degree_vector :
00501 */
00502
00503 std::vector<int> poly::degree_vector() const {
00504     POLY_GETIDENTITY(*this);
00505
00506     std::vector<int> degrees(AN.poly_num_vars);
00507     for (int i = 0; i < AN.poly_num_vars; i++) {
00508         degrees[i] = degree(i);
00509     }
00510
00511     return degrees;
00512 }
00513
00514 /*
00515     #] degree_vector :
00516     #[ compare_degree_vector :
00517 */
00518
00519 int poly::compare_degree_vector(const vector<int> & b) const {
00520     POLY_GETIDENTITY(*this);
00521
00522     if (terms[0] == 1) return -1;
00523
00524     for (int i = 0; i < AN.poly_num_vars; i++) {
00525         int d1 = degree(i);
00526         if (d1 != b[i]) return d1 - b[i];
00527     }
00528
00529     return 0;
00530 }
00531
00532 /*
00533     #] compare_degree_vector :
00534     #[ add :
00535 */
00536
00537 // addition of polynomials by merging
```

```
00538 void poly::add (const poly &a, const poly &b, poly &c) {
00539
00540     POLY_GETIDENTITY(a);
00541
00542   c.modp = a.modp;
00543   c.modn = a.modn;
00544
00545     WORD nmodq=0;
00546     UWORD *modq=NULL;
00547
00548     bool both_mod_small=false;
00549
00550     if (c.modp!=0) {
00551         if (c.modn == 1) {
00552             modq = (UWORD *)&c.modp;
00553             nmodq = 1;
00554             if (a.modp>0 && b.modp>0 && a.modn==1 && b.modn==1)
00555                 both_mod_small=true;
00556         }
00557         else {
00558             RaisPowCached(BHEAD c.modp,c.modn,&modq,&nmodq);
00559         }
00560     }
00561
00562     int ai=1,bi=1,ci=1;
00563
00564     while (ai<a[0] || bi<b[0]) {
00565
00566         c.check_memory(ci);
00567
00568         int cmp = ai<a[0] && bi<b[0] ? monomial_compare(BHEAD &a[ai],&b[bi]) : 0;
00569
00570         if (bi==b[0] || cmp>0) {
00571             // insert term from a
00572             c.termscopy(&a[ai],ci,a[ai]);
00573             ci+=a[ai];
00574             ai+=a[ai];
00575         }
00576         else if (ai==a[0] || cmp<0) {
00577             // insert term from b
00578             c.termscopy(&b[bi],ci,b[bi]);
00579             ci+=b[bi];
00580             bi+=b[bi];
00581         }
00582         else {
00583             // insert term from a+b
00584             c.termscopy(&a[ai],ci,MaX(a[ai],b[bi]));
00585             WORD nc=0;
00586             if (both_mod_small) {
00587                 c[ci+1+AN.poly_num_vars] = ((LONG)a[ai+1+AN.poly_num_vars]*a[ai+a[ai]-1]+
00588
      (LONG)b[bi+1+AN.poly_num_vars]*b[bi+b[bi]-1]) % c.modp;
00589                 if ((WORD)c[ci+1+AN.poly_num_vars] > +c.modp/2) c[ci+1+AN.poly_num_vars] -= c.modp;
00590                 if ((WORD)c[ci+1+AN.poly_num_vars] < -c.modp/2) c[ci+1+AN.poly_num_vars] += c.modp;
00591                 nc = (c[ci+1+AN.poly_num_vars]==0 ? 0 : SGN((WORD)c[ci+1+AN.poly_num_vars]));
00592                 c[ci+1+AN.poly_num_vars] = ABS((WORD)c[ci+1+AN.poly_num_vars]);

00593             }
00594             else {
00595                 AddLong((UWORD *)&a[ai+1+AN.poly_num_vars], a[ai+a[ai]-1],
00596                                 (UWORD *)&b[bi+1+AN.poly_num_vars], b[bi+b[bi]-1],
00597                                 (UWORD *)&c[ci+1+AN.poly_num_vars], &nc);
00598                 if (c.modp!=0) TakeNormalModulus((UWORD *)&c[ci+1+AN.poly_num_vars], &nc,
00599                                                                                 modq, nmodq,
      NOUNPACK);
00600             }
00601
00602             if (nc!=0) {
00603                 c[ci] = 2+AN.poly_num_vars+ABS(nc);
00604                 c[ci+c[ci]-1] = nc;
00605                 ci += c[ci];
00606             }
00607
00608             ai+=a[ai];
00609             bi+=b[bi];
00610         }
00611     }
00612
00613     c[0]=ci;
00614 }
00615
00616 /*
00617     #] add :
00618     #[ sub :
00619 */
00620
00621 // subtraction of polynomials by merging
```

```
00622 void poly::sub (const poly &a, const poly &b, poly &c) {
00623
00624     POLY_GETIDENTITY(a);
00625
00626   c.modp = a.modp;
00627   c.modn = a.modn;
00628
00629     WORD nmodq=0;
00630     UWORD *modq=NULL;
00631
00632     bool both_mod_small=false;
00633
00634     if (c.modp!=0) {
00635         if (c.modn == 1) {
00636             modq = (UWORD *)&c.modp;
00637             nmodq = 1;
00638             if (a.modp>0 && b.modp>0 && a.modn==1 && b.modn==1)
00639                 both_mod_small=true;
00640         }
00641         else {
00642             RaisPowCached(BHEAD c.modp,c.modn,&modq,&nmodq);
00643         }
00644     }
00645
00646     int ai=1,bi=1,ci=1;
00647
00648     while (ai<a[0] || bi<b[0]) {
00649
00650         c.check_memory(ci);
00651
00652         int cmp = ai<a[0] && bi<b[0] ? monomial_compare(BHEAD &a[ai],&b[bi]) : 0;
00653
00654         if (bi==b[0] || cmp>0) {
00655             // insert term from a
00656             c.termscopy(&a[ai],ci,a[ai]);
00657             ci+=a[ai];
00658             ai+=a[ai];
00659         }
00660         else if (ai==a[0] || cmp<0) {
00661             // insert term from b
00662             c.termscopy(&b[bi],ci,b[bi]);
00663             ci+=b[bi];
00664             bi+=b[bi];
00665             c[ci-1]*=-1;
00666         }
00667         else {
00668             // insert term from a+b
00669             c.termscopy(&a[ai],ci,MaX(a[ai],b[bi]));
00670             WORD nc=0;
00671             if (both_mod_small) {
00672                 c[ci+1+AN.poly_num_vars] = ((LONG)a[ai+1+AN.poly_num_vars]*a[ai+a[ai]-1]-
00673 (LONG)b[bi+1+AN.poly_num_vars]*b[bi+b[bi]-1]) % c.modp;
00674                 if ((WORD)c[ci+1+AN.poly_num_vars] > +c.modp/2) c[ci+1+AN.poly_num_vars] -= c.modp;
00675                 if ((WORD)c[ci+1+AN.poly_num_vars] < -c.modp/2) c[ci+1+AN.poly_num_vars] += c.modp;
00676                 nc = (c[ci+1+AN.poly_num_vars]==0 ? 0 : SGN((WORD)c[ci+1+AN.poly_num_vars]));
00677                 c[ci+1+AN.poly_num_vars] = ABS((WORD)c[ci+1+AN.poly_num_vars]);

00678             }
00679             else {
00680                 AddLong((UWORD *)&a[ai+1+AN.poly_num_vars], a[ai+a[ai]-1],
00681                                 (UWORD *)&b[bi+1+AN.poly_num_vars],-b[bi+b[bi]-1], // -b[...] causes
 subtraction
00682                                 (UWORD *)&c[ci+1+AN.poly_num_vars], &nc);
00683                 if (c.modp!=0) TakeNormalModulus((UWORD *)&c[ci+1+AN.poly_num_vars], &nc,
00684                                                                         modq, nmodq,
 NOUNPACK);
00685             }
00686
00687             if (nc!=0) {
00688                 c[ci] = 2+AN.poly_num_vars+ABS(nc);
00689                 c[ci+c[ci]-1] = nc;
00690                 ci += c[ci];
00691             }
00692
00693             ai+=a[ai];
00694             bi+=b[bi];
00695         }
00696     }
00697
00698     c[0]=ci;
00699 }
00700
00701 /*
00702     #] sub :
00703     #[ pop_heap :
00704 */
```

```
00705
00706 // pops the largest monomial from the heap and stores it in heap[n]
00707 void poly::pop_heap (PHEAD WORD **heap, int n) {
00708
00709     WORD *old = heap[0];
00710
00711     heap[0] = heap[--n];
00712
00713     int i=0;
00714     while (2*i+2<n && (monomial_compare(BHEAD heap[2*i+1]+3, heap[i]+3)>0 ||
00715                                         monomial_compare(BHEAD heap[2*i+2]+3, heap[i]+3)>0)) {
00716
00717         if (monomial_compare(BHEAD heap[2*i+1]+3, heap[2*i+2]+3)>0) {
00718             swap(heap[i], heap[2*i+1]);
00719             i=2*i+1;
00720         }
00721         else {
00722             swap(heap[i], heap[2*i+2]);
00723             i=2*i+2;
00724         }
00725     }
00726
00727     if (2*i+1<n && monomial_compare(BHEAD heap[2*i+1]+3, heap[i]+3)>0)
00728         swap(heap[i], heap[2*i+1]);
00729
00730     heap[n] = old;
00731 }
00732
00733 /*
00734     #] pop_heap :
00735     #[ push_heap :
00736 */
00737
00738 // pushes the monomial in heap[n] onto the heap
00739 void poly::push_heap (PHEAD WORD **heap, int n)  {
00740
00741     int i=n-1;
00742
00743     while (i>0 && monomial_compare(BHEAD heap[i]+3, heap[(i-1)/2]+3) > 0) {
00744         swap(heap[(i-1)/2], heap[i]);
00745         i=(i-1)/2;
00746     }
00747 }
00748
00749 /*
00750     #] push_heap :
00751     #[ mul_one_term :
00752 */
00753
00754 // multiplies a polynomial with a monomial
00755 void poly::mul_one_term (const poly &a, const poly &b, poly &c) {
00756
00757   POLY_GETIDENTITY(a);
00758
00759   int ci=1;
00760
00761     WORD nmodq=0;
00762     UWORD *modq=NULL;
00763
00764     bool both_mod_small=false;
00765
00766     if (c.modp!=0) {
00767         if (c.modn == 1) {
00768             modq = (UWORD *)&c.modp;
00769             nmodq = 1;
00770             if (a.modp>0 && b.modp>0 && a.modn==1 && b.modn==1)
00771                 both_mod_small=true;
00772         }
00773         else {
00774             RaisPowCached(BHEAD c.modp,c.modn,&modq,&nmodq);
00775         }
00776     }
00777
00778   for (int ai=1; ai<a[0]; ai+=a[ai])
00779     for (int bi=1; bi<b[0]; bi+=b[bi]) {
00780
00781             c.check_memory(ci);
00782
00783       for (int i=0; i<AN.poly_num_vars; i++)
00784         c[ci+1+i] = a[ai+1+i] + b[bi+1+i];
00785       WORD nc;
00786
00787             // if both polynomials are modulo p^1, use integer calculus
00788             if (both_mod_small) {
00789                 c[ci+1+AN.poly_num_vars] =
00790                     ((LONG)a[ai+1+AN.poly_num_vars] * a[ai+2+AN.poly_num_vars] *
00791                      b[bi+1+AN.poly_num_vars] * b[bi+2+AN.poly_num_vars]) % c.modp;
```

```
00792                 nc = (c[ci+1+AN.poly_num_vars]==0 ? 0 : 1);
00793             }
00794             else {
00795                 // otherwise, use form long calculus
00796                 MulLong((UWORD *)&a[ai+1+AN.poly_num_vars], a[ai+a[ai]-1],
00797                                 (UWORD *)&b[bi+1+AN.poly_num_vars], b[bi+b[bi]-1],
00798                                 (UWORD *)&c[ci+1+AN.poly_num_vars], &nc);
00799                 if (c.modp!=0) TakeNormalModulus((UWORD *)&c[ci+1+AN.poly_num_vars], &nc,
00800                                                                                 modq, nmodq,
      NOUNPACK);
00801             }
00802
00803             if (nc!=0) {
00804                 c[ci] = 2+AN.poly_num_vars+ABS(nc);
00805                 ci += c[ci];
00806
00807                 if (both_mod_small) {
00808                     if (c[ci-2] > +c.modp/2) c[ci-2] -= c.modp;
00809                     if (c[ci-2] < -c.modp/2) c[ci-2] += c.modp;
00810                     c[ci-1] = SGN(c[ci-2]);
00811                     c[ci-2] = ABS(c[ci-2]);
00812                 }
00813                 else {
00814                     c[ci-1] = nc;
00815                 }
00816             }
00817     }
00818
00819   c[0]=ci;
00820 }
00821
00822 /*
00823     #] mul_one_term :
00824     #[ mul_univar :
00825 */
00826
00827 // dense univariate multiplication, i.e., for each power find all
00828 // pairs of monomials that result in that power
00829 void poly::mul_univar (const poly &a, const poly &b, poly &c, int var) {
00830
00831     POLY_GETIDENTITY(a);
00832
00833     WORD nmodq=0;
00834     UWORD *modq=NULL;
00835
00836     bool both_mod_small=false;
00837
00838     if (c.modp!=0) {
00839         if (c.modn == 1) {
00840             modq = (UWORD *)&c.modp;
00841             nmodq = 1;
00842             if (a.modp>0 && b.modp>0 && a.modn==1 && b.modn==1)
00843                 both_mod_small=true;
00844         }
00845         else {
00846             RaisPowCached(BHEAD c.modp,c.modn,&modq,&nmodq);
00847         }
00848     }
00849
00850     poly t(BHEAD 0);
00851     WORD nt;
00852
00853     int ci=1;
00854
00855     // bounds on the powers in a*b
00856     int minpow = AN.poly_num_vars==0 ? 0 : a.last_monomial()[1+var] + b.last_monomial()[1+var];
00857     int maxpow = AN.poly_num_vars==0 ? 0 : a[2+var]+b[2+var];
00858     int afirst=1, blast=1;
00859
00860     for (int pow=maxpow; pow>=minpow; pow--) {
00861
00862         c.check_memory(ci);
00863
00864         WORD nc=0;
00865
00866         // adjust range in a or b
00867         if (a[afirst+1+var] + b[blast+1+var] > pow) {
00868             if (blast+b[blast] < b[0])
00869                 blast+=b[blast];
00870             else
00871                 afirst+=a[afirst];
00872         }
00873
00874         // find terms that result in the correct power
00875         for (int ai=afirst, bi=blast; ai<a[0] && bi>=1;) {
00876
00877             int thispow = AN.poly_num_vars==0 ? 0 : a[ai+1+var] + b[bi+1+var];
```

```
00878
00879             if (thispow == pow) {
00880
00881                 // if both polynomials are modulo p^1, use integer calculus
00882                 if (both_mod_small) {
00883                     c[ci+1+AN.poly_num_vars] =
00884                         ((nc==0 ? 0 : (LONG)c[ci+1+AN.poly_num_vars] * nc) +
00885                         (LONG)a[ai+1+AN.poly_num_vars] * a[ai+2+AN.poly_num_vars] *
00886                          b[bi+1+AN.poly_num_vars] * b[bi+2+AN.poly_num_vars]) % c.modp;
00887                     nc = (c[ci+1+AN.poly_num_vars]==0 ? 0 : 1);
00888                 }
00889                 else {
00890                     // otherwise, use form long calculus
00891
00892                     MulLong((UWORD *)&a[ai+1+AN.poly_num_vars], a[ai+a[ai]-1],
00893                                     (UWORD *)&b[bi+1+AN.poly_num_vars], b[bi+b[bi]-1],
00894                                     (UWORD *)&t[0], &nt);
00895
00896                     AddLong ((UWORD *)&t[0], nt,
00897                                     (UWORD *)&c[ci+1+AN.poly_num_vars], nc,
00898                                     (UWORD *)&c[ci+1+AN.poly_num_vars], &nc);
00899
00900                     if (c.modp!=0) TakeNormalModulus((UWORD *)&c[ci+1+AN.poly_num_vars], &nc,
00901                                                                                 modq, nmodq,
      NOUNPACK);
00902                 }
00903
00904                 ai += a[ai];
00905                 bi -= ABS(b[bi-1]) + 2 + AN.poly_num_vars;
00906             }
00907             else if (thispow > pow)
00908                 ai += a[ai];
00909             else
00910                 bi -= ABS(b[bi-1]) + 2 + AN.poly_num_vars;
00911         }
00912
00913         // add term to result
00914         if (nc != 0) {
00915             for (int j=0; j<AN.poly_num_vars; j++)
00916                 c[ci+1+j] = 0;
00917             if (AN.poly_num_vars > 0)
00918                 c[ci+1+var] = pow;
00919
00920             c[ci] = 2+AN.poly_num_vars+ABS(nc);
00921             ci += c[ci];
00922
00923             // if necessary, adjust to range [-p/2,p/2]
00924             if (both_mod_small) {
00925                 if (c[ci-2] > +c.modp/2) c[ci-2] -= c.modp;
00926                 if (c[ci-2] < -c.modp/2) c[ci-2] += c.modp;
00927                 c[ci-1] = SGN(c[ci-2]);
00928                 c[ci-2] = ABS(c[ci-2]);
00929             }
00930             else {
00931                 c[ci-1] = nc;
00932             }
00933         }
00934     }
00935
00936     c[0] = ci;
00937 }
00938
00939 /*
00940     #] mul_univar :
00941     #[ mul_heap :
00942 */
00943
00961 void poly::mul_heap (const poly &a, const poly &b, poly &c) {
00962
00963     POLY_GETIDENTITY(a);
00964
00965     WORD nmodq=0;
00966     UWORD *modq=NULL;
00967
00968     bool both_mod_small=false;
00969
00970     if (c.modp!=0) {
00971         if (c.modn == 1) {
00972             modq = (UWORD *)&c.modp;
00973             nmodq = 1;
00974             if (a.modp>0 && b.modp>0 && a.modn==1 && b.modn==1)
00975                 both_mod_small=true;
00976         }
00977         else {
00978             RaisPowCached(BHEAD c.modp,c.modn,&modq,&nmodq);
00979         }
00980     }
```

```
00981
00982     // find maximum powers in different variables
00983     WORD *maxpower  = AT.WorkPointer;
00984     AT.WorkPointer += AN.poly_num_vars;
00985     WORD *maxpowera = AT.WorkPointer;
00986     AT.WorkPointer += AN.poly_num_vars;
00987     WORD *maxpowerb = AT.WorkPointer;
00988     AT.WorkPointer += AN.poly_num_vars;
00989
00990     for (int i=0; i<AN.poly_num_vars; i++)
00991         maxpowera[i] = maxpowerb[i] = 0;
00992
00993     for (int ai=1; ai<a[0]; ai+=a[ai])
00994         for (int j=0; j<AN.poly_num_vars; j++)
00995             maxpowera[j] = MaX(maxpowera[j], a[ai+1+j]);
00996
00997     for (int bi=1; bi<b[0]; bi+=b[bi])
00998         for (int j=0; j<AN.poly_num_vars; j++)
00999             maxpowerb[j] = MaX(maxpowerb[j], b[bi+1+j]);
01000
01001     for (int i=0; i<AN.poly_num_vars; i++)
01002         maxpower[i] = maxpowera[i] + maxpowerb[i];
01003
01004     // if PROD(maxpower) small, use hash array
01005     bool use_hash = true;
01006     int nhash = 1;
01007
01008     for (int i=0; i<AN.poly_num_vars; i++) {
01009         if (nhash > POLY_MAX_HASH_SIZE / (maxpower[i]+1)) {
01010             nhash = 1;
01011             use_hash = false;
01012             break;
01013         }
01014         nhash *= maxpower[i]+1;
01015     }
01016
01017     // allocate heap and hash
01018     int nheap=a.number_of_terms();
01019
01020     WantAddPointers(nheap+nhash);
01021     WORD **heap = AT.pWorkSpace + AT.pWorkPointer;
01022
01023     for (int ai=1, i=0; ai<a[0]; ai+=a[ai], i++) {
01024         heap[i] = (WORD *) NumberMalloc("poly::mul_heap");
01025         heap[i][0] = ai;
01026         heap[i][1] = 1;
01027         heap[i][2] = -1;
01028         heap[i][3] = 0;
01029         for (int j=0; j<AN.poly_num_vars; j++)
01030             heap[i][4+j] = MAXPOSITIVE;
01031     }
01032
01033     WORD **hash = AT.pWorkSpace + AT.pWorkPointer + nheap;
01034     for (int i=0; i<nhash; i++)
01035         hash[i] = NULL;
01036
01037     int ci = 1;
01038
01039     // multiply
01040     while (nheap > 0) {
01041
01042         c.check_memory(ci);
01043
01044         pop_heap(BHEAD heap, nheap--);
01045         WORD *p = heap[nheap];
01046
01047         // if non-zero
01048         if (p[3] != 0) {
01049             if (use_hash) hash[p[2]] = NULL;
01050
01051             c[0] = ci;
01052
01053             // append this term to the result
01054             if (use_hash || ci==1 || monomial_compare(BHEAD p+3, c.last_monomial())!=0) {
01055                 p[4 + AN.poly_num_vars + ABS(p[3])] = p[3];
01056                 p[3] = 2 + AN.poly_num_vars + ABS(p[3]);
01057                 c.termscopy(&p[3],ci,p[3]);
01058                 ci += c[ci];
01059             }
01060             else {
01061                 // add this term to the last term of the result
01062                 ci = c.last_monomial_index();
01063                 WORD nc = c[ci+c[ci]-1];
01064
01065                 // if both polynomials are modulo p^1, use integer calculus
01066                 if (both_mod_small) {
01067                     c[ci+AN.poly_num_vars+1] = ((LONG)c[ci+AN.poly_num_vars+1]*nc +
```

```
    p[4+AN.poly_num_vars]*p[3]) % c.modp;
01068                     if (c[ci+1+AN.poly_num_vars]==0)
01069                         nc = 0;
01070                     else {
01071                         if (c[ci+1+AN.poly_num_vars] > +c.modp/2) c[ci+1+AN.poly_num_vars] -= c.modp;
01072                         if (c[ci+1+AN.poly_num_vars] < -c.modp/2) c[ci+1+AN.poly_num_vars] += c.modp;
01073                         nc = SGN(c[ci+1+AN.poly_num_vars]);
01074                         c[ci+1+AN.poly_num_vars] = ABS(c[ci+1+AN.poly_num_vars]);
01075                     }
01076                 }
01077                 else {
01078                     // otherwise, use form long calculus
01079                     AddLong ((UWORD *)&p[4+AN.poly_num_vars], p[3],
01080                                     (UWORD *)&c[ci+AN.poly_num_vars+1], nc,
01081                                     (UWORD *)&c[ci+AN.poly_num_vars+1],&nc);
01082
01083                     if (c.modp!=0) TakeNormalModulus((UWORD *)&c[ci+1+AN.poly_num_vars], &nc,
01084                                                                                 modq, nmodq,
    NOUNPACK);
01085                 }
01086
01087                 if (nc!=0) {
01088                     c[ci] = 2 + AN.poly_num_vars + ABS(nc);
01089                     ci += c[ci];
01090                     c[ci-1] = nc;
01091                 }
01092             }
01093         }
01094
01095         // add new term to the heap (ai, bi+1)
01096         while (p[1] < b[0]) {
01097
01098             for (int j=0; j<AN.poly_num_vars; j++)
01099                 p[4+j] = a[p[0]+1+j] + b[p[1]+1+j];
01100
01101             // if both polynomials are modulo p^1, use integer calculus
01102             if (both_mod_small) {
01103                 p[4+AN.poly_num_vars] = ((LONG)a[p[0]+1+AN.poly_num_vars]*a[p[0]+a[p[0]]-1]*
01104
    b[p[1]+1+AN.poly_num_vars]*b[p[1]+b[p[1]]-1]) % c.modp;
01105                 if (p[4+AN.poly_num_vars]==0)
01106                     p[3]=0;
01107                 else {
01108                     if (p[4+AN.poly_num_vars] > +c.modp/2) p[4+AN.poly_num_vars] -= c.modp;
01109                     if (p[4+AN.poly_num_vars] < -c.modp/2) p[4+AN.poly_num_vars] += c.modp;
01110                     p[3] = SGN(p[4+AN.poly_num_vars]);
01111                     p[4+AN.poly_num_vars] = ABS(p[4+AN.poly_num_vars]);
01112                 }
01113             }
01114             else {
01115                 // otherwise, use form long calculus
01116                 MulLong((UWORD *)&a[p[0]+1+AN.poly_num_vars], a[p[0]+a[p[0]]-1],
01117                                 (UWORD *)&b[p[1]+1+AN.poly_num_vars], b[p[1]+b[p[1]]-1],
01118                                 (UWORD *)&p[4+AN.poly_num_vars], &p[3]);
01119
01120                 if (c.modp!=0) TakeNormalModulus((UWORD *)&p[4+AN.poly_num_vars], &p[3],
01121                                                                             modq, nmodq,
    NOUNPACK);
01122             }
01123
01124             p[1] += b[p[1]];
01125
01126             if (use_hash) {
01127                 int ID = 0;
01128                 for (int i=0; i<AN.poly_num_vars; i++)
01129                     ID = (maxpower[i]+1)*ID + p[4+i];
01130
01131                 // if hash and unused, push onto heap
01132                 if (hash[ID] == NULL) {
01133                     p[2] = ID;
01134                     hash[ID] = p;
01135                     push_heap(BHEAD heap, ++nheap);
01136                     break;
01137                 }
01138                 else {
01139                     // if hash and used, add to heap element
01140                     WORD *h = hash[ID];
01141                     // if both polynomials are modulo p^1, use integer calculus
01142                     if (both_mod_small) {
01143                         h[4+AN.poly_num_vars] = ((LONG)p[4+AN.poly_num_vars]*p[3] +
    h[4+AN.poly_num_vars]*h[3]) % c.modp;
01144                         if (h[4+AN.poly_num_vars]==0)
01145                             h[3]=0;
01146                         else {
01147                             if (h[4+AN.poly_num_vars] > +c.modp/2) h[4+AN.poly_num_vars] -= c.modp;
01148                             if (h[4+AN.poly_num_vars] < -c.modp/2) h[4+AN.poly_num_vars] += c.modp;
01149                             h[3] = SGN(h[4+AN.poly_num_vars]);
```

```
01150                             h[4+AN.poly_num_vars] = ABS(h[4+AN.poly_num_vars]);
01151                         }
01152                     }
01153                     else {
01154                         // otherwise, use form long calculus
01155                         AddLong ((UWORD *)&p[4+AN.poly_num_vars],  p[3],
01156                                  (UWORD *)&h[4+AN.poly_num_vars],  h[3],
01157                                  (UWORD *)&h[4+AN.poly_num_vars], &h[3]);
01158
01159                         if (c.modp!=0) TakeNormalModulus((UWORD *)&h[4+AN.poly_num_vars], &h[3],
01160                                                                                           modq, nmodq,
      NOUNPACK);
01161                     }
01162                 }
01163             }
01164             else {
01165                 // if no hash, push onto heap
01166                 p[2] = -1;
01167                 push_heap(BHEAD heap, ++nheap);
01168                 break;
01169             }
01170         }
01171     }
01172
01173     c[0] = ci;
01174
01175     for (int ai=1, i=0; ai<a[0]; ai+=a[ai], i++)
01176         NumberFree(heap[i],"poly::mul_heap");
01177     AT.WorkPointer -= 3 * AN.poly_num_vars;
01178 }
01179
01180 /*
01181     #] mul_heap :
01182     #[ mul :
01183 */
01184
01195 void poly::mul (const poly &a, const poly &b, poly &c) {
01196
01197   c.modp = a.modp;
01198   c.modn = a.modn;
01199
01200     if (a.is_zero() || b.is_zero()) { c[0]=1; return; }
01201     if (a.is_one()) {
01202         c.check_memory(b[0]);
01203         c.termscopy(b.terms,0,b[0]);
01204         // normalize if necessary
01205         if (a.modp!=b.modp || a.modn!=b.modn) {
01206             c.modp=0;
01207             c.setmod(a.modp,a.modn);
01208         }
01209         return;
01210     }
01211     if (b.is_one()) {
01212         c.check_memory(a[0]);
01213         c.termscopy(a.terms,0,a[0]);
01214         return;
01215     }
01216
01217     int na=a.number_of_terms();
01218     int nb=b.number_of_terms();
01219
01220     if (na==1 || nb==1) {
01221         mul_one_term(a,b,c);
01222         return;
01223     }
01224
01225     int vara = a.is_dense_univariate();
01226     int varb = b.is_dense_univariate();
01227
01228     if (vara!=-2 && varb!=-2 && vara==varb) {
01229         mul_univar(a,b,c,vara);
01230         return;
01231     }
01232
01233     if (na < nb)
01234         mul_heap(a,b,c);
01235     else
01236         mul_heap(b,a,c);
01237 }
01238
01239 /*
01240     #] mul :
01241     #[ divmod_one_term : :
01242 */
01243
01244 // divides a polynomial by a monomial
01245 void poly::divmod_one_term (const poly &a, const poly &b, poly &q, poly &r, bool only_divides) {
```

```
01246
01247     POLY_GETIDENTITY(a);
01248
01249     int qi=1, ri=1;
01250
01251     WORD nmodq=0;
01252     UWORD *modq=NULL;
01253
01254     WORD nltbinv=0;
01255     UWORD *ltbinv=NULL;
01256
01257     bool both_mod_small=false;
01258
01259     if (q.modp!=0) {
01260         if (q.modn == 1) {
01261             modq = (UWORD *)&q.modp;
01262             nmodq = 1;
01263             if (a.modp>0 && b.modp>0 && a.modn==1 && b.modn==1)
01264                 both_mod_small=true;
01265         }
01266         else {
01267             RaisPowCached(BHEAD q.modp,q.modn,&modq,&nmodq);
01268         }
01269
01270         ltbinv = NumberMalloc("poly::div_one_term");
01271
01272         if (both_mod_small) {
01273             WORD ltb = b[b[1]]*b[2+AN.poly_num_vars];
01274             GetModInverses(ltb + (ltb<0?q.modp:0), q.modp, (WORD*)ltbinv, NULL);
01275             nltbinv = 1;
01276         }
01277         else
01278             GetLongModInverses(BHEAD (UWORD *)&b[2+AN.poly_num_vars], b[b[1]], modq, nmodq, ltbinv,
    &nltbinv, NULL, NULL);
01279     }
01280
01281     for (int ai=1; ai<a[0]; ai+=a[ai]) {
01282
01283         q.check_memory(qi);
01284         r.check_memory(ri);
01285
01286         // check divisibility of powers
01287         bool div=true;
01288         for (int j=0; j<AN.poly_num_vars; j++) {
01289             q[qi+1+j] = a[ai+1+j]-b[2+j];
01290             r[ri+1+j] = a[ai+1+j];
01291             if (q[qi+1+j] < 0) div=false;
01292         }
01293
01294         WORD nq,nr;
01295
01296         if (div) {
01297             // if variables are divisible, divide coefficient
01298             if (q.modp==0) {
01299                 DivLong((UWORD *)&a[ai+1+AN.poly_num_vars], a[ai+a[ai]-1],
01300                                 (UWORD *)&b[2+AN.poly_num_vars], b[b[1]],
01301                                 (UWORD *)&q[qi+1+AN.poly_num_vars], &nq,
01302                                 (UWORD *)&r[ri+1+AN.poly_num_vars], &nr);
01303             }
01304             else {
01305                 // if both polynomials are modulo p^1, use integer calculus
01306                 if (both_mod_small) {
01307                     q[qi+1+AN.poly_num_vars] =
01308                         ((LONG)a[ai+1+AN.poly_num_vars] * a[ai+a[ai]-1] * ltbinv[0] * nltbinv) %
    q.modp;
01309                     nq = (q[qi+1+AN.poly_num_vars]==0 ? 0 : 1);
01310                 }
01311                 else {
01312                     // otherwise, use form long calculus
01313                     MulLong((UWORD *)&a[ai+1+AN.poly_num_vars], a[ai+a[ai]-1],
01314                                     ltbinv, nltbinv,
01315                                     (UWORD *)&q[qi+1+AN.poly_num_vars], &nq);
01316                     TakeNormalModulus((UWORD *)&q[qi+1+AN.poly_num_vars], &nq,
01317                                                     modq,nmodq, NOUNPACK);
01318                 }
01319
01320                 nr=0;
01321             }
01322         }
01323         else {
01324             // if not, term becomes part of the remainder
01325             nq=0;
01326             nr=a[ai+a[ai]-1];
01327             r.termscopy(&a[ai+1+AN.poly_num_vars], ri+1+AN.poly_num_vars, ABS(nr));
01328         }
01329
01330         // add terms to quotient/remainder
```

```
01331         if (nq!=0) {
01332             q[qi] = 2+AN.poly_num_vars+ABS(nq);
01333             qi += q[qi];
01334 
01335             // if necessary, adjust to range [-p/2,p/2]
01336             if (both_mod_small) {
01337                 if (q[qi-2] > +q.modp/2) q[qi-2] -= q.modp;
01338                 if (q[qi-2] < -q.modp/2) q[qi-2] += q.modp;
01339                 q[qi-1] = SGN(q[qi-2]);
01340                 q[qi-2] = ABS(q[qi-2]);
01341             }
01342             else {
01343                 q[qi-1] = nq;
01344             }
01345         }
01346 
01347         if (nr != 0) {
01348             if (only_divides) { r = poly(BHEAD 1); ri=r[0]; break; }
01349             r[ri] = 2+AN.poly_num_vars+ABS(nr);
01350             ri += r[ri];
01351             r[ri-1] = nr;
01352         }
01353     }
01354 
01355     q[0]=qi;
01356     r[0]=ri;
01357 
01358     if (q.modp!=0 || ltbinv != NULL) NumberFree(ltbinv,"poly::div_one_term");
01359 }
01360 
01361 /*
01362     #] divmod_one_term :
01363     #[ divmod_univar : :
01364 */
01365 
01376 void poly::divmod_univar (const poly &a, const poly &b, poly &q, poly &r, int var, bool only_divides)
      {
01377 
01378     POLY_GETIDENTITY(a);
01379 
01380     WORD nmodq=0;
01381     UWORD *modq=NULL;
01382 
01383     WORD nltbinv=0;
01384     UWORD *ltbinv=NULL;
01385 
01386     bool both_mod_small=false;
01387 
01388     if (q.modp!=0) {
01389         if (q.modn == 1) {
01390             modq = (UWORD *)&q.modp;
01391             nmodq = 1;
01392             if (a.modp>0 && b.modp>0 && a.modn==1 && b.modn==1)
01393                 both_mod_small=true;
01394         }
01395         else {
01396             RaisPowCached(BHEAD q.modp,q.modn,&modq,&nmodq);
01397         }
01398         ltbinv = NumberMalloc("poly::div_univar");
01399 
01400         if (both_mod_small) {
01401             WORD ltb = b[b[1]]*b[2+AN.poly_num_vars];
01402             GetModInverses(ltb + (ltb<0?q.modp:0), q.modp, (WORD*)ltbinv, NULL);
01403             nltbinv = 1;
01404         }
01405         else
01406             GetLongModInverses(BHEAD (UWORD *)&b[2+AN.poly_num_vars], b[b[1]], modq, nmodq, ltbinv,
      &nltbinv, NULL, NULL);
01407     }
01408 
01409     WORD ns=0;
01410     WORD nt;
01411     UWORD *s = NumberMalloc("poly::div_univar");
01412     UWORD *t = NumberMalloc("poly::div_univar");
01413     s[0]=0;
01414 
01415     int bpow = b[2+var];
01416 
01417     int ai=1, qi=1, ri=1;
01418 
01419     for (int pow=a[2+var]; pow>=0; pow--) {
01420 
01421         q.check_memory(qi);
01422         r.check_memory(ri);
01423 
01424         // look for the correct power in a
01425         while (ai<a[0] && a[ai+1+var] > pow)
```

```
01426             ai+=a[ai];
01427
01428         // first term of the r.h.s. of the above equation
01429         if (ai<a[0] && a[ai+1+var] == pow) {
01430             ns = a[ai+a[ai]-1];
01431             WCOPY(s, &a[ai+1+AN.poly_num_vars], ABS(ns));
01432         }
01433         else {
01434             ns = 0;
01435         }
01436
01437         int bi=1, qj=qi;
01438
01439         // second term(s) of the r.h.s. of the above equation
01440         while (qj>1 && bi<b[0]) {
01441
01442             qj -= 2 + AN.poly_num_vars + ABS(q[qj-1]);
01443
01444             while (bi<b[0] && b[bi+1+var]+q[qj+1+var] > pow)
01445                 bi += b[bi];
01446
01447             if (bi<b[0] && b[bi+1+var]+q[qj+1+var] == pow) {
01448                 // if both polynomials are modulo p^1, use integer calculus
01449
01450                 if (both_mod_small) {
01451                     s[0] = ((WORD)s[0]*ns - (LONG)b[bi+1+AN.poly_num_vars] * b[bi+b[bi]-1] *
01452                                     q[qj+1+AN.poly_num_vars] * q[qj+q[qj]-1]) % q.modp;
01453                     ns = (s[0]==0 ? 0 : 1);
01454                 }
01455                 else {
01456                     // otherwise, use form long calculus
01457                     MulLong((UWORD *)&b[bi+1+AN.poly_num_vars], b[bi+b[bi]-1],
01458                                     (UWORD *)&q[qj+1+AN.poly_num_vars], q[qj+q[qj]-1],
01459                                     t, &nt);
01460                     nt *= -1;
01461                     AddLong(t,nt,s,ns,s,&ns);
01462                     if (q.modp!=0) TakeNormalModulus((UWORD *)s,&ns,modq, nmodq, NOUNPACK);
01463                 }
01464             }
01465         }
01466
01467         if (ns != 0) {
01468             // if necessary, adjust to range [-p/2,p/2]
01469             if (both_mod_small) {
01470                 s[0]*=ns;
01471                 if ((WORD)s[0] > +q.modp/2) s[0] -= q.modp;
01472                 if ((WORD)s[0] < -q.modp/2) s[0] += q.modp;
01473                 ns = SGN((WORD)s[0]);
01474                 s[0] = ABS((WORD)s[0]);
01475             }
01476
01477             if (pow >= bpow) {
01478                 // large power, so divide by b
01479                 if (q.modp == 0) {
01480                     DivLong(s, ns,
01481                                     (UWORD *)&b[2+AN.poly_num_vars],  b[b[1]],
01482                                     (UWORD *)&q[qi+1+AN.poly_num_vars], &ns, t, &nt);
01483                 }
01484                 else {
01485                     if (both_mod_small) {
01486                         q[qi+1+AN.poly_num_vars] = ((LONG)s[0]*ns*ltbinv[0]*nltbinv) % q.modp;
01487                         if ((WORD)q[qi+1+AN.poly_num_vars] > +q.modp/2) q[qi+1+AN.poly_num_vars] -=
      q.modp;
01488                         if ((WORD)q[qi+1+AN.poly_num_vars] < -q.modp/2) q[qi+1+AN.poly_num_vars] +=
      q.modp;
01489                         ns = (q[qi+1+AN.poly_num_vars]==0 ? 0 : SGN((WORD)q[qi+1+AN.poly_num_vars]));
01490                         q[qi+1+AN.poly_num_vars] = ABS((WORD)q[qi+1+AN.poly_num_vars]);

01491                     }
01492                     else {
01493                         MulLong(s, ns, ltbinv, nltbinv, (UWORD *)&q[qi+1+AN.poly_num_vars], &ns);
01494                         TakeNormalModulus((UWORD *)&q[qi+1+AN.poly_num_vars], &ns,
01495                                                     modq,nmodq, NOUNPACK);
01496                     }
01497                     nt=0;
01498                 }
01499             }
01500             else {
01501                 // small power, so remainder
01502                 WCOPY(t,s,ABS(ns));
01503                 nt = ns;
01504                 ns = 0;
01505             }
01506
01507             // add terms to quotient/remainder
01508             if (ns!=0) {
01509                 for (int i=0; i<AN.poly_num_vars; i++)
```

```
01510                   q[qi+1+i] = 0;
01511               q[qi+1+var] = pow-bpow;
01512
01513               q[qi] = 2+AN.poly_num_vars+ABS(ns);
01514               qi += q[qi];
01515               q[qi-1] = ns;
01516           }
01517
01518           if (nt != 0) {
01519               if (only_divides) { r=poly(BHEAD 1); ri=r[0]; break; }
01520               for (int i=0; i<AN.poly_num_vars; i++)
01521                   r[ri+1+i] = 0;
01522               r[ri+1+var] = pow;
01523
01524               for (int i=0; i<ABS(nt); i++)
01525                   r[ri+1+AN.poly_num_vars+i] = t[i];
01526
01527               r[ri] = 2+AN.poly_num_vars+ABS(nt);
01528               ri += r[ri];
01529               r[ri-1] = nt;
01530           }
01531       }
01532   }
01533
01534   q[0] = qi;
01535   r[0] = ri;
01536
01537   NumberFree(s,"poly::div_univar");
01538   NumberFree(t,"poly::div_univar");
01539
01540   if (q.modp!=0 || ltbinv!=NULL) NumberFree(ltbinv,"poly::div_univar");
01541 }
01542
01543 /*
01544     #] divmod_univar :
01545     #[ divmod_heap :
01546 */
01547
01579 void poly::divmod_heap (const poly &a, const poly &b, poly &q, poly &r, bool only_divides, bool
      check_div, bool &div_fail) {
01580
01581     POLY_GETIDENTITY(a);
01582
01583     div_fail = false;
01584     q[0] = r[0] = 1;
01585
01586     WORD nmodq=0;
01587     UWORD *modq=NULL;
01588
01589     WORD nltbinv=0;
01590     UWORD *ltbinv=NULL;
01591     LONG oldpWorkPointer = AT.pWorkPointer;
01592
01593     bool both_mod_small=false;
01594
01595     if (q.modp!=0) {
01596         if (q.modn == 1) {
01597             modq = (UWORD *)&q.modp;
01598             nmodq = 1;
01599             if (a.modp>0 && b.modp>0 && a.modn==1 && b.modn==1)
01600                 both_mod_small=true;
01601         }
01602         else {
01603             RaisPowCached(BHEAD q.modp,q.modn,&modq,&nmodq);
01604         }
01605         ltbinv = NumberMalloc("poly::div_heap-a");
01606
01607         if (both_mod_small) {
01608             WORD ltb = b[b[1]]*b[2+AN.poly_num_vars];
01609             GetModInverses(ltb + (ltb<0?q.modp:0), q.modp, (WORD*)ltbinv, NULL);
01610             nltbinv = 1;
01611         }
01612         else
01613             GetLongModInverses(BHEAD (UWORD *)&b[2+AN.poly_num_vars], b[b[1]], modq, nmodq, ltbinv,
      &nltbinv, NULL, NULL);
01614     }
01615
01616     // allocate heap
01617     int nb=b.number_of_terms();
01618     WantAddPointers(nb);
01619     WORD **heap = AT.pWorkSpace + AT.pWorkPointer;
01620     AT.pWorkPointer += nb;
01621
01622     for (int i=0; i<nb; i++)
01623         heap[i] = (WORD *) NumberMalloc("poly::div_heap-b");
01624     int nheap = 1;
01625     heap[0][0] = 1;
```

```
01626     heap[0][1] = 0;
01627     heap[0][2] = -1;
01628     WCOPY(&heap[0][3], &a[1], a[1]);
01629     heap[0][3] = a[a[1]];
01630 
01631     int qi=1, ri=1;
01632 
01633     int s = nb;
01634     WORD *t = (WORD *) NumberMalloc("poly::div_heap-c");
01635 
01636     // insert contains element that still have to be inserted to the heap
01637     // (exists to avoid code duplication).
01638     vector<pair<int,int> > insert;
01639 
01640     while (insert.size()>0 || nheap>0) {
01641 
01642         q.check_memory(qi);
01643         r.check_memory(ri);
01644 
01645         // collect a term t for the quotient/remainder
01646         t[0] = -1;
01647 
01648         do {
01649 
01650             WORD *p = heap[nheap];
01651             bool this_insert;
01652 
01653             if (insert.empty()) {
01654                 // extract element from the heap and prepare adding new ones
01655                 this_insert = false;
01656 
01657                 pop_heap(BHEAD heap, nheap--);
01658                 p = heap[nheap];
01659 
01660                 if (t[0] == -1) {
01661                     WCOPY(t, p, (5+ABS(p[3])+AN.poly_num_vars));
01662                 }
01663                 else {
01664                     // if both polynomials are modulo p^1, use integer calculus
01665                     if (both_mod_small) {
01666                         t[4+AN.poly_num_vars] = ((LONG)t[4+AN.poly_num_vars]*t[3] +
      p[4+AN.poly_num_vars]*p[3]) % q.modp;
01667                         if (t[4+AN.poly_num_vars]==0)
01668                             t[3]=0;
01669                         else {
01670                             if (t[4+AN.poly_num_vars] > +q.modp/2) t[4+AN.poly_num_vars] -= q.modp;
01671                             if (t[4+AN.poly_num_vars] < -q.modp/2) t[4+AN.poly_num_vars] += q.modp;
01672                             t[3] = SGN(t[4+AN.poly_num_vars]);
01673                             t[4+AN.poly_num_vars] = ABS(t[4+AN.poly_num_vars]);
01674                         }
01675                     }
01676                     else {
01677                         // otherwise, use form long calculus
01678                         AddLong ((UWORD *)&p[4+AN.poly_num_vars],  p[3],
01679                                  (UWORD *)&t[4+AN.poly_num_vars],  t[3],
01680                                  (UWORD *)&t[4+AN.poly_num_vars], &t[3]);
01681                         if (q.modp!=0) TakeNormalModulus((UWORD *)&t[4+AN.poly_num_vars], &t[3],
01682                                                                                          modq, nmodq,
      NOUNPACK);
01683                     }
01684                 }
01685             }
01686             else {
01687                 // prepare adding an element of insert to the heap
01688                 this_insert = true;
01689 
01690                 p[0] = insert.back().first;
01691                 p[1] = insert.back().second;
01692                 insert.pop_back();
01693             }
01694 
01695             // add elements to the heap
01696             while (true) {
01697                 // prepare the element
01698                 if (p[1]==0) {
01699                     p[0] += a[p[0]];
01700                     if (p[0]==a[0]) break;
01701                     WCOPY(&p[3], &a[p[0]], a[p[0]]);
01702                     p[3] = p[2+p[3]];
01703                 }
01704                 else {
01705                     if (!this_insert)
01706                         p[1] += q[p[1]];
01707                     this_insert = false;
01708 
01709                     if (p[1]==qi) { s++; break; }
01710 
```

```
01711                     for (int i=0; i<AN.poly_num_vars; i++)
01712                         p[4+i] = b[p[0]+1+i] + q[p[1]+1+i];
01713
01714                     // if both polynomials are modulo p^1, use integer calculus
01715                     if (both_mod_small) {
01716                         p[4+AN.poly_num_vars] = ((LONG)b[p[0]+1+AN.poly_num_vars]*b[p[0]+b[p[0]]-1]*
01717
      q[p[1]+1+AN.poly_num_vars]*q[p[1]+q[p[1]]-1]) % q.modp;
01718                         if (p[4+AN.poly_num_vars]==0)
01719                             p[3]=0;
01720                         else {
01721                             if (p[4+AN.poly_num_vars] > +q.modp/2) p[4+AN.poly_num_vars] -= q.modp;
01722                             if (p[4+AN.poly_num_vars] < -q.modp/2) p[4+AN.poly_num_vars] += q.modp;
01723                             p[3] = SGN(p[4+AN.poly_num_vars]);
01724                             p[4+AN.poly_num_vars] = ABS(p[4+AN.poly_num_vars]);
01725                         }
01726                     }
01727                     else {
01728                         // otherwise, use form long calculus
01729                         MulLong((UWORD *)&b[p[0]+1+AN.poly_num_vars], b[p[0]+b[p[0]]-1],
01730                                         (UWORD *)&q[p[1]+1+AN.poly_num_vars], q[p[1]+q[p[1]]-1],
01731                                         (UWORD *)&p[4+AN.poly_num_vars], &p[3]);
01732                         if (q.modp!=0) TakeNormalModulus((UWORD *)&p[4+AN.poly_num_vars], &p[3],
01733                                                                                                 modq, nmodq,
      NOUNPACK);
01734                     }
01735
01736                     p[3] *= -1;
01737                 }
01738
01739                 // no hashing
01740                 p[2] = -1;
01741
01742                 // add it to a heap element
01743                 swap (heap[nheap],p);
01744                 push_heap(BHEAD heap, ++nheap);
01745                 break;
01746             }
01747         }
01748         while (t[0]==-1 || (nheap>0 && monomial_compare(BHEAD heap[0]+3, t+3)==0));
01749
01750         if (t[3] == 0) continue;
01751
01752         // check divisibility
01753         bool div = true;
01754         for (int i=0; i<AN.poly_num_vars; i++)
01755             if (t[4+i] < b[2+i]) div=false;
01756
01757         if (!div) {
01758             // not divisible, so append it to the remainder
01759             if (only_divides) { r=poly(BHEAD 1); ri=r[0]; break; }
01760             t[4 + AN.poly_num_vars + ABS(t[3])] = t[3];
01761             t[3] = 2 + AN.poly_num_vars + ABS(t[3]);
01762             r.termscopy(&t[3], ri, t[3]);
01763             ri += t[3];
01764         }
01765         else {
01766             // divisible, so divide coefficient as well
01767             WORD nq, nr;
01768
01769             if (q.modp==0) {
01770                 DivLong((UWORD *)&t[4+AN.poly_num_vars], t[3],
01771                                 (UWORD *)&b[2+AN.poly_num_vars], b[b[1]],
01772                                 (UWORD *)&q[qi+1+AN.poly_num_vars], &nq,
01773                                 (UWORD *)&r[ri+1+AN.poly_num_vars], &nr);
01774                 if(check_div && nr != 0)
01775                 {
01776                     // non-zero remainder from coefficient division => result not expressible in
      integers
01777                     div_fail = true;
01778                     break;
01779                 }
01780             }
01781             else {
01782                 // if both polynomials are modulo p^1, use integer calculus
01783                 if (both_mod_small) {
01784                     q[qi+1+AN.poly_num_vars] = ((LONG)t[4+AN.poly_num_vars]*t[3]*ltbinv[0]*nltbinv) %
      q.modp;
01785                     if (q[qi+1+AN.poly_num_vars]==0)
01786                         nq=0;
01787                     else {
01788                         if (q[qi+1+AN.poly_num_vars] > +q.modp/2) q[qi+1+AN.poly_num_vars] -= q.modp;
01789                         if (q[qi+1+AN.poly_num_vars] < -q.modp/2) q[qi+1+AN.poly_num_vars] += q.modp;
01790                         nq = SGN(q[qi+1+AN.poly_num_vars]);
01791                         q[qi+1+AN.poly_num_vars] = ABS(q[qi+1+AN.poly_num_vars]);
01792                     }
01793                 }
```

```
01794                 else {
01795                     // otherwise, use form long calculus
01796                     MulLong((UWORD *)&t[4+AN.poly_num_vars], t[3], ltbinv, nltbinv, (UWORD
      *)&q[qi+1+AN.poly_num_vars], &nq);
01797                     TakeNormalModulus((UWORD *)&q[qi+1+AN.poly_num_vars], &nq, modq, nmodq, NOUNPACK);
01798                 }
01799
01800                 nr=0;
01801             }
01802
01803             // add terms to quotient and remainder
01804             if (nq != 0) {
01805                 int bi = 1;
01806                 for (int j=1; j<s; j++) {
01807                     bi += b[bi];
01808                     insert.push_back(make_pair(bi,qi));
01809                 }
01810                 s=1;
01811
01812                 q[qi] = 2+AN.poly_num_vars+ABS(nq);
01813                 for (int i=0; i<AN.poly_num_vars; i++)
01814                     q[qi+1+i] = t[4+i] - b[2+i];
01815                 qi += q[qi];
01816                 q[qi-1] = nq;
01817             }
01818
01819             if (nr != 0) {
01820                 r[ri] = 2+AN.poly_num_vars+ABS(nr);
01821                 for (int i=0; i<AN.poly_num_vars; i++)
01822                     r[ri+1+i] = t[4+i];
01823                 ri += r[ri];
01824                 r[ri-1] = nr;
01825             }
01826         }
01827     }
01828
01829     q[0] = qi;
01830     r[0] = ri;
01831
01832     for (int i=0; i<nb; i++)
01833         NumberFree(heap[i],"poly::div_heap-b");
01834
01835     NumberFree(t,"poly::div_heap-c");
01836
01837     if (q.modp!=0||ltbinv!=NULL) NumberFree(ltbinv,"poly::div_heap-a");
01838     AT.pWorkPointer = oldpWorkPointer;
01839 }
01840
01841 /*
01842     #] divmod_heap :
01843     #[ divmod :
01844 */
01845
01856 void poly::divmod (const poly &a, const poly &b, poly &q, poly &r, bool only_divides) {
01857
01858     q.modp = r.modp = a.modp;
01859     q.modn = r.modn = a.modn;
01860
01861     if (a.is_zero()) {
01862         q[0]=1;
01863         r[0]=1;
01864         return;
01865     }
01866     if (b.is_zero()) {
01867         MLOCK(ErrorMessageLock);
01868         MesPrint ((char*)"ERROR: division by zero polynomial");
01869         MUNLOCK(ErrorMessageLock);
01870         Terminate(1);
01871     }
01872     if (b.is_one()) {
01873         q.check_memory(a[0]);
01874         q.termscopy(a.terms,0,a[0]);
01875         r[0]=1;
01876         return;
01877     }
01878
01879     if (b.is_monomial()) {
01880         divmod_one_term(a,b,q,r,only_divides);
01881         return;
01882     }
01883
01884     int vara = a.is_dense_univariate();
01885     int varb = b.is_dense_univariate();
01886
01887     if (vara!=-2 && varb!=-2 && (vara==-1 || varb==-1 || vara==varb)) {
01888         divmod_univar(a,b,q,r,MaX(vara,varb),only_divides);
01889     }
```

```
01890     else {
01891         bool div_fail;
01892         divmod_heap(a,b,q,r,only_divides,false,div_fail);
01893     }
01894 }
01895
01896 /*
01897     #] divmod :
01898     #[ divides :
01899 */
01900
01901 // returns whether a exactly divides b
01902 bool poly::divides (const poly &a, const poly &b) {
01903     POLY_GETIDENTITY(a);
01904     poly q(BHEAD 0), r(BHEAD 0);
01905     divmod(b,a,q,r,true);
01906     return r.is_zero();
01907 }
01908
01909 /*
01910     #] divides :
01911     #[ div :
01912 */
01913
01914 // the quotient of two polynomials
01915 void poly::div (const poly &a, const poly &b, poly &c) {
01916     POLY_GETIDENTITY(a);
01917     poly d(BHEAD 0);
01918     divmod(a,b,c,d,false);
01919 }
01920
01921 /*
01922     #] div :
01923     #[ mod :
01924 */
01925
01926 // the remainder of division of two polynomials
01927 void poly::mod (const poly &a, const poly &b, poly &c) {
01928     POLY_GETIDENTITY(a);
01929     poly d(BHEAD 0);
01930     divmod(a,b,d,c,false);
01931 }
01932
01933 /*
01934     #] mod :
01935     #[ copy operator :
01936 */
01937
01938 // copy operator
01939 poly & poly::operator= (const poly &a) {
01940 #ifdef POLY_MOVE_DEBUG
01941     std::cout « "poly copy assign" « std::endl;
01942 #endif
01943     if (&a != this) {
01944         modp = a.modp;
01945         modn = a.modn;
01946         check_memory(a[0]);
01947         termscopy(a.terms,0,a[0]);
01948     }
01949
01950     return *this;
01951 }
01952
01953 /*
01954     #] copy operator :
01955     #[ operator overloads :
01956 */
01957
01958 // binary operators for polynomial arithmetic
01959 const poly poly::operator+ (const poly &a) const { POLY_GETIDENTITY(a); poly b(BHEAD 0);
      add(*this,a,b); return b; }
01960 const poly poly::operator- (const poly &a) const { POLY_GETIDENTITY(a); poly b(BHEAD 0);
      sub(*this,a,b); return b; }
01961 const poly poly::operator* (const poly &a) const { POLY_GETIDENTITY(a); poly b(BHEAD 0);
      mul(*this,a,b); return b; }
01962 const poly poly::operator/ (const poly &a) const { POLY_GETIDENTITY(a); poly b(BHEAD 0);
      div(*this,a,b); return b; }
01963 const poly poly::operator% (const poly &a) const { POLY_GETIDENTITY(a); poly b(BHEAD 0);
      mod(*this,a,b); return b; }
01964
01965 // assignment operators for polynomial arithmetic
01966 poly& poly::operator+= (const poly &a) { return *this = *this + a; }
01967 poly& poly::operator-= (const poly &a) { return *this = *this - a; }
01968 poly& poly::operator*= (const poly &a) { return *this = *this * a; }
01969 poly& poly::operator/= (const poly &a) { return *this = *this / a; }
01970 poly& poly::operator%= (const poly &a) { return *this = *this % a; }
01971
```

```
01972 // comparison operators
01973 bool poly::operator== (const poly &a) const {
01974     for (int i=0; i<terms[0]; i++)
01975         if (terms[i] != a[i]) return 0;
01976     return 1;
01977 }
01978
01979 bool poly::operator!= (const poly &a) const {   return !(*this == a); }
01980
01981 /*
01982     #] operator overloads :
01983     #[ num_terms :
01984 */
01985
01986 int poly::number_of_terms () const {
01987
01988     int res=0;
01989     for (int i=1; i<terms[0]; i+=terms[i])
01990         res++;
01991     return res;
01992 }
01993
01994 /*
01995     #] num_terms :
01996     #[ first_variable :
01997 */
01998
01999 // returns the lexicographically first variable of a polynomial
02000 int poly::first_variable () const {
02001
02002     POLY_GETIDENTITY(*this);
02003
02004     int var = AN.poly_num_vars;
02005     for (int j=0; j<var; j++)
02006         if (terms[2+j]>0) return j;
02007     return var;
02008 }
02009
02010 /*
02011     #] first_variable :
02012     #[ all_variables :
02013 */
02014
02015 // returns a list of all variables of a polynomial
02016 const vector<int> poly::all_variables () const {
02017
02018     POLY_GETIDENTITY(*this);
02019
02020     vector<bool> used(AN.poly_num_vars, false);
02021
02022     for (int i=1; i<terms[0]; i+=terms[i])
02023         for (int j=0; j<AN.poly_num_vars; j++)
02024             if (terms[i+1+j]>0) used[j] = true;
02025
02026     vector<int> vars;
02027     for (int i=0; i<AN.poly_num_vars; i++)
02028         if (used[i]) vars.push_back(i);
02029
02030     return vars;
02031 }
02032
02033 /*
02034     #] all_variables :
02035     #[ sign :
02036 */
02037
02038 // returns the sign of the leading coefficient
02039 int poly::sign () const {
02040     if (terms[0]==1) return 0;
02041     return terms[terms[1]] > 0 ? 1 : -1;
02042 }
02043
02044 /*
02045     #] sign :
02046     #[ degree :
02047 */
02048
02049 // returns the degree of x of a polynomial (deg=-1 iff a=0)
02050 int poly::degree (int x) const {
02051     int deg = -1;
02052     for (int i=1; i<terms[0]; i+=terms[i])
02053         deg = MaX(deg, terms[i+1+x]);
02054     return deg;
02055 }
02056
02057 /*
02058     #] degree :
```

```
02059     #[ total_degree :
02060 */
02061
02062 // returns the total degree of a polynomial (deg=-1 iff a=0)
02063 int poly::total_degree () const {
02064
02065     POLY_GETIDENTITY(*this);
02066
02067     int tot_deg = -1;
02068     for (int i=1; i<terms[0]; i+=terms[i]) {
02069         int deg=0;
02070         for (int j=0; j<AN.poly_num_vars; j++)
02071             deg += terms[i+1+j];
02072         tot_deg = MaX(deg, tot_deg);
02073     }
02074     return tot_deg;
02075 }
02076
02077 /*
02078     #] total_degree :
02079     #[ integer_lcoeff :
02080 */
02081
02082 // returns the integer coefficient of the leading monomial
02083 const poly poly::integer_lcoeff () const {
02084
02085     POLY_GETIDENTITY(*this);
02086
02087     poly res(BHEAD 0);
02088     res.modp = modp;
02089     res.modn = modn;
02090
02091     res.termscopy(&terms[1],1,terms[1]);
02092     res[0] = res[1] + 1; // length
02093     for (int i=0; i<AN.poly_num_vars; i++)
02094         res[2+i] = 0; // powers
02095
02096     return res;
02097 }
02098
02099 /*
02100     #] integer_lcoeff :
02101     #[ coefficient :
02102 */
02103
02104 // returns the polynomial coefficient of x^n
02105 const poly poly::coefficient (int x, int n) const {
02106
02107     POLY_GETIDENTITY(*this);
02108
02109     poly res(BHEAD 0);
02110     res.modp = modp;
02111     res.modn = modn;
02112     res[0] = 1;
02113
02114     for (int i=1; i<terms[0]; i+=terms[i])
02115         if (terms[i+1+x] == n) {
02116             res.check_memory(res[0]+terms[i]);
02117             res.termscopy(&terms[i], res[0], terms[i]);
02118             res[res[0]+1+x] -= n;  // power of x
02119             res[0] += res[res[0]]; // length
02120         }
02121
02122     return res;
02123 }
02124
02125 /*
02126     #] coefficient :
02127     #[ lcoeff_multivar :
02128 */
02129
02130 // returns the leading coefficient with the polynomial viewed as a
02131 // polynomial in x, so that the result is a polynomial in the
02132 // remaining variables
02133 const poly poly::lcoeff_univar (int x) const {
02134     return coefficient(x, degree(x));
02135 }
02136
02137 // returns the leading coefficient with the polynomial viewed as a
02138 // polynomial with coefficients in ZZ[x], so that the result
02139 // polynomial is in ZZ[x] too
02140 const poly poly::lcoeff_multivar (int x) const {
02141
02142     POLY_GETIDENTITY(*this);
02143
02144     poly res(BHEAD 0,modp,modn);
02145
```

```
02146     for (int i=1; i<terms[0]; i+=terms[i]) {
02147         bool same_powers = true;
02148         for (int j=0; j<AN.poly_num_vars; j++)
02149             if (j!=x && terms[i+1+j]!=terms[2+j]) {
02150                 same_powers = false;
02151                 break;
02152             }
02153         if (!same_powers) break;
02154 
02155         res.check_memory(res[0]+terms[i]);
02156         res.termscopy(&terms[i],res[0],terms[i]);
02157         for (int k=0; k<AN.poly_num_vars; k++)
02158             if (k!=x) res[res[0]+1+k]=0;
02159 
02160         res[0] += terms[i];
02161     }
02162 
02163     return res;
02164 }
02165 
02166 /*
02167     #] lcoeff_multivar :
02168     #[ derivative :
02169 */
02170 
02171 // returns the derivative with respect to x
02172 const poly poly::derivative (int x) const {
02173 
02174     POLY_GETIDENTITY(*this);
02175 
02176     poly b(BHEAD 0);
02177     int bi=1;
02178 
02179     for (int i=1; i<terms[0]; i+=terms[i]) {
02180 
02181         int power = terms[i+1+x];
02182 
02183         if (power > 0) {
02184             b.check_memory(bi);
02185             b.termscopy(&terms[i], bi, terms[i]);
02186             b[bi+1+x]--;
02187 
02188             WORD nb = b[bi+b[bi]-1];
02189             Product((UWORD *)&b[bi+1+AN.poly_num_vars], &nb, power);
02190 
02191             b[bi] = 2 + AN.poly_num_vars + ABS(nb);
02192             b[bi+b[bi]-1] = nb;
02193 
02194             bi += b[bi];
02195         }
02196     }
02197 
02198     b[0] = bi;
02199     b.setmod(modp, modn);
02200     return b;
02201 }
02202 
02203 /*
02204     #] derivative :
02205     #[ is_zero :
02206 */
02207 
02208 // returns whether the polynomial is zero
02209 bool poly::is_zero () const {
02210     return terms[0] == 1;
02211 }
02212 
02213 /*
02214     #] is_zero :
02215     #[ is_one :
02216 */
02217 
02218 // returns whether the polynomial is one
02219 bool poly::is_one () const {
02220 
02221     POLY_GETIDENTITY(*this);
02222 
02223     if (terms[0] != 4+AN.poly_num_vars) return false;
02224     if (terms[1] != 3+AN.poly_num_vars) return false;
02225     for (int i=0; i<AN.poly_num_vars; i++)
02226         if (terms[2+i] != 0) return false;
02227     if (terms[2+AN.poly_num_vars]!=1) return false;
02228     if (terms[3+AN.poly_num_vars]!=1) return false;
02229 
02230     return true;
02231 }
02232 
```

```
02233 /*
02234     #] is_one :
02235     #[ is_integer :
02236 */
02237
02238 // returns whether the polynomial is an integer
02239 bool poly::is_integer () const {
02240
02241     POLY_GETIDENTITY(*this);
02242
02243     if (terms[0] == 1) return true;
02244     if (terms[0] != terms[1]+1) return false;
02245
02246     for (int j=0; j<AN.poly_num_vars; j++)
02247         if (terms[2+j] != 0)
02248             return false;
02249
02250     return true;
02251 }
02252
02253 /*
02254     #] is_integer :
02255     #[ is_monomial :
02256 */
02257
02258 // returns whether the polynomial consist of one term
02259 bool poly::is_monomial () const {
02260     return terms[0]>1 && terms[0]==terms[1]+1;
02261 }
02262
02263 /*
02264     #] is_monomial :
02265     #[ is_dense_univariate :
02266 */
02267
02284 int poly::is_dense_univariate () const {
02285
02286     POLY_GETIDENTITY(*this);
02287
02288     int num_terms=0, res=-1;
02289
02290     // test univariate
02291     for (int i=1; i<terms[0]; i+=terms[i]) {
02292         for (int j=0; j<AN.poly_num_vars; j++)
02293             if (terms[i+1+j] > 0) {
02294                 if (res == -1) res = j;
02295                 if (res != j) return -2;
02296             }
02297
02298         num_terms++;
02299     }
02300
02301     // constant polynomial
02302     if (res == -1) return -1;
02303
02304     // test density
02305     int deg = terms[2+res];
02306     if (2*num_terms < deg+1) return -2;
02307
02308     return res;
02309 }
02310
02311 /*
02312     #] is_dense_univariate :
02313     #[ simple_poly (small) :
02314 */
02315
02316 // returns the polynomial (x-a)^b mod p^n with a small
02317 const poly poly::simple_poly (PHEAD int x, int a, int b, int p, int n) {
02318
02319     poly tmp(BHEAD 0,p,n);
02320
02321     int idx=1;
02322     tmp[idx++] = 3 + AN.poly_num_vars;                      // length
02323     for (int i=0; i<AN.poly_num_vars; i++)
02324         tmp[idx++] = i==x ? 1 : 0;                            // powers
02325     tmp[idx++] = 1;                                         // coefficient
02326     tmp[idx++] = 1;                                         // length coefficient
02327
02328     if (a != 0) {
02329         tmp[idx++] = 3 + AN.poly_num_vars;                    // length
02330         for (int i=0; i<AN.poly_num_vars; i++) tmp[idx++] = 0;  // powers
02331         tmp[idx++] = ABS(a);                                  // coefficient
02332         tmp[idx++] = -SGN(a);                                 // length coefficient
02333     }
02334
02335     tmp[0] = idx;                                           // length
```

```
02336
02337     if (b == 1) return tmp;
02338
02339     poly res(BHEAD 1,p,n);
02340
02341     while (b!=0) {
02342         if (b&1) res*=tmp;
02343         tmp*=tmp;
02344         b»=1;
02345     }
02346
02347     return res;
02348 }
02349
02350 /*
02351     #] simple_poly (small) :
02352     #[ simple_poly (large) :
02353 */
02354
02355 // returns the polynomial (x-a)^b mod p^n with a large
02356 const poly poly::simple_poly (PHEAD int x, const poly &a, int b, int p, int n) {
02357
02358     poly res(BHEAD 1,p,n);
02359     poly tmp(BHEAD 0,p,n);
02360
02361     int idx=1;
02362
02363     tmp[idx++] = 3 + AN.poly_num_vars;                                  // length
02364     for (int i=0; i<AN.poly_num_vars; i++)
02365         tmp[idx++] = i==x ? 1 : 0;                                        // powers
02366     tmp[idx++] = 1;                                                     // coefficient
02367     tmp[idx++] = 1;                                                     // length coefficient
02368
02369     if (!a.is_zero()) {
02370         tmp[idx++] = 2 + AN.poly_num_vars + ABS(a[a[0]-1]); // length
02371         for (int i=0; i<AN.poly_num_vars; i++) tmp[idx++] = 0;          // powers
02372         for (int i=0; i<ABS(a[a[0]-1]); i++)
02373             tmp[idx++] = a[2 + AN.poly_num_vars + i];                     // coefficient
02374         tmp[idx++] = -a[a[0]-1];                                        // length coefficient
02375     }
02376
02377     tmp[0] = idx;                                                       // length
02378
02379     while (b!=0) {
02380         if (b&1) res*=tmp;
02381         tmp*=tmp;
02382         b»=1;
02383     }
02384
02385     return res;
02386 }
02387
02388 /*
02389     #] simple_poly (large) :
02390     #[ get_variables :
02391 */
02392
02393 // gets all variables in the expressions and stores them in AN.poly_vars
02394 // (it is assumed that AN.poly_vars=NULL)
02395 void poly::get_variables (PHEAD const vector<WORD *> &es, bool with_arghead, bool sort_vars) {
02396     assert(AN.poly_vars == NULL);
02397
02398     AN.poly_num_vars = 0;
02399
02400     vector<int> vars;
02401     vector<int> degrees;
02402     map<int,int> var_to_idx;
02403
02404     // extract all variables
02405     for (int ei=0; ei<(int)es.size(); ei++) {
02406         const WORD *e = es[ei];
02407
02408         // fast notation
02409         if (*e == -SNUMBER) {
02410         }
02411         else if (*e == -SYMBOL) {
02412             if (!var_to_idx.count(e[1])) {
02413                 vars.push_back(e[1]);
02414                 var_to_idx[e[1]] = AN.poly_num_vars++;
02415                 degrees.push_back(1);
02416             }
02417         }
02418         // JD: Here we need to check for non-symbol/number terms in fast notation.
02419         else if (*e < 0) {
02420             MLOCK(ErrorMessageLock);
02421             MesPrint ((char*)"ERROR: polynomials and polyratfuns must contain symbols only");
02422             MUNLOCK(ErrorMessageLock);
```

```
02423             Terminate(1);
02424         }
02425         else {
02426             for (int i=with_arghead ? ARGHEAD : 0; with_arghead ? i<e[0] : e[i]!=0; i+=e[i]) {
02427                 if (i+1<i+e[i]-ABS(e[i+e[i]-1]) && e[i+1]!=SYMBOL) {
02428                     MLOCK(ErrorMessageLock);
02429                     MesPrint ((char*)"ERROR: polynomials and polyratfuns must contain symbols only");
02430                     MUNLOCK(ErrorMessageLock);
02431                     Terminate(1);
02432                 }
02433 
02434                 for (int j=i+3; j<i+e[i]-ABS(e[i+e[i]-1]); j+=2) {
02435                     if (!var_to_idx.count(e[j])) {
02436                         vars.push_back(e[j]);
02437                         var_to_idx[e[j]] = AN.poly_num_vars++;
02438                         degrees.push_back(e[j+1]);
02439                     }
02440                     else {
02441                         degrees[var_to_idx[e[j]]] = MaX(degrees[var_to_idx[e[j]]], e[j+1]);
02442                     }
02443                 }
02444             }
02445         }
02446     }
02447 
02448     // make sure an eventual FACTORSYMBOL appear as last
02449     if (var_to_idx.count(FACTORSYMBOL)) {
02450         int i = var_to_idx[FACTORSYMBOL];
02451         while (i+1<AN.poly_num_vars) {
02452             swap(vars[i], vars[i+1]);
02453             swap(degrees[i], degrees[i+1]);
02454             i++;
02455         }
02456         degrees[i] = -1; // makes sure it stays last
02457     }
02458 
02459     // AN.poly_vars will be deleted in calling functions from polywrap.c
02460     // For that we use the function poly_free_poly_vars
02461     // We add one to the allocation to avoid copying in poly_divmod
02462 
02463     if ( (AN.poly_num_vars+1)*sizeof(WORD) > (size_t)(AM.MaxTer) ) {
02464         // This can happen only in expressions with excessively many variables.
02465         AN.poly_vars = (WORD *)Malloc1((AN.poly_num_vars+1)*sizeof(WORD), "AN.poly_vars");
02466         AN.poly_vars_type = 1;
02467     }
02468     else {
02469         AN.poly_vars = TermMalloc("AN.poly_vars");
02470         AN.poly_vars_type = 0;
02471     }
02472 
02473     for (int i=0; i<AN.poly_num_vars; i++)
02474         AN.poly_vars[i] = vars[i];
02475 
02476     if (sort_vars) {
02477         // bubble sort variables in decreasing order of degree
02478         // (this seems better for factorization)
02479         for (int i=0; i<AN.poly_num_vars; i++)
02480             for (int j=0; j+1<AN.poly_num_vars; j++)
02481                 if (degrees[j] < degrees[j+1]) {
02482                     swap(degrees[j], degrees[j+1]);
02483                     swap(AN.poly_vars[j], AN.poly_vars[j+1]);
02484                 }
02485     }
02486     else {
02487         // sort lexicographically
02488         sort(AN.poly_vars, AN.poly_vars+AN.poly_num_vars - var_to_idx.count(FACTORSYMBOL));
02489     }
02490 }
02491 
02492 /*
02493     #] get_variables :
02494     #[ argument_to_poly :
02495 */
02496 
02497 // converts a form expression to a polynomial class "poly"
02498 const poly poly::argument_to_poly (PHEAD WORD *e, bool with_arghead, bool sort_univar, poly *denpoly)
    {
02499 
02500     map<int,int> var_to_idx;
02501     for (int i=0; i<AN.poly_num_vars; i++)
02502         var_to_idx[AN.poly_vars[i]] = i;
02503 
02504     poly res(BHEAD 0);
02505 
02506     // fast notation
02507     if (*e == -SNUMBER) {
02508         if (denpoly!=NULL) *denpoly = poly(BHEAD 1);
```

```
02509
02510         if (e[1] == 0) {
02511             res[0] = 1;
02512             return res;
02513         }
02514         else {
02515             res[0] = 4 + AN.poly_num_vars;
02516             res[1] = 3 + AN.poly_num_vars;
02517             for (int i=0; i<AN.poly_num_vars; i++)
02518                 res[2+i] = 0;
02519             res[2+AN.poly_num_vars] = ABS(e[1]);
02520             res[3+AN.poly_num_vars] = SGN(e[1]);
02521
02522             return res;
02523         }
02524     }
02525
02526     if (*e == -SYMBOL) {
02527         if (denpoly!=NULL) *denpoly = poly(BHEAD 1);
02528
02529         res[0] = 4 + AN.poly_num_vars;
02530         res[1] = 3 + AN.poly_num_vars;
02531         for (int i=0; i<AN.poly_num_vars; i++)
02532             res[2+i] = 0;
02533         res[2+var_to_idx.find(e[1])->second] = 1;
02534         res[2+AN.poly_num_vars] = 1;
02535         res[3+AN.poly_num_vars] = 1;
02536         return res;
02537     }
02538
02539     // find LCM of denominators of all terms
02540     WORD nden=1, npro=0, ngcd=0, ndum=0;
02541     UWORD *den = NumberMalloc("poly::argument_to_poly");
02542     UWORD *pro = NumberMalloc("poly::argument_to_poly");
02543     UWORD *gcd = NumberMalloc("poly::argument_to_poly");
02544     UWORD *dum = NumberMalloc("poly::argument_to_poly");
02545     den[0]=1;
02546
02547     for (int i=with_arghead ? ARGHEAD : 0; with_arghead ? i<e[0] : e[i]!=0; i+=e[i]) {
02548         int ncoe = ABS(e[i+e[i]-1]/2);
02549         UWORD *coe = (UWORD *)&e[i+e[i]-ncoe-1];
02550         while (ncoe>0 && coe[ncoe-1]==0) ncoe--;
02551         MulLong(den,nden,coe,ncoe,pro,&npro);
02552         GcdLong(BHEAD den,nden,coe,ncoe,gcd,&ngcd);
02553         DivLong(pro,npro,gcd,ngcd,den,&nden,dum,&ndum);
02554     }
02555
02556     if (denpoly!=NULL) *denpoly = poly(BHEAD den, nden);
02557
02558     int ri=1;
02559
02560     // ordinary notation
02561     for (int i=with_arghead ? ARGHEAD : 0; with_arghead ? i<e[0] : e[i]!=0; i+=e[i]) {
02562         res.check_memory(ri);
02563         WORD nc = e[i+e[i]-1];                                          // length coefficient
02564         for (int j=0; j<AN.poly_num_vars; j++)
02565             res[ri+1+j]=0;                                              // powers=0
02566         WCOPY(dum, &e[i+e[i]-ABS(nc)], ABS(nc));                        // coefficient to dummy
02567         nc /= 2;                                                        // remove denominator
02568         Mully(BHEAD dum, &nc, den, nden);                               // multiply with
    overall den
02569         res.termscopy((WORD *)dum, ri+1+AN.poly_num_vars, ABS(nc));     // coefficient to res
02570         res[ri] = ABS(nc) + AN.poly_num_vars + 2;                       // length
02571         res[ri+res[ri]-1] = nc;                                         // length coefficient
02572         for (int j=i+3; j<i+e[i]-ABS(e[i+e[i]-1]); j+=2)
02573             res[ri+1+var_to_idx.find(e[j])->second] = e[j+1];           // powers
02574         ri += res[ri];                                                  // length
02575     }
02576
02577     res[0] = ri;
02578
02579     // JD: For univariate cases, check whether the ordering is correct or not.
02580     // There are various ways to arrive here, but with an incorrect ordering, such
02581     // as interactions with collect, moving a polyratfun out of another function
02582     // argument, etc.
02583     // If the ordering is not correct, make sure we normalize. In multivariate cases,
02584     // always normalize as before.
02585     if (sort_univar == false && AN.poly_num_vars == 1) {
02586         if (res.number_of_terms() >= 2) {
02587             if(res[2] < res.last_monomial()[2]) {
02588                 sort_univar = true;
02589             }
02590         }
02591     }
02592
02593     if (sort_univar || AN.poly_num_vars>1) {
02594         res.normalize();
```

```
02595     }
02596
02597     NumberFree(den,"poly::argument_to_poly");
02598     NumberFree(pro,"poly::argument_to_poly");
02599     NumberFree(gcd,"poly::argument_to_poly");
02600     NumberFree(dum,"poly::argument_to_poly");
02601
02602     return res;
02603 }
02604
02605 /*
02606     #] argument_to_poly :
02607     #[ poly_to_argument :
02608 */
02609
02610 // converts a polynomial class "poly" to a form expression
02611 void poly::poly_to_argument (const poly &a, WORD *res, bool with_arghead) {
02612
02613     POLY_GETIDENTITY(a);
02614
02615     // special case: a=0
02616     if (a[0]==1) {
02617         if (with_arghead) {
02618             res[0] = -SNUMBER;
02619             res[1] = 0;
02620         }
02621         else {
02622             res[0] = 0;
02623         }
02624         return;
02625     }
02626
02627     if (with_arghead) {
02628         res[1] = AN.poly_num_vars>1 ? DIRTYFLAG : 0; // dirty flag
02629         for (int i=2; i<ARGHEAD; i++)
02630             res[i] = 0;                                 // remainder of arghead
02631     }
02632
02633     int L = with_arghead ? ARGHEAD : 0;
02634
02635     for (int i=1; i!=a[0]; i+=a[i]) {
02636
02637         res[L]=1; // length
02638
02639         bool first=true;
02640
02641         for (int j=0; j<AN.poly_num_vars; j++)
02642             if (a[i+1+j] > 0) {
02643                 if (first) {
02644                     first=false;
02645                     res[L+1] = 1; // symbols
02646                     res[L+2] = 2; // length
02647                 }
02648                 res[L+1+res[L+2]++] = AN.poly_vars[j]; // symbol
02649                 res[L+1+res[L+2]++] = a[i+1+j];  // power
02650             }
02651
02652         if (!first) res[L] += res[L+2]; // fix length
02653
02654         WORD nc = a[i+a[i]-1];
02655         WCOPY(&res[L+res[L]], &a[i+a[i]-1-ABS(nc)], ABS(nc)); // numerator
02656
02657         res[L] += ABS(nc);                                  // fix length
02658         memset(&res[L+res[L]], 0, ABS(nc)*sizeof(WORD)); // denominator one
02659         res[L+res[L]] = 1;                                  // denominator one
02660         res[L] += ABS(nc);                                  // fix length
02661         res[L+res[L]] = SGN(nc) * (2*ABS(nc)+1);          // length of coefficient
02662         res[L]++;                                           // fix length
02663         L += res[L];                                        // fix length
02664     }
02665
02666     if (with_arghead) {
02667         res[0] = L;
02668         // convert to fast notation if possible
02669         ToFast(res,res);
02670     }
02671     else {
02672         res[L] = 0;
02673     }
02674 }
02675
02676 /*
02677     #] poly_to_argument :
02678     #[ poly_to_argument_with_den :
02679 */
02680
02681 // converts a polynomial class "poly" divided by a number (nden, den) to a form expression
```

```
02682 // cf. poly::poly_to_argument()
02683 void poly::poly_to_argument_with_den (const poly &a, WORD nden, const UWORD *den, WORD *res, bool
      with_arghead) {
02684
02685     POLY_GETIDENTITY(a);
02686
02687     // special case: a=0
02688     if (a[0]==1) {
02689         if (with_arghead) {
02690             res[0] = -SNUMBER;
02691             res[1] = 0;
02692         }
02693         else {
02694             res[0] = 0;
02695         }
02696         return;
02697     }
02698
02699     if (with_arghead) {
02700         res[1] = AN.poly_num_vars>1 ? DIRTYFLAG : 0; // dirty flag
02701         for (int i=2; i<ARGHEAD; i++)
02702             res[i] = 0;                                  // remainder of arghead
02703     }
02704
02705     WORD nden1;
02706     UWORD *den1 = (UWORD *)NumberMalloc("poly_to_argument_with_den");
02707
02708     int L = with_arghead ? ARGHEAD : 0;
02709
02710     for (int i=1; i!=a[0]; i+=a[i]) {
02711
02712         res[L]=1; // length
02713
02714         bool first=true;
02715
02716         for (int j=0; j<AN.poly_num_vars; j++)
02717             if (a[i+1+j] > 0) {
02718                 if (first) {
02719                     first=false;
02720                     res[L+1] = 1; // symbols
02721                     res[L+2] = 2; // length
02722                 }
02723                 res[L+1+res[L+2]++] = AN.poly_vars[j]; // symbol
02724                 res[L+1+res[L+2]++] = a[i+1+j];  // power
02725             }
02726
02727         if (!first) res[L] += res[L+2]; // fix length
02728
02729         // numerator
02730         WORD nc = a[i+a[i]-1];
02731         WCOPY(&res[L+res[L]], &a[i+a[i]-1-ABS(nc)], ABS(nc));
02732
02733         // denominator
02734         nden1 = nden;
02735         WCOPY(den1, den, ABS(nden));
02736
02737         if (nden != 1 || den[0] != 1) {
02738             // remove gcd(num,den)
02739             Simplify(BHEAD (UWORD *)&res[L+res[L]], &nc, den1, &nden1);
02740         }
02741
02742         Pack((UWORD *)&res[L+res[L]], &nc, den1, nden1);  // format
02743         res[L] += 2*ABS(nc)+1;                            // fix length
02744         res[L+res[L]-1] = SGN(nc)*(2*ABS(nc)+1);          // length of coefficient
02745         L += res[L];                                      // fix length
02746     }
02747
02748     NumberFree(den1, "poly_to_argument_with_den");
02749
02750     if (with_arghead) {
02751         res[0] = L;
02752         // convert to fast notation if possible
02753         ToFast(res,res);
02754     }
02755     else {
02756         res[L] = 0;
02757     }
02758 }
02759
02760 /*
02761     #] poly_to_argument_with_den :
02762     #[ size_of_form_notation :
02763 */
02764
02765 // the size of the polynomial in form notation (without argheads and fast notation)
02766 int poly::size_of_form_notation () const {
02767
```

```
02768     POLY_GETIDENTITY(*this);
02769
02770     // special case: a=0
02771     if (terms[0]==1) return 0;
02772
02773     int len = 0;
02774
02775     for (int i=1; i!=terms[0]; i+=terms[i]) {
02776         len++;
02777         int npow = 0;
02778         for (int j=0; j<AN.poly_num_vars; j++)
02779             if (terms[i+1+j] > 0) npow++;
02780         if (npow > 0) len += 2*npow + 2;
02781         len += 2 * ABS(terms[i+terms[i]-1]) + 1;
02782     }
02783
02784     return len;
02785 }
02786
02787 /*
02788     #] size_of_form_notation :
02789     #[ size_of_form_notation_with_den :
02790 */
02791
02792 // the size of the polynomial divided by a number (its size is given by nden)
02793 // in form notation (without argheads and fast notation)
02794 // cf. poly::size_of_form_notation()
02795 int poly::size_of_form_notation_with_den (WORD nden) const {
02796
02797     POLY_GETIDENTITY(*this);
02798
02799     // special case: a=0
02800     if (terms[0]==1) return 0;
02801
02802     nden = ABS(nden);
02803     int len = 0;
02804
02805     for (int i=1; i!=terms[0]; i+=terms[i]) {
02806         len++;
02807         int npow = 0;
02808         for (int j=0; j<AN.poly_num_vars; j++)
02809             if (terms[i+1+j] > 0) npow++;
02810         if (npow > 0) len += 2*npow + 2;
02811         len += 2 * MaX(ABS(terms[i+terms[i]-1]), nden) + 1;
02812     }
02813
02814     return len;
02815 }
02816
02817 /*
02818     #] size_of_form_notation_with_den :
02819     #[ to_coefficient_list :
02820 */
02821
02822 // returns the coefficient list of a univariate polynomial
02823 const vector<WORD> poly::to_coefficient_list (const poly &a) {
02824
02825     POLY_GETIDENTITY(a);
02826
02827     if (a.is_zero()) return vector<WORD>();
02828
02829     int x = a.first_variable();
02830     if (x == AN.poly_num_vars) x=0;
02831
02832     vector<WORD> res(1+a[2+x],0);
02833
02834     for (int i=1; i<a[0]; i+=a[i])
02835         res[a[i+1+x]] = (a[i+a[i]-1] * a[i+a[i]-2]) % a.modp;
02836
02837     return res;
02838 }
02839
02840 /*
02841     #] to_coefficient_list :
02842     #[ coefficient_list_divmod :
02843 */
02844
02845 // divides two polynomials represented by coefficient lists
02846 const vector<WORD> poly::coefficient_list_divmod (const vector<WORD> &a, const vector<WORD> &b, WORD
      p, int divmod) {
02847
02848     int bsize = (int)b.size();
02849     while (b[bsize-1]==0) bsize--;
02850
02851     WORD inv;
02852     GetModInverses(b[bsize-1] + (b[bsize-1]<0?p:0), p, &inv, NULL);
02853
```

```
02854     vector<WORD> q(a.size(),0);
02855     vector<WORD> r(a);
02856 
02857     while ((int)r.size() >= bsize) {
02858         LONG mul = ((LONG)inv * r.back()) % p;
02859         int off = r.size()-bsize;
02860         q[off] = mul;
02861         for (int i=0; i<bsize; i++)
02862             r[off+i] = (r[off+i] - mul*b[i]) % p;
02863         while (r.size()>0 && r.back()==0)
02864             r.pop_back();
02865     }
02866 
02867     if (divmod==0) {
02868         while (q.size()>0 && q.back()==0)
02869             q.pop_back();
02870         return q;
02871     }
02872     else {
02873         while (r.size()>0 && r.back()==0)
02874             r.pop_back();
02875         return r;
02876     }
02877 }
02878 
02879 /*
02880     #] coefficient_list_divmod :
02881     #[ from_coefficient_list :
02882 */
02883 
02884 // converts a coefficient list to a "poly"
02885 const poly poly::from_coefficient_list (PHEAD const vector<WORD> &a, int x, WORD p) {
02886 
02887     poly res(BHEAD 0);
02888     int ri=1;
02889 
02890     for (int i=(int)a.size()-1; i>=0; i--)
02891         if (a[i] != 0) {
02892             res.check_memory(ri);
02893             res[ri] = AN.poly_num_vars+3;                // length
02894             for (int j=0; j<AN.poly_num_vars; j++)
02895                 res[ri+1+j]=0;                             // powers
02896             res[ri+1+x] = i;                             // power of x
02897             res[ri+1+AN.poly_num_vars] = ABS(a[i]);      // coefficient
02898             res[ri+1+AN.poly_num_vars+1] = SGN(a[i]);    // sign
02899             ri += res[ri];
02900         }
02901 
02902     res[0]=ri;                                       // total length
02903     res.setmod(p,1);
02904 
02905     return res;
02906 }
02907 
02908 /*
02909     #] from_coefficient_list :
02910 */
```

4.104 poly.h

```
00001 #pragma once
00002 /* #[ License : */
00003 /*
00004  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00005  *   When using this file you are requested to refer to the publication
00006  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00007  *   This is considered a matter of courtesy as the development was paid
00008  *   for by FOM the Dutch physics granting agency and we would like to
00009  *   be able to track its scientific use to convince FOM of its value
00010  *   for the community.
00011  *
00012  *   This file is part of FORM.
00013  *
00014  *   FORM is free software: you can redistribute it and/or modify it under the
00015  *   terms of the GNU General Public License as published by the Free Software
00016  *   Foundation, either version 3 of the License, or (at your option) any later
00017  *   version.
00018  *
00019  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00020  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00021  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00022  *   details.
00023  *
```

```
00024  *   You should have received a copy of the GNU General Public License along
00025  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00026  */
00027 /* #] License : */
00028
00029 extern "C" {
00030 #include "form3.h"
00031 }
00032
00033 #include <iostream>
00034 #include <string>
00035 #include <vector>
00036
00037 // for debugging
00038 // #define POLY_MOVE_DEBUG
00039
00040 // macros for tform
00041 #ifndef WITHPTHREADS
00042 #define POLY_GETIDENTITY(X)
00043 #define POLY_STOREIDENTITY
00044 #else
00045 #define POLY_GETIDENTITY(X) ALLPRIVATES *B = (X).Bpointer
00046 #define POLY_STOREIDENTITY Bpointer = B
00047 #endif
00048
00049 // maximum size of the hash table used for multiplication and division
00050
00051 const int POLY_MAX_HASH_SIZE = MiN(1«20, MAXPOSITIVE);
00052
00053 class poly {
00054
00055 public:
00056
00057     // variables
00058 #ifdef WITHPTHREADS
00059     ALLPRIVATES *Bpointer;
00060 #endif
00061
00062     WORD *terms;
00063     LONG size_of_terms;
00064     WORD modp,modn;
00065
00066     // constructors/destructor
00067     poly (PHEAD int, WORD=-1, WORD=1);
00068     poly (PHEAD const UWORD *, WORD, WORD=-1, WORD=1);
00069     poly (const poly &, WORD=-1, WORD=1);
00070     ~poly ();
00071
00072     // operators
00073     poly& operator+= (const poly&);
00074     poly& operator-= (const poly&);
00075     poly& operator*= (const poly&);
00076     poly& operator/= (const poly&);
00077     poly& operator%= (const poly&);
00078
00079     const poly operator+ (const poly&) const;
00080     const poly operator- (const poly&) const;
00081     const poly operator* (const poly&) const;
00082     const poly operator/ (const poly&) const;
00083     const poly operator% (const poly&) const;
00084
00085     bool operator== (const poly&) const;
00086     bool operator!= (const poly&) const;
00087     poly& operator= (const poly &);
00088     WORD& operator[] (int);
00089     const WORD& operator[] (int) const;
00090
00091 #if __cplusplus >= 201103L
00092     // support move semantics
00093     poly (poly &&, WORD=-1, WORD=1) noexcept;
00094     poly& operator= (poly &&) noexcept;
00095 #endif
00096
00097     // memory management
00098     void termscopy (const WORD *, int, int);
00099     void check_memory(int);
00100     void expand_memory(int);
00101
00102     // check type of polynomial
00103     bool is_zero () const;
00104     bool is_one () const;
00105     bool is_integer () const;
00106     bool is_monomial () const;
00107     int is_dense_univariate () const;
00108
00109     // properties
00110     int sign () const;
```

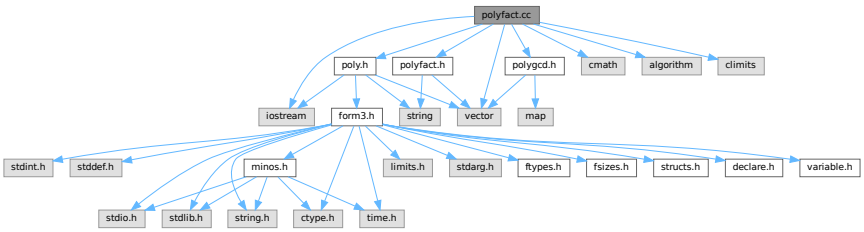
```
00111     int degree (int) const;
00112     int total_degree () const;
00113     int first_variable () const;
00114     int number_of_terms () const;
00115     const std::vector<int> all_variables () const;
00116     const poly integer_lcoeff () const;
00117     const poly lcoeff_univar (int) const;
00118     const poly lcoeff_multivar (int) const;
00119     const poly coefficient (int, int) const;
00120     const poly derivative (int) const;
00121 
00122     // modulo calculus
00123     void setmod(WORD, WORD=1);
00124     void coefficients_modulo (UWORD *, WORD, bool);
00125 
00126     // simple polynomials
00127     static const poly simple_poly (PHEAD int, int=0, int=1, int=0, int=1);
00128     static const poly simple_poly (PHEAD int, const poly&, int=1, int=0, int=1);
00129 
00130     // conversion from/to form notation
00131     static void get_variables (PHEAD const std::vector<WORD *> &, bool, bool);
00132     static const poly argument_to_poly (PHEAD WORD *, bool, bool, poly *den=NULL);
00133     static void poly_to_argument (const poly &, WORD *, bool);
00134     static void poly_to_argument_with_den (const poly &, WORD, const UWORD *, WORD *, bool);
00135     int size_of_form_notation () const;
00136     int size_of_form_notation_with_den (WORD) const;
00137     const poly & normalize ();
00138 
00139     // operations for coefficient lists
00140     static const poly from_coefficient_list (PHEAD const std::vector<WORD> &, int, WORD);
00141     static const std::vector<WORD> to_coefficient_list (const poly &);
00142     static const std::vector<WORD> coefficient_list_divmod (const std::vector<WORD> &, const
      std::vector<WORD> &, WORD, int);
00143 
00144     // string output for debugging
00145     const std::string to_string() const;
00146 
00147     // monomials
00148     static int monomial_compare (PHEAD const WORD *, const WORD *);
00149     WORD last_monomial_index () const;
00150     WORD* last_monomial () const;
00151     int compare_degree_vector(const poly &) const;
00152     std::vector<int> degree_vector() const;
00153     int compare_degree_vector(const std::vector<int> &) const;
00154 
00155     // (internal) mathematical operations
00156     static void add (const poly &, const poly &, poly &);
00157     static void sub (const poly &, const poly &, poly &);
00158     static void mul (const poly &, const poly &, poly &);
00159     static void div (const poly &, const poly &, poly &);
00160     static void mod (const poly &, const poly &, poly &);
00161     static void divmod (const poly &, const poly &, poly &, poly &, bool only_divides);
00162     static bool divides (const poly &, const poly &);
00163 
00164     static void mul_one_term (const poly &, const poly &, poly &);
00165     static void mul_univar (const poly &, const poly &, poly &, int);
00166     static void mul_heap (const poly &, const poly &, poly &);
00167 
00168     static void divmod_one_term (const poly &, const poly &, poly &, poly &, bool);
00169     static void divmod_univar (const poly &, const poly &, poly &, poly &, int, bool);
00170     static void divmod_heap (const poly &, const poly &, poly &, poly &, bool, bool, bool&);
00171 
00172     static void push_heap (PHEAD WORD **, int);
00173     static void pop_heap (PHEAD WORD **, int);
00174 };
00175 
00176 // comparison class for monomials (for std::sort)
00177 class monomial_larger {
00178 
00179 public:
00180 
00181 #ifndef WITHPTHREADS
00182     monomial_larger() {}
00183 #else
00184     ALLPRIVATES *B;
00185     monomial_larger(ALLPRIVATES *b): B(b) {}
00186 #endif
00187 
00188     bool operator()(const WORD *a, const WORD *b) {
00189         return poly::monomial_compare(BHEAD a, b) > 0;
00190     }
00191 };
00192 
00193 // stream operator
00194 std::ostream& operator<< (std::ostream &, const poly &);
00195 
00196 // inline function definitions
```

```
00197
00198 #if __cplusplus >= 201103L
00199
00200 inline poly::poly (poly &&a, WORD modp, WORD modn) noexcept {
00201 #ifdef POLY_MOVE_DEBUG
00202     std::cout « "poly move ctor" « std::endl;
00203 #endif
00204     POLY_GETIDENTITY(a);
00205     POLY_STOREIDENTITY;
00206     terms = a.terms;
00207     size_of_terms = a.size_of_terms;
00208     this->modp = a.modp;
00209     this->modn = a.modn;
00210     if (modp != -1) {
00211         setmod(modp,modn);
00212     }
00213     a.terms = nullptr;
00214     a.size_of_terms = -1;
00215 }
00216
00217 inline poly& poly::operator= (poly &&a) noexcept {
00218 #ifdef POLY_MOVE_DEBUG
00219     std::cout « "poly move assign" « std::endl;
00220 #endif
00221     if (this != &a) {
00222         WORD *old_terms = terms;
00223         LONG old_size_of_terms = size_of_terms;
00224         terms = a.terms;
00225         size_of_terms = a.size_of_terms;
00226         modp = a.modp;
00227         modn = a.modn;
00228         a.terms = old_terms;
00229         a.size_of_terms = old_size_of_terms;
00230     }
00231     return *this;
00232 }
00233
00234 #endif
00235
00236 /*   Checks whether the terms array is large enough to add another
00237  *   term (of size AM.MaxTal) to the polynomials. In case not, it is
00238  *   expanded.
00239  */
00240 inline void poly::check_memory (int i) {
00241     POLY_GETIDENTITY(*this);
00242     if (i + 3 + AN.poly_num_vars + AM.MaxTal >= size_of_terms) expand_memory(i + AM.MaxTal);
00243 //  Used to be i+2 but there should also be space for a trailing zero
00244 }
00245
00246 // indexing operators
00247 inline WORD& poly::operator[] (int i) {
00248     return terms[i];
00249 }
00250
00251 inline const WORD& poly::operator[] (int i) const {
00252     return terms[i];
00253 }
00254
00255 /*   Copies "num" WORD-sized terms from the pointer "source" to the
00256  *   current polynomial at index "dest"
00257  */
00258 inline void poly::termscopy (const WORD *source, int dest, int num) {
00259 #ifdef POLY_MOVE_DEBUG
00260     std::cout « "poly termscopy: num=" « num « std::endl;
00261 #endif
00262     memcpy (terms+dest, source, num*sizeof(WORD));
00263 }
```

4.105 polyfact.cc File Reference

```
#include "poly.h"
#include "polygcd.h"
#include "polyfact.h"
#include <cmath>
#include <vector>
#include <iostream>
#include <algorithm>
```

#include <climits>
Include dependency graph for polyfact.cc:



Functions

- ostream & operator<< (ostream &out, const factorized_poly &a)
- template<class T >
  ostream & operator<< (ostream &out, const vector< T > &v)

4.105.1 Detailed Description

Contains the routines for factorizing multivariate polynomials

Definition in file polyfact.cc.

4.105.2 Function Documentation

4.105.2.1 operator<<() [1/2]

```
ostream & operator<< (
            ostream & out,
            const factorized_poly & a )
```

Definition at line 104 of file polyfact.cc.

4.105.2.2 operator<<() [2/2]

```
template<class T >
ostream & operator<< (
            ostream & out,
            const vector< T > & v )
```

Definition at line 109 of file polyfact.cc.

4.106 polyfact.cc

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
00026  *
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
00031 /*
00032     #[ include :
00033 */
00034
00035 #include "poly.h"
00036 #include "polygcd.h"
00037 #include "polyfact.h"
00038
00039 #include <cmath>
00040 #include <vector>
00041 #include <iostream>
00042 #include <algorithm>
00043 #include <climits>
00044
00045 //#define DEBUG
00046
00047 #ifdef DEBUG
00048 #include "mytime.h"
00049 #endif
00050
00051 using namespace std;
00052
00053 /*
00054     #] include :
00055     #[ tostring :
00056 */
00057
00058 // Turns a factorized_poly into a readable string
00059 const string factorized_poly::tostring () const {
00060
00061     // empty
00062     if (factor.size()==0)
00063         return "no_factors";
00064
00065     string res;
00066
00067     // polynomial
00068     for (int i=0; i<(int)factor.size(); i++) {
00069         if (i>0) res += "*";
00070         res += "(";
00071         res += poly(factor[i],0,1).to_string();
00072         res += ")";
00073         if (power[i]>1) {
00074             res += "^";
00075             char tmp[100];
00076             snprintf (tmp,100,"%i",power[i]);
00077             res += tmp;
00078         }
00079     }
00080
00081     // modulo p^n
00082     if (factor[0].modp>0) {
00083         res += " (mod ";
00084         char tmp[12];
00085         snprintf (tmp,12,"%i",factor[0].modp);
```

```
00086         res += tmp;
00087         if (factor[0].modn > 1) {
00088             snprintf (tmp,12,"%i",factor[0].modn);
00089             res += "^";
00090             res += tmp;
00091         }
00092         res += ")";
00093     }
00094 
00095     return res;
00096 }
00097 
00098 /*
00099     #] tostring :
00100     #[ ostream operator :
00101 */
00102 
00103 // ostream operator for outputting a factorized_poly
00104 ostream& operator« (ostream &out, const factorized_poly &a) {
00105     return out « a.tostring();
00106 }
00107 
00108 // ostream operator for outputting a vector<T>
00109 template<class T> ostream& operator« (ostream &out, const vector<T> &v) {
00110     out«"{";
00111     for (int i=0; i<(int)v.size(); i++) {
00112         if (i>0) out«",";
00113         out«v[i];
00114     }
00115     out«"}";
00116     return out;
00117 }
00118 
00119 /*
00120     #] ostream operator :
00121     #[ add_factor :
00122 */
00123 
00124 // adds a factor f^p to a factorization
00125 void factorized_poly::add_factor(const poly &f, int p) {
00126     factor.push_back(f);
00127     power.push_back(p);
00128 }
00129 
00130 /*
00131     #] add_factor :
00132     #[ extended_gcd_Euclidean_lifted :
00133 */
00134 
00152 const vector<poly> polyfact::extended_gcd_Euclidean_lifted (const poly &a, const poly &b) {
00153 
00154 #ifdef DEBUGALL
00155     cout « "*** [" « thetime() « "]  CALL: extended_Euclidean_lifted("«a«","«b«")"«endl;
00156 #endif
00157 
00158     POLY_GETIDENTITY(a);
00159 
00160     // Calculate s,t,gcd (mod p) with the Extended Euclidean Algorithm.
00161     poly s(a,a.modp,1);
00162     poly t(b,b.modp,1);
00163     poly sa(BHEAD 1,a.modp,1);
00164     poly sb(BHEAD 0,b.modp,1);
00165     poly ta(BHEAD 0,a.modp,1);
00166     poly tb(BHEAD 1,b.modp,1);
00167 
00168     while (!t.is_zero()) {
00169         poly x(s/t);
00170         swap(s -=x*t , t);
00171         swap(sa-=x*ta, ta);
00172         swap(sb-=x*tb, tb);
00173     }
00174 
00175     // Normalize the result.
00176     sa /= s.integer_lcoeff();
00177     sb /= s.integer_lcoeff();
00178 
00179     // Lift the result to modulo p^n with p-adic Newton's iteration.
00180     poly samodp(sa);
00181     poly sbmodp(sb);
00182     poly term(BHEAD 1);
00183 
00184     sa.setmod(0,1);
00185     sb.setmod(0,1);
00186 
00187     poly amodp(a,a.modp,1);
00188     poly bmodp(b,a.modp,1);
00189     poly error(poly(BHEAD 1) - sa*a - sb*b);
```

```
00190     poly p(BHEAD a.modp);
00191
00192     for (int n=1; n<a.modn && !error.is_zero(); n++) {
00193         error /= p;
00194         term *= p;
00195         poly errormodp(error,a.modp,1);
00196         poly dsa((samodp * errormodp) % bmodp);
00197         // equivalent, but faster than the symmetric
00198         // poly dsb((sbmodp * errormodp) % amodp);
00199         poly dsb((errormodp - dsa*amodp) / bmodp);
00200         sa += term * dsa;
00201         sb += term * dsb;
00202         error -= a*dsa + b*dsb;
00203     }
00204
00205     sa.setmod(a.modp,a.modn);
00206     sb.setmod(a.modp,a.modn);
00207
00208     // Output the result
00209     vector<poly> res;
00210     res.push_back(sa);
00211     res.push_back(sb);
00212
00213 #ifdef DEBUGALL
00214     cout « "*** [" « thetime() « "]  RES : extended_Euclidean_lifted("«a«","«b«") = "«res«endl;
00215 #endif
00216
00217     return res;
00218 }
00219
00220 /*
00221     #] extended_gcd_Euclidean_lifted :
00222     #[ solve_Diophantine_univariate :
00223 */
00224
00256 const vector<poly> polyfact::solve_Diophantine_univariate (const vector<poly> &a, const poly &b) {
00257
00258 #ifdef DEBUGALL
00259     cout « "*** [" « thetime() « "]  CALL: solve_Diophantine_univariate(" «a«","«b«")"«endl;
00260 #endif
00261
00262     POLY_GETIDENTITY(b);
00263
00264     vector<poly> s(1,b);
00265
00266     for (int i=0; i+1<(int)a.size(); i++) {
00267         poly A(BHEAD 1,b.modp,b.modn);
00268         for (int j=i+1; j<(int)a.size(); j++) A *= a[j];
00269
00270         vector<poly> t(extended_gcd_Euclidean_lifted(a[i],A));
00271         poly prev(s.back());
00272         s.back() = t[1] * prev % a[i];
00273         s.push_back(t[0] * prev % A);
00274     }
00275
00276 #ifdef DEBUGALL
00277     cout « "*** [" « thetime() « "]  RES : solve_Diophantine_univariate(" «a«","«b«") = "«s«endl;
00278 #endif
00279
00280     return s;
00281 }
00282
00283 /*
00284     #] solve_Diophantine_univariate :
00285     #[ solve_Diophantine_multivariate :
00286 */
00287
00319 const vector<poly> polyfact::solve_Diophantine_multivariate (const vector<poly> &a, const poly &b,
      const vector<int> &x, const vector<int> &c, int d) {
00320
00321 #ifdef DEBUGALL
00322     cout « "*** [" « thetime() « "]  CALL: solve_Diophantine_multivariate("
      «a«","«b«","«x«","«c«","«d«")"«endl;
00323 #endif
00324
00325     POLY_GETIDENTITY(b);
00326
00327     if (b.is_zero()) return vector<poly>(a.size(),poly(BHEAD 0));
00328
00329     if (x.size() == 1) return solve_Diophantine_univariate(a,b);
00330
00331     // Reduce the polynomial with the homomorphism <xm-c{m-1}>
00332     poly simple(poly::simple_poly(BHEAD x.back(),c.back()));
00333
00334     vector<poly> ared (a);
00335     for (int i=0; i<(int)ared.size(); i++)
00336         ared[i] %= simple;
```

```
00337
00338     poly bred(b % simple);
00339     vector<int> xred(x.begin(),x.end()-1);
00340     vector<int> cred(c.begin(),c.end()-1);
00341
00342     // Solve the equation in one less variable
00343     vector<poly> s(solve_Diophantine_multivariate(ared,bred,xred,cred,d));
00344     if (s == vector<poly>()) return vector<poly>();
00345
00346     // Cache the Ai = product(aj | j!=i).
00347     vector<poly> A(a.size(), poly(BHEAD 1,b.modp,b.modn));
00348     for (int i=0; i<(int)a.size(); i++)
00349         for (int j=0; j<(int)a.size(); j++)
00350             if (i!=j) A[i] *= a[j];
00351
00352     // Add the powers (xm-c{m-1})^k with ideal-adic Newton iteration.
00353     poly term(BHEAD 1,b.modp,b.modn);
00354
00355     poly error(b);
00356     for (int i=0; i<(int)A.size(); i++)
00357         error -= s[i] * A[i];
00358
00359     for (int deg=1; deg<=d; deg++) {
00360
00361         if (error.is_zero()) break;
00362
00363         error /= simple;
00364         term *= simple;
00365
00366         vector<poly> ds(solve_Diophantine_multivariate(ared, error%simple, xred, cred, d));
00367         if (ds == vector<poly>()) return vector<poly>();
00368
00369         for (int i=0; i<(int)s.size(); i++) {
00370             s[i] += ds[i] * term;
00371             error -= ds[i] * A[i];
00372         }
00373     }
00374
00375     if (!error.is_zero()) return vector<poly>();
00376
00377 #ifdef DEBUGALL
00378     cout « "*** [" « thetime() « "]  RES : solve_Diophantine_multivariate("
      «a«","«b«","«x«","«c«","«d«") = "«s«endl;
00379 #endif
00380
00381     return s;
00382 }
00383
00384 /*
00385     #] solve_Diophantine_multivariate :
00386     #[ lift_coefficients :
00387 */
00388

00421 const vector<poly> polyfact::lift_coefficients (const poly &_A, const vector<poly> &_a) {
00422
00423 #ifdef DEBUG
00424     cout « "*** [" « thetime() « "]  CALL: lift_coefficients("«_A«","«_a«")"«endl;
00425 #endif
00426
00427     POLY_GETIDENTITY(_A);
00428
00429     poly A(_A);
00430     vector<poly> a(_a);
00431     poly term(BHEAD 1);
00432
00433     int x = A.first_variable();
00434
00435     // Replace the leading term of all factors with lterm(A) mod p
00436     poly lead(A.integer_lcoeff());
00437     for (int i=0; i<(int)a.size(); i++) {
00438         a[i] *= lead / a[i].integer_lcoeff();
00439         if (i>0) A*=lead;
00440     }
00441
00442     // Solve Diophantine equation
00443     vector<poly> s(solve_Diophantine_univariate(a,poly(BHEAD 1,A.modp,1)));
00444
00445     // Replace the leading term of all factors with lterm(A) mod p^n
00446     for (int i=0; i<(int)a.size(); i++) {
00447         a[i].setmod(A.modp,A.modn);
00448         a[i] += (lead - a[i].integer_lcoeff()) * poly::simple_poly(BHEAD x,0,a[i].degree(x));
00449     }
00450
00451     // Calculate the error, express it in terms of ai and add corrections.
00452     for (int k=2; k<=A.modn; k++) {
00453         term *= poly(BHEAD A.modp);
```

```
00454
00455         poly error(BHEAD -1);
00456         for (int i=0; i<(int)a.size(); i++) error *= a[i];
00457         error += A;
00458         if (error.is_zero()) break;
00459
00460         error /= term;
00461         error.setmod(A.modp,1);
00462
00463         for (int i=0; i<(int)a.size(); i++)
00464             a[i] += term * (error * s[i] % a[i]);
00465     }
00466
00467     // Fix leading coefficients by dividing out integer contents.
00468     for (int i=0; i<(int)a.size(); i++)
00469         a[i] /= polygcd::integer_content(poly(a[i],0,1));
00470
00471 #ifdef DEBUG
00472     cout « "*** [" « thetime() « "]  RES : lift_coefficients("«_A«","«_a«") = "«a«endl;
00473 #endif
00474
00475     return a;
00476 }
00477
00478 /*
00479     #] lift_coefficients :
00480         #[ predetermine :
00481 */
00482
00497 void polyfact::predetermine (int dep, const vector<vector<int> > &state, vector<vector<vector<int> > >
      &terms, vector<int> &term, int sumdeg) {
00498     // store the term
00499     if (dep == (int)state.size()) {
00500         terms[sumdeg].push_back(term);
00501         return;
00502     }
00503
00504     // recursively create new terms
00505     term.push_back(0);
00506
00507     for (int deg=0; sumdeg+deg<(int)state[dep].size(); deg++)
00508         if (state[dep][deg] > 0) {
00509             term.back() = deg;
00510             predetermine(dep+1, state, terms, term, sumdeg+deg);
00511         }
00512
00513     term.pop_back();
00514 }
00515
00516 /*
00517         #] predetermine :
00518     #[ lift_variables :
00519 */
00520
00558 const vector<poly> polyfact::lift_variables (const poly &A, const vector<poly> &_a, const vector<int>
      &x, const vector<int> &c, const vector<poly> &lc) {
00559
00560 #ifdef DEBUG
00561     cout « "*** [" « thetime() « "]  CALL: lift_variables("«A«","«_a«","«x«","«c«","«lc«")\n";
00562 #endif
00563
00564     // If univariate, don't lift
00565     if (x.size()<=1) return _a;
00566
00567     POLY_GETIDENTITY(A);
00568
00569     vector<poly> a(_a);
00570
00571     // First method: predetermine coefficients
00572
00573     // check feasibility, otherwise it tries too many possibilities
00574     int cnt = POLYFACT_MAX_PREDETERMINATION;
00575     for (int i=0; i<(int)a.size(); i++) {
00576         if (a[i].number_of_terms() == 0) return vector<poly>();
00577         cnt /= a[i].number_of_terms();
00578     }
00579
00580     if (cnt>0) {
00581         // state[n][d]: coefficient of x^d in a[n] is
00582         // 0: non-existent, 1: undetermined, 2: determined
00583         int D = A.degree(x[0]);
00584         vector<vector<int> > state(a.size(), vector<int>(D+1, 0));
00585
00586         for (int i=0; i<(int)a.size(); i++)
00587             for (int j=1; j<a[i][0]; j+=a[i][j])
00588                 state[i][a[i][j+1+x[0]]] = j==1 ? 2 : 1;
00589
```

```
00590         // collect all products of terms
00591         vector<vector<vector<int> > > terms(D+1);
00592         vector<int> term;
00593         predetermine(0,state,terms,term);
00594 
00595         // count the number of undetermined coefficients
00596         vector<int> num_undet(terms.size(),0);
00597         for (int i=0; i<(int)terms.size(); i++)
00598             for (int j=0; j<(int)terms[i].size(); j++)
00599                 for (int k=0; k<(int)terms[i][j].size(); k++)
00600                     if (state[k][terms[i][j][k]] == 1) num_undet[i]++;
00601 
00602         // replace the current leading coefficients by the correct ones
00603         for (int i=0; i<(int)a.size(); i++)
00604             a[i] += (lc[i] - a[i].lcoeff_univar(x[0])) * poly::simple_poly(BHEAD
      x[0],0,a[i].degree(x[0]));
00605 
00606         bool changed;
00607         do {
00608             changed = false;
00609 
00610             for (int i=0; i<(int)terms.size(); i++) {
00611                 // is only one coefficient in a equation is undetermined, solve
00612                 // the equation to determine this coefficient
00613                 if (num_undet[i] == 1) {
00614                     // generate equation
00615                     poly lhs(BHEAD 0), rhs(A.coefficient(x[0],i), A.modp, A.modn);
00616                     int which_idx=-1, which_deg=-1;
00617                     for (int j=0; j<(int)terms[i].size(); j++) {
00618                         poly coeff(BHEAD 1, A.modp, A.modn);
00619                         bool undet=false;
00620                         for (int k=0; k<(int)terms[i][j].size(); k++) {
00621                             if (state[k][terms[i][j][k]] == 1) {
00622                                 undet = true;
00623                                 which_idx=k;
00624                                 which_deg=terms[i][j][k];
00625                             }
00626                             else
00627                                 coeff *= a[k].coefficient(x[0], terms[i][j][k]);
00628                         }
00629                         if (undet)
00630                             lhs = coeff;
00631                         else
00632                             rhs -= coeff;
00633                     }
00634                     // solve equation
00635                     if (A.modn > 1) rhs.setmod(0,1);
00636                     if (lhs.is_zero() || !(rhs%lhs).is_zero()) return vector<poly>();
00637                     a[which_idx] += (rhs / lhs - a[which_idx].coefficient(x[0],which_deg)) *
      poly::simple_poly(BHEAD x[0],0,which_deg);
00638                     state[which_idx][which_deg] = 2;
00639 
00640                     // update number of undetermined coefficients
00641                     for (int j=0; j<(int)terms.size(); j++)
00642                         for (int k=0; k<(int)terms[j].size(); k++)
00643                             if (terms[j][k][which_idx] == which_deg)
00644                                 num_undet[j]--;
00645 
00646                     changed = true;
00647                 }
00648             }
00649         }
00650         while (changed);
00651 
00652         // if this is the complete result, skip lifting
00653         poly check(BHEAD 1, A.modn>1?0:A.modp, 1);
00654         for (int i=0; i<(int)a.size(); i++)
00655             check *= a[i];
00656 
00657         if (check == A) return a;
00658     }
00659 
00660     // Second method: Hensel lifting
00661 
00662     // Calculate A and lc's modulo Ii = <xi-c{i-1],...,xm-c{m-1}> (for i=2,...,m)
00663     vector<poly> simple(x.size(), poly(BHEAD 0));
00664     for (int i=(int)x.size()-2; i>=0; i--)
00665         simple[i] = poly::simple_poly(BHEAD x[i+1],c[i],1);
00666 
00667     // Calculate the maximum degree of A in x2,...,xm
00668     int maxdegA=0;
00669     for (int i=1; i<(int)x.size(); i++)
00670         maxdegA = MaX(maxdegA, A.degree(x[i]));
00671 
00672     // Iteratively add the variables x2,...,xm
00673     for (int xi=1; xi<(int)x.size(); xi++) {
00674         // replace the current leading coefficients by the correct ones
```

```
00675         for (int i=0; i<(int)a.size(); i++)
00676             a[i] += (lc[i] - a[i].lcoeff_univar(x[0])) * poly::simple_poly(BHEAD
      x[0],0,a[i].degree(x[0]));
00677 
00678         vector<poly> anew(a);
00679         for (int i=0; i<(int)anew.size(); i++)
00680             for (int j=xi-1; j<(int)c.size(); j++)
00681                 anew[i] %= simple[j];
00682 
00683         vector<int> xnew(x.begin(), x.begin()+xi);
00684         vector<int> cnew(c.begin(), c.begin()+xi-1);
00685         poly term(BHEAD 1,A.modp,A.modn);
00686 
00687         // Iteratively add the powers xi^k
00688         for (int deg=1, maxdeg=A.degree(x[xi]); deg<=maxdeg; deg++) {
00689 
00690             term *= simple[xi-1];
00691 
00692             // Calculate the error, express it in terms of ai and add corrections.
00693             poly error(BHEAD -1,A.modp,A.modn);
00694             for (int i=0; i<(int)a.size(); i++) error *= a[i];
00695             error += A;
00696             for (int i=xi; i<(int)c.size(); i++) error %= simple[i];
00697 
00698             if (error.is_zero()) break;
00699 
00700             error /= term;
00701             error %= simple[xi-1];
00702 
00703             vector<poly> s(solve_Diophantine_multivariate(anew,error,xnew,cnew,maxdegA));
00704             if (s == vector<poly>()) return vector<poly>();
00705 
00706             for (int i=0; i<(int)a.size(); i++)
00707                 a[i] += s[i] * term;
00708         }
00709 
00710         // check whether PRODUCT(a[i]) = A mod <xi-c{i-1},...,xm-c{m-1}> over the integers or ZZ/p
00711         poly check(BHEAD -1, A.modn>1?0:A.modp, 1);
00712         for (int i=0; i<(int)a.size(); i++) check *= a[i];
00713         check += A;
00714         for (int i=xi; i<(int)c.size(); i++) check %= simple[i];
00715         if (!check.is_zero()) return vector<poly>();
00716     }
00717 
00718 #ifdef DEBUG
00719     cout « "*** [" « thetime() « "]  RES : lift_variables("«A«","«_a«","«x«","«c«","«lc«") = " « a «
      endl;
00720 #endif
00721 
00722     return a;
00723 }
00724 
00725 /*
00726     #] lift_variables :
00727     #[ choose_prime :
00728 */
00729 
00741 WORD polyfact::choose_prime (const poly &a, const vector<int> &x, WORD p) {
00742 
00743 #ifdef DEBUG
00744     cout « "*** [" « thetime() « "]  CALL: choose_prime("«a«","«x«","«p«")"«endl;
00745 #endif
00746 
00747     POLY_GETIDENTITY(a);
00748 
00749     poly icont_lcoeff(polygcd::integer_content(a.lcoeff_univar(x[0])));
00750 
00751     if (p==0) p = POLYFACT_FIRST_PRIME;
00752 
00753     poly icont_lcoeff_modp(BHEAD 0);
00754 
00755     do {
00756 
00757         bool is_prime;
00758 
00759         do {
00760             p += 2;
00761             is_prime = true;
00762             for (int d=2; d*d<=p; d++)
00763                 if (p%d==0) { is_prime=false; break; }
00764         }
00765         while (!is_prime);
00766 
00767         icont_lcoeff_modp = icont_lcoeff;
00768         icont_lcoeff_modp.setmod(p,1);
00769     }
00770     while (icont_lcoeff_modp.is_zero());
```

```
00771
00772 #ifdef DEBUG
00773     cout « "*** [" « thetime() « "]  RES : choose_prime("«a«","«x«",?) = "«p«endl;
00774 #endif
00775
00776     return p;
00777 }
00778
00779 /*
00780     #] choose_prime :
00781     #[ choose_prime_power :
00782 */
00783
00798 WORD polyfact::choose_prime_power (const poly &a, WORD p) {
00799
00800 #ifdef DEBUG
00801     cout « "*** [" « thetime() « "]  CALL: choose_prime_power("«a«","«p«")"«endl;
00802 #endif
00803
00804     POLY_GETIDENTITY(a);
00805
00806     // analyse the polynomial for calculating the bound
00807     double maxdegree=0, maxlogcoeff=0, numterms=0;
00808
00809     for (int i=1; i<a[0]; i+=a[i]) {
00810         for (int j=0; j<AN.poly_num_vars; j++)
00811             maxdegree = MaX(maxdegree, a[i+1+j]);
00812
00813         maxlogcoeff = MaX(maxlogcoeff,
00814                                             log(1.0+(UWORD)a[i+a[i]-2]) +            // most
    significant digit + 1
00815                                             BITSINWORD*log(2.0)*(ABS(a[i+a[i]-1])-1)); // number of
    digits
00816         numterms++;
00817     }
00818
00819     WORD res = (WORD)ceil((log((sqrt(5.0)+1)/2)*maxdegree + maxlogcoeff + 0.5*log(numterms)) /
    log((double)p));
00820
00821 #ifdef DEBUG
00822     cout « "*** [" « thetime() « "]  CALL: choose_prime_power("«a«","«p«") = "«res«endl;
00823 #endif
00824
00825     return res;
00826 }
00827
00828 /*
00829     #] choose_prime_power :
00830     #[ choose_ideal :
00831 */
00832
00860 const vector<int> polyfact::choose_ideal (const poly &a, int p, const factorized_poly &lc, const
    vector<int> &x) {
00861
00862 #ifdef DEBUG
00863     cout « "*** [" « thetime() « "]  CALL: polyfact::choose_ideal("
00864              «a«","«p«","«lc«","«x«")"«endl;
00865 #endif
00866
00867     if (x.size()==1) return vector<int>();
00868
00869     POLY_GETIDENTITY(a);
00870
00871     vector<int> c(x.size()-1);
00872
00873     int dega = a.degree(x[0]);
00874     poly amodI(a);
00875
00876     // choose random c
00877     for (int i=0; i<(int)c.size(); i++) {
00878         c[i] = 1 + wranf(BHEAD0) % ((p-1) / POLYFACT_IDEAL_FRACTION);
00879         amodI %= poly::simple_poly(BHEAD x[i+1],c[i],1);
00880     }
00881
00882     poly amodIp(amodI);
00883     amodIp.setmod(p,1);
00884
00885     // check if leading coefficient is non-zero [equivalent to degree=old_degree]
00886     if (amodIp.degree(x[0]) != dega)
00887         return c = vector<int>();
00888
00889     // check if leading coefficient is squarefree [equivalent to gcd(a,a')==const]
00890     if (!polygcd::gcd_Euclidean(amodIp, amodIp.derivative(x[0])).is_integer())
00891         return c = vector<int>();
00892
00893     if (a.modp>0 && a.modn==1) return c;
00894
```

```
00895     // check for unique prime factors in each factor lc[i] of the leading coefficient
00896     vector<poly> d(1, polygcd::integer_content(amodI));
00897
00898     for (int i=0; i<(int)lc.factor.size(); i++) {
00899         // constant factor
00900         if (i==0 && lc.factor[i].is_integer()) {
00901             d[0] *= lc.factor[i];
00902             continue;
00903         }
00904
00905         // factor modulo I
00906         poly q(lc.factor[i]);
00907         for (int j=0; j<(int)c.size(); j++)
00908             q %= poly::simple_poly(BHEAD x[j+1],c[j]);
00909         if (q.sign() == -1) q *= poly(BHEAD -1);
00910
00911         // divide out common factors
00912         for (int j=(int)d.size()-1; j>=0; j--) {
00913             poly r(d[j]);
00914             while (!r.is_one()) {
00915                 r = polygcd::integer_gcd(r,q);
00916                 q /= r;
00917             }
00918         }
00919
00920         // check whether there is some factor left
00921         if (q.is_one()) return vector<int>();
00922         d.push_back(q);
00923     }
00924
00925 #ifdef DEBUG
00926     cout « "*** [" « thetime() « "]  RES : polyfact::choose_ideal("
00927              «a«","«p«","«lc«","«x«") = "«c«endl;
00928 #endif
00929
00930     return c;
00931 }
00932
00933 /*
00934     #] choose_ideal :
00935     #[ squarefree_factors_Yun :
00936 */
00937
00944 const factorized_poly polyfact::squarefree_factors_Yun (const poly &_a) {
00945
00946     factorized_poly res;
00947     poly a(_a);
00948
00949     int pow = 1;
00950     int x = a.first_variable();
00951
00952     poly b(a.derivative(x));
00953     poly c(polygcd::gcd(a,b));
00954
00955     while (true) {
00956         a /= c;
00957         b /= c;
00958         b -= a.derivative(x);
00959         if (b.is_zero()) break;
00960         c = polygcd::gcd(a,b);
00961         if (!c.is_one()) res.add_factor(c,pow);
00962         pow++;
00963     }
00964
00965     if (!a.is_one()) res.add_factor(a,pow);
00966     return res;
00967 }
00968
00969 /*
00970     #] squarefree_factors_Yun :
00971     #[ squarefree_factors_modp :
00972 */
00973
00980 const factorized_poly polyfact::squarefree_factors_modp (const poly &_a) {
00981
00982     factorized_poly res;
00983     poly a(_a);
00984
00985     int pow = 1;
00986     int x = a.first_variable();
00987     poly b(a.derivative(x));
00988
00989     // poly contains terms of the form c(x)^n (n!=c*p)
00990     if (!b.is_zero()) {
00991         poly c(polygcd::gcd(a,b));
00992         a /= c;
00993
```

```
00994         while (!a.is_one()) {
00995             b = polygcd::gcd(a,c);
00996             a /= b;
00997             if (!a.is_one()) res.add_factor(a,pow);
00998             pow++;
00999             a = b;
01000             c /= a;
01001         }
01002
01003         a = c;
01004     }
01005
01006     // polynomial contains terms of the form c(x)^p
01007     if (!a.is_one()) {
01008         for (int i=1; i<a[1]; i+=a[i])
01009             a[i+1+x] /= a.modp;
01010         factorized_poly res2(squarefree_factors(a));
01011         for (int i=0; i<(int)res2.factor.size(); i++) {
01012             res.factor.push_back(res2.factor[i]);
01013             res.power.push_back(a.modp*res2.power[i]);
01014         }
01015     }
01016
01017     return res;
01018 }
01019
01020 /*
01021     #] squarefree_factors_modp :
01022     #[ squarefree_factors :
01023 */
01024
01045 const factorized_poly polyfact::squarefree_factors (const poly &a) {
01046
01047 #ifdef DEBUG
01048     cout « "*** [" « thetime() « "]  CALL: squarefree_factors("«a«")\n";
01049 #endif
01050
01051     if (a.is_one()) return factorized_poly();
01052
01053     factorized_poly res;
01054
01055     if (a.modp==0)
01056         res = squarefree_factors_Yun(a);
01057     else
01058         res = squarefree_factors_modp(a);
01059
01060 #ifdef DEBUG
01061     cout « "*** [" « thetime() « "]  RES : squarefree_factors("«a«") = "«res«"\n";
01062 #endif
01063
01064     return res;
01065 }
01066
01067 /*
01068     #] squarefree_factors :
01069     #[ Berlekamp_Qmatrix :
01070 */
01071
01094 const vector<vector<WORD> > polyfact::Berlekamp_Qmatrix (const poly &_a) {
01095
01096 #ifdef DEBUG
01097     cout « "*** [" « thetime() « "]  CALL: Berlekamp_Qmatrix("«_a«")\n";
01098 #endif
01099
01100     if (_a.all_variables() == vector<int>())
01101         return vector<vector<WORD> >(0);
01102
01103     POLY_GETIDENTITY(_a);
01104
01105     poly a(_a);
01106     int x = a.first_variable();
01107     int n = a.degree(x);
01108     int p = a.modp;
01109
01110     poly lc(a.integer_lcoeff());
01111     a /= lc;
01112
01113     vector<vector<WORD> > Q(n, vector<WORD>(n));
01114
01115     // c is the vector of coefficients of the polynomial a
01116     vector<WORD> c(n+1,0);
01117     for (int j=1; j<a[0]; j+=a[j])
01118         c[a[j+1+x]] = a[j+1+AN.poly_num_vars] * a[j+2+AN.poly_num_vars];
01119
01120     // d is the vector of coefficients of x^i mod a, starting with i=0
01121     vector<WORD> d(n,0);
01122     d[0]=1;
```

```
01123
01124     for (int i=0; i<=(n-1)*p; i++) {
01125         // store the coefficients of x^(i*p) mod a
01126         if (i%p==0) Q[i/p] = d;
01127
01128         // transform d=x^i mod a into d=x^(i+1) mod a
01129         vector<WORD> e(n);
01130         for (int j=0; j<n; j++) {
01131             e[j] = (-(LONG)d[n-1]*c[j] + (j>0?d[j-1]:0)) % p;
01132             if (e[j]<0) e[j]+=p;
01133         }
01134
01135         d=e;
01136     }
01137
01138     // Q = Q - I
01139     for (int i=0; i<n; i++)
01140         Q[i][i] = (Q[i][i] - 1 + p) % p;
01141
01142     // Gaussian elimination
01143     for (int i=0; i<n; i++) {
01144         // Find pivot
01145         int ii=i; while (ii<n && Q[i][ii]==0) ii++;
01146         if (ii==n) continue;
01147
01148         for (int k=0; k<n; k++)
01149             swap(Q[k][ii],Q[k][i]);
01150
01151         // normalize row i, which becomes the pivot
01152         WORD mul;
01153         GetModInverses (Q[i][i], p, &mul, NULL);
01154         vector<int> idx;
01155
01156         for (int k=0; k<n; k++) if (Q[k][i] != 0) {
01157             // store indices of non-zero elements for sparse matrices
01158             idx.push_back(k);
01159             Q[k][i] = ((LONG)Q[k][i] * mul) % p;
01160         }
01161
01162         // reduce
01163         for (int j=0; j<n; j++)
01164             if (j!=i && Q[i][j]!=0) {
01165                 LONG mul = Q[i][j];
01166                 for (int k=0; k<(int)idx.size(); k++) {
01167                     Q[idx[k]][j] = (Q[idx[k]][j] - mul*Q[idx[k]][i]) % p;
01168                     if (Q[idx[k]][j] < 0) Q[idx[k]][j]+=p;
01169                 }
01170             }
01171     }
01172
01173     for (int i=0; i<n; i++) {
01174         // Q = Q - I
01175         Q[i][i] = Q[i][i]-1;
01176
01177         // reduce all coefficients in the range 0,1,...,p-1
01178         for (int j=0; j<n; j++) {
01179             Q[i][j] = -Q[i][j]%p;
01180             if (Q[i][j]<0) Q[i][j]+=p;
01181         }
01182     }
01183
01184 #ifdef DEBUG
01185     cout « "*** [" « thetime() « "]  RES : Berlekamp_Qmatrix("«_a«") = "«Q«"\n";
01186 #endif
01187
01188     return Q;
01189 }
01190
01191 /*
01192     #] Berlekamp_Qmatrix :
01193     #[ Berlekamp_find_factors :
01194 */
01195
01219 const vector<poly> polyfact::Berlekamp_find_factors (const poly &_a, const vector<vector<WORD> > &_q)
      {
01220
01221 #ifdef DEBUG
01222     cout « "*** [" « thetime() « "]  CALL: Berlekamp_find_factors("«_a«","«_q«")\n";
01223 #endif
01224
01225     if (_a.all_variables() == vector<int>()) return vector<poly>(1,_a);
01226
01227     POLY_GETIDENTITY(_a);
01228
01229     vector<vector<WORD> > q=_q;
01230
01231     int rank=0;
```

```
01232     for (int i=0; i<(int)q.size(); i++)
01233         if (q[i]!=vector<WORD>(q[i].size(),0)) rank++;
01234
01235     poly a(_a);
01236     int x = a.first_variable();
01237     int n = a.degree(x);
01238     int p = a.modp;
01239
01240     a /= a.integer_lcoeff();
01241
01242     // Vector of factors, represented as dense polynomials mod p
01243     vector<vector<WORD> > fac(1, vector<WORD>(n+1,0));
01244
01245     fac[0] = poly::to_coefficient_list(a);
01246     bool finished=false;
01247
01248     // Loop over the columns of q + constant, i.e., an exhaustive list of possible factors
01249     for (int i=1; i<n && !finished; i++) {
01250         if (q[i] == vector<WORD>(n,0)) continue;
01251
01252         for (int s=0; s<p && !finished; s++) {
01253             for (int j=0; j<(int)fac.size() && !finished; j++) {
01254                 vector<WORD> c = polygcd::coefficient_list_gcd(fac[j], q[i], p);
01255
01256                 // If a non-trivial factor is found, add it to the list
01257                 if (c.size()!=1 && c.size()!=fac[j].size()) {
01258                     fac.push_back(c);
01259                     fac[j] = poly::coefficient_list_divmod(fac[j], c, p, 0);
01260                     if ((int)fac.size() == rank) finished=true;
01261                 }
01262             }
01263
01264             // Increase the constant term by one
01265             q[i][0] = (q[i][0]+1) % p;
01266         }
01267     }
01268
01269     // Convert the densely represented polynomials to sparse ones
01270     vector<poly> res(fac.size(),poly(BHEAD 0, p));
01271     for (int i=0; i<(int)fac.size(); i++)
01272         res[i] = poly::from_coefficient_list(BHEAD fac[i],x,p);
01273
01274 #ifdef DEBUG
01275     cout « "*** [" « thetime() « "]  RES : Berlekamp_find_factors("«_a«","«_q«") = "«res«"\n";
01276 #endif
01277
01278     return res;
01279 }
01280
01281 /*
01282     #] Berlekamp_find_factors :
01283     #[ combine_factors :
01284 */
01285
01307 const vector<poly> polyfact::combine_factors (const poly &a, const vector<poly> &f) {
01308
01309 #ifdef DEBUG
01310     cout « "*** [" « thetime() « "]  CALL: combine_factors("«a«","«f«")\n";
01311 #endif
01312
01313     POLY_GETIDENTITY(a);
01314
01315     poly a0(a,0,1);
01316     vector<poly> res;
01317
01318     int num_used = 0;
01319     vector<bool> used(f.size(), false);
01320
01321     // Loop over all bitmasks with num=1,2,...,size(factors)/2 bits
01322     // set, that contain only unused factors
01323     for (int num=1; num<=(int)(f.size() - num_used)/2; num++) {
01324         vector<int> next(f.size() - num_used, 0);
01325         for (int i=0; i<num; i++) next[next.size()-1-i] = 1;
01326
01327         do {
01328             poly fac(BHEAD 1,a.modp,a.modn);
01329             for (int i=0, j=0; i<(int)f.size(); i++)
01330                 if (!used[i] && next[j++]) fac *= f[i];
01331             fac /= fac.integer_lcoeff();
01332             fac *= a.integer_lcoeff();
01333             fac /= polygcd::integer_content(poly(fac,0,1));
01334
01335             if ((a0 % fac).is_zero()) {
01336                 res.push_back(fac);
01337                 for (int i=0, j=0; i<(int)f.size(); i++)
01338                     if (!used[i]) used[i] = next[j++];
01339                 num_used += num;
```

```
01340                 num--;
01341                 break;
01342             }
01343         }
01344         while (next_permutation(next.begin(), next.end()));
01345     }
01346
01347     // All unused factors together form one more factor
01348     if (num_used != (int)f.size()) {
01349         poly fac(BHEAD 1,a.modp,a.modn);
01350         for (int i=0; i<(int)f.size(); i++)
01351             if (!used[i]) fac *= f[i];
01352         fac /= fac.integer_lcoeff();
01353         fac *= a.integer_lcoeff();
01354         fac /= polygcd::integer_content(poly(fac,0,1));
01355         res.push_back(fac);
01356     }
01357
01358 #ifdef DEBUG
01359     cout « "*** [" « thetime() « "]  RES : combine_factors("«a«","«f«") = "«res«"\n";
01360 #endif
01361
01362     return res;
01363 }
01364
01365 /*
01366     #] combine_factors :
01367     #[ factorize_squarefree :
01368 */
01369
01388 const vector<poly> polyfact::factorize_squarefree (const poly &a, const vector<int> &x) {
01389
01390 #ifdef DEBUG
01391     cout « "*** [" « thetime() « "]  CALL: factorize_squarefree("«a«")\n";
01392 #endif
01393
01394     POLY_GETIDENTITY(a);
01395
01396     WORD p=a.modp, n=a.modn;
01397
01398  try_again:
01399
01400     int bestp=0;
01401     int min_factors = INT_MAX;
01402     poly amodI(BHEAD 0), bestamodI(BHEAD 0);
01403     vector<int> c,d,bestc,bestd;
01404     vector<vector<WORD> > q,bestq;
01405
01406     // Factorize leading coefficient
01407     factorized_poly lc(factorize(a.lcoeff_univar(x[0])));
01408
01409     // Try a number of primes
01410     int prime_tries = 0;
01411
01412     while (prime_tries<POLYFACT_NUM_CONFIRMATIONS && min_factors>1) {
01413         if (a.modp == 0) {
01414             p = choose_prime(a,x,p);
01415             n = 0;
01416             if (a.degree(x[0]) % p == 0) continue;
01417
01418             // Univariate case: check whether the polynomial mod p is squarefree
01419             // Multivariate case: this check is done after choosing I (for efficiency)
01420             if (x.size()==1) {
01421                 poly amodp(a,p,1);
01422                 if (polygcd::gcd_Euclidean(amodp, amodp.derivative(x[0])).degree(x[0]) != 0)
01423                     continue;
01424             }
01425         }
01426
01427         // Try a number of ideals
01428         if (x.size()>1)
01429             for (int ideal_tries=0; ideal_tries<POLYFACT_MAX_IDEAL_TRIES; ideal_tries++) {
01430                 c = choose_ideal(a,p,lc,x);
01431                 if (c.size()>0) break;
01432             }
01433
01434         if (x.size()==1 || c.size()>0) {
01435             amodI = a;
01436             for (int i=0; i<(int)c.size(); i++)
01437                 amodI %= poly::simple_poly(BHEAD x[i+1],c[i]);
01438
01439             // Determine Q-matrix and its rank. Smaller rank is better.
01440             q = Berlekamp_Qmatrix(poly(amodI,p,1));
01441             int rank=0;
01442             for (int i=0; i<(int)q.size(); i++)
01443                 if (q[i]!=vector<WORD>(q[i].size(),0)) rank++;
01444
```

```
01445             if (rank<min_factors) {
01446                 bestp=p;
01447                 bestc=c;
01448                 bestq=q;
01449                 bestamodI=amodI;
01450                 min_factors = rank;
01451                 prime_tries = 0;
01452             }
01453
01454             if (rank==min_factors)
01455                 prime_tries++;
01456
01457 #ifdef DEBUG
01458     cout « "*** [" « thetime() « "]  (A) : factorize_squarefree("«a«
01459         ") try p=" « p « " #factors=" « rank « " (min="«min_factors«"x"«prime_tries«")" « endl;
01460 #endif
01461         }
01462     }
01463
01464     p=bestp;
01465     c=bestc;
01466     q=bestq;
01467     amodI=bestamodI;
01468
01469     // Determine to which power of p to lift
01470     if (n==0) {
01471         n = choose_prime_power(amodI,p);
01472         n = MaX(n, choose_prime_power(a,p));
01473     }
01474
01475     amodI.setmod(p,n);
01476
01477 #ifdef DEBUG
01478     cout « "*** [" « thetime() « "]  (B) : factorize_squarefree("«a«
01479         ") chosen c = " « c « ", p^n = "«p«"^"«n«endl;
01480     cout « "*** [" « thetime() « "]  (C) : factorize_squarefree("«a«
01481         ") #factors = " « min_factors « endl;
01482 #endif
01483
01484     // Find factors
01485     vector<poly> f(Berlekamp_find_factors(poly(amodI,p,1),q));
01486
01487     // Lift coefficients
01488     if (f.size() > 1) {
01489         f = lift_coefficients(amodI,f);
01490         if (f==vector<poly>()) {
01491 #ifdef DEBUG
01492             cout « "factor_squarefree failed (lift_coeff step) : " « endl;
01493 #endif
01494             goto try_again;
01495         }
01496
01497         // Combine factors
01498         if (a.modp==0) f = combine_factors(amodI,f);
01499     }
01500
01501     // Lift variables
01502     if (f.size() == 1)
01503         f = vector<poly>(1, a);
01504
01505     if (x.size() > 1 && f.size() > 1) {
01506
01507         // The correct leading coefficients of the factors can be
01508         // reconstructed from prime number factors of the leading
01509         // coefficients modulo I. This is possible since all factors of
01510         // the leading coefficient have unique prime factors for the ideal
01511         // I is chosen as such.
01512
01513         poly amodI(a);
01514         for (int i=0; i<(int)c.size(); i++)
01515             amodI %= poly::simple_poly(BHEAD x[i+1],c[i]);
01516         poly delta(polygcd::integer_content(amodI));
01517
01518         vector<poly> lcmodI(lc.factor.size(), poly(BHEAD 0));
01519         for (int i=0; i<(int)lc.factor.size(); i++) {
01520             lcmodI[i] = lc.factor[i];
01521             for (int j=0; j<(int)c.size(); j++)
01522                 lcmodI[i] %= poly::simple_poly(BHEAD x[j+1],c[j]);
01523         }
01524
01525         vector<poly> correct_lc(f.size(), poly(BHEAD 1,p,n));
01526
01527         for (int j=0; j<(int)f.size(); j++) {
01528             poly lc_f(f[j].integer_lcoeff() * delta);
01529             WORD nlc_f = lc_f[lc_f[1]];
01530             poly quo(BHEAD 0),rem(BHEAD 0);
01531             WORD nquo,nrem;
```

```
01532
01533             for (int i=(int)lcmodI.size()-1; i>=0; i--) {
01534
01535                 if (i==0 && lc.factor[i].is_integer()) continue;
01536
01537                 do {
01538                     DivLong((UWORD *)&lc_f[2+AN.poly_num_vars], nlc_f,
01539                                     (UWORD *)&lcmodI[i][2+AN.poly_num_vars], lcmodI[i][lcmodI[i][1]],
01540                                     (UWORD *)&quo[0], &nquo,
01541                                     (UWORD *)&rem[0], &nrem);
01542
01543                     if (nrem == 0) {
01544                         correct_lc[j] *= lc.factor[i];
01545                         lc_f.termscopy(&quo[0], 2+AN.poly_num_vars, ABS(nquo));
01546                         nlc_f = nquo;
01547                     }
01548                 }
01549                 while (nrem == 0);
01550             }
01551         }
01552
01553         for (int i=0; i<(int)correct_lc.size(); i++) {
01554             poly correct_modI(correct_lc[i]);
01555             for (int j=0; j<(int)c.size(); j++)
01556                 correct_modI %= poly::simple_poly(BHEAD x[j+1],c[j]);
01557
01558             poly d(polygcd::integer_gcd(correct_modI, f[i].integer_lcoeff()));
01559             correct_lc[i] *= f[i].integer_lcoeff() / d;
01560             delta /= correct_modI / d;
01561             f[i] *= correct_modI / d;
01562         }
01563
01564         // increase n, because of multiplying with delta
01565         if (!delta.is_one()) {
01566             poly deltapow(BHEAD 1);
01567             for (int i=1; i<(int)correct_lc.size(); i++)
01568                     deltapow *= delta;
01569             while (!deltapow.is_zero()) {
01570                 deltapow /= poly(BHEAD p);
01571                 n++;
01572             }
01573
01574             for (int i=0; i<(int)f.size(); i++) {
01575                 f[i].modn = n;
01576                 correct_lc[i].modn = n;
01577             }
01578         }
01579
01580         poly aa(a,p,n);
01581
01582         for (int i=0; i<(int)correct_lc.size(); i++) {
01583             correct_lc[i] *= delta;
01584             f[i] *= delta;
01585             if (i>0) aa *= delta;
01586         }
01587
01588         f = lift_variables(aa,f,x,c,correct_lc);
01589
01590         for (int i=0; i<(int)f.size(); i++)
01591             if (a.modp == 0)
01592                 f[i] /= polygcd::integer_content(poly(f[i],0,1));
01593             else
01594                 f[i] /= polygcd::content_univar(f[i], x[0]);
01595
01596         if (f==vector<poly>()) {
01597 #ifdef DEBUG
01598             cout « "factor_squarefree failed (lift_var step)" « endl;
01599 #endif
01600             goto try_again;
01601         }
01602     }
01603
01604     // set modulus of the factors correctly
01605     if (a.modp==0)
01606         for (int i=0; i<(int)f.size(); i++)
01607             f[i].setmod(0,1);
01608
01609     // Final check (not sure if this is necessary, but it doesn't hurt)
01610     poly check(BHEAD 1,a.modp,a.modn);
01611     for (int i=0; i<(int)f.size(); i++)
01612         check *= f[i];
01613
01614     if (check != a) {
01615 #ifdef DEBUG
01616         cout « "factor_squarefree failed (final check) : " « f « endl;
01617 #endif
01618         goto try_again;
```

```
01619     }
01620
01621 #ifdef DEBUG
01622     cout « "*** [" « thetime() « "]  RES : factorize_squarefree("«a«","«x«","«c«") = "«f«"\n";
01623 #endif
01624
01625     return f;
01626 }
01627
01628 /*
01629     #] factorize_squarefree :
01630     #[ factorize :
01631 */
01632
01641 const factorized_poly polyfact::factorize (const poly &a) {
01642
01643 #ifdef DEBUG
01644     cout « "*** [" « thetime() « "]  CALL: factorize("«a«")\n";
01645 #endif
01646     vector<int> x = a.all_variables();
01647
01648     // No variables, so just one factor
01649     if (x.size() == 0) {
01650         factorized_poly res;
01651         if (a.is_one()) return res;
01652         res.add_factor(a,1);
01653         return res;
01654     }
01655
01656     // Remove content
01657     poly conta(polygcd::content_univar(a,x[0]));
01658
01659     factorized_poly faca(factorize(conta));
01660
01661     poly ppa(a / conta);
01662
01663     // Find a squarefree factorization
01664     factorized_poly b(squarefree_factors(ppa));
01665
01666 #ifdef DEBUG
01667     cout « "*** [" « thetime() « "]  ... : factorize("«a«") : SFF = "«b«"\n";
01668 #endif
01669
01670     // Factorize each squarefree factor and build the "factorized_poly"
01671     for (int i=0; i<(int)b.factor.size(); i++) {
01672
01673         vector<poly> c(factorize_squarefree(b.factor[i], x));
01674
01675         for (int j=0; j<(int)c.size(); j++) {
01676             faca.factor.push_back(c[j]);
01677             faca.power.push_back(b.power[i]);
01678         }
01679     }
01680
01681 #ifdef DEBUG
01682     cout « "*** [" « thetime() « "]  RES : factorize("«a«") = "«faca«"\n";
01683 #endif
01684
01685     return faca;
01686 }
01687
01688 /*
01689     #] factorize :
01690 */
```

4.107 polyfact.h

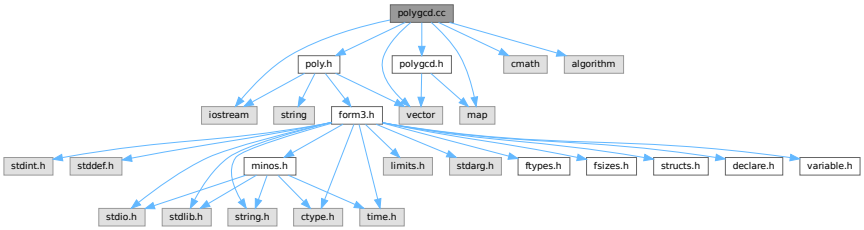
```
00001 #pragma once
00002 /* #[ License : */
00003 /*
00004  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00005  *   When using this file you are requested to refer to the publication
00006  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00007  *   This is considered a matter of courtesy as the development was paid
00008  *   for by FOM the Dutch physics granting agency and we would like to
00009  *   be able to track its scientific use to convince FOM of its value
00010  *   for the community.
00011  *
00012  *   This file is part of FORM.
00013  *
00014  *   FORM is free software: you can redistribute it and/or modify it under the
00015  *   terms of the GNU General Public License as published by the Free Software
00016  *   Foundation, either version 3 of the License, or (at your option) any later
```

```
00017  *   version.
00018  *
00019  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00020  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00021  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00022  *   details.
00023  *
00024  *   You should have received a copy of the GNU General Public License along
00025  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00026  */
00027 /* #] License : */
00028
00029 #include <vector>
00030 #include <string>
00031
00032 // First prime modulo which factorization is tried. Too small results
00033 // in more unsuccessful attempts; too large is slower.
00034 const int POLYFACT_FIRST_PRIME = 17;
00035
00036 // Fraction of [1,p) that is used for substitutions of variables. Too
00037 // small results in more unsuccessful attempts; too large is slower.
00038 const int POLYFACT_IDEAL_FRACTION = 5;
00039
00040 // Number of ideals that are tried before failure due to unlucky
00041 // choices is accepted.
00042 const int POLYFACT_MAX_IDEAL_TRIES = 3;
00043
00044 // Number of confirmations for the minimal number of factors before
00045 // Hensel lifting is started.
00046 const int POLYFACT_NUM_CONFIRMATIONS = 3;
00047
00048 // Maximum number of equations for predetermination of coefficients
00049 // for multivariate Hensel lifting
00050 const int POLYFACT_MAX_PREDETERMINATION = 10000;
00051
00052 class poly;
00053
00054 class factorized_poly {
00055     /*   Class for representing a factorized polynomial
00056      *   The polynomial is given by: PRODUCT(factor[i] ^ power[i])
00057      */
00058 public:
00059     std::vector<poly> factor;
00060     std::vector<int> power;
00061
00062     void add_factor(const poly &f, int p=1);
00063     const std::string tostring () const;
00064     friend std::ostream& operator« (std::ostream &out, const poly &p);
00065 };
00066
00067 std::ostream& operator« (std::ostream &out, const factorized_poly &a);
00068
00069 namespace polyfact {
00070
00071     // factorization routine
00072     const factorized_poly factorize (const poly &a);
00073
00074     // methods for squarefree factorization
00075     const factorized_poly squarefree_factors (const poly &a);
00076     const factorized_poly squarefree_factors_Yun (const poly &a);
00077     const factorized_poly squarefree_factors_modp (const poly &a);
00078
00079     // methods for choosing suitable reductions
00080     const std::vector<poly> factorize_squarefree (const poly &a, const std::vector<int> &x);
00081     WORD choose_prime (const poly &a, const std::vector<int> &x, WORD p=0);
00082     WORD choose_prime_power (const poly &a, WORD p);
00083     const std::vector<int> choose_ideal (const poly &a, int p, const factorized_poly &lc, const
      std::vector<int> &x);
00084
00085     // methods for univariate factorization
00086     const std::vector<std::vector<WORD> > Berlekamp_Qmatrix (const poly &a);
00087     const std::vector<poly> Berlekamp_find_factors (const poly &a, const std::vector<std::vector<WORD>
      > &Q);
00088     const std::vector<poly> combine_factors (const poly &a, const std::vector<poly> &f);
00089
00090     // methods for Hensel lifting
00091     const std::vector<poly> extended_gcd_Euclidean_lifted (const poly &a, const poly &b);
00092     const std::vector<poly> solve_Diophantine_univariate (const std::vector<poly> &a, const poly &b);
00093     const std::vector<poly> solve_Diophantine_multivariate (const std::vector<poly> &a, const poly &b,
      const std::vector<int> &x, const std::vector<int> &c, int d);
00094     const std::vector<poly> lift_coefficients (const poly &a, const std::vector<poly> &f);
00095     const std::vector<poly> lift_variables (const poly &a, const std::vector<poly> &f, const
      std::vector<int> &x, const std::vector<int> &c, const std::vector<poly> &lc);
00096     void predetermine (int dep, const std::vector<std::vector<int> > &state,
      std::vector<std::vector<std::vector<int> > > &terms, std::vector<int> &term, int sumdeg=0);
00097
00098 }
```

4.108 polygcd.cc File Reference

```
#include "poly.h"
#include "polygcd.h"
#include <iostream>
#include <vector>
#include <cmath>
#include <map>
#include <algorithm>
```

Include dependency graph for polygcd.cc:



Data Structures

- struct BracketInfo

Functions

- bool gcd_heuristic_possible (const poly &a)
- const poly gcd_linear_helper (const poly &a, const poly &b)

4.108.1 Detailed Description

Contains the routines for calculating greatest commons divisors of multivariate polynomials

Definition in file polygcd.cc.

4.108.2 Function Documentation

4.108.2.1 gcd_heuristic_possible()

```
bool gcd_heuristic_possible (
            const poly & a )
```

Heuristic greatest common divisor of multivariate polynomials

4.108.3 Description

Checks whether the heuristic seems possible by estimating

MAX_{terms} (coeff ^ PROD_{i=1..#vars} (pow_i+1))

and comparing this with GCD_HEURISTIC_MAX_DIGITS.

4.108.4 Notes

- For small polynomials, this consumes time and never triggers.

Definition at line 1142 of file polygcd.cc.

4.108.4.1 gcd_linear_helper()

```
const poly gcd_linear_helper (
             const poly & a,
             const poly & b )
```

Definition at line 1403 of file polygcd.cc.

4.109 polygcd.cc

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032 /*
00033     #[ include :
00034 */
00035
00036 #include "poly.h"
00037 #include "polygcd.h"
00038
00039 #include <iostream>
00040 #include <vector>
00041 #include <cmath>
00042 #include <map>
00043 #include <algorithm>
```

```
00044
00045 //#define DEBUG
00046 //#define DEBUGALL
00047
00048 #ifdef DEBUG
00049 #include "mytime.h"
00050 #endif
00051
00052 using namespace std;
00053
00054 /*
00055     #] include :
00056     #[ ostream operator :
00057 */
00058
00059 #ifdef DEBUG
00060 // ostream operator for outputting vector<T>s   for debugging purposes
00061 template<class T> ostream& operator« (ostream &out, const vector<T> &x) {
00062     out«"{";
00063     for (int i=0; i<(int)x.size(); i++) {
00064         if (i>0) out«",";
00065         out«x[i];
00066     }
00067     out«"}";
00068     return out;
00069 }
00070 #endif
00071
00072 /*
00073     #] ostream operator :
00074     #[ integer_gcd :
00075 */
00076
00091 const poly polygcd::integer_gcd (const poly &a, const poly &b) {
00092
00093 #ifdef DEBUGALL
00094     cout « "*** [" « thetime() « "]  CALL: integer_gcd(" « a « "," « b « ")" « endl;
00095 #endif
00096
00097     POLY_GETIDENTITY(a);
00098
00099     if (a.is_zero()) return b;
00100     if (b.is_zero()) return a;
00101
00102     poly c(BHEAD 0, a.modp, a.modn);
00103     WORD nc;
00104
00105     GcdLong(BHEAD
00106                     (UWORD *)&a[AN.poly_num_vars+2],a[a[0]-1],
00107                     (UWORD *)&b[AN.poly_num_vars+2],b[b[0]-1],
00108                     (UWORD *)&c[AN.poly_num_vars+2],&nc);
00109
00110     WORD x = 2 + AN.poly_num_vars + ABS(nc);
00111     c[1] = x;      // term length
00112     c[0] = x+1;    // total length
00113     c[x] = nc;     // coefficient length
00114
00115     for (int i=0; i<AN.poly_num_vars; i++)
00116         c[2+i] = 0; // powers
00117
00118 #ifdef DEBUGALL
00119     cout « "*** [" « thetime() « "]  RES : integer_gcd(" « a « "," « b « ") = " « c « endl;
00120 #endif
00121
00122     return c;
00123 }
00124
00125 /*
00126     #] integer_gcd :
00127     #[ integer_content :
00128 */
00129
00143 const poly polygcd::integer_content (const poly &a) {
00144
00145 #ifdef DEBUGALL
00146     cout « "*** [" « thetime() « "]  CALL: integer_content(" « a « ")" « endl;
00147 #endif
00148
00149     POLY_GETIDENTITY(a);
00150
00151     if (a.modp>0) return a.integer_lcoeff();
00152
00153     poly c(BHEAD 0, 0, 1);
00154     WORD *d = (WORD *)NumberMalloc("polygcd::integer_content");
00155     WORD nc=0;
00156
00157     for (int i=0; i<AN.poly_num_vars; i++)
```

```
00158         c[2+i] = 0;
00159
00160     for (int i=1; i<a[0]; i+=a[i]) {
00161
00162         WCOPY(d,&c[2+AN.poly_num_vars],nc);
00163
00164         GcdLong(BHEAD (UWORD *)d, nc,
00165                         (UWORD *)&a[i+1+AN.poly_num_vars], a[i+a[i]-1],
00166                         (UWORD *)&c[2+AN.poly_num_vars], &nc);
00167
00168         WORD x = 2 + AN.poly_num_vars + ABS(nc);
00169         c[1] = x;   // term length
00170         c[0] = x+1; // total length
00171         c[x] = nc;  // coefficient length
00172     }
00173
00174     if (a.sign() != c.sign()) c *= poly(BHEAD -1);
00175
00176     NumberFree(d,"polygcd::integer_content");
00177
00178 #ifdef DEBUGALL
00179     cout « "*** [" « thetime() « "]  RES : integer_content(" « a « ") = " « c « endl;
00180 #endif
00181
00182     return c;
00183 }
00184
00185 /*
00186     #] integer_content :
00187     #[ content_univar :
00188 */
00189
00205 const poly polygcd::content_univar (const poly &a, int x) {
00206
00207 #ifdef DEBUG
00208     cout « "*** [" « thetime() « "]  CALL: content_univar(" « a « "," « string(1,'a'+x) « ")" « endl;
00209 #endif
00210
00211     POLY_GETIDENTITY(a);
00212
00213     poly res(BHEAD 0, a.modp, a.modn);
00214
00215     for (int i=1; i<a[0];) {
00216         poly b(BHEAD 0, a.modp, a.modn);
00217         int deg = a[i+1+x];
00218
00219         for (; i<a[0] && a[i+1+x]==deg; i+=a[i]) {
00220             b.check_memory(b[0]+a[i]);
00221             b.termscopy(&a[i],b[0],a[i]);
00222             b[b[0]+1+x] = 0;
00223             b[0] += a[i];
00224         }
00225
00226         res = gcd(res, b);
00227
00228         if (res.is_integer()) {
00229             res = integer_content(a);
00230             break;
00231         }
00232     }
00233
00234     if (a.sign() != res.sign()) res *= poly(BHEAD -1);
00235
00236 #ifdef DEBUG
00237     cout « "*** [" « thetime() « "]  RES : content_univar(" « a « "," « string(1,'a'+x) « ") = " « res
      « endl;
00238 #endif
00239
00240     return res;
00241 }
00242
00243 /*
00244     #] content_univar :
00245         #[ content_multivar :
00246 */
00247
00263 const poly polygcd::content_multivar (const poly &a, int x) {
00264
00265 #ifdef DEBUGALL
00266     cout « "*** [" « thetime() « "]  CALL: content_multivar(" « a « "," « string(1,'a'+x) « ")" «
      endl;
00267 #endif
00268
00269     POLY_GETIDENTITY(a);
00270
00271     poly res(BHEAD 0, a.modp, a.modn);
00272
```

```
00273     for (int i=1,j; i<a[0]; i=j) {
00274         poly b(BHEAD 0, a.modp, a.modn);
00275 
00276         for (j=i; j<a[0]; j+=a[j]) {
00277             bool same_powers = true;
00278             for (int k=0; k<AN.poly_num_vars; k++)
00279                 if (k!=x && a[i+1+k]!=a[j+1+k]) {
00280                     same_powers = false;
00281                     break;
00282                 }
00283             if (!same_powers) break;
00284 
00285             b.check_memory(b[0]+a[j]);
00286             b.termscopy(&a[j],b[0],a[j]);
00287             for (int k=0; k<AN.poly_num_vars; k++)
00288                 if (k!=x) b[b[0]+1+k]=0;
00289 
00290             b[0] += a[j];
00291         }
00292 
00293         res = gcd_Euclidean(res, b);
00294 
00295         if (res.is_integer()) {
00296             res = poly(BHEAD a.sign(),a.modp,a.modn);
00297             break;
00298         }
00299     }
00300 
00301 #ifdef DEBUGALL
00302     cout « "*** [" « thetime() « "]  RES : content_multivar(" « a « "," « string(1,'a'+x) « ") = " «
      res « endl;
00303 #endif
00304 
00305     return res;
00306 }
00307 
00308 /*
00309     #] content_multivar :
00310         #[ coefficient_list_gcd :
00311 */
00312 
00325 const vector<WORD> polygcd::coefficient_list_gcd (const vector<WORD> &_a, const vector<WORD> &_b, WORD
      p) {
00326 
00327 #ifdef DEBUGALL
00328     cout « "*** [" « thetime() « "]  CALL: coefficient_list_gcd("«_a«","«_b«","«p«")"«endl;
00329 #endif
00330 
00331     vector<WORD> a(_a), b(_b);
00332 
00333     while (b.size() != 0) {
00334       a = poly::coefficient_list_divmod(a,b,p,1);
00335       swap(a,b);
00336     }
00337 
00338     while (a.back()==0) a.pop_back();
00339 
00340     WORD inv;
00341     GetModInverses(a.back() + (a.back()<0?p:0), p, &inv, NULL);
00342 
00343     for (int i=0; i<(int)a.size(); i++)
00344         a[i] = (LONG)inv*a[i] % p;
00345 
00346 #ifdef DEBUGALL
00347     cout « "*** [" « thetime() « "]  RES : coefficient_list_gcd("«_a«","«_b«","«p«") = "«a«endl;
00348 #endif
00349 
00350     return a;
00351 }
00352 
00353 /*
00354         #] coefficient_list_gcd :
00355     #[ gcd_Euclidean :
00356 */
00357 
00374 const poly polygcd::gcd_Euclidean (const poly &a, const poly &b) {
00375 
00376 #ifdef DEBUG
00377     cout « "*** [" « thetime() « "]  CALL: gcd_Euclidean("«a«","«b«")"«endl;
00378 #endif
00379 
00380     POLY_GETIDENTITY(a);
00381 
00382     if (a.is_zero()) return b;
00383     if (b.is_zero()) return a;
00384     if (a.is_integer() || b.is_integer())   return integer_gcd(a,b);
00385 
```

```
00386     poly res(BHEAD 0);
00387
00388     if (a.is_dense_univariate()>=-1 && b.is_dense_univariate()>=-1) {
00389         vector<WORD> coeff = coefficient_list_gcd(poly::to_coefficient_list(a),
00390
      poly::to_coefficient_list(b), a.modp);
00391         res = poly::from_coefficient_list(BHEAD coeff, a.first_variable(), a.modp);
00392     }
00393     else {
00394         res = a;
00395         poly rem(b);
00396         while (!rem.is_zero())
00397             swap(res%=rem, rem);
00398         res /= res.integer_lcoeff();
00399     }
00400
00401 #ifdef DEBUG
00402     cout « "*** [" « thetime() « "]  RES : gcd_Euclidean("«a«","«b«") = "«res«endl;
00403 #endif
00404
00405     return res;
00406 }
00407
00408 /*
00409     #] gcd_Euclidean :
00410         #[ chinese_remainder :
00411 */
00412
00426 const poly polygcd::chinese_remainder (const poly &a1, const poly &m1, const poly &a2, const poly &m2)
      {
00427
00428 #ifdef DEBUGALL
00429     cout « "*** [" « thetime() « "]  CALL: chinese_remainder(" « a1 « "," « m1 « "," « a2 « "," « m2 «
      ")" « endl;
00430 #endif
00431
00432     POLY_GETIDENTITY(a1);
00433
00434     WORD nx,ny,nz;
00435     UWORD *x = (UWORD *)NumberMalloc("polygcd::chinese_remainder");
00436     UWORD *y = (UWORD *)NumberMalloc("polygcd::chinese_remainder");
00437     UWORD *z = (UWORD *)NumberMalloc("polygcd::chinese_remainder");
00438
00439     GetLongModInverses(BHEAD (UWORD *)&m1[2+AN.poly_num_vars], m1[m1[1]],
00440                                         (UWORD *)&m2[2+AN.poly_num_vars], m2[m2[1]],
00441                                         (UWORD *)x, &nx, NULL, NULL);
00442
00443     AddLong((UWORD *)&a2[2+AN.poly_num_vars], a2.is_zero() ? 0 :  a2[a2[1]],
00444                     (UWORD *)&a1[2+AN.poly_num_vars], a1.is_zero() ? 0 : -a1[a1[1]],
00445                     y, &ny);
00446
00447     MulLong (x,nx,y,ny,z,&nz);
00448     MulLong (z,nz,(UWORD *)&m1[2+AN.poly_num_vars],m1[m1[1]],x,&nx);
00449
00450     AddLong (x,nx,(UWORD *)&a1[2+AN.poly_num_vars], a1.is_zero() ? 0 : a1[a1[1]],y,&ny);
00451
00452     MulLong ((UWORD *)&m1[2+AN.poly_num_vars], m1[m1[1]],
00453                      (UWORD *)&m2[2+AN.poly_num_vars], m2[m2[1]],
00454                      (UWORD *)z,&nz);
00455
00456     TakeNormalModulus (y,&ny,(UWORD *)z,nz,NOUNPACK);
00457
00458     poly res(BHEAD y,ny);
00459
00460     NumberFree(x,"polygcd::chinese_remainder");
00461     NumberFree(y,"polygcd::chinese_remainder");
00462     NumberFree(z,"polygcd::chinese_remainder");
00463
00464 #ifdef DEBUGALL
00465     cout « "*** [" « thetime() « "]  RES : chinese_remainder(" « a1 « "," « m1 « "," « a2 « "," « m2 «
      ") = " « res « endl;
00466 #endif
00467
00468     return res;
00469 }
00470
00471 /*
00472     #] chinese_remainder :
00473         #[ substitute :
00474 */
00475
00488 const poly polygcd::substitute(const poly &a, int x, int c) {
00489
00490     POLY_GETIDENTITY(a);
00491
00492     poly b(BHEAD 0);
00493
```

```
00494     if (a.is_zero()) {
00495         return b;
00496     }
00497
00498     bool zero=true;
00499     int bi=1;
00500
00501     // cache size is bounded by the degree in x, twice the number of terms of
00502     // the polynomial and a constant
00503     vector<WORD> cache(min(a.degree(x)+1,min(2*a.number_of_terms(),
00504
      POLYGCD_RAISPOWMOD_CACHE_SIZE)), 0);
00505
00506     for (int ai=1; ai<=a[0]; ai+=a[ai]) {
00507         // last term or different power, then add term to b iff non-zero
00508         if (!zero) {
00509             bool add=false;
00510             if (ai==a[0])
00511                 add=true;
00512             else {
00513                 for (int i=0; i<AN.poly_num_vars; i++)
00514                     if (i!=x && a[ai+1+i]!=b[bi+1+i]) {
00515                         zero=true;
00516                         add=true;
00517                         break;
00518                     }
00519             }
00520
00521             if (add) {
00522                 if (b[bi+AN.poly_num_vars+1] < 0)
00523                     b[bi+AN.poly_num_vars+1] += a.modp;
00524                 bi+=b[bi];
00525             }
00526
00527             if (ai==a[0]) break;
00528         }
00529
00530         b.check_memory(bi);
00531
00532         // create new term in b
00533         if (zero) {
00534             b[bi] = 3+AN.poly_num_vars;
00535             for (int i=0; i<AN.poly_num_vars; i++)
00536                 b[bi+1+i] = a[ai+1+i];
00537             b[bi+1+x] = 0;
00538             b[bi+AN.poly_num_vars+1] = 0;
00539             b[bi+AN.poly_num_vars+2] = 1;
00540         }
00541
00542         // add term of a to the current term in b
00543         LONG coeff = a[ai+1+AN.poly_num_vars] * a[ai+2+AN.poly_num_vars];
00544         int pow = a[ai+1+x];
00545
00546         if (pow<(int)cache.size()) {
00547             if (cache[pow]==0)
00548                 cache[pow] = RaisPowMod(c, pow, a.modp);
00549             coeff = (coeff * cache[pow]) % a.modp;
00550         }
00551         else {
00552             coeff = (coeff * RaisPowMod(c, pow, a.modp)) % a.modp;
00553         }
00554
00555         b[bi+AN.poly_num_vars+1] = (coeff + b[bi+AN.poly_num_vars+1]) % a.modp;
00556         if (b[bi+AN.poly_num_vars+1] != 0) zero=false;
00557     }
00558
00559     b[0]=bi;
00560     b.setmod(a.modp);
00561
00562     return b;
00563 }
00564
00565 /*
00566         #] substitute :
00567         #[ sparse_interpolation helper functions :
00568 */
00569
00570 // Returns a list of size #terms(a) with entries PROD(ci^powi, i=2..n)
00571 const vector<int> polygcd::sparse_interpolation_get_mul_list (const poly &a, const vector<int> &x,
      const vector<int> &c) {
00572     // cache size for variable x is bounded by the degree in x, twice
00573     // the number of terms of the polynomial and a constant
00574     vector<vector<WORD> > cache(c.size());
00575     int max_cache_size = min(2*a.number_of_terms(),POLYGCD_RAISPOWMOD_CACHE_SIZE);
00576     for (int i=0; i<(int)c.size(); i++)
00577         cache[i] = vector<WORD>(min(a.degree(x[i+1])+1,max_cache_size), 0);
00578
```

```
00579     vector<int> res;
00580     for (int i=1; i<a[0]; i+=a[i]) {
00581         LONG coeff=1;
00582         for (int j=0; j<(int)c.size(); j++) {
00583             int pow = a[i+1+x[j+1]];
00584             if (pow<(int)cache[j].size()) {
00585                 if (cache[j][pow]==0)
00586                     cache[j][pow] = RaisPowMod(c[j], pow, a.modp);
00587                 coeff = (coeff * cache[j][pow]) % a.modp;
00588             }
00589             else {
00590                 coeff = (coeff * RaisPowMod(c[j], pow, a.modp)) % a.modp;
00591             }
00592         }
00593         res.push_back(coeff);
00594     }
00595     return res;
00596 }
00597
00598 // Multiplies the coefficients of a with the entries of mul
00599 void polygcd::sparse_interpolation_mul_poly (poly &a, const vector<int> &mul) {
00600     for (int i=1,j=0; i<a[0]; i+=a[i],j++)
00601         a[i+a[i]-2] = ((LONG)a[i+a[i]-2]*mul[j]) % a.modp;
00602 }
00603
00604 // Sets all coefficients to the range 0..modp-1 and the powers of x2...xn to 0
00605 const poly polygcd::sparse_interpolation_reduce_poly (const poly &a, const vector<int> &x) {
00606     poly res(a);
00607     for (int i=1; i<a[0]; i+=a[i]) {
00608         for (int j=1; j<(int)x.size(); j++)
00609             res[i+1+x[j]]=0;
00610         if (res[i+a[i]-1]==-1) {
00611             res[i+a[i]-1]=1;
00612             res[i+a[i]-2]=a.modp-res[i+a[i]-2];
00613         }
00614     }
00615     return res;
00616 }
00617
00618 // Collects entries with equal powers, so that the result is a proper polynomial
00619 const poly polygcd::sparse_interpolation_fix_poly (const poly &a, int x) {
00620
00621     POLY_GETIDENTITY(a);
00622     poly res(BHEAD 0,a.modp,1);
00623
00624     int j=1;
00625     bool newterm=true;
00626
00627     for (int i=1; i<a[0]; i+=a[i]) {
00628         if (newterm)
00629             res.termscopy(&a[i], j, a[i]);
00630         else
00631             res[j+res[j]-2] = ((LONG)res[j+res[j]-2] + a[i+a[i]-2]) % a.modp;
00632
00633         newterm = i+a[i] == a[0] || res[j+1+x] != a[i+a[i]+1+x];
00634         if (newterm && res[j+res[j]-2]!=0) j += res[j];
00635     }
00636
00637     res[0]=j;
00638     return res;
00639 }
00640
00641 /*
00642         #] sparse_interpolation helper functions :
00643         #[ gcd_modular_sparse_interpolation :
00644 */
00645
00677 const poly polygcd::gcd_modular_sparse_interpolation (const poly &origa, const poly &origb, const
      vector<int> &x, const poly &s) {
00678
00679 #ifdef DEBUG
00680     cout « "*** [" « thetime() « "]  CALL: gcd_modular_sparse_interpolation("
00681              « origa « "," « origb « "," « x « "," « "," « s «")" « endl;
00682 #endif
00683
00684     POLY_GETIDENTITY(origa);
00685
00686     // strip multivariate content
00687     poly conta(content_multivar(origa,x.back()));
00688     poly contb(content_multivar(origb,x.back()));
00689     poly gcdconts(gcd_Euclidean(conta,contb));
00690     const poly& a = conta.is_one() ? origa : origa/conta;
00691     const poly& b = contb.is_one() ? origb : origb/contb;
00692
00693     // for non-monic cases, we need to normalize with the gcd of the lcoeffs of a poly in x[0]
00694     // or else the shape fitting does not work.
00695     // FIXME: the current implementation still rejects some valid shapes.
```

```
00696     poly lcgcd(BHEAD 1, a.modp);
00697     if (!s.lcoeff_univar(x[0]).is_integer()) {
00698         lcgcd = gcd_modular_dense_interpolation(a.lcoeff_univar(x[0]), b.lcoeff_univar(x[0]), x,
      poly(BHEAD 0));
00699     }
00700 
00701     // reduce polynomials
00702     poly ared(sparse_interpolation_reduce_poly(a,x));
00703     poly bred(sparse_interpolation_reduce_poly(b,x));
00704     poly sred(sparse_interpolation_reduce_poly(s,x));
00705     poly lred(sparse_interpolation_reduce_poly(lcgcd,x));
00706 
00707     // set all coefficients to 1
00708     for (int i=1; i<sred[0]; i+=sred[i]) {
00709         sred[i+sred[i]-2] = sred[i+sred[i]-1] = 1;
00710     }
00711 
00712     // generate random numbers and check there this set doesn't result
00713     // in a singular matrix
00714     vector<int> c(x.size()-1);
00715     vector<int> smul;
00716 
00717     bool duplicates;
00718     do {
00719         for (int i=0; i<(int)c.size(); i++)
00720             c[i] = 1 + wranf(BHEAD0) % (a.modp-1);
00721         smul = sparse_interpolation_get_mul_list(s,x,c);
00722 
00723         duplicates = false;
00724 
00725         int fr=0,to=0;
00726         for (int i=1; i<s[0];) {
00727             int pow = s[i+1+x[0]];
00728             while (i<s[0] && s[i+1+x[0]]==pow) i+=s[i], to++;
00729             for (int j=fr; j<to; j++)
00730                 for (int k=fr; k<j; k++)
00731                     if (smul[j] == smul[k])
00732                         duplicates = true;
00733             fr=to;
00734         }
00735     }
00736     while (duplicates);
00737 
00738     // get the lists to multiply the polynomials with every iteration
00739     vector<int> amul(sparse_interpolation_get_mul_list(a,x,c));
00740     vector<int> bmul(sparse_interpolation_get_mul_list(b,x,c));
00741     vector<int> lmul(sparse_interpolation_get_mul_list(lcgcd,x,c));
00742 
00743     vector<vector<vector<LONG> > > M;
00744     vector<vector<LONG> > V;
00745 
00746     int maxMsize=0;
00747 
00748     // create (empty) matrices
00749     for (int i=1; i<s[0]; i+=s[i]) {
00750         if (i==1 || s[i+1+x[0]]!=s[i+1+x[0]-s[i]]) {
00751             M.push_back(vector<vector<LONG> >());
00752             V.push_back(vector<LONG>());
00753         }
00754         M.back().push_back(vector<LONG>());
00755         V.back().push_back(0);
00756         maxMsize = max(maxMsize, (int)M.back().size());
00757     }
00758 
00759     // generate linear equations
00760     for (int numg=0; numg<maxMsize; numg++) {
00761 
00762         poly amodI(sparse_interpolation_fix_poly(ared,x[0]));
00763         poly bmodI(sparse_interpolation_fix_poly(bred,x[0]));
00764         poly lmodI(sparse_interpolation_fix_poly(lred,x[0]));
00765 
00766         // A fix for non-monic gcds. This could be slow if lmodI has many terms,
00767         // since it overfits the gcd now. Another gcd has to be run to remove the
00768         // extra terms.
00769         poly gcd(lmodI * gcd_Euclidean(amodI,bmodI));
00770 
00771         // if correct gcd
00772         if (!gcd.is_zero() && gcd[2+x[0]]==sred[2+x[0]]) {
00773 
00774             // for each power in the gcd, generate an equation if needed
00775             int gi=1, midx=0;
00776 
00777             for (int si=1; si<s[0];) {
00778                 // if the term exists, set Vi=coeff, otherwise Vi remains 0
00779                 if (gi<gcd[0] && gcd[gi+1+x[0]]==sred[si+1+x[0]]) {
00780                     if (numg < (int)V[midx].size())
00781                         V[midx][numg] = gcd[gi+gcd[gi]-1]*gcd[gi+gcd[gi]-2];
```

```
00782                     gi += gcd[gi];
00783                 }
00784
00785                 // add the coefficients of s to the matrix M
00786                 for (int i=0; i<(int)M[midx].size(); i++) {
00787                     if (numg < (int)M[midx].size())
00788                         M[midx][numg].push_back(sred[si+1+AN.poly_num_vars]);
00789                     si += s[si];
00790                 }
00791
00792                 midx++;
00793             }
00794         }
00795         else {
00796             // incorrect gcd
00797             if (!gcd.is_zero() && gcd[2+x[0]]<sred[2+x[0]])
00798                 return poly(BHEAD 0);
00799             numg--;
00800         }
00801
00802         // multiply polynomials by the lists to obtain new ones
00803         sparse_interpolation_mul_poly(ared,amul);
00804         sparse_interpolation_mul_poly(bred,bmul);
00805         sparse_interpolation_mul_poly(sred,smul);
00806         sparse_interpolation_mul_poly(lred,lmul);
00807     }
00808
00809     // solve the linear equations
00810     for (int i=0; i<(int)M.size(); i++) {
00811         int n = M[i].size();
00812
00813         // Gaussian elimination
00814         for (int j=0; j<n; j++) {
00815             for (int k=0; k<j; k++) {
00816                 LONG x = M[i][j][k];
00817                 for (int l=k; l<n; l++)
00818                     M[i][j][l] = (M[i][j][l] - M[i][k][l]*x) % a.modp;
00819                 V[i][j] = (V[i][j] - V[i][k]*x) % a.modp;
00820             }
00821
00822             // normalize row
00823             WORD x = M[i][j][j]; // WORD for GetModInverses
00824             GetModInverses(x + (x<0?a.modp:0), a.modp, &x, NULL);
00825             for (int k=0; k<n; k++)
00826                 M[i][j][k] = (M[i][j][k]*x) % a.modp;
00827             V[i][j] = (V[i][j]*x) % a.modp;
00828         }
00829
00830         // solve
00831         for (int j=n-1; j>=0; j--)
00832             for (int k=j+1; k<n; k++)
00833                 V[i][j] = (V[i][j] - M[i][j][k]*V[i][k]) % a.modp;
00834     }
00835
00836     // create coefficient list
00837     vector<LONG> coeff;
00838     for (int i=0; i<(int)V.size(); i++)
00839         for (int j=0; j<(int)V[i].size(); j++)
00840             coeff.push_back(V[i][j]);
00841
00842     // create resulting polynomial
00843     poly res(BHEAD 0);
00844     int ri=1, i=0;
00845     for (int si=1; si<s[0]; si+=s[si]) {
00846         res.check_memory(ri);
00847         res[ri] = 3 + AN.poly_num_vars;             // term length
00848         for (int j=0; j<AN.poly_num_vars; j++)
00849             res[ri+1+j] = s[si+1+j];                  // powers
00850         res[ri+1+AN.poly_num_vars] = ABS(coeff[i]); // coefficient
00851         res[ri+2+AN.poly_num_vars] = SGN(coeff[i]); // coefficient length
00852         i++;
00853         ri += res[ri];
00854     }
00855     res[0]=ri;                                    // total length
00856     res.setmod(a.modp,1);
00857
00858     if (!poly::divides(res, lcgcd * a) || !poly::divides(res, lcgcd * b)) {
00859         return poly(BHEAD 0); // bad shape
00860     } else {
00861         // refine gcd
00862         if (!poly::divides(res, a))
00863             res = gcd_modular_dense_interpolation(res, a, x, poly(BHEAD 0));
00864         if (!poly::divides(res, b))
00865             res = gcd_modular_dense_interpolation(res, b, x, poly(BHEAD 0));
00866     }
00867
00868 #ifdef DEBUG
```

```
00869     cout « "*** [" « thetime() « "]  RES : gcd_modular_sparse_interpolation("
00870              « a « "," « b « "," « x « "," « "," « s «") = " « res « endl;
00871 #endif
00872
00873     return gcdconts * res;
00874 }
00875
00876 /*
00877     #] gcd_modular_sparse_interpolation :
00878         #[ gcd_modular_dense_interpolation :
00879 */
00880
00899 const poly polygcd::gcd_modular_dense_interpolation (const poly &a, const poly &b, const vector<int>
      &x, const poly &s) {
00900
00901 #ifdef DEBUG
00902     cout « "*** [" « thetime() « "]  CALL: gcd_modular_dense_interpolation(" « a « "," « b « "," « x «
      "," « s «")" « endl;
00903 #endif
00904
00905     POLY_GETIDENTITY(a);
00906
00907     // if univariate, then use Euclidean algorithm
00908     if (x.size() == 1) {
00909         return gcd_Euclidean(a,b);
00910     }
00911
00912     // if shape is known, use sparse interpolation
00913     if (!s.is_zero()) {
00914         poly res = gcd_modular_sparse_interpolation (a,b,x,s);
00915         if (!res.is_zero()) return res;
00916         // apparently the shape was not correct. continue.
00917     }
00918
00919     // divide out multivariate content in last variable
00920     int X = x.back();
00921
00922     poly conta(content_multivar(a,X));
00923     poly contb(content_multivar(b,X));
00924     poly gcdconts(gcd_Euclidean(conta,contb));
00925     const poly& ppa = conta.is_one() ? a : poly(a/conta);
00926     const poly& ppb = contb.is_one() ? b : poly(b/contb);
00927
00928     // gcd of leading coefficients
00929     poly lcoeffa(ppa.lcoeff_multivar(X));
00930     poly lcoeffb(ppb.lcoeff_multivar(X));
00931     poly gcdlcoeffs(gcd_Euclidean(lcoeffa,lcoeffb));
00932
00933     // calculate the degree bound for each variable
00934     int m = MiN(ppa.degree(x[x.size() - 2]),ppb.degree(x[x.size() - 2]));
00935
00936     poly res(BHEAD 0);
00937     poly oldres(BHEAD 0);
00938     poly newshape(BHEAD 0);
00939     poly modpoly(BHEAD 1,a.modp);
00940
00941     while (true) {
00942         // generate random constants and substitute it
00943         int c = 1 + wranf(BHEAD0) % (a.modp-1);
00944         if (substitute(gcdlcoeffs,X,c).is_zero()) continue;
00945         if (substitute(modpoly,X,c).is_zero()) continue;
00946
00947         poly amodc(substitute(ppa,X,c));
00948         poly bmodc(substitute(ppb,X,c));
00949
00950         // calculate gcd recursively
00951         poly gcdmodc(gcd_modular_dense_interpolation(amodc,bmodc,vector<int>(x.begin(),x.end()-1),
      newshape));
00952         int n = gcdmodc.degree(x[x.size() - 2]);
00953
00954         // normalize
00955         gcdmodc = (gcdmodc * substitute(gcdlcoeffs,X,c)) / gcdmodc.integer_lcoeff();
00956         poly simple(poly::simple_poly(BHEAD X,c,1,a.modp)); // (X-c) mod p
00957
00958         // if power is smaller, the old one was wrong
00959         if ((res.is_zero() && n == m) || n < m) {
00960             m = n;
00961             res = gcdmodc;
00962             newshape = gcdmodc; // set a new shape (interpolation does not change it)
00963             modpoly = simple;
00964         }
00965         else if (n == m) {
00966             oldres = res;
00967             // equal powers, so interpolate results
00968             poly coeff_poly(substitute(modpoly,X,c));
00969             WORD coeff_word = coeff_poly[2+AN.poly_num_vars] * coeff_poly[3+AN.poly_num_vars];
00970             if (coeff_word < 0) coeff_word += a.modp;
```

```
00971
00972             GetModInverses(coeff_word, a.modp, &coeff_word, NULL);
00973
00974             res.setmod(a.modp); // make sure the mod is set before substituting
00975             res += poly(BHEAD coeff_word, a.modp, 1) * modpoly * (gcdmodc - substitute(res,X,c));
00976             modpoly *= simple;
00977         }
00978
00979         // check whether this is the complete gcd
00980         if (!res.is_zero() && res == oldres) {
00981             poly nres = res / content_multivar(res, X);
00982             if (poly::divides(nres,ppa) && poly::divides(nres,ppb)) {
00983 #ifdef DEBUG
00984                 cout « "*** [" « thetime() « "]  RES : gcd_modular_dense_interpolation(" « a « "," « b
      « ","
00985                         « x « "," « "," « s «") = " « gcdconts * nres « endl;
00986 #endif
00987                 return gcdconts * nres;
00988             }
00989
00990             // At this point, the gcd may be too large due to bad luck
00991             // TODO: create an efficient fail state that tries to find a smaller
00992             // polynomial without interpolating bad ones?
00993             newshape = poly(BHEAD 0); // reset the shape, important!
00994         }
00995     }
00996 }
00997
00998 /*
00999     #] gcd_modular_dense_interpolation : :
01000         #[ gcd_modular :
01001 */
01002
01019 const poly polygcd::gcd_modular (const poly &origa, const poly &origb, const vector<int> &x) {
01020
01021 #ifdef DEBUG
01022     cout « "*** [" « thetime() « "]  CALL: gcd_modular(" « origa « "," « origb « "," « x « ")" « endl;
01023 #endif
01024
01025     POLY_GETIDENTITY(origa);
01026
01027     if (origa.is_zero()) return origa.modp==0 ? origb : origb / origb.integer_lcoeff();
01028     if (origb.is_zero()) return origa.modp==0 ? origa : origa / origa.integer_lcoeff();
01029     if (origa==origb) return origa.modp==0 ? origa : origa / origa.integer_lcoeff();
01030
01031     poly ac = integer_content(origa);
01032     poly bc = integer_content(origb);
01033     const poly& a = ac.is_one() ? origa : poly(origa/ac);
01034     const poly& b = bc.is_one() ? origb : poly(origb/bc);
01035     poly ic = integer_gcd(ac, bc);
01036     poly g = integer_gcd(a.integer_lcoeff(), b.integer_lcoeff());
01037
01038     int pnum=0;
01039
01040     poly d(BHEAD 0);
01041     poly m1(BHEAD 1);
01042     int mindeg=MAXPOSITIVE;
01043
01044     while (true) {
01045         // choose a prime and solve modulo the prime
01046         WORD p = NextPrime(BHEAD pnum++);
01047         if (poly(a.integer_lcoeff(),p).is_zero()) continue;
01048         if (poly(b.integer_lcoeff(),p).is_zero()) continue;
01049
01050         poly c(gcd_modular_dense_interpolation(poly(a,p),poly(b,p),x,poly(d,p)));
01051         c = (c * poly(g,p)) / c.integer_lcoeff(); // normalize so that lcoeff(c) = g mod p
01052
01053         if (c.is_zero()) {
01054             // unlucky choices somewhere, so start all over again
01055             d = poly(BHEAD 0);
01056             m1 = poly(BHEAD 1);
01057             mindeg = MAXPOSITIVE;
01058             continue;
01059         }
01060
01061         if (!(poly(a,p)%c).is_zero()) continue;
01062         if (!(poly(b,p)%c).is_zero()) continue;
01063
01064         int deg = c.degree(x[0]);
01065
01066         if (deg < mindeg) {
01067             // small degree, so the old one is wrong
01068             d=c;
01069             d.modp=a.modp;
01070             d.modn=a.modn;
01071             m1 = poly(BHEAD p);
01072             mindeg=deg;
```

```
01073         }
01074         else if (deg == mindeg) {
01075             // same degree, so use Chinese Remainder Algorithm
01076             poly newd(BHEAD 0);
01077 
01078             for (int ci=1,di=1; ci<c[0]||di<d[0]; ) {
01079                 int comp = ci==c[0] ? -1 : di==d[0] ? +1 : poly::monomial_compare(BHEAD
      &c[ci],&d[di]);
01080                 poly a1(BHEAD 0),a2(BHEAD 0);
01081 
01082                 newd.check_memory(newd[0]);
01083 
01084                 if (comp <= 0) {
01085                     newd.termscopy(&d[di],newd[0],1+AN.poly_num_vars);
01086                     a1 = poly(BHEAD (UWORD *)&d[di+1+AN.poly_num_vars],d[di+d[di]-1]);
01087                     di+=d[di];
01088                 }
01089                 if (comp >= 0) {
01090                     newd.termscopy(&c[ci],newd[0],1+AN.poly_num_vars);
01091                     a2 = poly(BHEAD (UWORD *)&c[ci+1+AN.poly_num_vars],c[ci+c[ci]-1]);
01092                     ci+=c[ci];
01093                 }
01094 
01095                 poly e(chinese_remainder(a1,m1,a2,poly(BHEAD p)));
01096                 newd.termscopy(&e[2+AN.poly_num_vars], newd[0]+1+AN.poly_num_vars, ABS(e[e[1]])+1);
01097                 newd[newd[0]] = 2 + AN.poly_num_vars + ABS(e[e[1]]);
01098                 newd[0] += newd[newd[0]];
01099             }
01100 
01101             m1 *= poly(BHEAD p);
01102             d=newd;
01103         }
01104 
01105         // divide out spurious integer content
01106         poly ppd(d / integer_content(d));
01107 
01108         // check whether this is the complete gcd
01109         if (poly::divides(ppd,a) && poly::divides(ppd,b)) {
01110 #ifdef DEBUG
01111             cout « "*** [" « thetime() « "]  RES : gcd_modular(" « origa « "," « origb « "," « x « ")
      = "
01112                  « ic * ppd « endl;
01113 #endif
01114             return ic * ppd;
01115         }
01116 #ifdef DEBUG
01117         cout « "*** [" « thetime() « "] Retrying modular_gcd with new prime" « endl;
01118 #endif
01119     }
01120 }
01121 
01122 /*
01123     #] gcd_modular :
01124     #[ gcd_heuristic_possible :
01125 */
01126 
01142 bool gcd_heuristic_possible (const poly &a) {
01143 
01144     POLY_GETIDENTITY(a);
01145 
01146     double prod_deg = 1;
01147     for (int j=0; j<AN.poly_num_vars; j++)
01148         prod_deg *= a[2+j]+1;
01149 
01150     double digits = ABS(a[1+a[1]-1]);
01151   double lead = a[1+1+AN.poly_num_vars];
01152 
01153     return prod_deg*(digits-1+log(2*ABS(lead))/log(2.0)/(BITSINWORD/2)) <
      POLYGCD_HEURISTIC_MAX_DIGITS;
01154 }
01155 
01156 /*
01157     #] gcd_heuristic_possible :
01158     #[ gcd_heuristic :
01159 */
01160 
01161 
01188 const poly polygcd::gcd_heuristic (const poly &a, const poly &b, const vector<int> &x, int max_tries)
      {
01189 
01190 #ifdef DEBUG
01191     cout « "*** [" « thetime() « "]  CALL: gcd_heuristic("«a«","«b«","«x«")\n";
01192 #endif
01193 
01194     if (a.is_integer()) return integer_gcd(a,integer_content(b));
01195     if (b.is_integer()) return integer_gcd(integer_content(a),b);
01196 
```

```
01197     POLY_GETIDENTITY(a);
01198
01199     // Calculate xi = 2*min(max(coefficients a),max(coefficients b))+2
01200
01201     UWORD *pxi=NULL;
01202     WORD nxi=0;
01203
01204     for (int i=1; i<a[0]; i+=a[i]) {
01205         WORD na = ABS(a[i+a[i]-1]);
01206         if (BigLong((UWORD *)&a[i+a[i]-1-na], na, pxi, nxi) > 0) {
01207             pxi = (UWORD *)&a[i+a[i]-1-na];
01208             nxi = na;
01209         }
01210     }
01211
01212     for (int i=1; i<b[0]; i+=b[i]) {
01213         WORD nb = ABS(b[i+b[i]-1]);
01214         if (BigLong((UWORD *)&b[i+b[i]-1-nb], nb, pxi, nxi) > 0) {
01215             pxi = (UWORD *)&b[i+b[i]-1-nb];
01216             nxi = nb;
01217         }
01218     }
01219
01220     poly xi(BHEAD pxi,nxi);
01221
01222     // Addition of another random factor gives better performance
01223     xi = xi*poly(BHEAD 2) + poly(BHEAD 2 + wranf(BHEAD0)%POLYGCD_HEURISTIC_MAX_ADD_RANDOM);
01224
01225     // If degree*digits(xi) is too large, throw exception
01226     if (max(a.degree(x[0]),b.degree(x[0])) * xi[xi[1]] >= MiN(AM.MaxTal,
      POLYGCD_HEURISTIC_MAX_DIGITS)) {
01227 #ifdef DEBUG
01228         cout « "*** [" « thetime() « "]  RES : gcd_heuristic("«a«","«b«","«x«") = overflow\n";
01229 #endif
01230         throw(gcd_heuristic_failed());
01231     }
01232
01233     for (int times=0; times<max_tries; times++) {
01234
01235         // Recursively calculate the gcd
01236
01237         poly gamma(gcd_heuristic(a % poly::simple_poly(BHEAD x[0],xi),
01238                                                          b % poly::simple_poly(BHEAD x[0],xi),
01239                                                          vector<int>(x.begin()+1,x.end()),1));
01240
01241         // If a gcd is found, reconstruct the powers of x
01242         if (!gamma.is_zero()) {
01243             // res is construct is reverse order. idx/len are for reversing
01244             // it in the correct order
01245             poly res(BHEAD 0), c(BHEAD 0);
01246             vector<int> idx, len;
01247
01248             for (int power=0; !gamma.is_zero(); power++) {
01249
01250                 // calculate c = gamma % xi (c and gamma are polynomials, xi is integer)
01251                 c = gamma;
01252                 c.coefficients_modulo((UWORD *)&xi[2+AN.poly_num_vars], xi[xi[0]-1], false);
01253
01254                 // Add the terms c * x^power to res
01255                 res.check_memory(res[0]+c[0]);
01256                 res.termscopy(&c[1],res[0],c[0]-1);
01257                 for (int i=1; i<c[0]; i+=c[i])
01258                     res[res[0]-1+i+1+x[0]] = power;
01259
01260                 // Store idx/len for reversing
01261                 if (!c.is_zero()) {
01262                     idx.push_back(res[0]);
01263                     len.push_back(c[0]-1);
01264                 }
01265
01266                 res[0] += c[0]-1;
01267
01268                 // Divide gamma by xi
01269                 gamma = (gamma - c) / xi;
01270             }
01271
01272             // Reverse the resulting polynomial
01273             poly rev(BHEAD 0);
01274             rev.check_memory(res[0]);
01275
01276             rev[0] = 1;
01277             for (int i=idx.size()-1; i>=0; i--) {
01278                 rev.termscopy(&res[idx[i]], rev[0], len[i]);
01279                 rev[0] += len[i];
01280             }
01281             res = rev;
01282
```

```
01283             poly ppres = res / integer_content(res);
01284
01285             if ((a%ppres).is_zero() && (b%ppres).is_zero()) {
01286 #ifdef DEBUG
01287                 cout « "*** [" « thetime() « "]  RES : gcd_heuristic("«a«","«b«","«x«") = "«res«"\n";
01288 #endif
01289                 return res;
01290             }
01291         }
01292
01293         // Next xi by multiplying with the golden ratio to avoid correlated errors
01294         xi = xi * poly(BHEAD 28657) / poly(BHEAD 17711) + poly(BHEAD wranf(BHEAD0) %
      POLYGCD_HEURISTIC_MAX_ADD_RANDOM);
01295     }
01296
01297 #ifdef DEBUG
01298     cout « "*** [" « thetime() « "]  RES : gcd_heuristic("«a«","«b«","«x«") = failed\n";
01299 #endif
01300
01301     return poly(BHEAD 0);
01302 }
01303
01304 /*
01305     #] gcd_heuristic :
01306     #[ bracket :
01307 */
01308
01309 const map<vector<int>,poly> polygcd::full_bracket(const poly &a, const vector<int>& filter) {
01310     POLY_GETIDENTITY(a);
01311
01312     map<vector<int>,poly> bracket;
01313     for (int ai=1; ai<a[0]; ai+=a[ai]) {
01314         vector<int> varpattern(AN.poly_num_vars);
01315         for (int i=0; i<AN.poly_num_vars; i++)
01316             if (filter[i] == 1 && a[ai + i + 1] > 0)
01317                 varpattern[i] = a[ai + i + 1];
01318
01319         // create monomial
01320         poly mon(BHEAD 1);
01321         mon.setmod(a.modp);
01322         mon[0] = a[ai] + 1;
01323         for (int i=0; i<a[ai]; i++)
01324             if (i > 0 && i <= AN.poly_num_vars && varpattern[i - 1])
01325                 mon[1+i] = 0;
01326             else
01327                 mon[1+i] = a[ai+i];
01328
01329         map<vector<int>,poly>::iterator i = bracket.find(varpattern);
01330         if (i == bracket.end()) {
01331             bracket.insert(std::make_pair(varpattern, mon));
01332         } else {
01333             i->second += mon;
01334         }
01335     }
01336
01337     return bracket;
01338 }
01339
01340 const poly polygcd::bracket(const poly &a, const vector<int>& pattern, const vector<int>& filter) {
01341     POLY_GETIDENTITY(a);
01342
01343     poly bracket(BHEAD 0);
01344     for (int ai=1; ai<a[0]; ai+=a[ai]) {
01345         bool ispat = true;
01346         for (int i=0; i<AN.poly_num_vars; i++)
01347             if (filter[i] == 1 && pattern[i] != a[ai + i + 1]) {
01348                 ispat = false;
01349                 break;
01350             }
01351
01352         if (ispat) {
01353             poly mon(BHEAD 1);
01354             mon.setmod(a.modp);
01355             mon[0] = a[ai] + 1;
01356             for (int i=0; i<a[ai]; i++)
01357                 if (i > 0 && i <= AN.poly_num_vars && pattern[i - 1])
01358                     mon[1+i] = 0;
01359                 else
01360                     mon[1+i] = a[ai+i];
01361             bracket += mon;
01362         }
01363     }
01364
01365     return bracket;
01366 }
01367
01368 const map<vector<int>,int> polygcd::bracket_count(const poly &a, const vector<int>& filter) {
```

```
01369     POLY_GETIDENTITY(a);
01370
01371     map<vector<int>,int> bracket;
01372     for (int ai=1; ai<a[0]; ai+=a[ai]) {
01373         vector<int> varpattern(AN.poly_num_vars);
01374         for (int i=0; i<AN.poly_num_vars; i++)
01375             if (filter[i] == 1 && a[ai + i + 1] > 0)
01376                 varpattern[i] = a[ai + i + 1];
01377
01378         map<vector<int>,int>::iterator i = bracket.find(varpattern);
01379         if (i == bracket.end()) {
01380             bracket.insert(std::make_pair(varpattern, 0));
01381         } else {
01382             i->second++;
01383         }
01384     }
01385
01386     return bracket;
01387 }
01388
01389 struct BracketInfo {
01390     std::vector<int> pattern;
01391     int num_terms, dummy = 0;
01392     const poly* p;
01393
01394     BracketInfo(const std::vector<int>& pattern, int num_terms, const poly* p) : pattern(pattern),
      num_terms(num_terms), p(p) {}
01395     bool operator<(const BracketInfo& rhs) const { return num_terms > rhs.num_terms; } // biggest
      should be first!
01396 };
01397
01398 /*
01399     #] bracket :
01400     #[ gcd_linear:
01401 */
01402
01403 const poly gcd_linear_helper (const poly &a, const poly &b) {
01404     POLY_GETIDENTITY(a);
01405
01406     for (int i = 0; i < AN.poly_num_vars; i++)
01407         if (a.degree(i) == 1) {
01408             vector<int> filter(AN.poly_num_vars);
01409             filter[i] = 1;
01410
01411             // bracket the linear variable
01412             map<vector<int>,poly> ba = polygcd::full_bracket(a, filter);
01413
01414             poly subgcd(BHEAD 1);
01415             if (ba.size() == 2)
01416                 subgcd = polygcd::gcd_linear(ba.begin()->second, (++ba.begin())->second);
01417             else
01418                 subgcd = ba.begin()->second;
01419
01420             poly linfac = a / subgcd;
01421             if (poly::divides(linfac,b))
01422                 return linfac * polygcd::gcd_linear(subgcd, b / linfac);
01423
01424             return polygcd::gcd_linear(subgcd, b);
01425         }
01426
01427     return poly(BHEAD 0);
01428 }
01429
01434 const poly polygcd::gcd_linear (const poly &a, const poly &b) {
01435 #ifdef DEBUG
01436     cout « "*** [" « thetime() « "]  CALL: gcd_linear("«a«","«b«")\n";
01437 #endif
01438
01439     POLY_GETIDENTITY(a);
01440
01441     if (a.is_zero()) return a.modp==0 ? b : b / b.integer_lcoeff();
01442     if (b.is_zero()) return a.modp==0 ? a : a / a.integer_lcoeff();
01443     if (a==b) return a.modp==0 ? a : a / a.integer_lcoeff();
01444
01445     if (a.is_integer() || b.is_integer()) {
01446         if (a.modp > 0) return poly(BHEAD 1,a.modp,a.modn);
01447         return poly(integer_gcd(integer_content(a),integer_content(b)),0,1);
01448     }
01449
01450     poly h = gcd_linear_helper(a, b);
01451     if (!h.is_zero()) return h;
01452
01453     h = gcd_linear_helper(b, a);
01454     if (!h.is_zero()) return h;
01455
01456     vector<int> xa = a.all_variables();
01457     vector<int> xb = b.all_variables();
```

```
01458
01459     vector<int> used(AN.poly_num_vars,0);
01460     for (int i=0; i<(int)xa.size(); i++) used[xa[i]]++;
01461     for (int i=0; i<(int)xb.size(); i++) used[xb[i]]++;
01462     vector<int> x;
01463     for (int i=0; i<AN.poly_num_vars; i++)
01464         if (used[i]) x.push_back(i);
01465
01466     return gcd_modular(a,b,x);
01467 }
01468
01469 /*
01470     #] gcd_linear:
01471     #[ gcd :
01472 */
01473
01494 const poly polygcd::gcd (const poly &a, const poly &b) {
01495
01496 #ifdef DEBUG
01497     cout « "*** [" « thetime() « "]  CALL: gcd("«a«","«b«")\n";
01498 #endif
01499
01500     POLY_GETIDENTITY(a);
01501
01502     if (a.is_zero()) return a.modp==0 ? b : b / b.integer_lcoeff();
01503     if (b.is_zero()) return a.modp==0 ? a : a / a.integer_lcoeff();
01504     if (a==b) return a.modp==0 ? a : a / a.integer_lcoeff();
01505
01506     if (a.is_integer() || b.is_integer()) {
01507         if (a.modp > 0) return poly(BHEAD 1,a.modp,a.modn);
01508         return poly(integer_gcd(integer_content(a),integer_content(b)),0,1);
01509     }
01510
01511     // Generate a list of variables of a and b
01512     vector<int> xa = a.all_variables();
01513     vector<int> xb = b.all_variables();
01514
01515     vector<int> used(AN.poly_num_vars,0);
01516     for (int i=0; i<(int)xa.size(); i++) used[xa[i]]++;
01517     for (int i=0; i<(int)xb.size(); i++) used[xb[i]]++;
01518     vector<int> x;
01519     for (int i=0; i<AN.poly_num_vars; i++)
01520         if (used[i]) x.push_back(i);
01521
01522     if (a.is_monomial() || b.is_monomial()) {
01523
01524         poly res(BHEAD 1,a.modp,a.modn);
01525         if (a.modp == 0) res = integer_gcd(integer_content(a),integer_content(b));
01526
01527         for (int i=0; i<(int)x.size(); i++)
01528             res[2+x[i]] = 1«(BITSINWORD-2);
01529
01530         for (int i=1; i<a[0]; i+=a[i])
01531             for (int j=0; j<(int)x.size(); j++)
01532                 res[2+x[j]] = MiN(res[2+x[j]], a[i+1+x[j]]);
01533
01534         for (int i=1; i<b[0]; i+=b[i])
01535             for (int j=0; j<(int)x.size(); j++)
01536                 res[2+x[j]] = MiN(res[2+x[j]], b[i+1+x[j]]);
01537
01538         return res;
01539     }
01540
01541     // Calculate the contents, their gcd and the primitive parts
01542     poly conta(x.size()==1 ? integer_content(a) : content_univar(a,x[0]));
01543     poly contb(x.size()==1 ? integer_content(b) : content_univar(b,x[0]));
01544     poly gcdconts(x.size()==1 ? integer_gcd(conta,contb) : gcd(conta,contb));
01545     const poly& ppa = conta.is_one() ? a : poly(a/conta);
01546     const poly& ppb = contb.is_one() ? b : poly(b/contb);
01547
01548     if (ppa == ppb)
01549         return ppa * gcdconts;
01550
01551     poly res(BHEAD 0);
01552
01553 #ifdef POLYGCD_USE_HEURISTIC
01554     // Try the heuristic gcd algorithm
01555     if (a.modp==0 && gcd_heuristic_possible(a) && gcd_heuristic_possible(b)) {
01556         try {
01557             res = gcd_heuristic(ppa,ppb,x);
01558             if (!res.is_zero()) res /= integer_content(res);
01559         }
01560         catch (gcd_heuristic_failed) {}
01561     }
01562 #endif
01563
01564     // If gcd==0, the heuristic algorithm failed, so we do more extensive checks.
```

```
01565     // First, we filter out variables that appear in only one of the expressions.
01566     if (res.is_zero()) {
01567         bool unusedVars = false;
01568         for (unsigned int i = 0; i < used.size(); i++) {
01569             if (used[i] == 1) {
01570                 unusedVars = true;
01571                 break;
01572             }
01573         }
01574 
01575         // if there are no unused variables, go to the linear routine directly
01576         if (!unusedVars) {
01577             res = gcd_linear(ppa,ppb);
01578 #ifdef DEBUG
01579             cout « "New GCD attempt (unused vars): " « res « endl;
01580 #endif
01581         }
01582 
01583         // if res is not the gcd, it is 0 or larger than the gcd.
01584         // we bracket the expression in all the variables that appear only in one expr.
01585         // and we refine the gcd.
01586         bool diva = !res.is_zero() && poly::divides(res,ppa);
01587         bool divb = !res.is_zero() && poly::divides(res,ppb);
01588         if (!diva || !divb) {
01589             vector<BracketInfo> bracketinfo;
01590 
01591             if (!diva) {
01592                 map<vector<int>,int> ba = bracket_count(ppa, used);
01593                 for(map<vector<int>,int>::iterator it = ba.begin(); it != ba.end(); it++)
01594                     bracketinfo.push_back(BracketInfo(it->first, it->second, &ppa));
01595             }
01596 
01597             if (!divb) {
01598                 map<vector<int>,int> bb = bracket_count(ppb, used);
01599                 for(map<vector<int>,int>::iterator it = bb.begin(); it != bb.end(); it++)
01600                     bracketinfo.push_back(BracketInfo(it->first, it->second, &ppb));
01601             }
01602 
01603             // sort so that the smallest bracket will be last
01604             sort(bracketinfo.begin(), bracketinfo.end());
01605 
01606             if (res.is_zero()) {
01607                 res = bracket(*bracketinfo.back().p, bracketinfo.back().pattern, used);
01608                 bracketinfo.pop_back();
01609             }
01610 
01611             while (bracketinfo.size() > 0) {
01612                 poly subpoly(bracket(*bracketinfo.back().p, bracketinfo.back().pattern, used));
01613                 if (!poly::divides(res,subpoly)) {
01614                     // if we can filter out more variables, call gcd again
01615                     if (res.all_variables() != subpoly.all_variables())
01616                         res = gcd(subpoly,res);
01617                     else
01618                         res = gcd_linear(subpoly,res);
01619                 }
01620 
01621                 bracketinfo.pop_back();
01622             }
01623         }
01624 
01625         if (res.is_zero() || !poly::divides(res,ppa) || !poly::divides(res,ppb)) {
01626 /* INTERNAL_ERROR_EXCL_START */
01627             MesPrint("!>Bad gcd found.");
01628             std::cout « "Bad gcd:" « res « " for " « ppa « " " « ppb « std::endl;
01629             Terminate(1);
01630 /* INTERNAL_ERROR_EXCL_STOP */
01631         }
01632     }
01633 
01634     res *= gcdconts * poly(BHEAD res.sign());
01635 
01636 #ifdef DEBUG
01637     cout « "*** [" « thetime() « "]  RES : gcd("«a«","«b«") = "«res«endl;
01638 #endif
01639 
01640     return res;
01641 }
01642 
01643 /*
01644     #] gcd :
01645 */
```

4.110 polygcd.h

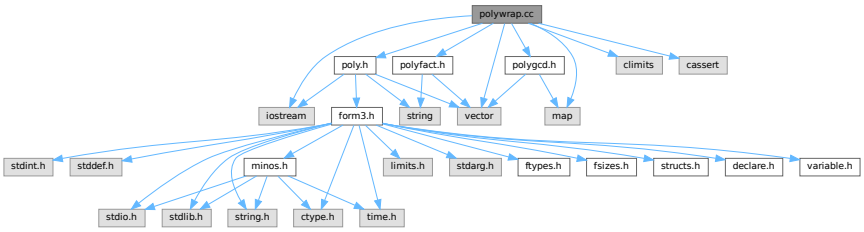
```
00001 #pragma once
00002 /* #[ License : */
00003 /*
00004  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00005  *   When using this file you are requested to refer to the publication
00006  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00007  *   This is considered a matter of courtesy as the development was paid
00008  *   for by FOM the Dutch physics granting agency and we would like to
00009  *   be able to track its scientific use to convince FOM of its value
00010  *   for the community.
00011  *
00012  *   This file is part of FORM.
00013  *
00014  *   FORM is free software: you can redistribute it and/or modify it under the
00015  *   terms of the GNU General Public License as published by the Free Software
00016  *   Foundation, either version 3 of the License, or (at your option) any later
00017  *   version.
00018  *
00019  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00020  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00021  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00022  *   details.
00023  *
00024  *   You should have received a copy of the GNU General Public License along
00025  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00026  */
00027 /* #] License : */
00028
00029 #include <vector>
00030 #include <map>
00031
00032 class poly; // polynomial class
00033 class gcd_heuristic_failed {}; // class for throwing exceptions
00034
00035 // whether or not to use the heuristic before Zippel's algorithm
00036 #define POLYGCD_USE_HEURISTIC
00037
00038 // maximum number of words in a coefficient for gcd_heuristic to continue
00039 const int POLYGCD_HEURISTIC_MAX_DIGITS = 1000;
00040
00041 // maximum number of retries after the heuristic has failed
00042 const int POLYGCD_HEURISTIC_MAX_TRIES = 10;
00043
00044 // a fudge factor, which improves efficiency
00045 const int POLYGCD_HEURISTIC_MAX_ADD_RANDOM = 10;
00046
00047 // maximum cached power in substitute_last and sparse_interpolation_get_mul_list
00048 const int POLYGCD_RAISPOWMOD_CACHE_SIZE = 1000;
00049
00050 namespace polygcd {
00051
00052     // functions to call the gcd routines
00053     const poly integer_gcd (const poly &a, const poly &b);
00054     const poly integer_content (const poly &a);
00055
00056     const poly gcd (const poly &a, const poly &b);
00057     const poly content_univar (const poly &a, int x);
00058     const poly content_multivar (const poly &a, int x);
00059
00060     const std::vector<WORD> coefficient_list_gcd (const std::vector<WORD> &a, const std::vector<WORD>
      &b, WORD p);
00061
00062     // internal functions
00063     const poly gcd_heuristic (const poly &a, const poly &b, const std::vector<int> &x, int
      max_tries=POLYGCD_HEURISTIC_MAX_TRIES);
00064     const poly gcd_Euclidean (const poly &a, const poly &b);
00065     const poly gcd_modular (const poly &a, const poly &b, const std::vector<int> &x);
00066     const poly gcd_modular_dense_interpolation (const poly &a, const poly &b, const std::vector<int>
      &x, const poly &s);
00067     const poly gcd_modular_sparse_interpolation (const poly &a, const poly &b, const std::vector<int>
      &x, const poly &s);
00068
00069     const std::vector<int> sparse_interpolation_get_mul_list (const poly &a, const std::vector<int>
      &x, const std::vector<int> &c);
00070     void sparse_interpolation_mul_poly (poly &a, const std::vector<int> &m);
00071     const poly sparse_interpolation_reduce_poly (const poly &a, const std::vector<int> &x);
00072     const poly sparse_interpolation_fix_poly (const poly &a, int x);
00073
00074     const poly chinese_remainder (const poly &a1, const poly &m1, const poly &a2, const poly &m2);
00075     const poly substitute(const poly &a, int x, int c);
00076     const std::map<std::vector<int>,poly> full_bracket(const poly &a, const std::vector<int>& filter);
00077     const poly bracket(const poly &a, const std::vector<int>& pattern, const std::vector<int>&
      filter);
00078     const std::map<std::vector<int>,int> bracket_count(const poly &a, const std::vector<int>& filter);
00079     const poly gcd_linear (const poly &a, const poly &b);
```

```
00080 }
```

4.111 polywrap.cc File Reference

```
#include "poly.h"
#include "polygcd.h"
#include "polyfact.h"
#include <iostream>
#include <vector>
#include <map>
#include <climits>
#include <cassert>
```

Include dependency graph for polywrap.cc:



Functions

- WORD poly_determine_modulus (PHEAD bool multi_error, bool is_fun_arg, string message)
- WORD * poly_gcd (PHEAD WORD *a, WORD *b, WORD fit)
- WORD * poly_divmod (PHEAD WORD *a, WORD *b, int divmod, WORD fit)
- WORD * poly_div (PHEAD WORD *a, WORD *b, WORD fit)
- WORD * poly_rem (PHEAD WORD *a, WORD *b, WORD fit)
- void poly_ratfun_read (WORD *a, poly &num, poly &den)
- void poly_sort (PHEAD WORD *a)
- WORD * poly_ratfun_add (PHEAD WORD *t1, WORD *t2)
- int poly_ratfun_normalize (PHEAD WORD *term)
- void poly_fix_minus_signs (factorized_poly &a)
- WORD * poly_factorize (PHEAD WORD *argin, WORD *argout, bool with_arghead, bool is_fun_arg)
- int poly_factorize_argument (PHEAD WORD *argin, WORD *argout)
- WORD * poly_factorize_dollar (PHEAD WORD *argin)
- int poly_factorize_expression (EXPRESSIONS expr)
- int poly_unfactorize_expression (EXPRESSIONS expr)
- WORD * poly_inverse (PHEAD WORD *arga, WORD *argb)
- WORD * poly_mul (PHEAD WORD *a, WORD *b)
- void poly_free_poly_vars (PHEAD const char *text)

Variables

- const int POLYWRAP_DENOMPOWER_INCREASE_FACTOR = 2

4.111.1 Detailed Description

Contains methods to call the polynomial methods (written in C++) from the rest of Form (written in C). These include polynomial gcd computation, factorization and polyratfuns.

Definition in file polywrap.cc.

4.111.2 Function Documentation

4.111.2.1 poly_determine_modulus()

```
WORD poly_determine_modulus (
            PHEAD bool multi_error,
            bool is_fun_arg,
            string message )
```

Modulus for polynomial algebra

4.111.3 Description

This method determines whether polynomial algebra is done with a modulus or not. This depends on AC.ncmod. If only_funargs is set it also depends on (AC.modmode & ALSOFUNARGS).

The program terminates if the feature is not implemented. Polynomial algebra modulo M $>$ WORDSIZE in not implemented. If multi_error is set, multivariate algebra mod M is not implemented.

4.111.4 Notes

- If AC.ncmod$>$0 and only_funargs=true and AC.modmode&ALSOFUNARGS=false, AN.ncmod is set to zero, for otherwise RaisPow calculates mod M.

Definition at line 79 of file polywrap.cc.

Referenced by poly_factorize(), poly_factorize_expression(), poly_gcd(), poly_ratfun_add(), and poly_ratfun_normalize().

4.111.4.1 poly_gcd()

```
WORD * poly_gcd (
            PHEAD WORD * a,
            WORD * b,
            WORD fit )
```

Polynomial gcd

4.111.5 Description

This method calculates the greatest common divisor of two polynomials, given by two zero-terminated Form-style term lists.

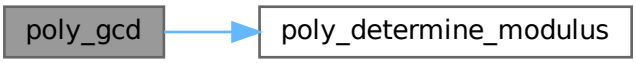
4.111.6 Notes

- The result is written at newly allocated memory
- Called from ratio.c
- Calls polygcd::gcd

Definition at line 124 of file polywrap.cc.

References poly_determine_modulus().

Here is the call graph for this function:



4.111.6.1 poly_divmod()

```
WORD * poly_divmod (
              PHEAD WORD * a,
              WORD * b,
              int divmod,
              WORD fit )
```

Definition at line 203 of file polywrap.cc.

4.111.6.2 poly_div()

```
WORD * poly_div (
              PHEAD WORD * a,
              WORD * b,
              WORD fit )
```

Definition at line 437 of file polywrap.cc.

4.111.6.3 poly_rem()

```
WORD * poly_rem (
              PHEAD WORD * a,
              WORD * b,
              WORD fit )
```

Definition at line 465 of file polywrap.cc.

4.111.6.4 poly_ratfun_read()

```
void poly_ratfun_read (
             WORD * a,
             poly & num,
             poly & den )
```

Read a PolyRatFun

4.111.7 Description

This method reads a polyratfun starting at the pointer a. The resulting numerator and denominator are written in num and den. If MUSTCLEANPRF, the result is normalized.

4.111.8 Notes

- Calls polygcd::gcd

Definition at line 498 of file polywrap.cc.

Referenced by poly_ratfun_add(), and poly_ratfun_normalize().

4.111.8.1 poly_sort()

```
void poly_sort (
             PHEAD WORD * a )
```

Sort the polynomial terms

4.111.9 Description

Sorts the terms of a polynomial in Form poly(rat)fun order, i.e. lexicographical order with highest degree first.

4.111.10 Notes

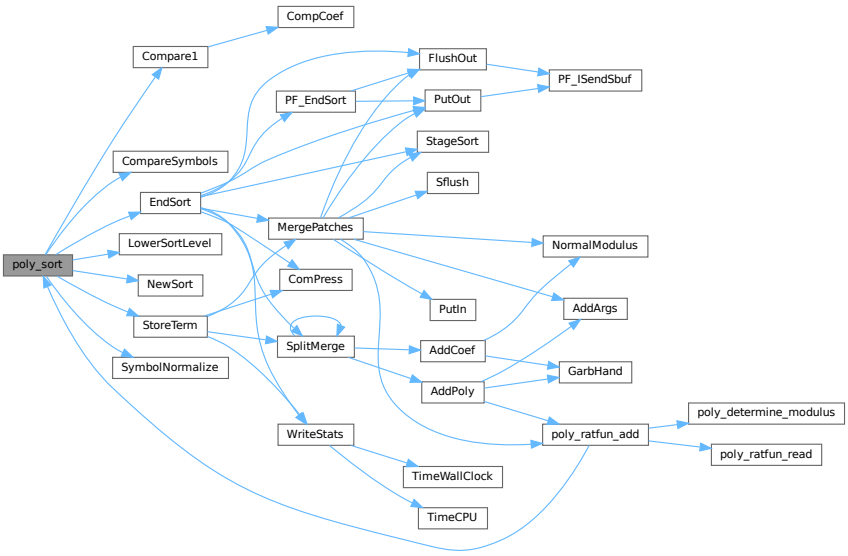
- Uses Form sort routines with custom compare

Definition at line 592 of file polywrap.cc.

References Compare1(), CompareSymbols(), EndSort(), LowerSortLevel(), NewSort(), StoreTerm(), and SymbolNormalize().

Referenced by poly_ratfun_add(), and poly_ratfun_normalize().

Here is the call graph for this function:



4.111.10.1 poly_ratfun_add()

```
WORD * poly_ratfun_add (
            PHEAD WORD * t1,
            WORD * t2 )
```

Addition of PolyRatFuns

4.111.11 Description

This method gets two pointers to polyratfuns with up to two arguments each and calculates the sum.

4.111.12 Notes

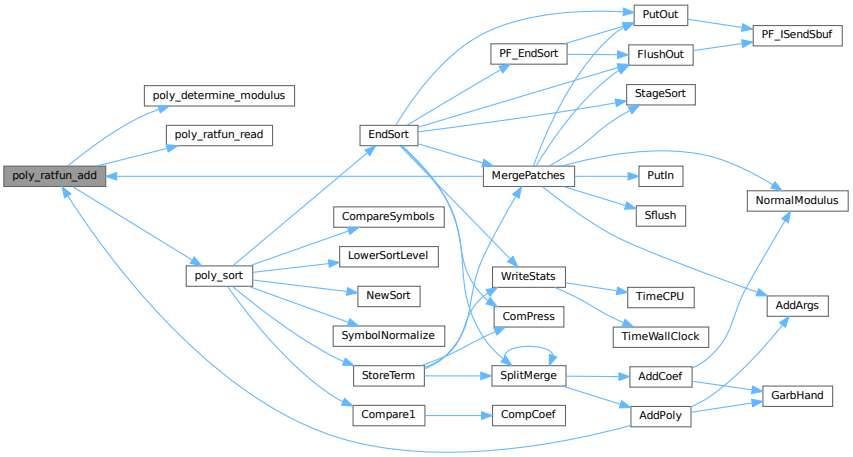
- The result is written at the workpointer
- Called from sort.c and threads.c
- Calls poly::operators and polygcd::gcd

Definition at line 635 of file polywrap.cc.

References poly_determine_modulus(), poly_ratfun_read(), and poly_sort().

Referenced by AddPoly(), and MergePatches().

Here is the call graph for this function:



4.111.12.1 poly_ratfun_normalize()

```
int poly_ratfun_normalize (
            PHEAD WORD * term )
```

Multiplication/normalization of PolyRatFuns

4.111.13 Description

This method searches a term for multiple polyratfuns and multiplies their contents. The result is properly normalized. Normalization also works for terms with a single polyratfun.

4.111.14 Notes

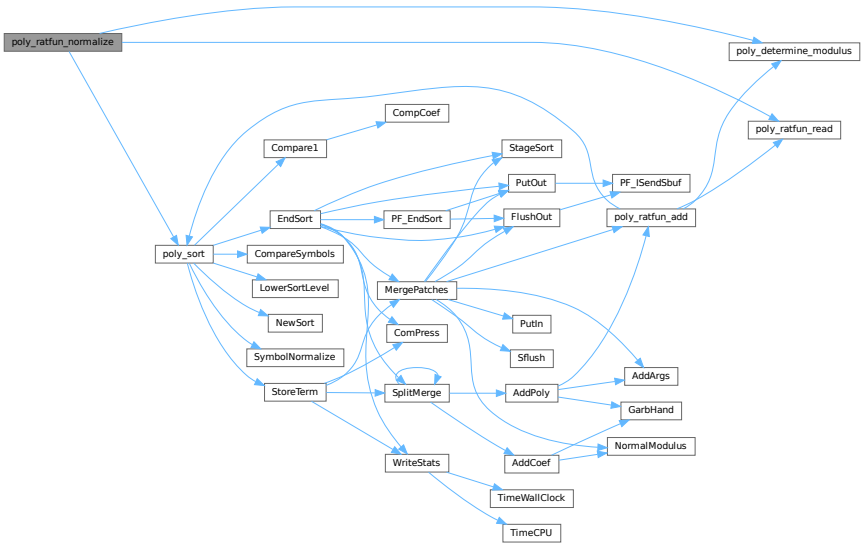
- The result overwrites the original term
- Called from proces.c
- Calls poly::operators and polygcd::gcd

Definition at line 771 of file polywrap.cc.

References poly_determine_modulus(), poly_ratfun_read(), and poly_sort().

Referenced by PolyFunMul(), and PrepPoly().

Here is the call graph for this function:



4.111.14.1 poly_fix_minus_signs()

```
void poly_fix_minus_signs (
            factorized_poly & a )
```

Definition at line 923 of file polywrap.cc.

4.111.14.2 poly_factorize()

```
WORD * poly_factorize (
            PHEAD WORD * argin,
            WORD * argout,
            bool with_arghead,
            bool is_fun_arg )
```

Factorization of function arguments / dollars

4.111.15 Description

This method factorizes a Form style argument or zero-terminated term list.

4.111.16 Notes

- Called from poly_factorize_{argument,dollar}
- Calls polyfact::factorize

Definition at line 988 of file polywrap.cc.

References poly_determine_modulus().

Referenced by poly_factorize_argument(), and poly_factorize_dollar().

Here is the call graph for this function:

4.111.16.1 poly_factorize_argument()

```
int poly_factorize_argument (
              PHEAD WORD * argin,
              WORD * argout )
```

Factorization of function arguments

4.111.17 Description

This method factorizes the Form-style argument argin.

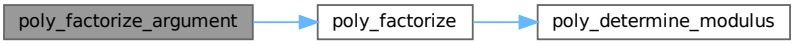
4.111.18 Notes

- The result is written at argout
- Called from argument.c
- Calls poly_factorize

Definition at line 1114 of file polywrap.cc.

References poly_factorize().

Here is the call graph for this function:



4.111.18.1 poly_factorize_dollar()

```
WORD * poly_factorize_dollar (
             PHEAD WORD * argin )
```

Factorization of dollar variables

4.111.19 Description

This method factorizes a dollar variable.

4.111.20 Notes

- The result is written at newly allocated memory.
- Called from dollar.c
- Calls poly_factorize

Definition at line 1148 of file polywrap.cc.

References poly_factorize().

Here is the call graph for this function:

4.111.20.1 poly_factorize_expression()

```
int poly_factorize_expression (
            EXPRESSIONS expr )
```

Factorization of expressions

4.111.21 Description

This method factorizes an expression.

4.111.22 Notes

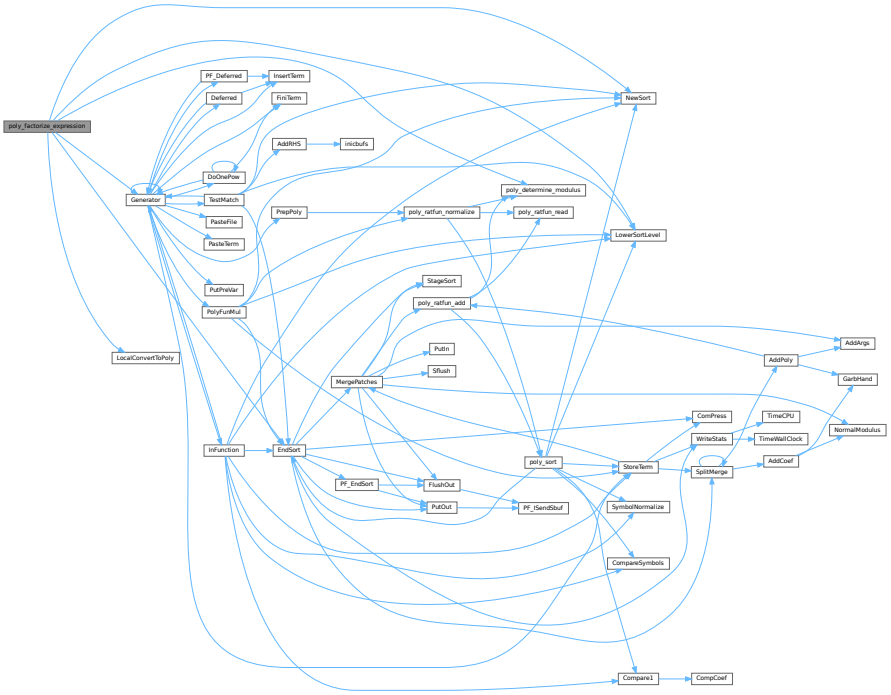
- The result overwrites the input expression
- Called from proces.c
- Calls polyfact::factorize

Definition at line 1180 of file polywrap.cc.

References EndSort(), Generator(), FiLe::handle, LocalConvertToPoly(), LowerSortLevel(), NewSort(), and poly_determine_modulus().

Referenced by PF_Processor(), and Processor().

Here is the call graph for this function:



4.111.22.1 poly_unfactorize_expression()

```
int poly_unfactorize_expression (
            EXPRESSIONS expr )
```

Unfactorization of expressions

4.111.23 Description

This method expands a factorized expression.

4.111.24 Notes

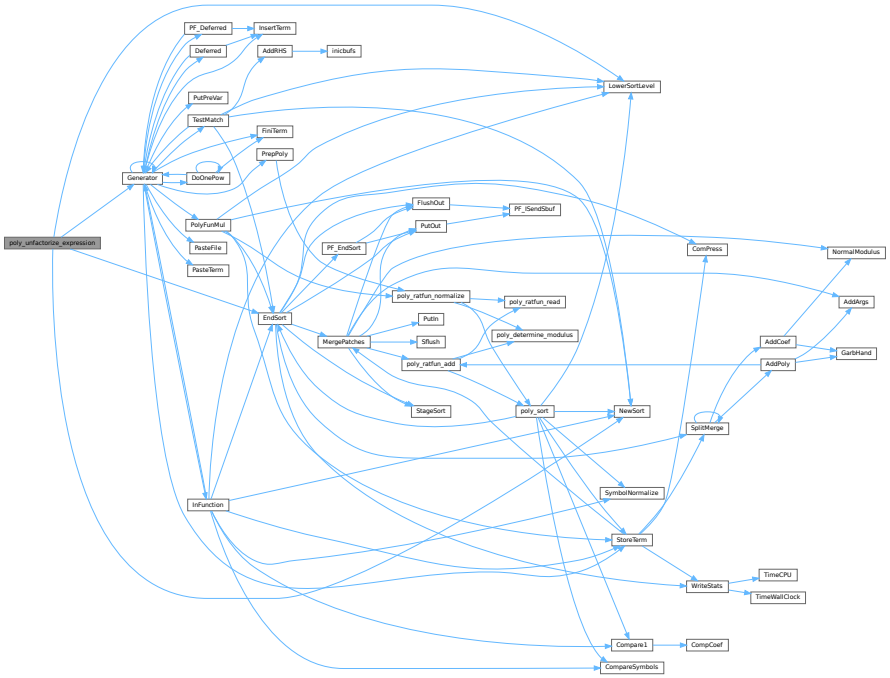
- The result overwrites the input expression
- Called from proces.c

Definition at line 1545 of file polywrap.cc.

References EndSort(), Generator(), FiLe::handle, LowerSortLevel(), and NewSort().

Referenced by PF_Processor(), and Processor().

Here is the call graph for this function:



4.111.24.1 poly_inverse()

```
WORD * poly_inverse (
              PHEAD WORD * arga,
              WORD * argb )
```

Definition at line 1757 of file polywrap.cc.

4.111.24.2 poly_mul()

```
WORD * poly_mul (
              PHEAD WORD * a,
              WORD * b )
```

Definition at line 1962 of file polywrap.cc.

4.111.24.3 poly_free_poly_vars()

```
void poly_free_poly_vars (
              PHEAD const char * text )
```

Definition at line 2028 of file polywrap.cc.

4.111.25 Variable Documentation

4.111.25.1 POLYWRAP_DENOMPOWER_INCREASE_FACTOR

```
const int POLYWRAP_DENOMPOWER_INCREASE_FACTOR = 2
```

Definition at line 52 of file polywrap.cc.

4.112 polywrap.cc

Go to the documentation of this file.

```
00001
00008 /* #[ License : */
00009 /*
00010  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00011  *   When using this file you are requested to refer to the publication
00012  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013  *   This is considered a matter of courtesy as the development was paid
00014  *   for by FOM the Dutch physics granting agency and we would like to
00015  *   be able to track its scientific use to convince FOM of its value
00016  *   for the community.
00017  *
00018  *   This file is part of FORM.
00019  *
00020  *   FORM is free software: you can redistribute it and/or modify it under the
00021  *   terms of the GNU General Public License as published by the Free Software
00022  *   Foundation, either version 3 of the License, or (at your option) any later
00023  *   version.
00024  *
00025  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00027  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00028  *   details.
00029  *
```

```
00030  *   You should have received a copy of the GNU General Public License along
00031  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00032  */
00033 /* #] License : */
00034
00035 #include "poly.h"
00036 #include "polygcd.h"
00037 #include "polyfact.h"
00038
00039 #include <iostream>
00040 #include <vector>
00041 #include <map>
00042 #include <climits>
00043 #include <cassert>
00044
00045 //#define DEBUG
00046
00047 #ifdef DEBUG
00048 #include "mytime.h"
00049 #endif
00050
00051 // denompower is increased by this factor when divmod_heap fails
00052 const int POLYWRAP_DENOMPOWER_INCREASE_FACTOR = 2;
00053
00054 using namespace std;
00055
00056 /*
00057     #[ poly_determine_modulus :
00058 */
00059
00079 WORD poly_determine_modulus (PHEAD bool multi_error, bool is_fun_arg, string message) {
00080
00081     if (AC.ncmod==0) return 0;
00082
00083     if (!is_fun_arg || (AC.modmode & ALSOFUNARGS)) {
00084
00085         if (ABS(AC.ncmod)>1) {
00086             MLOCK(ErrorMessageLock);
00087             MesPrint ((char*)"ERROR: %s with modulus > WORDSIZE not implemented",message.c_str());
00088             MUNLOCK(ErrorMessageLock);
00089             Terminate(-1);
00090         }
00091
00092         if (multi_error && AN.poly_num_vars>1) {
00093             MLOCK(ErrorMessageLock);
00094             MesPrint ((char*)"ERROR: multivariate %s with modulus not implemented",message.c_str());
00095             MUNLOCK(ErrorMessageLock);
00096             Terminate(-1);
00097         }
00098
00099         return *AC.cmod;
00100     }
00101
00102     AN.ncmod = 0;
00103     return 0;
00104 }
00105
00106 /*
00107     #] poly_determine_modulus :
00108     #[ poly_gcd :
00109 */
00110
00124 WORD *poly_gcd(PHEAD WORD *a, WORD *b, WORD fit) {
00125
00126 #ifdef WITHFLINT
00127     if ( AC.FlintPolyFlag && AC.ncmod==0 ) {
00128         WORD *ret = flint_gcd(BHEAD a, b, fit);
00129         return ret;
00130     }
00131 #endif
00132
00133 #ifdef DEBUG
00134     cout << "*** [" << thetime() << "]  CALL : poly_gcd" << endl;
00135 #endif
00136
00137 //
00138 //MesPrint("Calling poly_gcd with:");
00139 //{
00140 //  WORD *at = a;
00141 //  MesPrint("      a:");
00142 //  while ( *at ) {
00143 //      MesPrint("   %a",*at,at);
00144 //      at += *at;
00145 //  }
00146 //  MesPrint("      b:");
00147 //  at = b;
00148 //  while ( *at ) {
```

```
00149 //      MesPrint("   %a",*at,at);
00150 //      at += *at;
00151 //  }
00152 //}
00153
00154     // Extract variables
00155     vector<WORD *> e;
00156     e.reserve(2);
00157     e.push_back(a);
00158     e.push_back(b);
00159     poly::get_variables(BHEAD e, false, true);
00160
00161     // Check for modulus calculus
00162     WORD modp=poly_determine_modulus(BHEAD true, true, "polynomial GCD");
00163
00164     // Convert to polynomials
00165     poly pa(poly::argument_to_poly(BHEAD a, false, true), modp, 1);
00166     poly pb(poly::argument_to_poly(BHEAD b, false, true), modp, 1);
00167
00168     // Calculate gcd
00169     poly gcd(polygcd::gcd(pa,pb));
00170
00171     // Allocate new memory and convert to Form notation
00172     int newsize = (gcd.size_of_form_notation()+1);
00173     WORD *res;
00174     if ( fit ) {
00175         if ( newsize*sizeof(WORD) >= (size_t)(AM.MaxTer) ) {
00176             MLOCK(ErrorMessageLock);
00177             MesPrint("poly_gcd: Term too complex (%d words). Maybe increasing MaxTermSize (%d words)
    can help",
00178                 newsize, AM.MaxTer/sizeof(WORD));
00179             MUNLOCK(ErrorMessageLock);
00180             Terminate(-1);
00181         }
00182         res = TermMalloc("poly_gcd");
00183     }
00184     else {
00185         res = (WORD *)Malloc1(newsize*sizeof(WORD), "poly_gcd");
00186     }
00187     poly::poly_to_argument(gcd, res, false);
00188
00189     poly_free_poly_vars(BHEAD "AN.poly_vars_gcd");
00190
00191     // reset modulo calculation
00192     AN.ncmod = AC.ncmod;
00193     return res;
00194 }
00195
00196 /*
00197     #] poly_gcd :
00198     #[ poly_divmod :
00199
00200     if fit == 1 the answer must fit inside a term.
00201 */
00202
00203 WORD *poly_divmod(PHEAD WORD *a, WORD *b, int divmod, WORD fit) {
00204
00205 #ifdef DEBUG
00206     cout << "*** [" << thetime() << "]  CALL : poly_divmod" << endl;
00207 #endif
00208
00209     // check for modulus calculus
00210     WORD modp=poly_determine_modulus(BHEAD false, true, "polynomial division");
00211
00212     // get variables
00213     vector<WORD *> e;
00214     e.reserve(2);
00215     e.push_back(a);
00216     e.push_back(b);
00217     poly::get_variables(BHEAD e, false, false);
00218
00219     // add extra variables to keep track of denominators
00220     const int DENOMSYMBOL = MAXPOSITIVE;
00221
00222 //  WORD *new_poly_vars = (WORD *)Malloc1((AN.poly_num_vars+1)*sizeof(WORD), "AN.poly_vars");
00223 //  WCOPY(new_poly_vars, AN.poly_vars, AN.poly_num_vars);
00224 //  new_poly_vars[AN.poly_num_vars] = DENOMSYMBOL;
00225 //  if (AN.poly_num_vars > 0)
00226 //      M_free(AN.poly_vars, "AN.poly_vars");
00227 //  AN.poly_num_vars++;
00228 //  AN.poly_vars = new_poly_vars;
00229     AN.poly_vars[AN.poly_num_vars++] = DENOMSYMBOL;
00230
00231     // convert to polynomials
00232     poly dena(BHEAD 0);
00233     poly denb(BHEAD 0);
00234     poly pa(poly::argument_to_poly(BHEAD a, false, true, &dena), modp, 1);
```

```
00235     poly pb(poly::argument_to_poly(BHEAD b, false, true, &denb), modp, 1);
00236
00237     // remove contents
00238     poly numres(polygcd::integer_content(pa));
00239     poly denres(polygcd::integer_content(pb));
00240     pa /= numres;
00241     pb /= denres;
00242
00243     if (divmod==0) {
00244         numres *= denb;
00245         denres *= dena;
00246     }
00247     else {
00248         denres = dena;
00249     }
00250
00251     poly gcdres(polygcd::integer_gcd(numres,denres));
00252     numres /= gcdres;
00253     denres /= gcdres;
00254
00255     // determine lcoeff(b)
00256     poly lcoeffb(pb.integer_lcoeff());
00257
00258     poly pres(BHEAD 0);
00259
00260     if (!lcoeffb.is_one()) {
00261
00262         if (AN.poly_num_vars > 2) {
00263             // the original polynomial is multivariate (one dummy variable has
00264             // been added), so it is not trivial to determine which power of
00265             // lcoeff(b) can be in the answer
00266
00267             int denompower = 0, prevdenompower = 0;
00268             poly pq(BHEAD 0);
00269             poly pr(BHEAD 0);
00270
00271             // try denompower = 0, if this fails increase denompower until division succeeds
00272             bool div_fail = true;
00273             do
00274             {
00275                 if(denompower < prevdenompower)
00276                 {
00277                     // denompower increased beyond INT_MAX
00278 /* INTERNAL_ERROR_EXCL_START */
00279                     MLOCK(ErrorMessageLock);
00280                     MesPrint ((char*)"!>ERROR: pseudo-division failed in poly_divmod (denompower >
    INT_MAX)");
00281                     MUNLOCK(ErrorMessageLock);
00282                     Terminate(1);
00283 /* INTERNAL_ERROR_EXCL_STOP */
00284                 }
00285
00286                 if(denompower != 0)
00287                 {
00288                     // multiply a by lcoeffb^(denompower-prevdenompower)
00289                     WORD n = lcoeffb[lcoeffb[1]];
00290                     RaisPow(BHEAD (UWORD *)&lcoeffb[2+AN.poly_num_vars], &n,
    denompower-prevdenompower);
00291                     lcoeffb[1] = 2 + AN.poly_num_vars + ABS(n);
00292                     lcoeffb[0] = 1 + lcoeffb[1];
00293                     lcoeffb[lcoeffb[1]] = n;
00294
00295                     pa *= lcoeffb;
00296                     denres *= lcoeffb;
00297                 }
00298
00299                 // try division
00300                 poly ppow(BHEAD 0);
00301                 poly::divmod_heap(pa,pb,pq,pr,false,true,div_fail); // sets div_fail
00302
00303                 // increase denompower for next iteration
00304                 prevdenompower = denompower;
00305                 denompower = (denompower==0 ? 1 : denompower*POLYWRAP_DENOMPOWER_INCREASE_FACTOR+1 );
    // generates 2^n-1 when POLYWRAP_DENOMPOWER_INCREASE_FACTOR = 2
00306             }
00307             while(div_fail);
00308
00309             pres = (divmod==0 ? pq : pr);
00310
00311         }
00312         else {
00313             // one variable, so the power is the difference of the degrees
00314
00315             int denompower = MaX(0, pa.degree(0) - pb.degree(0) + 1);
00316
00317             // multiply a by that power
00318             WORD n = lcoeffb[lcoeffb[1]];
```

```
00319             RaisPow(BHEAD (UWORD *)&lcoeffb[2+AN.poly_num_vars], &n, denompower);
00320             lcoeffb[1] = 2 + AN.poly_num_vars + ABS(n);
00321             lcoeffb[0] = 1 + lcoeffb[1];
00322             lcoeffb[lcoeffb[1]] = n;
00323 
00324             pa *= lcoeffb;
00325             denres *= lcoeffb;
00326 
00327             pres = (divmod==0 ? pa/pb : pa%pb);
00328 
00329         }
00330 
00331     }
00332     else {
00333 
00334         pres = (divmod==0 ? pa/pb : pa%pb);
00335 
00336     }
00337 
00338     // convert to Form notation
00339     // NOTE: this part can be rewritten with poly::size_of_form_notation_with_den()
00340     // and poly::poly_to_argument_with_den().
00341     WORD *res;
00342 
00343     // special case: a=0
00344     if (pres[0]==1) {
00345         if ( fit ) {
00346             res = TermMalloc("poly_divmod");
00347         }
00348         else {
00349             res = (WORD *)Malloc1(sizeof(WORD), "poly_divmod");
00350         }
00351         res[0] = 0;
00352     }
00353     else {
00354         pres *= numres;
00355 
00356         WORD nden;
00357         UWORD *den = (UWORD *)NumberMalloc("poly_divmod");
00358 
00359         // allocate the memory; note that this overestimates the size,
00360         // since the estimated denominators are too large
00361         int ressize = pres.size_of_form_notation() + pres.number_of_terms()*2*ABS(denres[denres[1]]) +
    1;
00362 
00363         if ( fit ) {
00364             if ( ressize*sizeof(WORD) > (size_t)(AM.MaxTer) ) {
00365                 MLOCK(ErrorMessageLock);
00366                 MesPrint("poly_divmod: Term too complex (%d words). Maybe increasing MaxTermSize (%d
    words) can help",
00367                     ressize, AM.MaxTer/sizeof(WORD));
00368                 MUNLOCK(ErrorMessageLock);
00369                 Terminate(-1);
00370             }
00371             res = TermMalloc("poly_divmod");
00372         }
00373         else {
00374             res = (WORD *)Malloc1(ressize*sizeof(WORD), "poly_divmod");
00375         }
00376         int L=0;
00377 
00378         for (int i=1; i!=pres[0]; i+=pres[i]) {
00379 
00380             res[L]=1; // length
00381             bool first = true;
00382 
00383             for (int j=0; j<AN.poly_num_vars; j++)
00384                 if (pres[i+1+j] > 0) {
00385                     if (first) {
00386                         first = false;
00387                         res[L+1] = 1; // symbols
00388                         res[L+2] = 2; // length
00389                     }
00390                     res[L+1+res[L+2]++] = AN.poly_vars[j]; // symbol
00391                     res[L+1+res[L+2]++] = pres[i+1+j];     // power
00392                 }
00393 
00394             if (!first) res[L] += res[L+2]; // fix length
00395 
00396             // numerator
00397             WORD nnum = pres[i+pres[i]-1];
00398             WCOPY(&res[L+res[L]], &pres[i+pres[i]-1-ABS(nnum)], ABS(nnum));
00399 
00400             // calculate denominator
00401             nden = denres[denres[1]];
00402             WCOPY(den, &denres[2+AN.poly_num_vars], ABS(nden));
00403 
```

```
00404             if (nden!=1 || den[0]!=1)
00405                 Simplify(BHEAD (UWORD *)&res[L+res[L]], &nnum, den, &nden); // gcd(num,den)
00406             Pack((UWORD *)&res[L+res[L]], &nnum, den, nden);              // format
00407             res[L] += 2*ABS(nnum)+1;                                      // fix length
00408             res[L+res[L]-1] = SGN(nnum)*(2*ABS(nnum)+1);                  // length of coefficient
00409             L += res[L];                                                  // fix length
00410         }
00411
00412         res[L] = 0;
00413
00414         NumberFree(den,"poly_divmod");
00415     }
00416
00417     // clean up
00418     poly_free_poly_vars(BHEAD "AN.poly_vars_divmod");
00419
00420     // reset modulo calculation
00421     AN.ncmod = AC.ncmod;
00422
00423     return res;
00424 }
00425
00426 /*
00427     #] poly_divmod :
00428     #[ poly_div :
00429
00430     Routine divides the expression in arg1 by the expression in arg2.
00431     We did not take out special cases.
00432     The arguments are zero terminated sequences of term(s).
00433     The action is to divide arg1 by arg2: [arg1/arg2].
00434     The answer should be a buffer (allocated by Malloc1) with a zero
00435     terminated sequence of terms (or just zero).
00436 */
00437 WORD *poly_div(PHEAD WORD *a, WORD *b, WORD fit) {
00438
00439 #ifdef WITHFLINT
00440     if ( AC.FlintPolyFlag && AC.ncmod==0 ) {
00441         WORD *ret = flint_div(BHEAD a, b, fit);
00442         return ret;
00443     }
00444 #endif
00445
00446 #ifdef DEBUG
00447     cout << "*** [" << thetime() << "]  CALL : poly_div" << endl;
00448 #endif
00449
00450     return poly_divmod(BHEAD a, b, 0, fit);
00451 }
00452
00453 /*
00454     #] poly_div :
00455     #[ poly_rem :
00456
00457     Routine divides the expression in arg1 by the expression in arg2
00458     and takes the remainder.
00459     We did not take out special cases.
00460     The arguments are zero terminated sequences of term(s).
00461     The action is to divide arg1 by arg2 and take the remainder: [arg1%arg2].
00462     The answer should be a buffer (allocated by Malloc1) with a zero
00463     terminated sequence of terms (or just zero).
00464 */
00465 WORD *poly_rem(PHEAD WORD *a, WORD *b, WORD fit) {
00466
00467 #ifdef WITHFLINT
00468     if ( AC.FlintPolyFlag && AC.ncmod==0 ) {
00469         WORD *ret = flint_rem(BHEAD a, b, fit);
00470         return ret;
00471     }
00472 #endif
00473
00474 #ifdef DEBUG
00475     cout << "*** [" << thetime() << "]  CALL : poly_rem" << endl;
00476 #endif
00477
00478     return poly_divmod(BHEAD a, b, 1, fit);
00479 }
00480
00481 /*
00482     #] poly_rem :
00483     #[ poly_ratfun_read :
00484 */
00485
00498 void poly_ratfun_read (WORD *a, poly &num, poly &den) {
00499
00500 #ifdef DEBUG
00501     cout << "*** [" << thetime() << "]  CALL : poly_ratfun_read" << endl;
00502 #endif
```

```
00503
00504     POLY_GETIDENTITY(num);
00505
00506     int modp = num.modp;
00507
00508     WORD *astop = a+a[1];
00509
00510     bool clean = (a[2] & MUSTCLEANPRF) == 0;
00511
00512     a += FUNHEAD;
00513     if (a >= astop) {
00514         MLOCK(ErrorMessageLock);
00515         MesPrint ((char*)"ERROR: PolyRatFun cannot have zero arguments");
00516         MUNLOCK(ErrorMessageLock);
00517         Terminate(-1);
00518     }
00519
00520     poly den_num(BHEAD 1),den_den(BHEAD 1);
00521
00522     num = poly::argument_to_poly(BHEAD a, true, !clean, &den_num);
00523     num.setmod(modp,1);
00524     NEXTARG(a);
00525
00526     if (a < astop) {
00527         den = poly::argument_to_poly(BHEAD a, true, !clean, &den_den);
00528         den.setmod(modp,1);
00529         NEXTARG(a);
00530     }
00531     else {
00532         den = poly(BHEAD 1, modp, 1);
00533     }
00534
00535     if (a < astop) {
00536         MLOCK(ErrorMessageLock);
00537         MesPrint ((char*)"ERROR: PolyRatFun cannot have more than two arguments");
00538         MUNLOCK(ErrorMessageLock);
00539         Terminate(-1);
00540     }
00541
00542     // JD: At this point, num and den are certainly sorted into the correct order by
00543     // poly::argument_to_poly, but we can't rely on the clean flag to know if there
00544     // are any negative powers. Check for them, and set clean = false if there are any.
00545     vector<WORD> minpower(AN.poly_num_vars, MAXPOSITIVE);
00546
00547     for (int i=1; i<num[0]; i+=num[i]) {
00548         for (int j=0; j<AN.poly_num_vars; j++) {
00549             minpower[j] = MiN(minpower[j], num[i+1+j]);
00550         }
00551     }
00552     for (int i=1; i<den[0]; i+=den[i]) {
00553         for (int j=0; j<AN.poly_num_vars; j++) {
00554             minpower[j] = MiN(minpower[j], den[i+1+j]);
00555             if ( minpower[j] < 0 ) clean = false;
00556         }
00557     }
00558
00559     if (!clean) {
00560         for (int i=1; i<num[0]; i+=num[i])
00561             for (int j=0; j<AN.poly_num_vars; j++)
00562                 num[i+1+j] -= minpower[j];
00563         for (int i=1; i<den[0]; i+=den[i])
00564             for (int j=0; j<AN.poly_num_vars; j++)
00565                 den[i+1+j] -= minpower[j];
00566
00567         num *= den_den;
00568         den *= den_num;
00569
00570         poly gcd = polygcd::gcd(num,den);
00571         num /= gcd;
00572         den /= gcd;
00573     }
00574 }
00575
00576 /*
00577     #] poly_ratfun_read :
00578     #[ poly_sort :
00579 */
00580
00592 void poly_sort(PHEAD WORD *a) {
00593
00594 #ifdef DEBUG
00595     cout << "*** [" << thetime() << "]  CALL : poly_sort" << endl;
00596 #endif
00597     if (NewSort(BHEAD0)) { Terminate(-1); }
00598     AR.CompareRoutine = (COMPAREDUMMY)(&CompareSymbols);
00599
00600     for (int i=ARGHEAD; i<a[0]; i+=a[i]) {
```

```
00601         if (SymbolNormalize(a+i)<0 || StoreTerm(BHEAD a+i)) {
00602             AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
00603             LowerSortLevel();
00604             Terminate(-1);
00605         }
00606     }
00607 
00608     if (EndSort(BHEAD a+ARGHEAD,1) < 0) {
00609         AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
00610         Terminate(-1);
00611     }
00612 
00613     AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
00614     a[1] = 0; // set dirty flag to zero
00615 }
00616 
00617 /*
00618     #] poly_sort :
00619     #[ poly_ratfun_add :
00620 */
00621 
00635 WORD *poly_ratfun_add (PHEAD WORD *t1, WORD *t2) {
00636 
00637 #ifdef WITHFLINT
00638     if ( AC.FlintPolyFlag && AC.ncmod==0 ) {
00639         WORD *ret = flint_ratfun_add(BHEAD t1, t2);
00640         return ret;
00641     }
00642 #endif
00643 
00644     if ( AR.PolyFunExp == 1 ) return PolyRatFunSpecial(BHEAD t1, t2);
00645 
00646 #ifdef DEBUG
00647     cout « "*** [" « thetime() « "]  CALL : poly_ratfun_add" « endl;
00648 #endif
00649 
00650     WORD *oldworkpointer = AT.WorkPointer;
00651 
00652     // Extract variables
00653     vector<WORD *> e;
00654     e.reserve(4);
00655 
00656     for (WORD *t=t1+FUNHEAD; t<t1+t1[1];) {
00657         e.push_back(t);
00658         NEXTARG(t);
00659     }
00660     for (WORD *t=t2+FUNHEAD; t<t2+t2[1];) {
00661         e.push_back(t);
00662         NEXTARG(t);
00663     }
00664 
00665     assert(e.size() == 4);
00666 
00667     poly::get_variables(BHEAD e, true, true);
00668 
00669     // Check for modulus calculus
00670     WORD modp=poly_determine_modulus(BHEAD true, true, "PolyRatFun");
00671 
00672     // Find numerators / denominators
00673     poly num1(BHEAD 0,modp,1), den1(BHEAD 0,modp,1), num2(BHEAD 0,modp,1), den2(BHEAD 0,modp,1);
00674 
00675     poly_ratfun_read(t1, num1, den1);
00676     poly_ratfun_read(t2, num2, den2);
00677 
00678     poly num(BHEAD 0),den(BHEAD 0),gcd(BHEAD 0);
00679 
00680     // Calculate result
00681     if (den1 != den2) {
00682         gcd = polygcd::gcd(den1,den2);
00683 #ifdef OLDADDITION
00684         num = num1*(den2/gcd) + num2*(den1/gcd);
00685         den = (den1/gcd)*den2;
00686         gcd = polygcd::gcd(num,den);
00687 #else
00688         den = den1/gcd;
00689         num = num1*(den2/gcd) + num2*den;
00690         den = den*den2;
00691         gcd = polygcd::gcd(num,gcd);
00692 #endif
00693     }
00694     else {
00695         num = num1 + num2;
00696         den = den1;
00697         gcd = polygcd::gcd(num,den);
00698     }
00699 
00700     num /= gcd;
```

```
00701     den /= gcd;
00702
00703     // Fix sign
00704     if (den.sign() == -1) { num*=poly(BHEAD -1); den*=poly(BHEAD -1); }
00705
00706     // Check size: include FUNHEAD for the prf itself, an ARGHEAD each for num and den,
00707     // and 3 for the final coeff "1/1". We don't know here what the rest of the term looks like,
00708     // but it certainly has at least its total size (so +1):
00709     if ((num.size_of_form_notation() + den.size_of_form_notation() + FUNHEAD + 2*ARGHEAD + 3 + 1)
00710         > AM.MaxTer/(int)sizeof(WORD)) {
00711
00712         MLOCK(ErrorMessageLock);
00713         MesPrint ("ERROR: PolyRatFun doesn't fit in a term");
00714         MesPrint ("(1) num size = %d, den size = %d, rest = %d, MaxTermSize = %d words",
00715                 num.size_of_form_notation()+ARGHEAD,
00716                 den.size_of_form_notation()+ARGHEAD,
00717                 FUNHEAD + 3 + 1,
00718                 AM.MaxTer/sizeof(WORD));
00719         MUNLOCK(ErrorMessageLock);
00720         Terminate(-1);
00721     }
00722
00723     // Format result in Form notation
00724     WORD *t = oldworkpointer;
00725
00726     *t++ = AR.PolyFun;                    // function
00727     *t++ = 0;                             // length (to be determined)
00728 //  *t++ &= ~MUSTCLEANPRF;                // clean polyratfun
00729     *t++ = 0;
00730     FILLFUN3(t);                          // header
00731     poly::poly_to_argument(num,t, true); // argument 1 (numerator)
00732     if (*t>0 && t[1]==DIRTYFLAG)           // to Form order
00733         poly_sort(BHEAD t);
00734     t += (*t>0 ? *t : 2);
00735     poly::poly_to_argument(den,t, true); // argument 2 (denominator)
00736     if (*t>0 && t[1]==DIRTYFLAG)           // to Form order
00737         poly_sort(BHEAD t);
00738     t += (*t>0 ? *t : 2);
00739
00740     oldworkpointer[1] = t - oldworkpointer; // length
00741     AT.WorkPointer = t;
00742
00743     poly_free_poly_vars(BHEAD "AN.poly_vars_ratfun_add");
00744
00745     // reset modulo calculation
00746     AN.ncmod = AC.ncmod;
00747
00748     return oldworkpointer;
00749 }
00750
00751 /*
00752     #] poly_ratfun_add :
00753     #[ poly_ratfun_normalize :
00754 */
00755
00771 int poly_ratfun_normalize (PHEAD WORD *term) {
00772
00773 #ifdef WITHFLINT
00774     if ( AC.FlintPolyFlag && AC.ncmod==0 ) {
00775         flint_ratfun_normalize(BHEAD term);
00776         return 0;
00777     }
00778 #endif
00779
00780 #ifdef DEBUG
00781     cout << "*** [" << thetime() << "]  CALL : poly_ratfun_normalize" << endl;
00782 #endif
00783
00784     // Strip coefficient
00785     WORD *tstop = term + *term;
00786     int ncoeff = tstop[-1];
00787     tstop -= ABS(ncoeff);
00788
00789     // if only one clean polyratfun, return immediately
00790     int num_polyratfun = 0;
00791
00792     for (WORD *t=term+1; t<tstop; t+=t[1])
00793         if (*t == AR.PolyFun) {
00794             num_polyratfun++;
00795             if ((t[2] & MUSTCLEANPRF) != 0)
00796                 num_polyratfun = INT_MAX;
00797             if (num_polyratfun > 1) break;
00798         }
00799
00800     if (num_polyratfun <= 1) return 0;
00801
00802     WORD oldsorttype = AR.SortType;
```

```
00803     AR.SortType = SORTHIGHFIRST;
00804
00805 /*
00806     When there are polyratfun's with only one variable: rename them
00807     temporarily to TMPPOLYFUN.
00808 */
00809     for (WORD *t=term+1; t<tstop; t+=t[1]) {
00810         if (*t == AR.PolyFun && (t[1] == FUNHEAD+t[FUNHEAD]
00811             || t[1] == FUNHEAD+2 ) ) { *t = TMPPOLYFUN; }
00812     }
00813
00814
00815     // Extract all variables in the polyfuns
00816     vector<WORD *> e;
00817
00818     for (WORD *t=term+1; t<tstop; t+=t[1]) {
00819         if (*t == AR.PolyFun)
00820             for (WORD *t2 = t+FUNHEAD; t2<t+t[1];) {
00821                 e.push_back(t2);
00822                 NEXTARG(t2);
00823             }
00824     }
00825     poly::get_variables(BHEAD e, true, true);
00826
00827     // Check for modulus calculus
00828     WORD modp=poly_determine_modulus(BHEAD true, true, "PolyRatFun");
00829
00830     // Accumulate total denominator/numerator and copy the remaining terms
00831     // We start with 'trivial' polynomials
00832     poly num1(BHEAD (UWORD *)tstop, ncoeff/2, modp, 1);
00833     poly den1(BHEAD (UWORD *)tstop+ABS(ncoeff/2), ABS(ncoeff)/2, modp, 1);
00834
00835     WORD *s = term+1;
00836
00837     for (WORD *t=term+1; t<tstop;)
00838         if (*t == AR.PolyFun) {
00839
00840             poly num2(BHEAD 0,modp,1);
00841             poly den2(BHEAD 0,modp,1);
00842             poly_ratfun_read(t,num2,den2);
00843
00844             if ((t[2] & MUSTCLEANPRF) != 0) { // first normalize
00845                 poly gcd1(polygcd::gcd(num2,den2));
00846                 num2 = num2/gcd1;
00847                 den2 = den2/gcd1;
00848             }
00849             t += t[1];
00850             poly gcd1(polygcd::gcd(num1,den2));
00851             poly gcd2(polygcd::gcd(num2,den1));
00852
00853             num1 = (num1 / gcd1) * (num2 / gcd2);
00854             den1 = (den1 / gcd2) * (den2 / gcd1);
00855         }
00856         else {
00857             int i = t[1];
00858             if ( s != t ) { NCOPY(s,t,i) }
00859             else { t += i; s += i; }
00860         }
00861
00862     // Fix sign
00863     if (den1.sign() == -1) { num1*=poly(BHEAD -1); den1*=poly(BHEAD -1); }
00864
00865     // Check size: include FUNHEAD for the prf itself, an ARGHEAD each for num and den,
00866     // s-term for the copied term so far, and 3 for final coeff "1/1"
00867     if ((num1.size_of_form_notation() + den1.size_of_form_notation() + FUNHEAD + 2*ARGHEAD
00868         + s-term + 3) > AM.MaxTer/(int)sizeof(WORD)) {
00869
00870         MLOCK(ErrorMessageLock);
00871         MesPrint ("ERROR: PolyRatFun doesn't fit in a term");
00872         MesPrint ("(2) num size = %d, den size = %d, rest = %d, MaxTermSize = %d words",
00873                 num1.size_of_form_notation()+ARGHEAD,
00874                 den1.size_of_form_notation()+ARGHEAD,
00875                 FUNHEAD + s-term + 3,
00876                 AM.MaxTer/sizeof(WORD));
00877         MUNLOCK(ErrorMessageLock);
00878         Terminate(-1);
00879     }
00880
00881     // Format result in Form notation
00882     WORD *t = s;
00883     *t++ = AR.PolyFun;                  // function
00884     *t++ = 0;                           // size (to be determined)
00885     *t++ &= ~MUSTCLEANPRF;                  // clean polyratfun
00886     FILLFUN3(t);                        // header
00887     poly::poly_to_argument(num1,t,true); // argument 1 (numerator)
00888     if (*t>0 && t[1]==DIRTYFLAG)        // to Form order
00889         poly_sort(BHEAD t);
```

```
00890     t += (*t>0 ? *t : 2);
00891     poly::poly_to_argument(den1,t,true); // argument 2 (denominator)
00892     if (*t>0 && t[1]==DIRTYFLAG)         // to Form order
00893         poly_sort(BHEAD t);
00894     t += (*t>0 ? *t : 2);
00895
00896     s[1] = t - s;                        // function length
00897
00898     *t++ = 1;                            // term coefficient
00899     *t++ = 1;
00900     *t++ = 3;
00901
00902     term[0] = t-term;                    // term length
00903
00904     poly_free_poly_vars(BHEAD "AN.poly_vars_ratfun_normalize");
00905
00906     // reset modulo calculation
00907     AN.ncmod = AC.ncmod;
00908
00909     tstop = term + *term; tstop -= ABS(tstop[-1]);
00910     for (WORD *t=term+1; t<tstop; t+=t[1]) {
00911         if (*t == TMPPOLYFUN ) *t = AR.PolyFun;
00912     }
00913
00914     AR.SortType = oldsorttype;
00915     return 0;
00916 }
00917
00918 /*
00919     #] poly_ratfun_normalize :
00920     #[ poly_fix_minus_signs :
00921 */
00922
00923 void poly_fix_minus_signs (factorized_poly &a) {
00924
00925     if ( a.factor.empty() ) return;
00926
00927     POLY_GETIDENTITY(a.factor[0]);
00928
00929     int overall_sign = +1;
00930
00931     // find term with maximum power of highest symbol
00932     for (int i=0; i<(int)a.factor.size(); i++) {
00933
00934         int maxvar = -1;
00935         int maxpow = -1;
00936         int sign = +1;
00937
00938         WORD *tstop = a.factor[i].terms; tstop += *tstop;
00939         for (WORD *t=a.factor[i].terms+1; t<tstop; t+=*t)
00940             for (int j=0; j<AN.poly_num_vars; j++) {
00941                 int var = AN.poly_vars[j];
00942                 int pow = t[1+j];
00943                 if (pow>0 && (var>maxvar || (var==maxvar && pow>maxpow))) {
00944                     maxvar = var;
00945                     maxpow = pow;
00946                     sign = SGN(*(t+*t-1));
00947                 }
00948             }
00949
00950         // if negative coefficient, multiply by -1
00951         if (sign==-1) {
00952             a.factor[i] *= poly(BHEAD sign);
00953             if (a.power[i] % 2 == 1) overall_sign*=-1;
00954         }
00955     }
00956
00957     // if overall minus sign
00958     if (overall_sign == -1) {
00959         // look at constant factor and multiply by -1
00960         for (int i=0; i<(int)a.factor.size(); i++)
00961             if (a.factor[i].is_integer()) {
00962                 a.factor[i] *= poly(BHEAD -1);
00963                 return;
00964             }
00965
00966         // otherwise, add a factor of -1
00967         a.add_factor(poly(BHEAD -1), 1);
00968     }
00969 }
00970
00971 /*
00972     #] poly_fix_minus_signs :
00973     #[ poly_factorize :
00974 */
00975
00988 WORD *poly_factorize (PHEAD WORD *argin, WORD *argout, bool with_arghead, bool is_fun_arg) {
```

```
00989
00990 #ifdef DEBUG
00991     cout « "*** [" « thetime() « "]  CALL : poly_factorize" « endl;
00992 #endif
00993
00994     poly::get_variables(BHEAD vector<WORD*>(1,argin), with_arghead, true);
00995     poly den(BHEAD 0);
00996     poly a(poly::argument_to_poly(BHEAD argin, with_arghead, true, &den));
00997
00998     // check for modulus calculus
00999     WORD modp=poly_determine_modulus(BHEAD true, is_fun_arg, "polynomial factorization");
01000     a.setmod(modp,1);
01001
01002     // factorize
01003     factorized_poly f(polyfact::factorize(a));
01004     poly_fix_minus_signs(f);
01005
01006     poly num(BHEAD 1);
01007     for (int i=0; i<(int)f.factor.size(); i++)
01008         if (f.factor[i].is_integer())
01009             num = f.factor[i];
01010
01011     // determine size
01012     int len = with_arghead ? ARGHEAD : 0;
01013
01014     if (!num.is_one() || !den.is_one()) {
01015         len++;
01016         len += MaX(ABS(num[num[1]]), den[den[1]])*2+1;
01017         len += with_arghead ? ARGHEAD : 1;
01018     }
01019
01020     for (int i=0; i<(int)f.factor.size(); i++) {
01021         if (!f.factor[i].is_integer()) {
01022             len += f.power[i] * f.factor[i].size_of_form_notation();
01023             len += f.power[i] * (with_arghead ? ARGHEAD : 1);
01024         }
01025     }
01026
01027     len++;
01028
01029     if (argout != NULL) {
01030         // check size
01031         if (len >= AM.MaxTer) {
01032             MLOCK(ErrorMessageLock);
01033             MesPrint ("ERROR: factorization doesn't fit in a term (len = %d, MaxTermSize = %d words)",
01034                 len/sizeof(WORD), AM.MaxTer/sizeof(WORD));
01035             MUNLOCK(ErrorMessageLock);
01036             Terminate(-1);
01037         }
01038     }
01039     else {
01040         // allocate size
01041         argout = (WORD*) Malloc1(len*sizeof(WORD), "poly_factorize");
01042     }
01043
01044     WORD *old_argout = argout;
01045
01046     // constant factor
01047     if (!num.is_one() || !den.is_one()) {
01048         int n = max(ABS(num[num[1]]), ABS(den[den[1]]));
01049
01050         if (with_arghead) {
01051             *argout++ = ARGHEAD + 2 + 2*n;
01052             for (int i=1; i<ARGHEAD; i++)
01053                 *argout++ = 0;
01054         }
01055
01056         *argout++ = 2 + 2*n;
01057
01058         for (int i=0; i<n; i++)
01059             *argout++ = i<ABS(num[num[1]]) ? num[2+AN.poly_num_vars+i] : 0;
01060         for (int i=0; i<n; i++)
01061             *argout++ = i<ABS(den[den[1]]) ? den[2+AN.poly_num_vars+i] : 0;
01062
01063         *argout++ = SGN(num[num[1]]) * (2*n+1);
01064
01065         if (!with_arghead)
01066             *argout++ = 0;
01067         else {
01068             if (ToFast(old_argout, old_argout))
01069                 argout = old_argout+2;
01070         }
01071     }
01072
01073     // non-constant factors
01074     for (int i=0; i<(int)f.factor.size(); i++)
01075         if (!f.factor[i].is_integer())
```

```
01076             for (int j=0; j<f.power[i]; j++) {
01077                 poly::poly_to_argument(f.factor[i],argout,with_arghead);
01078 
01079                 if (with_arghead)
01080                     argout += *argout > 0 ? *argout : 2;
01081                 else {
01082                     while (*argout!=0) argout+=*argout;
01083                     argout++;
01084                 }
01085             }
01086 
01087     *argout=0;
01088 
01089     poly_free_poly_vars(BHEAD "AN.poly_vars_factorize");
01090 
01091     // reset modulo calculation
01092     AN.ncmod = AC.ncmod;
01093 
01094     return old_argout;
01095 }
01096 
01097 /*
01098     #] poly_factorize :
01099     #[ poly_factorize_argument :
01100 */
01101 
01114 int poly_factorize_argument(PHEAD WORD *argin, WORD *argout) {
01115 
01116 #ifdef DEBUG
01117     cout << "*** [" << thetime() << "]  CALL : poly_factorize_argument" << endl;
01118 #endif
01119 
01120 #ifdef WITHFLINT
01121     if ( AC.FlintPolyFlag && AC.ncmod==0 ) {
01122         flint_factorize_argument(BHEAD argin, argout);
01123         return 0;
01124     }
01125 #endif
01126 
01127     poly_factorize(BHEAD argin,argout,true,true);
01128     return 0;
01129 }
01130 
01131 /*
01132     #] poly_factorize_argument :
01133     #[ poly_factorize_dollar :
01134 */
01135 
01148 WORD *poly_factorize_dollar (PHEAD WORD *argin) {
01149 
01150 #ifdef DEBUG
01151     cout << "*** [" << thetime() << "]  CALL : poly_factorize_dollar" << endl;
01152 #endif
01153 
01154 #ifdef WITHFLINT
01155     if ( AC.FlintPolyFlag && AC.ncmod==0 ) {
01156         return flint_factorize_dollar(BHEAD argin);
01157     }
01158 #endif
01159 
01160     return poly_factorize(BHEAD argin,NULL,false,false);
01161 }
01162 
01163 /*
01164     #] poly_factorize_dollar :
01165     #[ poly_factorize_expression :
01166 */
01167 
01180 int poly_factorize_expression(EXPRESSIONS expr) {
01181 
01182 #ifdef DEBUG
01183     cout << "*** [" << thetime() << "]  CALL : poly_factorize_expression" << endl;
01184 #endif
01185 
01186     GETIDENTITY;
01187 
01188     if (AT.WorkPointer + AM.MaxTer > AT.WorkTop ) {
01189         MLOCK(ErrorMessageLock);
01190         MesWork();
01191         MUNLOCK(ErrorMessageLock);
01192         Terminate(-1);
01193     }
01194 
01195     WORD *term = AT.WorkPointer;
01196     WORD startebuf = cbuf[AT.ebufnum].numrhs;
01197     FILEHANDLE *file;
01198     POSITION pos;
```

```
01199
01200     FILEHANDLE *oldinfile = AR.infile;
01201     FILEHANDLE *oldoutfile = AR.outfile;
01202     WORD oldBracketOn = AR.BracketOn;
01203     WORD *oldBrackBuf = AT.BrackBuf;
01204     WORD oldbracketindexflag = AT.bracketindexflag;
01205     char oldCommercial[COMMERCIALSIZE+2];
01206
01207     strcpy(oldCommercial, (char*)AC.Commercial);
01208     strcpy((char*)AC.Commercial, "factorize");
01209
01210     // locate is the input
01211     if (expr->status == HIDDENGEXPRESSION || expr->status == HIDDENLEXPRESSION ||
01212               expr->status == INTOHIDEGEXPRESSION || expr->status == INTOHIDELEXPRESSION) {
01213         AR.InHiBuf = 0; file = AR.hidefile; AR.GetFile = 2;
01214     }
01215     else {
01216         AR.InInBuf = 0; file = AR.outfile; AR.GetFile = 0;
01217     }
01218
01219     // read and write to expression file
01220     AR.infile = AR.outfile = file;
01221
01222     // dummy indices are not allowed
01223     if (expr->numdummies > 0) {
01224         MesPrint("ERROR: factorization with dummy indices not implemented");
01225         Terminate(-1);
01226     }
01227
01228     // determine whether the expression in on file or in memory
01229     if (file->handle >= 0) {
01230         pos = expr->onfile;
01231         SeekFile(file->handle,&pos,SEEK_SET);
01232         if (ISNOTEQUALPOS(pos,expr->onfile)) {
01233 /* INTERNAL_ERROR_EXCL_START */
01234             MesPrint("!>ERROR: something wrong in scratch file [poly_factorize_expression]");
01235             Terminate(-1);
01236 /* INTERNAL_ERROR_EXCL_STOP */
01237         }
01238         file->POposition = expr->onfile;
01239         file->POfull = file->PObuffer;
01240         if (expr->status == HIDDENGEXPRESSION)
01241             AR.InHiBuf = 0;
01242         else
01243             AR.InInBuf = 0;
01244     }
01245     else {
01246         file->POfill = (WORD *)((UBYTE *)(file->PObuffer)+BASEPOSITION(expr->onfile));
01247     }
01248
01249     SetScratch(AR.infile, &(expr->onfile));
01250
01251     // read the first header term
01252     WORD size = GetTerm(BHEAD term);
01253     if (size <= 0) {
01254 /* INTERNAL_ERROR_EXCL_START */
01255         MesPrint ("!>ERROR: something wrong with expression [poly_factorize_expression]");
01256         Terminate(-1);
01257 /* INTERNAL_ERROR_EXCL_STOP */
01258     }
01259
01260     // store position: this is where the output will go
01261     pos = expr->onfile;
01262     ADDPOS(pos, size*sizeof(WORD));
01263
01264     // use polynomial as buffer, because it is easy to extend
01265     poly buffer(BHEAD 0);
01266     int bufpos = 0;
01267     int sumcommu = 0;
01268
01269     // read all terms
01270     while (GetTerm(BHEAD term)) {
01271         // substitute non-symbols by extra symbols
01272         sumcommu += DoesCommu(term);
01273         if ( sumcommu > 1 ) {
01274             MesPrint("ERROR: Cannot factorize an expression with more than one noncommuting object");
01275             Terminate(-1);
01276         }
01277         buffer.check_memory(bufpos);
01278         if (LocalConvertToPoly(BHEAD term, buffer.terms + bufpos, startebuf,0) < 0) {
01279 /* INTERNAL_ERROR_EXCL_START */
01280             MesPrint("!>ERROR: in LocalConvertToPoly [factorize_expression]");
01281             Terminate(-1);
01282 /* INTERNAL_ERROR_EXCL_STOP */
01283         }
01284         bufpos += *(buffer.terms + bufpos);
01285     }
```

```
01286     buffer[bufpos] = 0;
01287
01288     // parse the polynomial
01289
01290     AN.poly_num_vars = 0;
01291     poly::get_variables(BHEAD vector<WORD*>(1,buffer.terms), false, true);
01292     poly den(BHEAD 0);
01293     poly a(poly::argument_to_poly(BHEAD buffer.terms, false, true, &den));
01294
01295     // check for modulus calculus
01296     WORD modp=poly_determine_modulus(BHEAD true, false, "polynomial factorization");
01297     a.setmod(modp,1);
01298
01299     // create output
01300     SetScratch(file, &pos);
01301     NewSort(BHEAD0);
01302
01303     CBUF *C = cbuf+AC.cbufnum;
01304     CBUF *CC = cbuf+AT.ebufnum;
01305
01306     // turn brackets on. We force the existence of a bracket index.
01307     WORD nexpr = expr - Expressions;
01308     AR.BracketOn = 1;
01309     AT.BrackBuf = AM.BracketFactors;
01310     AT.bracketindexflag = 1;
01311     ClearBracketIndex(-nexpr-2); // Clears the index made during primary generation
01312     OpenBracketIndex(nexpr);     // Set up a new index
01313
01314     if (a.is_zero()) {
01315         expr->numfactors = 1;
01316     }
01317     else if (a.is_one() && den.is_one()) {
01318         expr->numfactors = 1;
01319
01320         term[0] = 8;
01321         term[1] = SYMBOL;
01322         term[2] = 4;
01323         term[3] = FACTORSYMBOL;
01324         term[4] = 1;
01325         term[5] = 1;
01326         term[6] = 1;
01327         term[7] = 3;
01328
01329         AT.WorkPointer += *term;
01330         Generator(BHEAD term, C->numlhs);
01331         AT.WorkPointer = term;
01332     }
01333     else {
01334         factorized_poly fac;
01335         bool iszero = false;
01336
01337         if (!(expr->vflags & ISFACTORIZED)) {
01338             // factorize the polynomial
01339             fac = polyfact::factorize(a);
01340             poly_fix_minus_signs(fac);
01341         }
01342         else {
01343             int factorsymbol=-1;
01344             for (int i=0; i<AN.poly_num_vars; i++)
01345                 if (AN.poly_vars[i] == FACTORSYMBOL)
01346                     factorsymbol = i;
01347
01348             poly denpow(BHEAD 1);
01349
01350             // already factorized, so factorize the factors
01351             for (int i=1; i<=expr->numfactors; i++) {
01352                 poly origfac(a.coefficient(factorsymbol, i));
01353                 factorized_poly fac2;
01354                 if (origfac.is_zero())
01355                     iszero=true;
01356                 else {
01357                     fac2 = polyfact::factorize(origfac);
01358                     poly_fix_minus_signs(fac2);
01359                     denpow *= den;
01360                 }
01361                 for (int j=0; j<(int)fac2.power.size(); j++)
01362                     fac.add_factor(fac2.factor[j], fac2.power[j]);
01363             }
01364
01365             // update denominator, since each factor was scaled
01366             den=denpow;
01367         }
01368
01369         expr->numfactors = 0;
01370
01371         // coefficient
01372         poly num(BHEAD 1);
```

```
01373         for (int i=0; i<(int)fac.factor.size(); i++)
01374             if (fac.factor[i].is_integer())
01375                 num *= fac.factor[i];
01376 
01377         poly gcd(polygcd::integer_gcd(num,den));
01378         den/=gcd;
01379         num/=gcd;
01380 
01381         if (iszero)
01382             expr->numfactors++;
01383 
01384         if (!iszero || (expr->vflags & KEEPZERO)) {
01385             if (!num.is_one() || !den.is_one()) {
01386                 expr->numfactors++;
01387 
01388                 int n = max(ABS(num[num[1]]), ABS(den[den[1]]));
01389 
01390                 term[0] = 6 + 2*n;
01391                 term[1] = SYMBOL;
01392                 term[2] = 4;
01393                 term[3] = FACTORSYMBOL;
01394                 term[4] = expr->numfactors;
01395                 for (int i=0; i<n; i++) {
01396                     term[5+i]   = i<ABS(num[num[1]]) ? num[2+AN.poly_num_vars+i] : 0;
01397                     term[5+n+i] = i<ABS(den[den[1]]) ? den[2+AN.poly_num_vars+i] : 0;
01398                 }
01399                 term[5+2*n] = SGN(num[num[1]]) * (2*n+1);
01400                 AT.WorkPointer += *term;
01401                 Generator(BHEAD term, C->numlhs);
01402                 AT.WorkPointer = term;
01403             }
01404 
01405             vector<poly> fac_arg(fac.factor.size(), poly(BHEAD 0));
01406 
01407             // convert the non-constant factors to Form-style arguments
01408             for (int i=0; i<(int)fac.factor.size(); i++)
01409                 if (!fac.factor[i].is_integer()) {
01410                     buffer.check_memory(fac.factor[i].size_of_form_notation()+1);
01411                     poly::poly_to_argument(fac.factor[i], buffer.terms, false);
01412                     NewSort(BHEAD0);
01413 
01414                     for (WORD *t=buffer.terms; *t!=0; t+=*t) {
01415                         // substitute extra symbols
01416                         if (ConvertFromPoly(BHEAD t, term, numxsymbol,
      CC->numrhs-startebuf+numxsymbol,
01417                                                                 startebuf-numxsymbol, 1) <= 0 ) {
01418 /* INTERNAL_ERROR_EXCL_START */
01419                             MesPrint("!>ERROR: in ConvertFromPoly [factorize_expression]");
01420                             Terminate(-1);
01421                             return(-1);
01422 /* INTERNAL_ERROR_EXCL_STOP */
01423                         }
01424 
01425                         // store term
01426                         AT.WorkPointer += *term;
01427                         Generator(BHEAD term, C->numlhs);
01428                         AT.WorkPointer = term;
01429                     }
01430 
01431                     // sort and store in buffer
01432                     WORD *buffer;
01433                     if (EndSort(BHEAD (WORD *)((void *)(&buffer)),2) < 0) return -1;
01434 
01435                     LONG bufsize=0;
01436                     for (WORD *t=buffer; *t!=0; t+=*t)
01437                         bufsize+=*t;
01438 
01439                     fac_arg[i].check_memory(bufsize+ARGHEAD+1);
01440 
01441                     for (int j=0; j<ARGHEAD; j++)
01442                         fac_arg[i].terms[j] = 0;
01443                     fac_arg[i].terms[0] = ARGHEAD + bufsize;
01444                     WCOPY(fac_arg[i].terms+ARGHEAD, buffer, bufsize);
01445                     M_free(buffer, "polynomial factorization");
01446                 }
01447 
01448             // compare and sort the factors in Form notation
01449             vector<int> order;
01450             vector<vector<int> > comp(fac.factor.size(), vector<int>(fac.factor.size(), 0));
01451 
01452             for (int i=0; i<(int)fac.factor.size(); i++)
01453                 if (!fac.factor[i].is_integer()) {
01454                     order.push_back(i);
01455 
01456                     for (int j=i+1; j<(int)fac.factor.size(); j++)
01457                         if (!fac.factor[j].is_integer()) {
01458                             comp[i][j] = CompArg(fac_arg[j].terms, fac_arg[i].terms);
```

```
01459                             comp[j][i] = -comp[i][j];
01460                         }
01461                 }
01462 
01463             for (int i=0; i<(int)order.size(); i++)
01464                 for (int j=0; j+1<(int)order.size(); j++)
01465                     if (comp[order[i]][order[j]] == 1)
01466                         swap(order[i],order[j]);
01467 
01468             // create the final expression
01469             for (int i=0; i<(int)order.size(); i++)
01470                 for (int j=0; j<fac.power[order[i]]; j++) {
01471 
01472                     expr->numfactors++;
01473 
01474                     WORD *tstop = fac_arg[order[i]].terms + *fac_arg[order[i]].terms;
01475                     for (WORD *t=fac_arg[order[i]].terms+ARGHEAD; t<tstop; t+=*t) {
01476 
01477                         WCOPY(term+4, t, *t);
01478 
01479                         // add special symbol "factor_"
01480                         *term = *(term+4) + 4;
01481                         *(term+1) = SYMBOL;
01482                         *(term+2) = 4;
01483                         *(term+3) = FACTORSYMBOL;
01484                         *(term+4) = expr->numfactors;
01485 
01486                         // store term
01487                         AT.WorkPointer += *term;
01488                         Generator(BHEAD term, C->numlhs);
01489                         AT.WorkPointer = term;
01490                     }
01491                 }
01492         }
01493     }
01494 
01495     // final sorting
01496     if (EndSort(BHEAD NULL,0) < 0) {
01497         LowerSortLevel();
01498         Terminate(-1);
01499     }
01500 
01501     // set factorized flag
01502     if (expr->numfactors > 0)
01503         expr->vflags |= ISFACTORIZED;
01504 
01505     // clean up
01506     AR.infile = oldinfile;
01507     AR.outfile = oldoutfile;
01508     AR.BracketOn = oldBracketOn;
01509     AT.BrackBuf = oldBrackBuf;
01510     AT.bracketindexflag = oldbracketindexflag;
01511     strcpy((char*)AC.Commercial, oldCommercial);
01512 
01513     poly_free_poly_vars(BHEAD "AN.poly_vars_factorize_expression");
01514 
01515     return 0;
01516 }
01517 
01518 /*
01519     #] poly_factorize_expression :
01520     #[ poly_unfactorize_expression :
01521 */
01522 
01535 #if ( SUBEXPSIZE == 5 )
01536 static WORD genericterm[] = {38,1,4,FACTORSYMBOL,0
01537     ,EXPRESSION,15,0,1,0,13,10,8,1,4,FACTORSYMBOL,0,1,1,3
01538     ,EXPRESSION,15,0,1,0,13,10,8,1,4,FACTORSYMBOL,0,1,1,3
01539     ,1,1,3,0};
01540 static WORD genericterm2[] = {23,1,4,FACTORSYMBOL,0
01541     ,EXPRESSION,15,0,1,0,13,10,8,1,4,FACTORSYMBOL,0,1,1,3
01542     ,1,1,3,0};
01543 #endif
01544 
01545 int poly_unfactorize_expression(EXPRESSIONS expr)
01546 {
01547     GETIDENTITY;
01548     int i, j, nfac = expr->numfactors, nfacp, nexpr = expr - Expressions;
01549     int expriszero = 0;
01550 
01551     FILEHANDLE *oldinfile = AR.infile;
01552     FILEHANDLE *oldoutfile = AR.outfile;
01553     char oldCommercial[COMMERCIALSIZE+2];
01554 
01555     WORD *oldworkpointer = AT.WorkPointer;
01556     WORD *term = AT.WorkPointer, *t, *w, size;
01557 
```

```
01558     FILEHANDLE *file;
01559     POSITION pos, oldpos;
01560
01561     WORD oldBracketOn = AR.BracketOn;
01562     WORD *oldBrackBuf = AT.BrackBuf;
01563     CBUF *C = cbuf+AC.cbufnum;
01564
01565     if ( ( expr->vflags & ISFACTORIZED ) == 0 ) return(0);
01566
01567     if ( AT.WorkPointer + AM.MaxTer > AT.WorkTop ) {
01568         MLOCK(ErrorMessageLock);
01569         MesWork();
01570         MUNLOCK(ErrorMessageLock);
01571         Terminate(-1);
01572     }
01573
01574     oldpos = AS.OldOnFile[nexpr];
01575     AS.OldOnFile[nexpr] = expr->onfile;
01576
01577     strcpy(oldCommercial, (char*)AC.Commercial);
01578     strcpy((char*)AC.Commercial, "unfactorize");
01579 /*
01580     locate the input
01581 */
01582     if ( expr->status == HIDDENGEXPRESSION || expr->status == HIDDENLEXPRESSION ||
01583             expr->status == INTOHIDEGEXPRESSION || expr->status == INTOHIDELEXPRESSION ) {
01584         AR.InHiBuf = 0; file = AR.hidefile; AR.GetFile = 2;
01585     }
01586     else {
01587         AR.InInBuf = 0; file = AR.outfile; AR.GetFile = 0;
01588     }
01589 /*
01590     read and write to expression file
01591 */
01592     AR.infile = AR.outfile = file;
01593 /*
01594     set the input file to the correct position
01595 */
01596     if ( file->handle >= 0 ) {
01597         pos = expr->onfile;
01598         SeekFile(file->handle,&pos,SEEK_SET);
01599         if (ISNOTEQUALPOS(pos,expr->onfile)) {
01600 /* INTERNAL_ERROR_EXCL_START */
01601             MesPrint("!>ERROR: something wrong in scratch file unfactorize_expression");
01602             Terminate(-1);
01603 /* INTERNAL_ERROR_EXCL_STOP */
01604         }
01605         file->POposition = expr->onfile;
01606         file->POfull = file->PObuffer;
01607         if ( expr->status == HIDDENGEXPRESSION )
01608             AR.InHiBuf = 0;
01609         else
01610             AR.InInBuf = 0;
01611     }
01612     else {
01613         file->POfill = (WORD *)((UBYTE *)(file->PObuffer)+BASEPOSITION(expr->onfile));
01614     }
01615     SetScratch(AR.infile, &(expr->onfile));
01616 /*
01617     Test for whether the first factor is zero.
01618 */
01619     if ( GetFirstBracket(term,nexpr) < 0 ) Terminate(-1);
01620     if ( term[4] != 1 || *term != 8 || term[1] != SYMBOL || term[3] != FACTORSYMBOL || term[4] != 1 )
      {
01621         expriszero = 1;
01622     }
01623     SetScratch(AR.infile, &(expr->onfile));
01624 /*
01625     Read the prototype. After this we have the file ready for the output at pos.
01626 */
01627     size = GetTerm(BHEAD term);
01628     if ( size <= 0 ) {
01629 /* INTERNAL_ERROR_EXCL_START */
01630         MesPrint ("!>ERROR: something wrong with expression unfactorize_expression");
01631         Terminate(-1);
01632 /* INTERNAL_ERROR_EXCL_STOP */
01633     }
01634     pos = expr->onfile;
01635     ADDPOS(pos, size*sizeof(WORD));
01636 /*
01637     Set the brackets straight
01638 */
01639     AR.BracketOn = 1;
01640     AT.BrackBuf = AM.BracketFactors;
01641     AT.bracketinfo = 0;
01642     while ( nfac > 2 ) {
01643         nfacp = nfac - nfac%2;
```

```
01644 /*
01645         Prepare the bracket index. We have:
01646             e->bracketinfo:    the old input bracket index
01647             e->newbracketinfo: the bracket index made for our current input
01648         We need to keep e->bracketinfo in case other workers need it (InParallel)
01649         Hence we work with AT.bracketinfo which takes priority.
01650         Note that in Processor we forced a newbracketinfo to be made.
01651 */
01652         if ( AT.bracketinfo != 0 ) ClearBracketIndex(-1);
01653         AT.bracketinfo = expr->newbracketinfo;
01654         OpenBracketIndex(nexpr);
01655 /*
01656         Now emulate the terms:
01657             sum_(i,0,nfacp,2,factor_^(i/2+1)*F[factor_^(i+1)]*F[factor_^(i+2)])
01658             +factor_^(nfacp/2+1)*F[factor_^nfac]
01659 */
01660         NewSort(BHEAD0);
01661         if ( expriszero == 0 ) {
01662             for ( i = 0; i < nfacp; i += 2 ) {
01663                 t = genericterm; w = term = oldworkpointer;
01664                 j = *t; NCOPY(w,t,j);
01665                 term[4] = i/2+1;
01666                 term[7] = nexpr;
01667                 term[16] = i+1;
01668                 term[22] = nexpr;
01669                 term[31] = i+2;
01670                 AT.WorkPointer = term + *term;
01671                 Generator(BHEAD term, C->numlhs);
01672             }
01673             if ( nfac > nfacp ) {
01674                 t = genericterm2; w = term = oldworkpointer;
01675                 j = *t; NCOPY(w,t,j);
01676                 term[4] = i/2+1;
01677                 term[7] = nexpr;
01678                 term[16] = nfac;
01679                 AT.WorkPointer = term + *term;
01680                 Generator(BHEAD term, C->numlhs);
01681             }
01682         }
01683         if ( EndSort(BHEAD AM.S0->sBuffer,0) < 0 ) {
01684             LowerSortLevel();
01685             Terminate(-1);
01686         }
01687 /*
01688         Set the file back into reading position
01689 */
01690         SetScratch(file, &pos);
01691         nfac = (nfac+1)/2;
01692         if ( expriszero ) { nfac = 1; }
01693     }
01694     if ( AT.bracketinfo != 0 ) ClearBracketIndex(-1);
01695     AT.bracketinfo = expr->newbracketinfo;
01696     expr->newbracketinfo = 0;
01697 /*
01698     Reset the brackets to make them ready for the final pass
01699 */
01700     AR.BracketOn = oldBracketOn;
01701     AT.BrackBuf = oldBrackBuf;
01702     if ( AR.BracketOn ) OpenBracketIndex(nexpr);
01703 /*
01704     We distinguish two cases: nfac == 2 and nfac == 1
01705     After preparing the term we skip the factor_ part.
01706 */
01707     NewSort(BHEAD0);
01708     if ( expriszero == 0 ) {
01709         if ( nfac == 1 ) {
01710             t = genericterm2; w = term = oldworkpointer;
01711             j = *t; NCOPY(w,t,j);
01712             term[7] = nexpr;
01713             term[16] = nfac;
01714         }
01715         else if ( nfac == 2 ) {
01716             t = genericterm; w = term = oldworkpointer;
01717             j = *t; NCOPY(w,t,j);
01718             term[7] = nexpr;
01719             term[16] = 1;
01720             term[22] = nexpr;
01721             term[31] = 2;
01722         }
01723         else {
01724             AS.OldOnFile[nexpr] = oldpos;
01725             return(-1);
01726         }
01727         term[4] = term[0]-4;
01728         term += 4;
01729         AT.WorkPointer = term + *term;
01730         Generator(BHEAD term, C->numlhs);
```

```
01731     }
01732     if ( EndSort(BHEAD AM.S0->sBuffer,0) < 0 ) {
01733         LowerSortLevel();
01734         Terminate(-1);
01735     }
01736 /*
01737     Final Cleanup
01738 */
01739     expr->numfactors = 0;
01740     expr->vflags &= ~ISFACTORIZED;
01741     if ( AT.bracketinfo != 0 ) ClearBracketIndex(-1);
01742 
01743     AR.infile = oldinfile;
01744     AR.outfile = oldoutfile;
01745     strcpy((char*)AC.Commercial, oldCommercial);
01746     AT.WorkPointer = oldworkpointer;
01747     AS.OldOnFile[nexpr] = oldpos;
01748 
01749     return(0);
01750 }
01751 
01752 /*
01753     #] poly_unfactorize_expression :
01754     #[ poly_inverse :
01755 */
01756 
01757 WORD *poly_inverse(PHEAD WORD *arga, WORD *argb) {
01758 
01759 #ifdef WITHFLINT
01760     if ( AC.FlintPolyFlag && AC.ncmod==0 ) {
01761         return flint_inverse(BHEAD arga, argb);
01762     }
01763 #endif
01764 
01765 #ifdef DEBUG
01766     cout « "*** [" « thetime() « "]  CALL : poly_inverse" « endl;
01767 #endif
01768 
01769     // Extract variables
01770     vector<WORD *> e;
01771     e.reserve(2);
01772     e.push_back(arga);
01773     e.push_back(argb);
01774     poly::get_variables(BHEAD e, false, true);
01775 
01776     if (AN.poly_num_vars > 1) {
01777         MLOCK(ErrorMessageLock);
01778         MesPrint ((char*)"ERROR: multivariate polynomial inverse is generally impossible");
01779         MUNLOCK(ErrorMessageLock);
01780         Terminate(-1);
01781     }
01782 
01783     poly finalden(BHEAD 1), finalres(BHEAD 1);
01784     int ressize = 0;
01785     WORD *res = NULL;
01786 
01787     // Convert to polynomials
01788     poly dena(BHEAD 0); // We need to keep the overall denominator of arga, to multiply the result
01789     poly a(poly::argument_to_poly(BHEAD arga, false, true, &dena));
01790     poly b(poly::argument_to_poly(BHEAD argb, false, true));
01791 
01792     // Divide out the integer content, FORM has not already done this.
01793     poly content_a(BHEAD 0), content_b(BHEAD 0);
01794     content_a = polygcd::integer_content(a);
01795     content_b = polygcd::integer_content(b);
01796     a /= content_a;
01797     b /= content_b;
01798 
01799     poly invamodp(BHEAD 0), invbmodp(BHEAD 0);
01800     WORD modp = 0;
01801 
01802     // Special cases:
01803     // Possibly strange that we give 1 for inverse_(x1,1) but here we take MMA's convention.
01804     if ((a.is_one() && b.is_one()) || a.is_one()) {
01805         finalres = poly(BHEAD 1);
01806     }
01807     else if (b.is_one()) {
01808         finalres = poly(BHEAD 0);
01809     }
01810     else {
01811         // Check for modulus calculus
01812         modp=poly_determine_modulus(BHEAD true, true, "polynomial inverse");
01813         const bool mod_calc = modp==0 ? false : true;
01814         a.setmod(modp,1);
01815         b.setmod(modp,1);
01816 
01817         // Check the gcd of a,b: if it is != 1, the inverse does not exist.
```

```
01818         poly gcd(polygcd::gcd(a,b));
01819         if (!gcd.is_one()) {
01820             MLOCK(ErrorMessageLock);
01821             if (mod_calc) {
01822                 MesPrint ((char*)"ERROR: polynomial inverse does not exist (mod %d)", modp);
01823             }
01824             else {
01825                 MesPrint ((char*)"ERROR: polynomial inverse does not exist");
01826             }
01827             MUNLOCK(ErrorMessageLock);
01828             Terminate(-1);
01829         }
01830 
01831         bool inv_exists = true;
01832         do {
01833             // If we are not using modulus calculus, find a suitable prime for xgcd:
01834             if (!mod_calc) {
01835                 vector<int> x(1,0);
01836                 modp = polyfact::choose_prime(a.integer_lcoeff()*b.integer_lcoeff(), x, modp);
01837             }
01838 
01839             poly amodp(a,modp,1);
01840             poly bmodp(b,modp,1);
01841 
01842             // Calculate gcd
01843             vector<poly> xgcd(polyfact::extended_gcd_Euclidean_lifted(amodp,bmodp));
01844             invamodp = poly(xgcd[0]);
01845             invbmodp = poly(xgcd[1]);
01846 
01847             inv_exists = ((invamodp * amodp) % bmodp).is_one();
01848             if (!inv_exists && mod_calc) {
01849                 // Control should not reach here!
01850                 MLOCK(ErrorMessageLock);
01851                 MesPrint ((char*)"ERROR: polynomial inverse does not exist (mod %d) B", modp);
01852                 MUNLOCK(ErrorMessageLock);
01853                 Terminate(-1);
01854             }
01855 
01856         // If the inverse does not exist and we are not working in modulus calculus,
01857         // choose a new prime and try again. This loop should always terminate, as
01858         // we have already checked that gcd(a,b) == 1.
01859         } while (!inv_exists);
01860 
01861         // estimate of the size of the Form notation; might be extended later
01862         ressize = invamodp.size_of_form_notation()+1;
01863         res = (WORD *)Malloc1(ressize*sizeof(WORD), "poly_inverse");
01864 
01865         // initialize polynomials to store the result
01866         poly primepower(BHEAD modp);
01867         poly inva(invamodp,modp,1);
01868         poly invb(invbmodp,modp,1);
01869 
01870         while (true) {
01871             // convert to Form notation
01872             int j=0;
01873             WORD n=0;
01874             for (int i=1; i<inva[0]; i+=inva[i]) {
01875 
01876                 // check whether res should be extended
01877                 while (ressize < j + 2*ABS(inva[i+inva[i]-1]) + (inva[i+1]>0?4:0) + 3) {
01878                     int newressize = 2*ressize;
01879 
01880                     WORD *newres = (WORD *)Malloc1(newressize*sizeof(WORD), "poly_inverse");
01881                     WCOPY(newres, res, ressize);
01882                     M_free(res, "poly_inverse");
01883                     res = newres;
01884                     ressize = newressize;
01885                 }
01886 
01887                 res[j] = 1;
01888                 if (inva[i+1]>0) {
01889                     res[j+res[j]++] = SYMBOL;
01890                     res[j+res[j]++] = 4;
01891                     res[j+res[j]++] = AN.poly_vars[0];
01892                     res[j+res[j]++] = inva[i+1];
01893                 }
01894                 MakeLongRational(BHEAD (UWORD *)&inva[i+2], inva[i+inva[i]-1],
01895                                                     (UWORD*)&primepower.terms[3],
      primepower.terms[primepower.terms[1]],
01896                                                     (UWORD *)&res[j+res[j]], &n);
01897                 res[j] += 2*ABS(n);
01898                 res[j+res[j]++] = SGN(n)*(2*ABS(n)+1);
01899                 j += res[j];
01900             }
01901             res[j]=0;
01902 
01903             // if modulus calculus is set, this is the answer
```

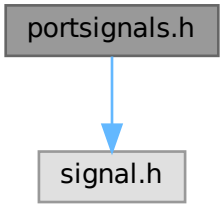
```
01904             if (a.modp != 0) break;
01905
01906             // otherwise check over integers
01907             poly den(BHEAD 0);
01908             poly check(poly::argument_to_poly(BHEAD res, false, true, &den));
01909             // Shortcut: if b's lcoeff doesn't divide the lcoeff of check*a,
01910             // b certainly doesn't divide check*a:
01911             if (poly::divides(b.integer_lcoeff(), check.integer_lcoeff()*a.integer_lcoeff())) {
01912                 check = check*a - den;
01913                 if (poly::divides(b, check)) break;
01914             }
01915
01916             // if incorrect, lift with quadratic p-adic Newton's iteration.
01917             poly error((poly(BHEAD 1) - a*inva - b*invb) / primepower);
01918             poly errormodpp(error, modp, inva.modn);
01919
01920             inva.modn *= 2;
01921             invb.modn *= 2;
01922
01923             poly dinva((inva * errormodpp) % b);
01924             poly dinvb((invb * errormodpp) % a);
01925
01926             inva += dinva * primepower;
01927             invb += dinvb * primepower;
01928
01929             primepower *= primepower;
01930         }
01931
01932         // One more round trip from form -> poly -> form, to multiply by dena in an easy way
01933         finalres = poly(poly::argument_to_poly(BHEAD res, false, true, &finalden));
01934     }
01935
01936     finalres *= dena;
01937     // The overall denominator additionally needs to be multiplied by content_a:
01938     finalden *= content_a;
01939     const WORD finalden_size = finalden.terms[finalden.terms[1]];
01940     const int finalsize = finalres.size_of_form_notation_with_den(finalden_size)+1;
01941     if (ressize < finalsize) {
01942         if (res != NULL) {
01943             M_free(res, "poly_inverse");
01944         }
01945         res = (WORD *)Malloc1(finalsize*sizeof(WORD), "poly_inverse");
01946     }
01947     poly::poly_to_argument_with_den(finalres, finalden_size,
01948         (UWORD*)&(finalden.terms[finalden.terms[1] - ABS(finalden_size)]), res, false);
01949
01950     // clean up and reset modulo calculation
01951     poly_free_poly_vars(BHEAD "AN.poly_vars_inverse");
01952
01953     AN.ncmod = AC.ncmod;
01954     return res;
01955 }
01956
01957 /*
01958     #] poly_inverse :
01959     #[ poly_mul :
01960 */
01961
01962 WORD *poly_mul(PHEAD WORD *a, WORD *b) {
01963
01964 #ifdef DEBUG
01965     cout << "*** [" << thetime() << "]  CALL : poly_mul" << endl;
01966 #endif
01967
01968 #ifdef WITHFLINT
01969     if ( AC.FlintPolyFlag && AC.ncmod==0 ) {
01970         return flint_mul(BHEAD a, b);
01971     }
01972 #endif
01973
01974     // Extract variables
01975     vector<WORD *> e;
01976     e.reserve(2);
01977     e.push_back(a);
01978     e.push_back(b);
01979     poly::get_variables(BHEAD e, false, false);  // TODO: any performance effect by sort_vars=true?
01980
01981     // Convert to polynomials
01982     poly dena(BHEAD 0);
01983     poly denb(BHEAD 0);
01984     poly pa(poly::argument_to_poly(BHEAD a, false, true, &dena));
01985     poly pb(poly::argument_to_poly(BHEAD b, false, true, &denb));
01986
01987     // Check for modulus calculus
01988     WORD modp = poly_determine_modulus(BHEAD true, true, "polynomial multiplication");
01989     pa.setmod(modp, 1);
01990
```

```
01991     // NOTE: mul_ is currently implemented by translating negative powers of
01992     // symbols to extra symbols. For future improvement, it may be better to
01993     // compute
01994     //   (content(a) * content(b)) * (a/content(a)) * (b/content(b))
01995     // to avoid introducing extra symbols for "mixed" cases, e.g.,
01996     // (1+x) * (1/x) -> (1+x) * (1+Z1_).
01997     assert(dena.is_integer());
01998     assert(denb.is_integer());
01999     assert(modp == 0 || dena.is_one());
02000     assert(modp == 0 || denb.is_one());
02001
02002     // multiplication
02003     pa *= pb;
02004
02005     // convert to Form notation
02006     WORD *res;
02007     if (dena.is_one() && denb.is_one()) {
02008         res = (WORD *)Malloc1((pa.size_of_form_notation() + 1) * sizeof(WORD), "poly_mul");
02009         poly::poly_to_argument(pa, res, false);
02010     } else {
02011         dena *= denb;
02012         res = (WORD *)Malloc1((pa.size_of_form_notation_with_den(dena[dena[1]]) + 1) * sizeof(WORD),
      "poly_mul");
02013         poly::poly_to_argument_with_den(pa, dena[dena[1]], (const UWORD *)&dena[2+AN.poly_num_vars],
      res, false);
02014     }
02015
02016     // clean up and reset modulo calculation
02017     poly_free_poly_vars(BHEAD "AN.poly_vars_mul");
02018     AN.ncmod = AC.ncmod;
02019
02020     return res;
02021 }
02022
02023 /*
02024     #] poly_mul :
02025     #[ poly_free_poly_vars :
02026 */
02027
02028 void poly_free_poly_vars(PHEAD const char *text)
02029 {
02030     if ( AN.poly_vars_type == 0 ) {
02031         TermFree(AN.poly_vars, text);
02032     }
02033     else {
02034         M_free(AN.poly_vars, text);
02035     }
02036     AN.poly_num_vars = 0;
02037     AN.poly_vars = 0;
02038 }
02039
02040 /*
02041     #] poly_free_poly_vars :
02042 */
```

4.113 portsignals.h File Reference

`#include <signal.h>`

Include dependency graph for portsignals.h:



Macros

- #define FATAL_SIG_ERROR 4
- #define NSIG (1024)
- #define SIGSEGV (NSIG+1)
- #define SIGFPE (NSIG+2)
- #define SIGILL (NSIG+3)
- #define SIGEMT (NSIG+4)
- #define SIGSYS (NSIG+5)
- #define SIGPIPE (NSIG+6)
- #define SIGLOST (NSIG+7)
- #define SIGXCPU (NSIG+8)
- #define SIGXFSZ (NSIG+9)
- #define SIGTERM (NSIG+10)
- #define SIGINT (NSIG+11)
- #define SIGQUIT (NSIG+12)
- #define SIGHUP (NSIG+13)
- #define SIGALRM (NSIG+14)
- #define SIGVTALRM (NSIG+15)
- #define SIGPROF (NSIG+16)

4.113.1 Detailed Description

Contains definitions for signals used/intercepted in FORM.

Some systems (especially LINUX) have not enough signals available so some of the (!documented!) signals are not defined. This file contains the definition of all signals used in the program. If the signal is not defined we define it as unused (>NSIG).

The include of signal.h must be first, before we try to define undefined signals.

Definition in file portsignals.h.

4.113.2 Macro Definition Documentation

4.113.2.1 FATAL_SIG_ERROR

```
#define FATAL_SIG_ERROR 4
```

Definition at line 47 of file portsignals.h.

4.113.2.2 NSIG

```
#define NSIG (1024)
```

Definition at line 53 of file portsignals.h.

4.113.2.3 SIGSEGV

```
#define SIGSEGV (NSIG+1)
```

Definition at line 57 of file portsignals.h.

4.113.2.4 SIGFPE

```
#define SIGFPE (NSIG+2)
```

Definition at line 60 of file portsignals.h.

4.113.2.5 SIGILL

```
#define SIGILL (NSIG+3)
```

Definition at line 63 of file portsignals.h.

4.113.2.6 SIGEMT

```
#define SIGEMT (NSIG+4)
```

Definition at line 66 of file portsignals.h.

4.113.2.7 SIGSYS

```
#define SIGSYS (NSIG+5)
```

Definition at line 69 of file portsignals.h.

4.113.2.8 SIGPIPE

```
#define SIGPIPE (NSIG+6)
```

Definition at line 72 of file portsignals.h.

4.113.2.9 SIGLOST

```
#define SIGLOST (NSIG+7)
```

Definition at line 75 of file portsignals.h.

4.113.2.10 SIGXCPU

```
#define SIGXCPU (NSIG+8)
```

Definition at line 78 of file portsignals.h.

4.113.2.11 SIGXFSZ

```
#define SIGXFSZ (NSIG+9)
```

Definition at line 81 of file portsignals.h.

4.113.2.12 SIGTERM

```
#define SIGTERM (NSIG+10)
```

Definition at line 84 of file portsignals.h.

4.113.2.13 SIGINT

```
#define SIGINT (NSIG+11)
```

Definition at line 87 of file portsignals.h.

4.113.2.14 SIGQUIT

```
#define SIGQUIT (NSIG+12)
```

Definition at line 90 of file portsignals.h.

4.113.2.15 SIGHUP

```
#define SIGHUP (NSIG+13)
```

Definition at line 93 of file portsignals.h.

4.113.2.16 SIGALRM

```
#define SIGALRM (NSIG+14)
```

Definition at line 96 of file portsignals.h.

4.113.2.17 SIGVTALRM

```
#define SIGVTALRM (NSIG+15)
```

Definition at line 99 of file portsignals.h.

4.113.2.18 SIGPROF

```
#define SIGPROF (NSIG+16)
```

Definition at line 102 of file portsignals.h.

4.114 portsignals.h

Go to the documentation of this file.

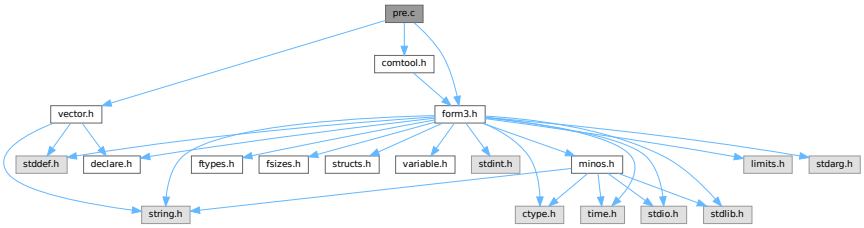
```
00001 #ifndef PORTSIGNAL_H
00002 #define PORTSIGNAL_H
00003
00018 /* #[ License : */
00019 /*
00020  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00021  *   When using this file you are requested to refer to the publication
00022  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00023  *   This is considered a matter of courtesy as the development was paid
00024  *   for by FOM the Dutch physics granting agency and we would like to
00025  *   be able to track its scientific use to convince FOM of its value
00026  *   for the community.
00027  *
00028  *   This file is part of FORM.
00029  *
00030  *   FORM is free software: you can redistribute it and/or modify it under the
00031  *   terms of the GNU General Public License as published by the Free Software
00032  *   Foundation, either version 3 of the License, or (at your option) any later
00033  *   version.
00034  *
00035  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00036  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00037  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00038  *   details.
00039  *
00040  *   You should have received a copy of the GNU General Public License along
00041  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00042  */
00043 /* #] License : */
00044
00045 #include <signal.h>
00046
00047 #define FATAL_SIG_ERROR 4
00048
00049 #ifndef NSIG
00050 /*
00051     The value of NSIG must be enough to fall outside the range of defined signals
00052 */
00053 #define NSIG (1024)
00054 #endif
00055
00056 #ifndef SIGSEGV
00057 #define SIGSEGV (NSIG+1)
00058 #endif
00059 #ifndef SIGFPE
00060 #define SIGFPE (NSIG+2)
00061 #endif
00062 #ifndef SIGILL
00063 #define SIGILL (NSIG+3)
00064 #endif
00065 #ifndef SIGEMT
00066 #define SIGEMT (NSIG+4)
00067 #endif
00068 #ifndef SIGSYS
00069 #define SIGSYS (NSIG+5)
00070 #endif
00071 #ifndef SIGPIPE
00072 #define SIGPIPE  (NSIG+6)
00073 #endif
00074 #ifndef SIGLOST
00075 #define SIGLOST (NSIG+7)
00076 #endif
00077 #ifndef SIGXCPU
00078 #define SIGXCPU  (NSIG+8)
00079 #endif
00080 #ifndef SIGXFSZ
00081 #define SIGXFSZ (NSIG+9)
00082 #endif
00083 #ifndef SIGTERM
00084 #define SIGTERM (NSIG+10)
00085 #endif
00086 #ifndef SIGINT
00087 #define SIGINT (NSIG+11)
00088 #endif
00089 #ifndef SIGQUIT
00090 #define SIGQUIT (NSIG+12)
00091 #endif
00092 #ifndef SIGHUP
00093 #define SIGHUP (NSIG+13)
00094 #endif
00095 #ifndef SIGALRM
00096 #define SIGALRM (NSIG+14)
```

```
00097 #endif
00098 #ifndef SIGVTALRM
00099 #define SIGVTALRM (NSIG+15)
00100 #endif
00101 #ifndef SIGPROF
00102 #define SIGPROF (NSIG+16)
00103 #endif
00104
00105 #endif
```

4.115 pre.c File Reference

```
#include "form3.h"
#include "comtool.h"
#include "vector.h"
```

Include dependency graph for pre.c:



Macros

- #define STRINGIFY(x) STRINGIFY__(x)
- #define STRINGIFY__(x) #x
- #define SKIPBUFSIZE 20
- #define KILL "kill"
- #define KILLALL "killall"
- #define DAEMON "daemon"
- #define SHELL "shell"
- #define STDERR "stderr"
- #define TRUE_EXPR "true"
- #define FALSE_EXPR "false"
- #define NOSHELL "noshell"
- #define TERMINAL "terminal"

Functions

- UBYTE GetInput (void)
- void ClearPushback (void)
- UBYTE GetChar (int level)
- void CharOut (UBYTE c)
- void UnsetAllowDelay (void)
- UBYTE ∗ GetPreVar (UBYTE ∗name, int flag)
- int PutPreVar (UBYTE ∗name, UBYTE ∗value, UBYTE ∗args, int mode)
- void PopPreVars (int tonumber)
- void IniModule (int type)

- void IniSpecialModule (int type)
- void PreProcessor (void)
- int PreProInstruction (void)
- int LoadInstruction (int mode)
- int LoadStatement (int type)
- int ExpandTripleDots (int par)
- KEYWORD * FindKeyWord (UBYTE *theword, KEYWORD *table, int size)
- KEYWORD * FindInKeyWord (UBYTE *theword, KEYWORD *table, int size)
- int TheDefine (UBYTE *s, int mode)
- int DoCommentChar (UBYTE *s)
- int DoPreAssign (UBYTE *s)
- int DoDefine (UBYTE *s)
- int DoRedefine (UBYTE *s)
- int ClearMacro (UBYTE *name)
- int TheUndefine (UBYTE *name)
- int DoUndefine (UBYTE *s)
- int DoInclude (UBYTE *s)
- int DoReverseInclude (UBYTE *s)
- int Include (UBYTE *s, int type)
- int DoPreExchange (UBYTE *s)
- int DoCall (UBYTE *s)
- int DoDebug (UBYTE *s)
- int DoTerminate (UBYTE *s)
- int DoContinueDo (UBYTE *s)
- int DoDo (UBYTE *s)
- int DoBreakDo (UBYTE *s)
- int DoElse (UBYTE *s)
- int DoElseif (UBYTE *s)
- int DoEnddo (UBYTE *s)
- int DoEndif (UBYTE *s)
- int DoEndprocedure (UBYTE *s)
- int DoIf (UBYTE *s)
- int DoIfdef (UBYTE *s, int par)
- int DoIfydef (UBYTE *s)
- int DoIfndef (UBYTE *s)
- int DoInside (UBYTE *s)
- int DoEndInside (UBYTE *s)
- int DoMessage (UBYTE *s)
- int DoPipe (UBYTE *s)
- int DoPrcExtension (UBYTE *s)
- int DoPreOut (UBYTE *s)
- int DoPrePrintTimes (UBYTE *s)
- int DoPreSortReallocate (UBYTE *s)
- int DoPreAppend (UBYTE *s)
- int DoPreCreate (UBYTE *s)
- int DoPreRemove (UBYTE *s)
- int DoPreClose (UBYTE *s)
- int DoPreWrite (UBYTE *s)
- int DoProcedure (UBYTE *s)
- int DoPreBreak (UBYTE *s)
- int DoPreCase (UBYTE *s)
- int DoPreDefault (UBYTE *s)
- int DoPreEndSwitch (UBYTE *s)
- int DoPreSwitch (UBYTE *s)
- int DoPreShow (UBYTE *s)

- int DoSystem (UBYTE *s)
- int PreLoad (PRELOAD *p, UBYTE *start, UBYTE *stop, int mode, char *message)
- int PreSkip (UBYTE *start, UBYTE *stop, int mode)
- void StartPrepro (void)
- int EvalPreIf (UBYTE *s)
- UBYTE * PreIfEval (UBYTE *s, int *value)
- int PreCmp (int type, int val, UBYTE *t, int type2, int val2, UBYTE *t2, int cmpop)
- int PreEq (int type, int val, UBYTE *t, int type2, int val2, UBYTE *t2, int eqop)
- UBYTE * pParseObject (UBYTE *s, int *type, LONG *val2)
- UBYTE * PreCalc (void)
- UBYTE * PreEval (UBYTE *s, LONG *x)
- void AddToPreTypes (int type)
- void MessPreNesting (int par)
- int DoPreAddSeparator (UBYTE *s)
- int DoPreRmSeparator (UBYTE *s)
- int DoExternal (UBYTE *s)
- int DoPrompt (UBYTE *s)
- int DoSetExternal (UBYTE *s)
- int DoSetExternalAttr (UBYTE *s)
- int DoRmExternal (UBYTE *s)
- int DoFromExternal (UBYTE *s)
- int DoToExternal (UBYTE *s)
- UBYTE * defineChannel (UBYTE *s, HANDLERS *h)
- int writeToChannel (int wtype, UBYTE *s, HANDLERS *h)
- int DoFactDollar (UBYTE *s)
- WORD GetDollarNumber (UBYTE **inp, DOLLARS d)
- int DoSetRandom (UBYTE *s)
- int DoOptimize (UBYTE *s)
- int DoClearOptimize (UBYTE *s)
- int DoSkipExtraSymbols (UBYTE *s)
- int DoPreReset (UBYTE *s)
- int DoPreAppendPath (UBYTE *s)
- int DoPrePrependPath (UBYTE *s)
- int DoTimeOutAfter (UBYTE *s)
- int DoNamespace (UBYTE *s)
- int DoEndNamespace (UBYTE *s)
- UBYTE * SkipName (UBYTE *s)
- UBYTE * ConstructName (UBYTE *s, UBYTE type)
- int DoUse (UBYTE *s)
- int UserFlags (UBYTE *s, int par)
- int DoClearUserFlag (UBYTE *s)
- int DoSetUserFlag (UBYTE *s)

4.115.1 Detailed Description

This is the preprocessor and all its routines.

Definition in file pre.c.

4.115.2 Macro Definition Documentation

4.115.2.1 STRINGIFY

```
#define STRINGIFY(
              x ) STRINGIFY__(x)
```

Definition at line 4342 of file pre.c.

4.115.2.2 STRINGIFY__

```
#define STRINGIFY__(
              x ) #x
```

Definition at line 4343 of file pre.c.

4.115.2.3 SKIPBUFSIZE

```
#define SKIPBUFSIZE 20
```

Definition at line 4466 of file pre.c.

4.115.2.4 KILL

```
#define KILL "kill"
```

Definition at line 5686 of file pre.c.

4.115.2.5 KILLALL

```
#define KILLALL "killall"
```

Definition at line 5687 of file pre.c.

4.115.2.6 DAEMON

```
#define DAEMON "daemon"
```

Definition at line 5688 of file pre.c.

4.115.2.7 SHELL

```
#define SHELL "shell"
```

Definition at line 5689 of file pre.c.

4.115.2.8 STDERR

```
#define STDERR "stderr"
```

Definition at line 5690 of file pre.c.

4.115.2.9 TRUE_EXPR

```
#define TRUE_EXPR "true"
```

Definition at line 5692 of file pre.c.

4.115.2.10 FALSE_EXPR

```
#define FALSE_EXPR "false"
```

Definition at line 5693 of file pre.c.

4.115.2.11 NOSHELL

```
#define NOSHELL "noshell"
```

Definition at line 5694 of file pre.c.

4.115.2.12 TERMINAL

```
#define TERMINAL "terminal"
```

Definition at line 5695 of file pre.c.

4.115.3 Function Documentation

4.115.3.1 GetInput()

```
UBYTE GetInput (
            void  )
```

Definition at line 138 of file pre.c.

4.115.3.2 ClearPushback()

```
void ClearPushback (
            void  )
```

Definition at line 167 of file pre.c.

4.115.3.3 GetChar()

```
UBYTE GetChar (
            int level )
```

Definition at line 185 of file pre.c.

4.115.3.4 CharOut()

```
void CharOut (
            UBYTE c )
```

Definition at line 495 of file pre.c.

4.115.3.5 UnsetAllowDelay()

```
void UnsetAllowDelay (
            void  )
```

Definition at line 516 of file pre.c.

4.115.3.6 GetPreVar()

```
UBYTE * GetPreVar (
            UBYTE * name,
            int flag )
```

Definition at line 542 of file pre.c.

4.115.3.7 PutPreVar()

```
int PutPreVar (
            UBYTE * name,
            UBYTE * value,
            UBYTE * args,
            int mode )
```

Inserts/Updates a preprocessor variable in the name administration.

Parameters

name	Character string with the variable name.
value	Character string with a possible value. Special case: if this argument is zero, then we have no value. Note: This is different from having an empty argument! This should only occur when the name starts with a ?
args	Character string with possible arguments.
mode	=0: always create a new name entry, =1: try to do a redefinition if possible.

Returns

Index of used entry in name list.

Definition at line 724 of file pre.c.

Referenced by ClearOptimize(), Generator(), Optimize(), PF_BroadcastRedefinedPreVars(), StartVariables(), and TheDefine().

4.115.3.8 PopPreVars()

```
void PopPreVars (
            int tonumber )
```

Definition at line 826 of file pre.c.

4.115.3.9 IniModule()

```
void IniModule (
            int type )
```

Definition at line 841 of file pre.c.

4.115.3.10 IniSpecialModule()

```
void IniSpecialModule (
            int type )
```

Definition at line 943 of file pre.c.

4.115.3.11 PreProcessor()

```
void PreProcessor (
            void  )
```

Definition at line 953 of file pre.c.

4.115.3.12 PreProInstruction()

```
int PreProInstruction (
            void  )
```

Definition at line 1172 of file pre.c.

4.115.3.13 LoadInstruction()

```
int LoadInstruction (
            int mode )
```

Definition at line 1269 of file pre.c.

4.115.3.14 LoadStatement()

```
int LoadStatement (
            int type )
```

Definition at line 1472 of file pre.c.

4.115.3.15 ExpandTripleDots()

```
int ExpandTripleDots (
            int par )
```

Definition at line 1610 of file pre.c.

4.115.3.16 FindKeyWord()

```
KEYWORD * FindKeyWord (
            UBYTE * theword,
            KEYWORD * table,
            int size )
```

Definition at line 1978 of file pre.c.

4.115.3.17 FindInKeyWord()

```
KEYWORD * FindInKeyWord (
            UBYTE * theword,
            KEYWORD * table,
            int size )
```

Definition at line 2009 of file pre.c.

4.115.3.18 TheDefine()

```
int TheDefine (
            UBYTE * s,
            int mode )
```

Preprocessor assignment. Possible arguments and values are treated and the new preprocessor variable is put into the name administration.

Parameters

s	Pointer to the character string following the preprocessor command.
mode	Bitmask. 0-bit clear: always create a new name entry, 0-bit set: try to redefine an existing name, 1-bit set: ignore preprocessor if/switch status.

Returns

zero: no errors, negative number: errors.

Definition at line 2039 of file pre.c.

References PutPreVar().

Here is the call graph for this function:

4.115.3.19 DoCommentChar()

```
int DoCommentChar (
            UBYTE * s )
```

Definition at line 2112 of file pre.c.

4.115.3.20 DoPreAssign()

```
int DoPreAssign (
            UBYTE * s )
```

Definition at line 2143 of file pre.c.

4.115.3.21 DoDefine()

```
int DoDefine (
            UBYTE * s )
```

Definition at line 2174 of file pre.c.

4.115.3.22 DoRedefine()

```
int DoRedefine (
            UBYTE * s )
```

Definition at line 2184 of file pre.c.

4.115.3.23 ClearMacro()

```
int ClearMacro (
            UBYTE * name )
```

Definition at line 2196 of file pre.c.

4.115.3.24 TheUndefine()

```
int TheUndefine (
            UBYTE * name )
```

Definition at line 2224 of file pre.c.

4.115.3.25 DoUndefine()

```
int DoUndefine (
            UBYTE * s )
```

Definition at line 2278 of file pre.c.

4.115.3.26 DoInclude()

```
int DoInclude (
            UBYTE * s )
```

Definition at line 2330 of file pre.c.

4.115.3.27 DoReverseInclude()

```
int DoReverseInclude (
            UBYTE * s )
```

Definition at line 2337 of file pre.c.

4.115.3.28 Include()

```
int Include (
            UBYTE * s,
            int type )
```

Definition at line 2344 of file pre.c.

4.115.3.29 DoPreExchange()

```
int DoPreExchange (
            UBYTE * s )
```

Definition at line 2505 of file pre.c.

4.115.3.30 DoCall()

```
int DoCall (
            UBYTE * s )
```

Definition at line 2566 of file pre.c.

4.115.3.31 DoDebug()

```
int DoDebug (
            UBYTE * s )
```

Definition at line 2742 of file pre.c.

4.115.3.32 DoTerminate()

```
int DoTerminate (
            UBYTE * s )
```

Definition at line 2779 of file pre.c.

4.115.3.33 DoContinueDo()

```
int DoContinueDo (
            UBYTE * s )
```

Jumps forward to the corresponding `#enddo` of the specified number of outer `#do` loops.

Syntax:

```
#continuedo [<number>=1]
```

If `number` is omitted, it defaults to 1. If `number` is zero then the instruction has no effect.

Definition at line 2813 of file pre.c.

4.115.3.34 DoDo()

```
int DoDo (
            UBYTE * s )
```

Definition at line 2877 of file pre.c.

4.115.3.35 DoBreakDo()

```
int DoBreakDo (
            UBYTE * s )
```

Definition at line 3092 of file pre.c.

4.115.3.36 DoElse()

```
int DoElse (
            UBYTE * s )
```

Definition at line 3170 of file pre.c.

4.115.3.37 DoElseif()

```
int DoElseif (
            UBYTE * s )
```

Definition at line 3207 of file pre.c.

4.115.3.38 DoEnddo()

```
int DoEnddo (
            UBYTE * s )
```

Definition at line 3242 of file pre.c.

4.115.3.39 DoEndif()

```
int DoEndif (
            UBYTE * s )
```

Definition at line 3388 of file pre.c.

4.115.3.40 DoEndprocedure()

```
int DoEndprocedure (
            UBYTE * s )
```

Definition at line 3416 of file pre.c.

4.115.3.41 DoIf()

```
int DoIf (
            UBYTE * s )
```

Definition at line 3443 of file pre.c.

4.115.3.42 DoIfdef()

```
int DoIfdef (
            UBYTE * s,
            int par )
```

Definition at line 3467 of file pre.c.

4.115.3.43 DoIfydef()

```
int DoIfydef (
            UBYTE * s )
```

Definition at line 3492 of file pre.c.

4.115.3.44 DoIfndef()

```
int DoIfndef (
            UBYTE * s )
```

Definition at line 3502 of file pre.c.

4.115.3.45 DoInside()

```
int DoInside (
            UBYTE * s )
```

Definition at line 3527 of file pre.c.

4.115.3.46 DoEndInside()

```
int DoEndInside (
            UBYTE * s )
```

Definition at line 3620 of file pre.c.

4.115.3.47 DoMessage()

```
int DoMessage (
            UBYTE * s )
```

Definition at line 3752 of file pre.c.

4.115.3.48 DoPipe()

```
int DoPipe (
            UBYTE * s )
```

Definition at line 3766 of file pre.c.

4.115.3.49 DoPrcExtension()

```
int DoPrcExtension (
            UBYTE * s )
```

Definition at line 3789 of file pre.c.

4.115.3.50 DoPreOut()

```
int DoPreOut (
            UBYTE * s )
```

Definition at line 3823 of file pre.c.

4.115.3.51 DoPrePrintTimes()

```
int DoPrePrintTimes (
            UBYTE * s )
```

Definition at line 3846 of file pre.c.

4.115.3.52 DoPreSortReallocate()

```
int DoPreSortReallocate (
            UBYTE * s )
```

Definition at line 3860 of file pre.c.

4.115.3.53 DoPreAppend()

```
int DoPreAppend (
            UBYTE * s )
```

Definition at line 3880 of file pre.c.

4.115.3.54 DoPreCreate()

```
int DoPreCreate (
            UBYTE * s )
```

Definition at line 3923 of file pre.c.

4.115.3.55 DoPreRemove()

```
int DoPreRemove (
            UBYTE * s )
```

Definition at line 3963 of file pre.c.

4.115.3.56 DoPreClose()

```
int DoPreClose (
            UBYTE * s )
```

Definition at line 3996 of file pre.c.

4.115.3.57 DoPreWrite()

```
int DoPreWrite (
            UBYTE * s )
```

Definition at line 4040 of file pre.c.

4.115.3.58 DoProcedure()

```
int DoProcedure (
            UBYTE * s )
```

Definition at line 4081 of file pre.c.

4.115.3.59 DoPreBreak()

```
int DoPreBreak (
            UBYTE * s )
```

Definition at line 4138 of file pre.c.

4.115.3.60 DoPreCase()

```
int DoPreCase (
            UBYTE * s )
```

Definition at line 4162 of file pre.c.

4.115.3.61 DoPreDefault()

```
int DoPreDefault (
            UBYTE * s )
```

Definition at line 4201 of file pre.c.

4.115.3.62 DoPreEndSwitch()

```
int DoPreEndSwitch (
            UBYTE * s )
```

Definition at line 4225 of file pre.c.

4.115.3.63 DoPreSwitch()

```
int DoPreSwitch (
            UBYTE * s )
```

Definition at line 4252 of file pre.c.

4.115.3.64 DoPreShow()

```
int DoPreShow (
            UBYTE * s )
```

Definition at line 4306 of file pre.c.

4.115.3.65 DoSystem()

```
int DoSystem (
            UBYTE * s )
```

Definition at line 4345 of file pre.c.

4.115.3.66 PreLoad()

```
int PreLoad (
            PRELOAD * p,
            UBYTE * start,
            UBYTE * stop,
            int mode,
            char * message )
```

Definition at line 4385 of file pre.c.

4.115.3.67 PreSkip()

```
int PreSkip (
            UBYTE * start,
            UBYTE * stop,
            int mode )
```

Definition at line 4468 of file pre.c.

4.115.3.68 StartPrepro()

```
void StartPrepro (
            void  )
```

Definition at line 4523 of file pre.c.

4.115.3.69 EvalPreIf()

```
int EvalPreIf (
            UBYTE * s )
```

Definition at line 4551 of file pre.c.

4.115.3.70 PreIfEval()

```
UBYTE * PreIfEval (
             UBYTE * s,
             int * value )
```

Definition at line 4586 of file pre.c.

4.115.3.71 PreCmp()

```
int PreCmp (
             int type,
             int val,
             UBYTE * t,
             int type2,
             int val2,
             UBYTE * t2,
             int cmpop )
```

Definition at line 4715 of file pre.c.

4.115.3.72 PreEq()

```
int PreEq (
             int type,
             int val,
             UBYTE * t,
             int type2,
             int val2,
             UBYTE * t2,
             int eqop )
```

Definition at line 4739 of file pre.c.

4.115.3.73 pParseObject()

```
UBYTE * pParseObject (
             UBYTE * s,
             int * type,
             LONG * val2 )
```

Definition at line 4768 of file pre.c.

4.115.3.74 PreCalc()

```
UBYTE * PreCalc (
             void  )
```

Definition at line 5204 of file pre.c.

4.115.3.75 PreEval()

```
UBYTE * PreEval (
            UBYTE * s,
            LONG * x )
```

Definition at line 5282 of file pre.c.

4.115.3.76 AddToPreTypes()

```
void AddToPreTypes (
            int type )
```

Definition at line 5432 of file pre.c.

4.115.3.77 MessPreNesting()

```
void MessPreNesting (
            int par )
```

Definition at line 5450 of file pre.c.

4.115.3.78 DoPreAddSeparator()

```
int DoPreAddSeparator (
            UBYTE * s )
```

Definition at line 5473 of file pre.c.

4.115.3.79 DoPreRmSeparator()

```
int DoPreRmSeparator (
            UBYTE * s )
```

Definition at line 5498 of file pre.c.

4.115.3.80 DoExternal()

```
int DoExternal (
            UBYTE * s )
```

Definition at line 5515 of file pre.c.

4.115.3.81 DoPrompt()

```
int DoPrompt (
            UBYTE * s )
```

Definition at line 5597 of file pre.c.

4.115.3.82 DoSetExternal()

```
int DoSetExternal (
            UBYTE * s )
```

Definition at line 5630 of file pre.c.

4.115.3.83 DoSetExternalAttr()

```
int DoSetExternalAttr (
            UBYTE * s )
```

Definition at line 5700 of file pre.c.

4.115.3.84 DoRmExternal()

```
int DoRmExternal (
            UBYTE * s )
```

Definition at line 5817 of file pre.c.

4.115.3.85 DoFromExternal()

```
int DoFromExternal (
            UBYTE * s )
```

Definition at line 5882 of file pre.c.

4.115.3.86 DoToExternal()

```
int DoToExternal (
            UBYTE * s )
```

Definition at line 6077 of file pre.c.

4.115.3.87 defineChannel()

```
UBYTE * defineChannel (
            UBYTE * s,
            HANDLERS * h )
```

Definition at line 6124 of file pre.c.

4.115.3.88 writeToChannel()

```
int writeToChannel (
            int wtype,
            UBYTE * s,
            HANDLERS * h )
```

Definition at line 6159 of file pre.c.

4.115.3.89 DoFactDollar()

```
int DoFactDollar (
            UBYTE * s )
```

Definition at line 6724 of file pre.c.

4.115.3.90 GetDollarNumber()

```
WORD GetDollarNumber (
            UBYTE ** inp,
            DOLLARS d )
```

Definition at line 6761 of file pre.c.

4.115.3.91 DoSetRandom()

```
int DoSetRandom (
            UBYTE * s )
```

Definition at line 6851 of file pre.c.

4.115.3.92 DoOptimize()

```
int DoOptimize (
            UBYTE * s )
```

Definition at line 6894 of file pre.c.

4.115.3.93 DoClearOptimize()

```
int DoClearOptimize (
            UBYTE * s )
```

Definition at line 7031 of file pre.c.

4.115.3.94 DoSkipExtraSymbols()

```
int DoSkipExtraSymbols (
            UBYTE * s )
```

Definition at line 7051 of file pre.c.

4.115.3.95 DoPreReset()

```
int DoPreReset (
            UBYTE * s )
```

Definition at line 7090 of file pre.c.

4.115.3.96 DoPreAppendPath()

```
int DoPreAppendPath (
            UBYTE * s )
```

Appends the given path (absolute or relative to the current file directory) to the FORM path.

Syntax: #appendpath <path>

Definition at line 7248 of file pre.c.

4.115.3.97 DoPrePrependPath()

```
int DoPrePrependPath (
            UBYTE * s )
```

Prepends the given path (absolute or relative to the current file directory) to the FORM path.

Syntax: #prependpath <path>

Definition at line 7265 of file pre.c.

4.115.3.98 DoTimeOutAfter()

```
int DoTimeOutAfter (
            UBYTE * s )
```

Definition at line 7277 of file pre.c.

4.115.3.99 DoNamespace()

```
int DoNamespace (
            UBYTE * s )
```

Definition at line 7329 of file pre.c.

4.115.3.100 DoEndNamespace()

```
int DoEndNamespace (
            UBYTE * s )
```

Definition at line 7377 of file pre.c.

4.115.3.101 SkipName()

```
UBYTE * SkipName (
            UBYTE * s )
```

Definition at line 7400 of file pre.c.

4.115.3.102 ConstructName()

```
UBYTE * ConstructName (
             UBYTE * s,
             UBYTE type )
```

Definition at line 7500 of file pre.c.

4.115.3.103 DoUse()

```
int DoUse (
             UBYTE * s )
```

Definition at line 7594 of file pre.c.

4.115.3.104 UserFlags()

```
int UserFlags (
             UBYTE * s,
             int par )
```

Definition at line 7642 of file pre.c.

4.115.3.105 DoClearUserFlag()

```
int DoClearUserFlag (
             UBYTE * s )
```

Definition at line 7749 of file pre.c.

4.115.3.106 DoSetUserFlag()

```
int DoSetUserFlag (
             UBYTE * s )
```

Definition at line 7759 of file pre.c.

4.116 pre.c

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
00026  *
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
00031 /*
00032     #[ Includes :
00033 */
00034 #include "form3.h"
00035 #include "comtool.h"
00036 #ifdef WITHFLOAT
00037 #include "math.h"
00038 #endif
00039
00040 static UBYTE pushbackchar = 0;
00041 static int oldmode = 0;
00042 static int stopdelay = 0;
00043 static STREAM *oldstream = 0;
00044 static UBYTE underscore[2] = {'_',0};
00045 static PREVAR *ThePreVar = 0;
00046
00047 static int ExitDoLoops(int, const char *);
00048
00049 static KEYWORD precommands[] = {
00050      {"add"           , DoPreAdd       , 0, 0}
00051     ,{"addseparator"  , DoPreAddSeparator,0,0}
00052     ,{"append"        , DoPreAppend    , 0, 0}
00053     ,{"appendpath"    , DoPreAppendPath, 0, 0}
00054     ,{"assign"        , DoPreAssign    , 0, 0}
00055     ,{"break"         , DoPreBreak     , 0, 0}
00056     ,{"breakdo"       , DoBreakDo      , 0, 0}
00057     ,{"call"          , DoCall         , 0, 0}
00058     ,{"case"          , DoPreCase      , 0, 0}
00059     ,{"clearflag"     , DoClearUserFlag, 0, 0}
00060     ,{"clearoptimize", DoClearOptimize, 0, 0}
00061     ,{"close"         , DoPreClose     , 0, 0}
00062     ,{"closedictionary", DoPreCloseDictionary,0,0}
00063     ,{"commentchar"   , DoCommentChar  , 0, 0}
00064     ,{"continuedo"  ,  DoContinueDo    , 0, 0}
00065     ,{"create"        , DoPreCreate    , 0, 0}
00066     ,{"debug"         , DoDebug        , 0, 0}
00067     ,{"default"       , DoPreDefault   , 0, 0}
00068     ,{"define"        , DoDefine       , 0, 0}
00069     ,{"do"            , DoDo           , 0, 0}
00070     ,{"else"          , DoElse         , 0, 0}
00071     ,{"elseif"        , DoElseif       , 0, 0}
00072     ,{"enddo"         , DoEnddo        , 0, 0}
00073 #ifdef WITHFLOAT
00074     ,{"endfloat"      , DoEndFloat     , 0, 0}
00075 #endif
00076     ,{"endif"         , DoEndif        , 0, 0}
00077     ,{"endinside"     , DoEndInside    , 0, 0}
00078     ,{"endnamespace"  , DoEndNamespace , 0, 0}
00079     ,{"endprocedure"  , DoEndprocedure , 0, 0}
00080     ,{"endswitch"     , DoPreEndSwitch , 0, 0}
00081     ,{"exchange"      , DoPreExchange  , 0, 0}
00082     ,{"external"      , DoExternal     , 0, 0}
00083     ,{"factdollar"    , DoFactDollar   , 0, 0}
00084     ,{"fromexternal"  , DoFromExternal , 0, 0}
00085     ,{"if"            , DoIf           , 0, 0}
```

```
00086     ,{"ifdef"        , DoIfydef       , 0, 0}
00087     ,{"ifndef"       , DoIfndef       , 0, 0}
00088     ,{"include"      , DoInclude      , 0, 0}
00089     ,{"inside"       , DoInside       , 0, 0}
00090     ,{"message"      , DoMessage      , 0, 0}
00091     ,{"namespace"    , DoNamespace    , 0, 0}
00092     ,{"opendictionary", DoPreOpenDictionary,0,0}
00093     ,{"optimize"     , DoOptimize     , 0, 0}
00094     ,{"pipe"         , DoPipe         , 0, 0}
00095     ,{"preout"       , DoPreOut       , 0, 0}
00096     ,{"prependpath"  , DoPrePrependPath,0, 0}
00097     ,{"printtimes"   , DoPrePrintTimes, 0, 0}
00098     ,{"procedure"    , DoProcedure    , 0, 0}
00099     ,{"procedureextension" , DoPrcExtension   , 0, 0}
00100     ,{"prompt"       , DoPrompt       , 0, 0}
00101     ,{"redefine"     , DoRedefine     , 0, 0}
00102     ,{"remove"       , DoPreRemove    , 0, 0}
00103     ,{"reset"        , DoPreReset     , 0, 0}
00104     ,{"reverseinclude"   , DoReverseInclude   , 0, 0}
00105     ,{"rmexternal"   , DoRmExternal   , 0, 0}
00106     ,{"rmseparator"  , DoPreRmSeparator,0, 0}
00107     ,{"setexternal"  , DoSetExternal  , 0, 0}
00108     ,{"setexternalattr"  , DoSetExternalAttr  , 0, 0}
00109     ,{"setflag"      , DoSetUserFlag  , 0, 0}
00110     ,{"setrandom"    , DoSetRandom    , 0, 0}
00111     ,{"show"         , DoPreShow      , 0, 0}
00112     ,{"skipextrasymbols" , DoSkipExtraSymbols , 0, 0}
00113     ,{"sortreallocate", DoPreSortReallocate , 0, 0}
00114 #ifdef WITHFLOAT
00115     ,{"startfloat"   , DoStartFloat   , 0, 0}
00116 #endif
00117     ,{"switch"       , DoPreSwitch    , 0, 0}
00118     ,{"system"       , DoSystem       , 0, 0}
00119     ,{"terminate"    , DoTerminate    , 0, 0}
00120     ,{"timeoutafter" , DoTimeOutAfter , 0, 0}
00121     ,{"toexternal"   , DoToExternal   , 0, 0}
00122     ,{"undefine"     , DoUndefine     , 0, 0}
00123     ,{"use"          , DoUse          , 0, 0}
00124     ,{"usedictionary", DoPreUseDictionary,0,0}
00125     ,{"write"        , DoPreWrite     , 0, 0}
00126 };
00127
00128 /*
00129     #] Includes :
00130     # [ PreProcessor :
00131         #[ GetInput :
00132
00133         Gets one input character. If we reach the end of a stream
00134         we pop to the previous stream and try again.
00135         If there are no more streams we let this be known.
00136 */
00137
00138 UBYTE GetInput(void)
00139 {
00140     UBYTE c;
00141     while ( AC.CurrentStream ) {
00142         c = GetFromStream(AC.CurrentStream);
00143         if ( c != ENDOFSTREAM ) {
00144 #ifdef WITHMPI
00145             if ( PF.me == MASTER
00146                  && AC.NoShowInput <= 0
00147                  && AC.CurrentStream->type != PREVARSTREAM )
00148 #else
00149             if ( AC.NoShowInput <= 0 && AC.CurrentStream->type != PREVARSTREAM )
00150 #endif
00151                 CharOut(c);
00152             return(c);
00153         }
00154         AC.CurrentStream = CloseStream(AC.CurrentStream);
00155         if ( stopdelay && AC.CurrentStream == oldstream ) {
00156             stopdelay = 0; AP.AllowDelay = 1;
00157         }
00158     }
00159     return(ENDOFINPUT);
00160 }
00161
00162 /*
00163         #] GetInput :
00164         #[ ClearPushback :
00165 */
00166
00167 void ClearPushback(void)
00168 {
00169     pushbackchar = 0;
00170 }
00171
00172 /*
```

```
00173         #] ClearPushback :
00174         #[ GetChar :
00175
00176         Reads one character. If it encounters a quote it immediately
00177         takes the whole preprocessor variable and opens a stream
00178         for it and starts reading the stream.
00179         Note that we have to take special precautions for escaped quotes.
00180         That is why we remember the previous character. We allow the
00181         (dubious?) construction of ending a stream with a backslash and
00182         then using it to escape an object in the parent stream.
00183 */
00184
00185 UBYTE GetChar(int level)
00186 {
00187     UBYTE namebuf[MAXPRENAMESIZE+2], c, *s, *t;
00188     static UBYTE lastchar, charinbuf = 0;
00189     int i, j, raiselow, olddelay;
00190     STREAM *stream;
00191     if ( level > 0 ) {
00192         lastchar = ' `';
00193         goto higherlevel;
00194     }
00195     if ( pushbackchar ) { c = pushbackchar; pushbackchar = 0; return(c); }
00196     if ( charinbuf ) { c = charinbuf; charinbuf = 0; return(c); }
00197     c = GetInput();
00198     for(;;) {
00199         if ( c == '\\' ) {
00200             charinbuf = GetInput();
00201             if ( charinbuf != LINEFEED ) {
00202                 pushbackchar = charinbuf;
00203                 charinbuf = 0;
00204                 break;
00205             }
00206             charinbuf = 0;  /* Escaped linefeed -> skip leading blanks */
00207             while ( ( c = GetInput() ) == ' ' || c == '\t' ) {}
00208         }
00209         else if ( c == '\"' || c == ' `' ) {
00210             if ( AP.DelayPrevar == 1 && c == '\" ) {
00211                 AP.DelayPrevar = 0;
00212                 break;
00213             }
00214             lastchar = c;
00215 higherlevel:
00216             c = GetInput();
00217             if ( c == '!' && lastchar == ' `' ) {
00218                 if ( stopdelay == 0 ) oldstream = AC.CurrentStream;
00219                 AP.AllowDelay = 0;
00220                 stopdelay = 1;
00221                 c = GetInput();
00222             }
00223             if ( c == '~' && lastchar == ' `' ) {
00224                 if ( AP.AllowDelay ) {
00225                     pushbackchar = c;
00226                     c = lastchar;
00227                     AP.DelayPrevar = 1;
00228                     break;
00229                 }
00230             }
00231             else {
00232                 pushbackchar = c;
00233             }
00234             olddelay = AP.DelayPrevar;
00235             AP.DelayPrevar = 0;
00236             i = 0; lastchar = 0;
00237             for (;;) {
00238                 if ( pushbackchar ) { c = pushbackchar; pushbackchar = 0; }
00239                 else { c = GetInput(); }
00240                 if ( c == ENDOFINPUT || ( ( c == '\" || c == LINEFEED )
00241                 && lastchar != '\\' ) ) {
00242                     break;
00243                 }
00244                 if ( c == '{' ) { /* Try the preprocessor calculator */
00245                     if ( PreCalc() == 0 ) Terminate(-1);
00246                     c = GetInput();    /* This is either a { or a number */
00247                     if ( c == '{' ) {
00248                         MesPrint("@Illegal set inside preprocessor variable name");
00249                         Terminate(-1);
00250                     }
00251                 }
00252                 if ( c == ' `' && lastchar != '\\' ) {
00253                     c = GetChar(1);
00254                     if ( c == ENDOFINPUT || ( ( c == '\" || c == LINEFEED )
00255                     && lastchar != '\\' ) ) {
00256                         break;
00257                     }
00258                 }
00259                 if ( lastchar == '\\' ) { i--; lastchar = 0; }
```

```
00260                   else lastchar = c;
00261                   namebuf[i++] = c;
00262                   if ( i > MAXPRENAMESIZE ) {
00263                       namebuf[i] = 0;
00264                       Error1("Preprocessor variable name too long: ",namebuf);
00265                   }
00266               }
00267               namebuf[i++] = 0;
00268               if ( c != '\"' ) {
00269                   Error1("Unmatched quotes for preprocessor variable",namebuf);
00270               }
00271               AP.DelayPrevar = olddelay;
00272               if ( namebuf[0] == '$' ) {
00273                   raiselow = PRENOACTION;
00274                   if ( AP.PreproFlag && *AP.preStart) {
00275                       s = EndOfToken(AP.preStart);
00276                       c = *s; *s = 0;
00277                       if ( ( StrICmp(AP.preStart,(UBYTE *)"ifdef") == 0
00278                       || StrICmp(AP.preStart,(UBYTE *)"ifndef") == 0 )
00279                       && GetDollar(namebuf+1) < 0 ) {
00280                           *s = c; c = ' ';
00281                           break;
00282                       }
00283                       *s = c;
00284                   }
00285                   else {
00286                       s = EndOfToken(namebuf+1);
00287                       if ( *s == '[' ) { while ( *s ) s++; }
00288                   }
00289                   if ( *s == '-' && s[1] == '-' && s[2] == 0 )
00290                       raiselow = PRELOWERAFTER;
00291                   else if ( *s == '+' && s[1] == '+' && s[2] == 0 )
00292                       raiselow = PRERAISEAFTER;
00293                   c = *s; *s = 0;
00294                   if ( OpenStream(namebuf+1,DOLLARSTREAM,0,raiselow) == 0 ) {
00295                       *s = c;
00296                       MesPrint("@Undefined variable %s used as preprocessor variable",
00297                           namebuf);
00298                       Terminate(-1);
00299                   }
00300                   *s = c;
00301               }
00302               else {
00303                   raiselow = PRENOACTION;
00304                   if ( AP.PreproFlag && *AP.preStart) {
00305                       s = EndOfToken(AP.preStart);
00306                       c = *s; *s = 0;
00307                       if ( ( StrICmp(AP.preStart,(UBYTE *)"ifdef") == 0
00308                       || StrICmp(AP.preStart,(UBYTE *)"ifndef") == 0 )
00309                       && GetPreVar(namebuf,WITHOUTERROR) == 0 ) {
00310                           *s = c; c = ' ';
00311                           break;
00312                       }
00313                       *s = c;
00314                   }
00315                   s = EndOfToken(namebuf);
00316                   if ( *s == '_' ) s++;
00317                   if ( *s == '-' && s[1] == '-' && s[2] == 0 )
00318                       raiselow = PRELOWERAFTER;
00319                   else if ( *s == '+' && s[1] == '+' && s[2] == 0 )
00320                       raiselow = PRERAISEAFTER;
00321                   else if ( *s == '(' && namebuf[i-2] == ')' ) {
00322 /*
00323                       Now count the arguments and separate them by zeroes
00324                       Check on the ?var construction and if present, reset
00325                       some comma's.
00326                       Make the assignments of the variables
00327                       Run the macro.
00328                       Undefine the variables
00329 */
00330                       int nargs = 1;
00331                       PREVAR *p;
00332                       size_t p_offset;
00333                       *s++ = 0; namebuf[i-2] = 0;
00334                       if ( StrICmp(namebuf,(UBYTE *)"random_") == 0 ) {
00335                           UBYTE *ranvalue;
00336                           ranvalue = PreRandom(s);
00337                           PutPreVar(namebuf,ranvalue,0,1);
00338                           M_free(ranvalue,"PreRandom");
00339                           goto dostream;
00340                       }
00341                       else if ( StrICmp(namebuf,(UBYTE *)"tolower_") == 0 ) {
00342                           UBYTE *ss = s;
00343                           while ( *ss ) { *ss = (UBYTE)(tolower(*ss)); ss++; }
00344                           PutPreVar(namebuf,s,0,1);
00345                           goto dostream;
00346                       }
```

```
00347                   else if ( StrICmp(namebuf,(UBYTE *)"toupper_") == 0 ) {
00348                       UBYTE *ss = s;
00349                       while ( *ss ) { *ss = (UBYTE)(toupper(*ss)); ss++; }
00350                       PutPreVar(namebuf,s,0,1);
00351                       goto dostream;
00352                   }
00353                   else if ( StrICmp(namebuf,(UBYTE *)"takeleft_") == 0 ) {
00354                       UBYTE *ss = s;
00355                       int x = 0, nsize;
00356                       while ( *ss != ',' && *ss ) ss++;
00357                       nsize = ss-s;
00358                       if ( *ss ) {
00359                           *ss++ = 0;
00360                           while ( FG.cTable[*ss] == 1 ) x = 10*x + (*ss++ - '0');
00361                           if ( x > nsize ) x = nsize;
00362                       }
00363                       else x = 0;
00364                       PutPreVar(namebuf,s+x,0,1);
00365                       goto dostream;
00366                   }
00367                   else if ( StrICmp(namebuf,(UBYTE *)"takeright_") == 0 ) {
00368                       UBYTE *ss = s;
00369                       int x = 0, nsize;
00370                       while ( *ss != ',' && *ss ) ss++;
00371                       nsize = ss-s;
00372                       if ( *ss ) {
00373                           *ss++ = 0;
00374                           while ( FG.cTable[*ss] == 1 ) x = 10*x + (*ss++ - '0');
00375                           if ( x > nsize ) x = nsize;
00376                       }
00377                       else x = 0;
00378                       x = nsize - x;
00379                       s[x] = 0;
00380                       PutPreVar(namebuf,s,0,1);
00381                       goto dostream;
00382                   }
00383                   else if ( StrICmp(namebuf,(UBYTE *)"keepleft_") == 0 ) {
00384                       UBYTE *ss = s;
00385                       int x = 0, nsize;
00386                       while ( *ss != ',' && *ss ) ss++;
00387                       nsize = ss-s;
00388                       if ( *ss ) {
00389                           *ss++ = 0;
00390                           while ( FG.cTable[*ss] == 1 ) x = 10*x + (*ss++ - '0');
00391                           if ( x > nsize ) x = nsize;
00392                       }
00393                       else x = nsize;
00394                       s[x] = 0;
00395                       PutPreVar(namebuf,s,0,1);
00396                       goto dostream;
00397                   }
00398                   else if ( StrICmp(namebuf,(UBYTE *)"keepright_") == 0 ) {
00399                       UBYTE *ss = s;
00400                       int x = 0, nsize;
00401                       while ( *ss != ',' && *ss ) ss++;
00402                       nsize = ss-s;
00403                       if ( *ss ) {
00404                           *ss++ = 0;
00405                           while ( FG.cTable[*ss] == 1 ) x = 10*x + (*ss++ - '0');
00406                           if ( x > nsize ) x = nsize;
00407                       }
00408                       else x = nsize;
00409                       x = nsize-x;
00410                       PutPreVar(namebuf,s+x,0,1);
00411                       goto dostream;
00412                   }
00413                   while ( *s ) {
00414                       if ( *s == '\\' ) s++;
00415                       if ( *s == ',' ) { *s = 0; nargs++; }
00416                       s++;
00417                   }
00418                   GetPreVar(namebuf,WITHERROR);
00419                   p = ThePreVar;
00420                   if ( p == 0 ) {
00421                       MesPrint("@Illegal use of arguments in preprocessor variable %s",namebuf);
00422                       Terminate(-1);
00423                   }
00424                   if ( p->nargs <= 0 || ( p->wildarg == 0 && nargs != p->nargs )
00425                       || ( p->wildarg > 0 && nargs < p->nargs-1 ) ) {
00426                       MesPrint("@Arguments of macro %s do not match",namebuf);
00427                       Terminate(-1);
00428                   }
00429                   if ( p->wildarg > 0 ) {
00430 /*
00431                       Change some zeroes into commas
00432 */
00433                       s = namebuf;
```

```
00434                           for ( j = 0; j < p->wildarg; j++ ) {
00435                               while ( *s ) s++;
00436                               s++;
00437                           }
00438                           for ( j = 0; j < nargs-p->nargs; j++ ) {
00439                               while ( *s ) s++;
00440                               *s++ = ',';
00441                           }
00442                       }
00443 /*
00444                           Now we can make the assignments
00445 */
00446                       s = namebuf;
00447                       while ( *s ) s++;
00448                       s++;
00449                       t = p->argnames;
00450                       p_offset = p - PreVar;
00451                       for ( j = 0; j < p->nargs; j++ ) {
00452                           if ( ( nargs == p->nargs-1 ) && ( *t == '?' ) ) {
00453                               PutPreVar(t,0,0,0);
00454                           }
00455                           else {
00456                               PutPreVar(t,s,0,0);
00457                               while ( *s ) s++;
00458                               s++;
00459                           }
00460                           p = PreVar + p_offset;
00461                           while ( *t ) t++;
00462                           t++;
00463                       }
00464                   }
00465 dostream:;
00466                   if ( ( stream = OpenStream(namebuf,PREVARSTREAM,0,raiselow) ) == 0 ) {
00467 /*
00468                       Eat comma before or after. This is `no value'
00469 */
00470                   }
00471                   else if ( stream->inbuffer == 0 ) {
00472                       c = GetInput();
00473                       if ( level > 0 && c == '\'' ) return(c);
00474                       goto endofloop;
00475                   }
00476               }
00477               c = GetInput();
00478           }
00479           else if ( c == '{' ) { /* Try the preprocessor calculator */
00480               if ( PreCalc() == 0 ) Terminate(-1);
00481               c = GetInput();    /* This is either a { or a number */
00482               break;
00483           }
00484           else break;
00485 endofloop:;
00486       }
00487       return(c);
00488 }
00489
00490 /*
00491           #] GetChar :
00492           #[ CharOut :
00493 */
00494
00495 void CharOut(UBYTE c)
00496 {
00497       if ( c == LINEFEED ) {
00498           AM.OutBuffer[AP.InOutBuf++] = c;
00499           WriteString(INPUTOUT,AM.OutBuffer,AP.InOutBuf);
00500           AP.InOutBuf = 0;
00501       }
00502       else {
00503           if ( AP.InOutBuf >= AM.OutBufSize || c == LINEFEED ) {
00504               WriteString(INPUTOUT,AM.OutBuffer,AP.InOutBuf);
00505               AP.InOutBuf = 0;
00506           }
00507           AM.OutBuffer[AP.InOutBuf++] = c;
00508       }
00509 }
00510
00511 /*
00512           #] CharOut :
00513           #[ UnsetAllowDelay :
00514 */
00515
00516 void UnsetAllowDelay(void)
00517 {
00518       if ( ThePreVar != 0 ) {
00519           if ( ThePreVar->nargs > 0 ) AP.AllowDelay = 0;
00520       }
```

```
00521 }
00522
00523 /*
00524         #] UnsetAllowDelay :
00525         #[ GetPreVar :
00526
00527         We use the model of a heap. If the same name has been used more
00528         than once the last definition is used. This gives the impression
00529         of local variables.
00530
00531         There are two types: The regular ones and the expression variables.
00532         The last ones are like UNCHANGED_exprname and ZERO_exprname or
00533         UNCHANGED_ and ZERO_.
00534 */
00535
00536 static UBYTE *yes = (UBYTE *)"1";
00537 static UBYTE *no  = (UBYTE *)"0";
00538 static UBYTE numintopolynomial[12];
00539 #include "vector.h"
00540 static Vector(UBYTE, exprstr);  /* Used for numactiveexprs_ and activeexprnames_. */
00541
00542 UBYTE *GetPreVar(UBYTE *name, int flag)
00543 {
00544     GETIDENTITY
00545     int i, mode;
00546     WORD number;
00547     UBYTE *t, c = 0, *tt = 0;
00548     t = name; while ( *t ) t++;
00549     if ( t[-1] == '-' && t[-2] == '-' && t-2 > name && t[-3] != '_' ) {
00550         t -= 2; c = *t; *t = 0; tt = t;
00551     }
00552     else if ( t[-1] == '+' && t[-2] == '+' && t-2 > name && t[-3] != '_' ) {
00553         t -= 2; c = *t; *t = 0; tt = t;
00554     }
00555     else if ( StrICmp(name,(UBYTE *)"time_") == 0 ) {
00556         UBYTE millibuf[24];
00557         LONG millitime, timepart;
00558         int timepart1, timepart2;
00559         static char timestring[40];
00560 /*      millitime = TimeCPU(1); */
00561         millitime = GetRunningTime();
00562         timepart = millitime%1000;
00563         millitime /= 1000;
00564         timepart /= 10;
00565         timepart1 = timepart / 10;
00566         timepart2 = timepart % 10;
00567         NumToStr(millibuf,millitime);
00568         snprintf(timestring,40,"%s.%1d%1d",millibuf,timepart1,timepart2);
00569         return((UBYTE *)timestring);
00570     }
00571     else if ( ( StrICmp(name,(UBYTE *)"timer_") == 0 )
00572           || ( StrICmp(name,(UBYTE *)"stopwatch_") == 0 ) ) {
00573         static char timestring[40];
00574         snprintf(timestring,40,"%ld",(long int)(GetRunningTime() - AP.StopWatchZero));
00575         return((UBYTE *)timestring);
00576     }
00577     else if ( StrICmp(name, (UBYTE *)"numactiveexprs_") == 0 ) {
00578         /* the number of active expressions */
00579         int n = 0;
00580         for ( i = 0; i < NumExpressions; i++ ) {
00581             EXPRESSIONS e = Expressions + i;
00582             switch ( e->status ) {
00583                 case LOCALEXPRESSION:
00584                 case GLOBALEXPRESSION:
00585                 case UNHIDELEXPRESSION:
00586                 case UNHIDEGEXPRESSION:
00587                 case INTOHIDELEXPRESSION:
00588                 case INTOHIDEGEXPRESSION:
00589                     n++;
00590                     break;
00591             }
00592         }
00593         VectorReserve(exprstr, 41);  /* up to 128-bit */
00594         LongCopy(n, (char *)VectorPtr(exprstr));
00595         return VectorPtr(exprstr);
00596     }
00597     else if ( StrICmp(name, (UBYTE *)"activeexprnames_") == 0 ) {
00598         /* the list of active expressions separated by commas */
00599         int j = 0;
00600         VectorReserve(exprstr, 16);  /* at least 1 character for '\0' */
00601         for ( i = 0; i < NumExpressions; i++ ) {
00602             UBYTE *p, *s;
00603             int len, k;
00604             EXPRESSIONS e = Expressions + i;
00605             switch ( e->status ) {
00606                 case LOCALEXPRESSION:
00607                 case GLOBALEXPRESSION:
```

```
00608                 case UNHIDELEXPRESSION:
00609                 case UNHIDEGEXPRESSION:
00610                 case INTOHIDELEXPRESSION:
00611                 case INTOHIDEGEXPRESSION:
00612                     s = AC.exprnames->namebuffer + e->name;
00613                     len = StrLen(s);
00614                     VectorSize(exprstr) = j;  /* j bytes must be copied in extending the buffer. */
00615                     VectorReserve(exprstr, j + len * 2 + 1);
00616                     p = VectorPtr(exprstr);
00617                     if ( j > 0 ) p[j++] = ',';
00618                     for ( k = 0; k < len; k++ ) {
00619                         if ( s[k] == ',' || s[k] == '|' ) p[j++] = '\\';
00620                         p[j++] = s[k];
00621                     }
00622                     break;
00623             }
00624         }
00625         VectorPtr(exprstr)[j] = '\0';
00626         return VectorPtr(exprstr);
00627     }
00628     else if ( StrICmp(name, (UBYTE *)"path_") == 0 ) {
00629         /* the current FORM path (for debugging both in .c and .frm) */
00630         if ( AM.Path ) {
00631             return(AM.Path);
00632         }
00633         else {
00634             return((UBYTE *)"");
00635         }
00636     }
00637     t = name;
00638     while ( *t && *t != '_' ) t++;
00639     for ( i = NumPre-1; i >= 0; i-- ) {
00640         if ( *t == '_' && ( StrICmp(name,PreVar[i].name) == 0 ) ) {
00641             if ( c ) *tt = c;
00642             ThePreVar = PreVar+i;
00643             return(PreVar[i].value);
00644         }
00645         else if ( StrCmp(name,PreVar[i].name) == 0 ) {
00646             if ( c ) *tt = c;
00647             ThePreVar = PreVar+i;
00648             return(PreVar[i].value);
00649         }
00650     }
00651     if ( *t == '_' ) {
00652         if ( StrICmp(name,(UBYTE *)"EXTRASYMBOLS_") == 0 ) goto extrashort;
00653         *t = 0;
00654         if ( StrICmp(name,(UBYTE *)"UNCHANGED") == 0 ) mode = 1;
00655         else if ( StrICmp(name,(UBYTE *)"ZERO") == 0 ) mode = 0;
00656         else if ( StrICmp(name,(UBYTE *)"SHOWINPUT") == 0 ) {
00657             *t++ = '_';
00658             if ( c ) *tt = c;
00659             if ( AC.NoShowInput > 0 ) return(no);
00660             else return(yes);
00661         }
00662         else if ( StrICmp(name,(UBYTE *)"EXTRASYMBOLS") == 0 ) {
00663             *t++ = '_';
00664 extrashort:;
00665             number = cbuf[AM.sbufnum].numrhs;
00666             t = numintopolynomial;
00667             NumCopy(number,t);
00668             return(numintopolynomial);
00669         }
00670         else mode = -1;
00671         *t++ = '_';
00672         if ( mode >= 0 ) {
00673             ThePreVar = 0;
00674             if ( *t ) {
00675                 if ( GetName(AC.exprnames,t,&number,NOAUTO) == CEXPRESSION ) {
00676                     if ( c ) *tt = c;
00677                     if ( ( Expressions[number].vflags & ( 1 << mode ) ) != 0 )
00678                         return(yes);
00679                     else return(no);
00680                 }
00681             }
00682             else {
00683 /*
00684                 Here we have to test all active results.
00685                 These are in `negative' so the flags have to be zero.
00686 */
00687                 if ( c ) *tt = c;
00688                 if ( ( AR.expflags & ( 1 << mode ) ) == 0 ) return(yes);
00689                 else return(no);
00690             }
00691         }
00692     }
00693     if ( ( t = (UBYTE *)(getenv((char *)(name))) ) != 0 ) {
00694         if ( c ) *tt = c;
```

```
00695         ThePreVar = 0;
00696         return(t);
00697     }
00698     if ( c ) *tt = c;
00699     if ( flag == WITHERROR ) {
00700         Error1("Undefined preprocessor variable",name);
00701     }
00702     return(0);
00703 }
00704
00705 /*
00706         #] GetPreVar :
00707         #[ PutPreVar :
00708 */
00709
00724 int PutPreVar(UBYTE *name, UBYTE *value, UBYTE *args, int mode)
00725 {
00726     int i, ii, num = 2, nnum = 2, numargs = 0;
00727     UBYTE *s, *t, *u = 0;
00728     PREVAR *p;
00729     if ( value == 0 && name[0] != '?' ) {
00730         MesPrint("@Illegal empty value for preprocessor variable %s",name);
00731         Terminate(-1);
00732     }
00733     if ( args ) {
00734         s = args; num++;
00735         while ( *s ) {
00736             if ( *s != ' ' && *s != '\t' ) num++;
00737             s++;
00738         }
00739     }
00740     if ( mode == 1 ) {
00741         i = NumPre;
00742         while ( --i >= 0 ) {
00743             if ( StrCmp(name,PreVar[i].name) == 0 ) {
00744                 u = PreVar[i].name;
00745                 break;
00746             }
00747         }
00748     }
00749     else i = -1;
00750     if ( i < 0 ) { p = (PREVAR *)FromList(&AP.PreVarList); ii = p - PreVar; }
00751     else         { p = &(PreVar[i]);             ii = i; }
00752     if ( value ) {
00753         s = value; while ( *s ) { s++; num++; }
00754     }
00755     else num = 1;
00756     if ( i >= 0 ) {
00757         if ( p->value ) {
00758             s = p->value;
00759             while ( *s ) { s++; nnum++; }
00760         }
00761         else nnum = 1;
00762         if ( nnum >= num ) {
00763 /*
00764             We can keep this in place
00765 */
00766             if ( value && p->value ) {
00767                 s = value;
00768                 t = p->value;
00769                 while ( *s ) *t++ = *s++;
00770                 *t = 0;
00771             }
00772             else p->value = 0;
00773             return(i);
00774         }
00775     }
00776     s = name;  while ( *s ) { s++; num++; }
00777     t = (UBYTE *)Malloc1(num,"PreVariable");
00778     p->name = t;
00779     s = name;  while ( *s ) *t++ = *s++; *t++ = 0;
00780     if ( value ) {
00781         p->value = t;
00782         s = value; while ( *s ) *t++ = *s++; *t = 0;
00783         if ( AM.atstartup && t[-1] == '\n' ) t[-1] = 0;
00784     }
00785     else p->value = 0;
00786     p->wildarg = 0;
00787     if ( args ) {
00788         int first = 1;
00789         t++; p->argnames = t;
00790         s = args;
00791         while ( *s ) {
00792             if ( *s == ' ' || *s == '\t' ) { s++; continue; }
00793             if ( *s == ',' ) {
00794                 s++; *t++ = 0; numargs++;
00795                 while ( *s == ' ' || *s == '\t' ) s++;
```

```
00796                 if ( *s == '?' ) {
00797                     if ( p->wildarg > 0 ) {
00798                         Error0("More than one ?var in #define");
00799                     }
00800                     p->wildarg = numargs;
00801                 }
00802             }
00803             else if ( *s == '?' && first ) {
00804                 p->wildarg = 1; *t++ = *s++;
00805             }
00806             else { *t++ = *s++; }
00807             first = 0;
00808         }
00809         *t = 0;
00810         numargs++;
00811         p->nargs = numargs;
00812     }
00813     else {
00814         p->nargs = 0;
00815         p->argnames = 0;
00816     }
00817     if ( u ) M_free(u,"replace PreVar value");
00818     return(ii);
00819 }
00820
00821 /*
00822         #] PutPreVar :
00823         #[ PopPreVars :
00824 */
00825
00826 void PopPreVars(int tonumber)
00827 {
00828     PREVAR *p = &(PreVar[NumPre]);
00829     while ( NumPre > tonumber ) {
00830         NumPre--; p--;
00831         M_free(p->name,"popping PreVar");
00832         p->name = p->value = 0;
00833     }
00834 }
00835
00836 /*
00837         #] PopPreVars :
00838         #[ IniModule :
00839 */
00840
00841 void IniModule(int type)
00842 {
00843     GETIDENTITY
00844     WORD **w, i;
00845     CBUF *C = cbuf+AC.cbufnum;
00846     /*[05nov2003 mt]:*/
00847 #ifdef WITHMPI
00848     /* To prevent
00849      *   (1) FlushOut() and PutOut() on the slaves to send a mess to the master
00850      *       compiling a module,
00851      *   (2) EndSort() called from poly_factorize_expression() on the master
00852      *       waits for the slaves.
00853      */
00854     PF.parallel=0;
00855     /*BTW, this was the bug preventing usage of more than 1 expression!*/
00856 #endif
00857
00858     AR.BracketOn = 0;
00859     AR.StoreData.dirtyflag = 0;
00860     AC.bracketindexflag = 0;
00861     AT.bracketindexflag = 0;
00862
00863 /*[06nov2003 mt]:*/
00864 #ifdef WITHMPI
00865     /* This flag may be set in the procedure tokenize(). */
00866     AC.RhsExprInModuleFlag = 0;
00867 /*[20oct2009 mt]:*/
00868     PF.mkSlaveInfile=0;
00869     PF.slavebuf.PObuffer=NULL;
00870     for(i=0; i<NumExpressions; i++)
00871         Expressions[i].vflags &= ~ISINRHS;
00872 /*:[20oct2009 mt]*/
00873 #endif
00874 /*:[06nov2003 mt]*/
00875
00876     /*[19nov2003 mt]:*/
00877     /*The module counter:*/
00878     (AC.CModule)++;
00879     /*:[19nov2003 mt]*/
00880
00881     if ( !type ) {
00882         if ( C->rhs ) {
```

```
00883             w = C->rhs; i = C->maxrhs;
00884             do { *w++ = 0; } while ( --i > 0 );
00885         }
00886         if ( C->lhs ) {
00887             w = C->lhs; i = C->maxlhs;
00888             do { *w++ = 0; } while ( --i > 0 );
00889         }
00890     }
00891     C->numlhs = C->numrhs = 0;
00892     ClearTree(AC.cbufnum);
00893     while ( AC.NumLabels > 0 ) {
00894         AC.NumLabels--;
00895         if ( AC.LabelNames[AC.NumLabels] ) M_free(AC.LabelNames[AC.NumLabels],"LabelName");
00896     }
00897
00898     C->Pointer = C->Buffer;
00899
00900     AC.Commercial[0] = 0;
00901
00902     AC.IfStack = AC.IfHeap;
00903     AC.arglevel = 0;
00904     AC.termlevel = 0;
00905     AC.IfLevel = 0;
00906     AC.WhileLevel = 0;
00907     AC.RepLevel = 0;
00908     AC.insidelevel = 0;
00909     AC.dolooplevel = 0;
00910     AC.MustTestTable = 0;
00911     AO.PrintType = 0;               /* Otherwise statistics can get spoiled */
00912     AC.ComDefer = 0;
00913     AC.CollectFun = 0;
00914     AM.S0->PolyWise = 0;
00915     AC.SymChangeFlag = 0;
00916     AP.lhdollarerror = 0;
00917     AR.PolyFun = AC.lPolyFun;
00918     AR.PolyFunInv = AC.lPolyFunInv;
00919     AR.PolyFunType = AC.lPolyFunType;
00920     AR.PolyFunExp = AC.lPolyFunExp;
00921     AR.PolyFunVar = AC.lPolyFunVar;
00922     AR.PolyFunPow = AC.lPolyFunPow;
00923     AC.mparallelflag = AC.parallelflag | AM.hparallelflag;
00924     AC.inparallelflag = 0;
00925     AC.mProcessBucketSize = AC.ProcessBucketSize;
00926     NumPotModdollars = 0;
00927     AC.topolynomialflag = 0;
00928 #ifdef WITHPTHREADS
00929     if ( AM.totalnumberofthreads > 1 ) AS.MultiThreaded = 1;
00930     else AS.MultiThreaded = 0;
00931     for ( i = 1; i < AM.totalnumberofthreads; i++ ) {
00932         AB[i]->T.S0->PolyWise = 0;
00933     }
00934 #endif
00935     OpenTemp();
00936 }
00937
00938 /*
00939         #] IniModule :
00940         #[ IniSpecialModule :
00941 */
00942
00943 void IniSpecialModule(int type)
00944 {
00945     DUMMYUSE(type);
00946 }
00947
00948 /*
00949         #] IniSpecialModule :
00950         #[ PreProcessor :
00951 */
00952
00953 void PreProcessor(void)
00954 {
00955     int moduletype = FIRSTMODULE;
00956     int specialtype = 0;
00957     int error1 = 0, error2 = 0, retcode, retval;
00958     UBYTE c, *t, *s;
00959     AP.StopWatchZero = GetRunningTime();
00960     AC.compiletype = 0;
00961     AP.PreContinuation = 0;
00962     AP.PreAssignLevel = 0;
00963     AP.gNumPre = NumPre;
00964     AC.iPointer = AC.iBuffer;
00965     AC.iPointer[0] = 0;
00966
00967     if ( AC.CheckpointFlag == -1 ) DoRecovery(&moduletype);
00968     AC.CheckpointStamp = Timer(0);
00969
```

```
00970     for(;;) {
00971 /*      if ( A.StatisticsFlag ) CharOut(LINEFEED); */
00972
00973         IniModule(moduletype);
00974
00975         /*Re-define preprocessor variable CMODULE_ as a current module number, starting from 1*/
00976         /*The module counter is AC.CModule, it is incremented in IniModule*/
00977         {
00978             UBYTE buf[24];/*64/Log_2[10] = 19.3, this is enough for any integer*/
00979             NumToStr(buf,AC.CModule);
00980             PutPreVar((UBYTE *)"CMODULE_",buf,0,1);
00981         }
00982
00983         if ( specialtype ) IniSpecialModule(specialtype);
00984
00985         for(;;) {   /* Read a single line/statement */
00986             c = GetChar(0);
00987             if ( c == AP.ComChar ) {  /* This line is commentary */
00988                 LoadInstruction(5);
00989                 if ( AC.CurrentStream->FoldName ) {
00990                     t = AP.preStart;
00991                     if ( *t && t[1] && t[2] == '#' && t[3] == ']' ) {
00992                         t += 4;
00993                         while ( *t == ' ' || *t == '\t' ) t++;
00994                         s = AC.CurrentStream->FoldName;
00995                         while ( *s == *t ) { s++; t++; }
00996                         if ( *s == 0 && ( *t == ' ' || *t == '\t'
00997                         || *t == ':' ) ) {
00998                             while ( *t == ' ' || *t == '\t' ) t++;
00999                             if ( *t == ':' ) {
01000                                 AC.CurrentStream = CloseStream(AC.CurrentStream);
01001                             }
01002                         }
01003                     }
01004                 }
01005                 *AP.preStart = 0;
01006                 continue;
01007             }
01008             while ( c == ' ' || c == '\t' ) c = GetChar(0);
01009             if ( c == LINEFEED ) continue;
01010             if ( c == ENDOFINPUT ) {
01011 /*              CharOut(LINEFEED); */
01012                 Warning(".end instruction generated");
01013                 moduletype = ENDMODULE; specialtype = 0;
01014                 goto endmodule; /* Fake one */
01015             }
01016             if ( c == '#' ) {
01017                 if ( PreProInstruction() ) { error1++; error2++; AP.preError++; }
01018                 *AP.preStart = 0;
01019             }
01020             else if ( c == '.' ) {
01021                 if ( ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) ||
01022                 ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) ) {
01023                     LoadInstruction(1);
01024                     continue;
01025                 }
01026                 if ( ModuleInstruction(&moduletype,&specialtype) ) { error2++; AP.preError++; }
01027                 if ( specialtype ) SetSpecialMode(moduletype,specialtype);
01028                 if ( AP.PreInsideLevel != 0 ) {
01029                     MesPrint("@end of module instructions may not be used inside");
01030                     MesPrint("@the scope of a %#inside %#endinside construction.");
01031                     Terminate(-1);
01032                 }
01033                 if ( AC.RepLevel > 0 ) {
01034                     MesPrint("&EndRepeat statement(s) missing");
01035                     error2++; AP.preError++;
01036                 }
01037                 if ( AC.tablecheck == 0 ) {
01038                     AC.tablecheck = 1;
01039                     if ( TestTables() ) { error2++; AP.preError++; }
01040                 }
01041                 if ( AP.PreContinuation ) {
01042                     error1++; error2++;
01043                     MesPrint("&Unfinished statement. Missing ;?");
01044                 }
01045                 if ( moduletype == GLOBALMODULE ) MakeGlobal();
01046                 else {
01047 endmodule:          if ( error2 == 0 && AM.qError == 0 ) {
01048                         retcode = ExecModule(moduletype);
01049 #ifdef WITHMPI
01050                         if(PF.slavebuf.PObuffer!=NULL){
01051                             M_free(PF.slavebuf.PObuffer,"PF inbuf");
01052                             PF.slavebuf.PObuffer=NULL;
01053                         }
01054 #endif
01055                         UpdatePositions();
01056                         if ( retcode < 0 ) error1++;
```

```
01057                       if ( retcode ) { error2++; AP.preError++; }
01058                   }
01059                   else {
01060                       EXPRESSIONS e;
01061                       WORD j;
01062                       for ( j = 0, e = Expressions; j < NumExpressions; j++, e++ ) {
01063                           if ( e->replace == NEWLYDEFINEDEXPRESSION ) e->replace =
    REGULAREXPRESSION;
01064                       }
01065                   }
01066                   switch ( moduletype ) {
01067                       case STOREMODULE:
01068                           if ( ExecStore() ) error1++;
01069                           break;
01070                       case CLEARMODULE:
01071                           FullCleanUp();
01072                           error1 = error2 = AP.preError = 0;
01073                           AM.atstartup = 1;
01074                           PutPreVar((UBYTE *)"DATE_",(UBYTE *)MakeDate(),0,1);
01075                           AM.atstartup = 0;
01076                           if ( AM.resetTimeOnClear ) {
01077 #ifdef WITHPTHREADS
01078                               ClearAllThreads();
01079 #endif
01080                               AM.SumTime += TimeCPU(1);
01081                               TimeCPU(0);
01082                           }
01083                           AP.StopWatchZero = GetRunningTime();
01084                           break;
01085                       case ENDMODULE:
01086                           Terminate( -( error1 | error2 ) );
01087                   }
01088               }
01089               AC.tablecheck = 0;
01090               AC.compiletype = 0;
01091               if ( AC.exprfillwarning > 0 ) {
01092                   AC.exprfillwarning = 0;
01093               }
01094               if ( AC.CheckpointFlag && error1 == 0 && error2 == 0 ) DoCheckpoint(moduletype);
01095               break;  /* start a new module */
01096           }
01097           else {
01098               if ( ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) ||
01099               ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) ) {
01100                   pushbackchar = c;
01101                   LoadInstruction(5);
01102                   continue;
01103               }
01104               UngetChar(c);
01105               if ( AP.PreContinuation ) {
01106                   retval = LoadStatement(OLDSTATEMENT);
01107               }
01108               else {
01109                   AC.CurrentStream->prevline = AC.CurrentStream->linenumber;
01110                   retval = LoadStatement(NEWSTATEMENT);
01111               }
01112               if ( retval < 0 ) {
01113                   error1++;
01114                   if ( retval == -1 ) AP.PreContinuation = 0;
01115                   else AP.PreContinuation = 1;
01116                   TryRecover(0);
01117               }
01118               else if ( retval > 0 ) AP.PreContinuation = 0;
01119               else AP.PreContinuation = 1;
01120               if ( error1 == 0 && !AP.PreContinuation ) {
01121                   if ( ( AP.PreDebug & PREPROONLY ) == 0 ) {
01122                       int onpmd = NumPotModdollars;
01123 #ifdef WITHMPI
01124                       WORD oldRhsExprInModuleFlag = AC.RhsExprInModuleFlag;
01125                       if ( AP.PreAssignFlag ) AC.RhsExprInModuleFlag = 0;
01126 #endif
01127                       if ( AP.PreOut || ( AP.PreDebug & DUMPTOCOMPILER )
01128                               == DUMPTOCOMPILER )
01129                                   MesPrint(" %s",AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]);
01130                       retcode = CompileStatement(AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]);
01131                       if ( retcode < 0 ) error1++;
01132                       if ( retcode ) { error2++; AP.preError++; }
01133                       if ( AP.PreAssignFlag ) {
01134                           if ( retcode == 0 ) {
01135                               if ( ( retcode = CatchDollar(0) ) < 0 ) error1++;
01136                               else if ( retcode > 0 ) { error2++; AP.preError++; }
01137                           }
01138                           else CatchDollar(-1);
01139                           POPPREASSIGNLEVEL;
01140                           if ( AP.PreAssignLevel <=0 )
01141                               AP.PreAssignFlag = 0;
01142                           NumPotModdollars = onpmd;
```

```
01143 #ifdef WITHMPI
01144                               AC.RhsExprInModuleFlag = oldRhsExprInModuleFlag;
01145 #endif
01146                           }
01147                       }
01148                       else {
01149                           MesPrint(" %s",AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]);
01150                       }
01151                   }
01152                   else if ( !AP.PreContinuation ) {
01153                       if ( AP.PreAssignLevel > 0 ) {
01154                           POPPREASSIGNLEVEL;
01155                           if ( AP.PreAssignLevel <=0 )
01156                               AP.PreAssignFlag = 0;
01157                       }
01158                   }
01159 /*
01160                   if ( !AP.PreContinuation ) AP.PreAssignFlag = 0;
01161 */
01162               }
01163           }
01164       }
01165 }
01166
01167 /*
01168           #] PreProcessor :
01169           #[ PreProInstruction :
01170 */
01171
01172 int PreProInstruction(void)
01173 {
01174     UBYTE *s, *t;
01175     KEYWORD *key;
01176     AP.PreproFlag = 1;
01177     AP.preFill = 0;
01178     AP.AllowDelay = 0;
01179     AP.DelayPrevar = 0;
01180
01181     oldmode = 0;
01182     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) {
01183         LoadInstruction(3);
01184         if ( ( StrICmp(AP.preStart,(UBYTE *)"case") == 0
01185         || StrICmp(AP.preStart,(UBYTE *)"default") == 0 )
01186         && AP.PreSwitchModes[AP.PreSwitchLevel] == SEARCHINGPRECASE ) {
01187             LoadInstruction(0);
01188         }
01189         else if ( StrICmp(AP.preStart,(UBYTE *)"assign ") == 0 ) {}
01190         else { LoadInstruction(1); }
01191     }
01192     else if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) {
01193         LoadInstruction(3);
01194         if ( ( StrICmp(AP.preStart,(UBYTE *)"else") == 0
01195         || StrICmp(AP.preStart,(UBYTE *)"elseif") == 0 )
01196         && AP.PreIfStack[AP.PreIfLevel] == LOOKINGFORELSE ) {
01197             LoadInstruction(0);
01198         }
01199         else if ( StrICmp(AP.preStart,(UBYTE *)"assign ") == 0 ) {}
01200         else {
01201             LoadInstruction(1);
01202         }
01203     }
01204     else {
01205         LoadInstruction(0);
01206     }
01207     AP.PreproFlag = 0;
01208     t = AP.preStart;
01209     if ( *t == '-' ) {
01210         if ( AP.PreSwitchModes[AP.PreSwitchLevel] == EXECUTINGPRESWITCH
01211             && AP.PreIfStack[AP.PreIfLevel] == EXECUTINGIF )
01212             AC.NoShowInput = 1;
01213     }
01214     else if ( *t == '+' ) {
01215         if ( AP.PreSwitchModes[AP.PreSwitchLevel] == EXECUTINGPRESWITCH
01216             && AP.PreIfStack[AP.PreIfLevel] == EXECUTINGIF )
01217             AC.NoShowInput = 0;
01218     }
01219     else if ( *t == ':' ) {
01220         AP.FoundFileSetupCount--;
01221         if ( AP.FoundFileSetupCount < 0 ) {
01222             // This implies that TryFileSetups found fewer #: setup lines than
01223             // are present in the file. This means that some were specified
01224             // out-of-order, i.e. after a non-comment non-#: line. Warn.
01225             MesPrint("Warning: ignoring out-of-place setup command:");
01226             MesPrint("@%#%s", t);
01227         }
01228     }
01229     else {
```

```
01230 retry:;
01231         key = FindKeyWord(t,precommands,sizeof(precommands)/sizeof(KEYWORD));
01232         s = EndOfToken(t);
01233         if ( key == 0 ) {
01234             if ( *s == ';' ) {
01235                 *s = 0; goto retry;
01236             }
01237             else {
01238                 *s = 0;
01239                 MesPrint("@Unrecognized preprocessor instruction: %s",t);
01240                 return(-1);
01241             }
01242         }
01243         while ( *s == ' ' || *s == '\t' || *s == ',' ) s++;
01244         t = s;
01245         while ( *t ) t++;
01246         while ( ( t[-1] == ';' ) && ( t[-2] != '\\' ) ) {
01247             t--; *t = 0;
01248         }
01249         return((key->func)(s));
01250     }
01251     return(0);
01252 }
01253
01254 /*
01255         #] PreProInstruction :
01256         #[ LoadInstruction :
01257
01258         0:  preprocessor instruction that may involve matching of brackets
01259         1:  runs straight to end-of-line
01260         2:  runs to ;
01261         3:  only gets one word without `' interpretation.
01262         5:  with pushbackchar, but inside commentary. -> 1
01263
01264 To be added:
01265     In define, redefine, call and listed do we may have delayed substitution
01266     of preprocessor variables.
01267 */
01268
01269 int LoadInstruction(int mode)
01270 {
01271     UBYTE *s, *sstart, *t, c, cp;
01272     LONG position, fillpos = 0;
01273     int bralevel = 0, parlevel = 0, first = 1;
01274     int quotelevel = 0;
01275     if ( AP.preFill ) {
01276         s = AP.preFill;
01277         AP.preFill = 0;
01278         if ( s[1] != LINEFEED && s[1] != ENDOFINPUT ) {
01279             s[0] = s[1]; s++;
01280         }
01281         else { oldmode = mode; return(0); }
01282     }
01283     else { s = AP.preStart; }
01284     sstart = s; *s = 0;
01285     for(;;) {
01286         if ( ( mode & 1 ) == 1 ) {
01287             if ( pushbackchar && ( mode == 3 || mode == 5 ) ) {
01288                 c = pushbackchar; pushbackchar = 0;
01289             }
01290             else c = GetInput();
01291         }
01292         else {
01293             c = GetChar(0);
01294         }
01295
01296         if ( mode == 2 && c == ';' ) break;
01297         if ( ( mode == 1 || mode == 5 ) && c == LINEFEED ) break;
01298         if ( mode == 3 && FG.cTable[c] != 0 ) {
01299             if ( c == '$' ) {
01300                 pushbackchar = '$';
01301                 *s++ = 'a'; *s++ = 's'; *s++ = 's'; *s++ = 'i';
01302                 *s++ = 'g'; *s++ = 'n'; *s++ = ' '; *s = 0;
01303             }
01304             if ( c == '\"' || c == '`' ) { /* we do not expand preprocessor variables */
01305                 mode = 1;
01306             }
01307             else {
01308                 AP.preFill = s; *s++ = 0; *s = c;
01309                 oldmode = mode;
01310                 return(0);
01311             }
01312         }
01313         if ( mode == 0 && first ) {
01314             if ( c == '$' ) {
01315 dodollar:       s = sstart;
01316                 *s++ = 'a'; *s++ = 's'; *s++ = 's'; *s++ = 'i';
```

```
01317                 *s++ = 'g'; *s++ = 'n'; *s = 0;
01318                 pushbackchar = c;
01319                 oldmode = mode;
01320                 return(0);
01321             }
01322             if ( c == ' ' || c == '\t' || c == ',' ) {}
01323             else first = 0;
01324         }
01325         else if ( mode == 1 && first && c == '$' && oldmode == 3 ) goto dodollar;
01326         if ( c == ENDOFINPUT || ( c == LINEFEED
01327 /*      && bralevel == 0  */
01328         && quotelevel == 0 ) ) {
01329             if ( mode == 2 && c == ENDOFINPUT ) {
01330                 MesPrint("@Unexpected end of instruction");
01331                 oldmode = mode;
01332                 return(-1);
01333             }
01334 /*
01335             if ( mode == 0 && bralevel ) {
01336                 MesPrint("@Unmatched brackets");
01337                 oldmode = mode;
01338                 return(-1);
01339             }
01340 */
01341             if ( mode != 2 ) break;
01342         }
01343         if ( quotelevel ) {
01344             if ( c == '\\' ) {
01345                 if ( ( mode == 1 ) || ( mode == 5 ) ) c = GetInput();
01346                 else {
01347                     c = GetChar(0);
01348                 }
01349                 if ( c == ENDOFINPUT ) {
01350                     MesPrint("@Unmatched \"");
01351                     if ( mode == 2 && c == ENDOFINPUT ) {
01352                         MesPrint("@Unexpected end of instruction");
01353                     }
01354 /*
01355                     if ( mode == 0 && bralevel ) {
01356                         MesPrint("@Unmatched brackets");
01357                     }
01358 */
01359                     oldmode = mode;
01360                     return(-1);
01361                 }
01362                 else if ( c == LINEFEED ) {}
01363                 else if ( c == '"' ) { *s++ = '\\'; }
01364                 else {
01365                     *s++ = '\\';
01366                 }
01367             }
01368             else if ( c == '"' ) {
01369                 quotelevel = 0;
01370                 AP.AllowDelay = 0;
01371             }
01372         }
01373         else if ( c == '\\' ) {
01374             if ( ( mode == 1 ) || ( mode == 5 ) ) cp = GetInput();
01375             else {
01376                 cp = GetChar(0);
01377             }
01378             if ( cp == LINEFEED ) continue;
01379             if ( mode != 2 || cp != ';' ) *s++ = c;
01380             c = cp;
01381         }
01382         else if ( c == '"' ) {
01383 /*
01384             Now look back in the buffer and determine what the keyword is.
01385             If it is define or redefine, put AllowDelay to 1.
01386 */
01387             t = AP.preStart;
01388             while ( FG.cTable[*t] <= 1 ) t++;
01389             cp = *t; *t = 0;
01390             if ( ( StrICmp(AP.preStart,(UBYTE *)"define") == 0 )
01391                 || ( StrICmp(AP.preStart,(UBYTE *)"redefine") == 0 ) ) {
01392                 AP.AllowDelay = 1;
01393                 oldstream = AC.CurrentStream;
01394             }
01395             *t = cp;
01396             quotelevel = 1;
01397         }
01398         else if ( quotelevel == 0 && bralevel == 0 && c == '(' ) {
01399             t = AP.preStart;
01400             while ( FG.cTable[*t] <= 1 ) t++;
01401             cp = *t; *t = 0;
01402             if ( ( parlevel == 0 )
01403                  && ( StrICmp(AP.preStart,(UBYTE *)"call") == 0 ) ) {
```

```
01404                 AP.AllowDelay = 1;
01405                 oldstream = AC.CurrentStream;
01406             }
01407             *t = cp;
01408             parlevel++;
01409         }
01410         else if ( quotelevel == 0 && bralevel == 0 && c == ')' ) {
01411             parlevel--;
01412         }
01413         else if ( quotelevel == 0 && parlevel == 0 && c == '{' ) {
01414             t = AP.preStart;
01415             while ( FG.cTable[*t] <= 1 ) t++;
01416             cp = *t; *t = 0;
01417             if ( ( bralevel == 0 )
01418                 && ( ( StrICmp(AP.preStart,(UBYTE *)"call") == 0 )
01419                 || ( StrICmp(AP.preStart,(UBYTE *)"do") == 0 ) ) ) {
01420                 AP.AllowDelay = 1;
01421                 oldstream = AC.CurrentStream;
01422             }
01423             *t = cp;
01424             bralevel++;
01425         }
01426         else if ( quotelevel == 0 && parlevel == 0 && c == '}' ) {
01427             bralevel--;
01428             if ( bralevel < 0 ) {
01429                 if ( mode != 5 ) {
01430                     MesPrint("@Unmatched brackets");
01431                     oldmode = mode;
01432                     return(-1);
01433                 }
01434                 bralevel = 0;
01435             }
01436         }
01437         if ( s >= (AP.preStop-1) ) {
01438             UBYTE **ppp;
01439             position = s - AP.preStart;
01440             if ( AP.preFill ) fillpos = AP.preFill - AP.preStart;
01441             ppp = &(AP.preStart); /* to avoid a compiler warning */
01442             if ( DoubleLList((void ***)ppp,&AP.pSize,sizeof(UBYTE),
01443             "instruction buffer") ) { *s = 0; oldmode = mode; return(-1); }
01444             AP.preStop = AP.preStart + AP.pSize-3;
01445             s = AP.preStart + position;
01446             if ( AP.preFill ) AP.preFill = fillpos + AP.preStart;
01447         }
01448         *s++ = c;
01449     }
01450     *s = 0;
01451     oldmode = mode;
01452     if ( mode == 0 ) {
01453         if ( ExpandTripleDots(1) < 0 ) return(-1);
01454     }
01455     return(0);
01456 }
01457
01458 /*
01459         #] LoadInstruction :
01460         #[ LoadStatement :
01461
01462         Puts the current string together in the input buffer.
01463         Does things like placing comma's where needed and expand ...
01464         We force a comma after the keyword. Before 8-sep-2009 the program might
01465         not put a comma if a + or - followed. And then the compiler ate
01466         the + or - and we needed repair code in the routines that used the
01467         + or - (Print, modulus, multiply and (a)bracket). This worked but
01468         the problem was with statements like  Dimension -4; which then would
01469         be processed as Dimension 4; (JV)
01470 */
01471
01472 int LoadStatement(int type)
01473 {
01474     UBYTE *s, c, cp;
01475     int retval = 0, stringlevel = 0, newstatement = 0;
01476     if ( type == NEWSTATEMENT ) { AP.eat = 1; newstatement = 1;
01477         s = AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]; }
01478     else { s = AC.iPointer; *s = 0; c = ' '; goto blank; }
01479     *s = 0;
01480     for(;;) {
01481         c = GetChar(0);
01482         if ( c == ENDOFINPUT ) { retval = -1; break; }
01483         if ( stringlevel == 0 ) {
01484             if ( c == LINEFEED ) {
01485                 if ( AP.eat < 0 ) { s--; AP.eat = 0; }
01486                 retval = 0; break;
01487             }
01488             if ( c == ';' ) {
01489                 if ( AP.eat < 0 ) { s--; AP.eat = 0; }
01490                 while ( ( c = GetChar(0) ) == ' ' || c == '\t' ) {}
```

```
01491                 if ( c != LINEFEED ) UngetChar(c);
01492                 retval = 1;
01493                 break;
01494             }
01495         }
01496         if ( c == '\\' ) {
01497             cp = GetChar(0);
01498             if ( cp == LINEFEED ) continue;
01499             *s++ = c;
01500             c = cp;
01501         }
01502         if ( c == '"' ) {
01503             if ( stringlevel == 0 ) stringlevel = 1;
01504             else stringlevel = 0;
01505             AP.eat = 0;
01506         }
01507         else if ( stringlevel == 0 ) {
01508             if ( c == '\t' ) c = ' ';
01509             if ( c == ' ' ) {
01510 blank:          if ( newstatement < 0 ) newstatement = 0;
01511                 if ( AP.eat && ( newstatement == 0 ) ) continue;
01512                 c = ',';
01513                 AP.eat = -2;
01514                 if ( newstatement > 0 ) newstatement = -1;
01515             }
01516             else if ( chartype[c] <= 3 ) {
01517                 AP.eat = 0;
01518                 if ( newstatement < 0 ) newstatement = 0;
01519             }
01520             else if ( c == ',' ) {
01521                 if ( newstatement > 0 ) {
01522                     newstatement = -1;
01523                     AP.eat = -2;
01524                 }
01525 /*              else if ( AP.eat == -2 ) { s--; } */
01526                 else if ( AP.eat == -2 ) { AP.eat = 1; continue; }
01527                 else { goto doall; }
01528             }
01529             else {
01530 doall:;         if ( AP.eat < 0 ) {
01531                     if ( newstatement == 0 ) s--;
01532                     else { newstatement = 0; }
01533                 }
01534                 else if ( newstatement == 1 ) newstatement = 0;
01535                 AP.eat = 1;
01536                 if ( c == '*' && s > AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel] && s[-1] == '*' )
    {
01537                     s[-1] = '^';
01538                     continue;
01539                 }
01540             }
01541         }
01542         if ( s >= AC.iStop ) {
01543             if ( !AP.iBufError ) {
01544                 LONG position = s - AC.iBuffer;
01545                 LONG position2 = AC.iPointer - AC.iBuffer;
01546                 UBYTE **ppp = &(AC.iBuffer); /* to avoid a compiler warning */
01547                 if ( DoubleLList((void ***)ppp,&AC.iBufferSize
01548                 ,sizeof(UBYTE),"statement buffer") ) {
01549                     *s = 0; retval = -1; AP.iBufError = 1;
01550                 }
01551                 AC.iPointer = AC.iBuffer + position2;
01552                 AC.iStop = AC.iBuffer + AC.iBufferSize-2;
01553                 s = AC.iBuffer + position;
01554             }
01555             if ( AP.iBufError ) {
01556                 for(;;){
01557                     c = GetChar(0);
01558                     if ( c == ENDOFINPUT ) { retval = -1; break; }
01559                     if ( c == '"' ) {
01560                         if ( stringlevel > 0 ) stringlevel = 0;
01561                         else stringlevel = 1;
01562                     }
01563                     else if ( c == LINEFEED && !stringlevel ) { retval = -2; break; }
01564                     else if ( c == ';' && !stringlevel ) {
01565                         while ( ( c = GetChar(0) ) == ' ' || c == '\t' ) {}
01566                         if ( c != LINEFEED ) UngetChar(c);
01567                         retval = -1;
01568                         break;
01569                     }
01570                     else if ( c == '\\' ) c = GetChar(0);
01571                 }
01572                 break;
01573             }
01574         }
01575         *s++ = c;
01576     }
```

```
01577     AC.iPointer = s;
01578     *s = 0;
01579     if ( stringlevel > 0 ) {
01580         MesPrint("@Unbalanced \". Runaway string");
01581         retval = -1;
01582     }
01583     if ( retval == 1 ) {
01584         if ( ExpandTripleDots(0) < 0 ) retval = -1;
01585     }
01586     return(retval);
01587 }
01588
01589 /*
01590         #] LoadStatement :
01591         #[ ExpandTripleDots :
01592 */
01593
01594 static inline int IsSignChar(UBYTE c)
01595 {
01596     return c == '+' || c == '-';
01597 }
01598
01599 static inline int IsAlphanumericChar(UBYTE c)
01600 {
01601     return FG.cTable[c] == 0 || FG.cTable[c] == 1;
01602 }
01603
01604 static inline int CanParseSignedNumber(const UBYTE *s)
01605 {
01606     while ( IsSignChar(*s) ) s++;
01607     return FG.cTable[*s] == 1;
01608 }
01609
01610 int ExpandTripleDots(int par)
01611 {
01612     UBYTE *s, *s1, *s2, *n1, *n2, *t1, *t2, *startp, operator1, operator2, c, cc;
01613     UBYTE *nBuffer, *strngs, *Buffer, *Stop;
01614     LONG withquestion, x1, x2, y1, y2, number, inc, newsize, pow, fullsize;
01615     int i, error = 0, i1 ,i2, ii, *nums = 0;
01616
01617     if ( par == 0 ) {
01618         Buffer = AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]; Stop = AC.iStop;
01619     }
01620     else {
01621         Buffer = AP.preStart; Stop = AP.preStop;
01622     }
01623     s = Buffer; while ( *s ) s++;
01624     fullsize = s - Buffer;
01625     if ( fullsize < 7 ) return(error);
01626
01627     s = Buffer+2;
01628     while ( *s ) {
01629         if ( *s != '.' || ( s[-1] != ',' && FG.cTable[s[-1]] != 5 ) )
01630             { s++; continue; }
01631         if ( s[-1] == '%' || s[-1] == '^' || s[1] != '.' || s[2] != '.' )
01632             { s++; continue; }
01633         s1 = s - 2;
01634         s += 3;
01635         if ( *s != s[-4] && ( *s != '+' || s[-4] != '-' )
01636          && ( *s != '-' || s[-4] != '+' ) ) {
01637             MesPrint("&Improper operators for ...");
01638             error = -1;
01639         }
01640         operator1 = s[-4];
01641         operator2 = *s++;
01642         if ( operator1 == ':' ) operator1 = '.';
01643         if ( operator2 == ':' ) operator2 = '.';
01644 /*
01645         We have now O1...O2 (O stands for operator)
01646         Full syntax is
01647         [str]#1[?]O1...O2[str]#2[?]   (Special case)
01648         in which both strings are identical and if one ? then also the other.
01649         <pattern1>O1...O2<pattern2>   (General case)
01650         in which the difference in the patterns is just numerical.
01651 */
01652         s2 = s;         /* the beginning of the second string */
01653         if ( *s2 != '<' || *s1 != '>' ) {   /* Special case */
01654             startp = s1+1;
01655             withquestion = ( *s1 == '?' ); s1--;
01656             while ( FG.cTable[*s1] == 1 && s1 >= Buffer ) s1--;
01657             n1 = s1+1;      /* Beginning of first number */
01658             if ( FG.cTable[*n1] != 1 ) {
01659                 MesPrint("&No first number in ... operator");
01660                 error = -1;
01661             }
01662             while ( FG.cTable[*s1] <= 1 && s1 >= Buffer ) s1--;
01663             s1++;
```

```
01664 /*
01665             We have now the first string from s1 to n1, number from n1
01666 */
01667             t1 = s1; t2 = s2;
01668             while ( t1 < n1 && *t1 == *t2 ) { t1++; t2++; }
01669             n2 = t2;
01670             if ( FG.cTable[*t2] != 1 ) {
01671                 MesPrint("&No second number in ... operator");
01672                 error = -1;
01673             }
01674             x2 = 0;
01675             while ( FG.cTable[*t2] == 1 ) x2 = 10*x2 + *t2++ - '0';
01676             x1 = 0;
01677             while ( FG.cTable[*t1] == 1 ) x1 = 10*x1 + *t1++ - '0';
01678             if ( withquestion != ( *t2 == '?' ) ) {
01679                 MesPrint("&Improper use of ? in ... operator");
01680                 if ( *t2 == '?' ) t2++;
01681                 error = -1;
01682             }
01683             else if ( withquestion ) t2++;
01684             if ( FG.cTable[*t2] <= 2 ) {
01685                 MesPrint("&Illegal object after ... construction");
01686                 error = -1;
01687             }
01688             c = *n1; *n1 = 0; s = t2;
01689             if ( error ) continue;
01690 /*
01691             At this point the syntax has been fulfilled. We have
01692             str in s1.
01693             x1,x2 are #1,#2
01694             operator1,operator2 are the two operators.
01695             s points at whatever comes after.
01696             Expansion will have to be computed.
01697 */
01698             if ( x2 < x1 ) { number = x1-x2; inc = -1; y1 = x2; y2 = x1; }
01699             else           { number = x2-x1; inc =  1; y1 = x1; y2 = x2; }
01700             newsize = (number+1)*(n1-s1)            /* the strings   */
01701                      + number                       /* the operators */
01702                      +(number+1)*(withquestion?1:0) /* questionmarks */
01703                      +(number+1);                   /* last digits   */
01704             pow = 10;
01705             for ( i = 1; i < 10; i++, pow *= 10 ) {
01706                 if ( y1 >= pow )      newsize += number+1;
01707                 else if ( y2 >= pow ) newsize += y2-pow+1;
01708                 else break;
01709             }
01710             while ( Buffer+(fullsize+newsize-(s-s1)) >= Stop ) {
01711                 LONG strpos = s1-Buffer;
01712                 LONG endstr = n1-Buffer;
01713                 LONG startq = startp - Buffer;
01714                 LONG position = s - Buffer;
01715                 UBYTE **ppp;
01716                 if ( par == 0 ) {
01717                     LONG position2 = AC.iPointer - AC.iBuffer;
01718                     ppp = &(AC.iBuffer); /* to avoid a compiler warning */
01719                     if ( DoubleLList((void ***)ppp,&AC.iBufferSize
01720                         ,sizeof(UBYTE),"statement buffer") ) {
01721                             Terminate(-1);
01722                     }
01723                     AC.iPointer = AC.iBuffer + position2;
01724                     AC.iStop = AC.iBuffer + AC.iBufferSize-2;
01725                     Buffer = AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]; Stop = AC.iStop;
01726                 }
01727                 else {
01728                     LONG fillpos = 0;
01729                     if ( AP.preFill ) fillpos = AP.preFill - AP.preStart;
01730                     ppp = &(AP.preStart); /* to avoid a compiler warning */
01731                     if ( DoubleLList((void ***)ppp,&AP.pSize,sizeof(UBYTE),
01732                         "instruction buffer") ) {
01733                             Terminate(-1);
01734                     }
01735                     AP.preStop = AP.preStart + AP.pSize-3;
01736                     if ( AP.preFill ) AP.preFill = fillpos + AP.preStart;
01737                     Buffer = AP.preStart; Stop = AP.preStop;
01738                 }
01739                 s  = Buffer + position;
01740                 n1 = Buffer + endstr;
01741                 s1 = Buffer + strpos;
01742                 startp = Buffer + startq;
01743             }
01744 /*
01745             We have space for the expansion in the buffer.
01746             There are two cases: new size >  old size
01747                                  old size >= new size
01748             Note that whereever we move things, it will be at least startp.
01749 */
01750             if ( newsize > (s-s1) ) {
```

```
01751                 t2 = Buffer + fullsize;
01752                 t1 = t2 + (newsize - (s-s1));
01753                 *t1 = 0;
01754                 while ( t2 > s ) { *--t1 = *--t2; }
01755             }
01756             else if ( newsize < (s-s1) ) {
01757                 t1 = s1 + newsize; t2 = s; s = t1;
01758                 while ( *t2 ) *t1++ = *t2++;
01759                 *t1 = 0;
01760             }
01761             for ( x1 += inc, t1 = startp; number > 0; number--, x1 += inc ) {
01762                 *t1++ = operator1;
01763                 cc = operator1; operator1 = operator2; operator2 = cc;
01764                 t2 = s1; while ( *t2 ) *t1++ = *t2++;
01765                 x2 = x1; n2 = t1;
01766                 do {
01767                     *t1++ = '0' + x2 % 10;
01768                     x2 /= 10;
01769                 } while ( x2 );
01770                 s2 = t1 - 1;
01771                 while ( s2 > n2 ) { cc = *s2; *s2 = *n2; *n2++ = cc; s2--; }
01772                 if ( withquestion ) *t1++ = '?';
01773             }
01774             fullsize += newsize - ( s - s1 );
01775             *n1 = c;
01776         }
01777         else {      /* General case. Find the patterns first */
01778             t1 = s1; s1--;
01779             while ( s1 > Buffer ) {
01780                 if ( *s1 == '<' ) break;
01781                 s1--;
01782             }
01783             t2 = s2;
01784             while ( *t2 ) {
01785                 if ( *t2 == '>' ) break;
01786                 t2++;
01787             }
01788             if ( *s1 != '<' || *t2 != '>' ) {
01789                 MesPrint("&Illegal attempt to use ... operator");
01790                 return(-1);
01791             }
01792             s1++; s2++;     /* Pointers to the patterns */
01793             nums = (int *)Malloc1((t1-s1)*2*(sizeof(int)+sizeof(UBYTE))
01794                         ,"Expand ...");
01795             strngs = (UBYTE *)(nums + 2*(t1-s1));
01796             n1 = s1; n2 = s2; ii = -1; i = 0;
01797             s = strngs;
01798             while ( n1 < t1 || n2 < t2 ) {
01799                 /* Check the next characters can be parsed as numbers including signs. */
01800                 if ( CanParseSignedNumber(n1) && CanParseSignedNumber(n2) ) {
01801                     /*
01802                      * Don't allow the cases that one has the sign and the other doesn't,
01803                      * and the meaning changes without the sign. For example,
01804                      *   <f(1)>+...+<f(3)>       Allowed
01805                      *   <f(-2)>+...+<f(2)>      Allowed
01806                      *   <f(x-2)>+...+<f(x+2)>   Allowed
01807                      *   <f(x-2)>+...+<f(x2)>    Not allowed
01808                      */
01809                     int sign1 = IsSignChar(*n1);
01810                     int sign2 = IsSignChar(*n2);
01811                     int inword1 = s1 < n1 && IsAlphanumericChar(n1[-1]);
01812                     int inword2 = s2 < n2 && IsAlphanumericChar(n2[-1]);
01813                     if ( ( sign1 ^ sign2 ) && ( inword1 || inword2 ) ) break;  /* Not allowed. */
01814                     if ( sign1 || sign2 ) {
01815                         *s++ = '+';  /* Marker indicating we need the sign. */
01816                     }
01817                 } else {
01818                     /* If they are not numbers, they should be same. */
01819                     if ( *n1 == *n2 ) { *s++ = *n1++; n2++; continue; }
01820                     else break;
01821                 }
01822                 ParseSignedNumber(x1,n1)
01823                 ParseSignedNumber(x2,n2)
01824                 if ( x1 == x2 ) {
01825                     if ( s != strngs && ( s[-1] == '+' || s[-1] == '-' ) ) {
01826                         /* We need the sign. */
01827                         s--;
01828                         if ( x1 >= 0 ) {
01829                             *s++ = '+';
01830                         }
01831                     }
01832                     s = NumCopy(x1, s);
01833                 }
01834                 else {
01835                     nums[2*i] = x1; nums[2*i+1] = x2;
01836                     i++; *s++ = 0;
01837                 }
```

```
01838             }
01839             if ( n1 < t1 || n2 < t2 ) {
01840                 MesPrint("&Improper use of ... operator.");
01841 theend:         M_free(nums,"Expand ...");
01842                 return(-1);
01843             }
01844             *s = 0;
01845             if ( i == 0 ) ii = 0;
01846             else {
01847                 ii = nums[0] - nums[1];
01848                 if ( ii < 0 ) ii = -ii;
01849                 for ( x1 = 1; x1 < i; x1++ ) {
01850                     x2 = nums[2*x1]-nums[2*x1+1];
01851                     if ( x2 < 0 ) x2 = -x2;
01852                     if ( x2 != ii ) {
01853                         MesPrint("&Improper synchronization of numbers in ... operator");
01854                         goto theend;
01855                     }
01856                 }
01857             }
01858             ii++;
01859 /*
01860             We have now proper syntax.
01861             There are i+1 strings in strngs and i pairs of numbers
01862             in nums. Each time a start value and a finish value.
01863             We have ii steps. If ii <= 2, it will fit in the existing
01864             allocation. But this is hardly useful.
01865             We make a new allocation and copy from the old.
01866             Compute space.
01867 */
01868             x2 = s - strngs - i;  /* -1 for end-of-string and +1 for the operator*/
01869             for ( i1 = 0; i1 < i; i1++ ) {
01870                 i2 = nums[2*i1];
01871                 x1 = nums[2*i1+1];
01872                 if ( i2 < 0 ) i2 = -i2;
01873                 if ( x1 < 0 ) x1 = -x1;
01874                 if ( x1 > i2 ) i2 = x1;
01875                 x1 = 2;
01876                 while ( i2 > 0 ) { i2 /= 10; x1++; }
01877                 x2 += x1;
01878             }
01879             x2 *= ii;   /* Space for the expanded string (a bit more) */
01880             x2 += fullsize;
01881             x2 += 5;        /* This will definitely hold everything */
01882             x2 += sizeof(UBYTE *);
01883             x2 = x2 - (x2 & (sizeof(UBYTE *)-1));
01884
01885             nBuffer = (UBYTE *)Malloc1(x2,"input buffer");
01886             n1 = nBuffer; s = Buffer; s1--;
01887             while ( s < s1 ) *n1++ = *s++;
01888 /*
01889             Solution of the special case that no comma was generated
01890             due to the presence of < to start the pattern.
01891             We get a comma when the word before ends in an alphanumeric
01892             character, a _ or a ] and the word inside starts with an
01893             alphanumeric character, a [ (or an _ (for future considerations))
01894 */
01895             if ( ( ( n1 > nBuffer ) && ( ( FG.cTable[n1[-1]] <= 1 )
01896             || ( n1[-1] == '_' ) || ( n1[-1] == ']' ) ) ) &&
01897             ( ( FG.cTable[strngs[0]] <= 1 ) || ( strngs[0] == '[' )
01898              || ( strngs[0] == '_' ) ) ) *n1++ = ',';
01899
01900             for ( i1 = 0; i1 < ii; i1++ ) {
01901                 s = strngs; while ( *s ) *n1++ = *s++;
01902                 for ( i2 = 0; i2 < i; i2++ ) {
01903                     if ( n1 > nBuffer && IsSignChar(n1[-1]) ) {
01904                         /* We need the sign of counters. */
01905                         n1--;
01906                         if ( nums[2*i2] >= 0 ) {
01907                             *n1++ = '+';
01908                         }
01909                     }
01910                     n1 = NumCopy((WORD)(nums[2*i2]),n1);
01911                     if ( nums[2*i2] > nums[2*i2+1] ) nums[2*i2]--;
01912                     else nums[2*i2]++;
01913                     s++; while ( *s ) *n1++ = *s++;
01914                 }
01915                 if ( ( i1 & 1 ) == 0 ) *n1++ = operator1;
01916                 else *n1++ = operator2;
01917             }
01918             n1--;   /* drop the trailing operator */
01919             s = t2 + 1; n2 = n1;
01920 /*
01921             Similar extra comma
01922 */
01923             if ( ( ( ( FG.cTable[n1[-1]] <= 1 )
01924             || ( n1[-1] == '_' ) || ( n1[-1] == ']' ) ) ) &&
```

```
01925             ( ( FG.cTable[s[0]] <= 1 ) || ( s[0] == '[' )
01926              || ( s[0] == '_' ) ) ) *n1++ = ',';
01927
01928             while ( *s ) *n1++ = *s++;
01929             *n1 = 0;
01930             if ( par == 0 ) {
01931                 LONG nnn1 = n1-nBuffer;
01932                 LONG nnn2 = n2-nBuffer;
01933                 LONG nnn3;
01934                 while ( AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel] + x2 >= AC.iStop ) {
01935                     LONG position = s-Buffer;
01936                     LONG position2 = AC.iPointer - AC.iBuffer;
01937                     UBYTE **ppp;
01938                     ppp = &(AC.iBuffer); /* to avoid a compiler warning */
01939                     if ( DoubleLList((void ***)ppp,&AC.iBufferSize
01940                         ,sizeof(UBYTE),"statement buffer") ) {
01941                             Terminate(-1);
01942                     }
01943                     AC.iPointer = AC.iBuffer + position2;
01944                     AC.iStop = AC.iBuffer + AC.iBufferSize-2;
01945                     Buffer = AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]; Stop = AC.iStop;
01946                     s  = Buffer + position;
01947                 }
01948 /*
01949                 This can be improved. We only have to start from the first term.
01950 */
01951                 for ( nnn3 = 0; nnn3 < nnn1; nnn3++ ) Buffer[nnn3] = nBuffer[nnn3];
01952                 Buffer[nnn3] = 0;
01953                 n1 = Buffer + nnn1;
01954                 n2 = Buffer + nnn2;
01955                 M_free(nBuffer,"input buffer");
01956                 M_free(nums,"Expand ...");
01957             }
01958             else { /* Comes here only inside a real preprocessor instruction */
01959                 AP.preStop = nBuffer + x2 - 2;
01960                 AP.pSize = x2;
01961                 M_free(AP.preStart,"input buffer");
01962                 M_free(nums,"Expand ...");
01963                 AP.preStart = nBuffer;
01964                 Buffer = AP.preStart; Stop = AP.preStop;
01965             }
01966             fullsize = n1 - Buffer;
01967             s = n2;
01968         }
01969     }
01970     return(error);
01971 }
01972
01973 /*
01974         #] ExpandTripleDots :
01975         #[ FindKeyWord :
01976 */
01977
01978 KEYWORD *FindKeyWord(UBYTE *theword, KEYWORD *table, int size)
01979 {
01980     int low,med,hi;
01981     UBYTE *s1, *s2;
01982     low = 0;
01983     hi = size-1;
01984     while ( hi >= low ) {
01985         med = (hi+low)/2;
01986         s1 = (UBYTE *)(table[med].name);
01987         s2 = theword;
01988         while ( *s1 && tolower(*s1) == tolower(*s2) ) { s1++; s2++; }
01989         if ( *s1 == 0 &&
01990 /*[30apr2004 mt]:*/
01991 /* The bug!:
01992                 FG.cTable[*s2] != 1 && FG.cTable[*s2] != 2
01993 */
01994                 FG.cTable[*s2] != 0 && FG.cTable[*s2] != 1
01995 /*      ( *s2 == ' ' || *s2 == '\t' || *s2 == 0 || *s2 == ',' || *s2 == '(' ) */
01996             )
01997             return(table+med);
01998         if ( tolower(*s2) > tolower(*s1) ) low = med+1;
01999         else hi = med - 1;
02000     }
02001     return(0);
02002 }
02003
02004 /*
02005         #] FindKeyWord :
02006         #[ FindInKeyWord :
02007 */
02008
02009 KEYWORD *FindInKeyWord(UBYTE *theword, KEYWORD *table, int size)
02010 {
02011     int i;
```

```
02012     UBYTE *s1, *s2;
02013     for ( i = 0; i < size; i++ ) {
02014         s1 = (UBYTE *)(table[i].name);
02015         s2 = theword;
02016         while ( *s1 && tolower(*s1) == tolower(*s2) ) { s1++; s2++; }
02017         if ( *s2 == 0 || *s2 == ' ' || *s2 == ',' || *s2 == '\t' )
02018             return(table+i);
02019     }
02020     return(0);
02021 }
02022
02023 /*
02024         #] FindInKeyWord :
02025         #[ TheDefine :
02026 */
02027
02039 int TheDefine(UBYTE *s, int mode)
02040 {
02041     UBYTE *name, *value, *valpoin, *args = 0, c;
02042     if ( ( mode & 2 ) == 0 ) {
02043         if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
02044         if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
02045     }
02046     else { mode &= ~2; }
02047     name = s;
02048     if ( chartype[*s] != 0 ) goto illname;
02049     s++;
02050     while ( chartype[*s] <= 1 ) s++;
02051     value = s;
02052     while ( *s == ' ' || *s == '\t' ) s++;
02053     c = *s; *value = 0;
02054     if ( c == 0 ) {
02055         if ( PutPreVar(name,(UBYTE *)"1",0,mode) < 0 ) return(-1);
02056         return(0);
02057     }
02058     if ( c == '(' ) {   /* arguments. scan for correctness */
02059         s++; args = s;
02060         for (;;) {
02061             if ( chartype[*s] != 0 ) goto illarg;
02062             s++;
02063             while ( chartype[*s] <= 1 ) s++;
02064             while ( *s == ' ' || *s == '\t' ) s++;
02065             if ( *s == ')' ) break;
02066             if ( *s != ',' ) goto illargs;
02067             s++;
02068             while ( *s == ' ' || *s == '\t' ) s++;
02069         }
02070         *s++ = 0;
02071         while ( *s == ' ' || *s == '\t' ) s++;
02072         c = *s;
02073     }
02074     if ( c == '"' ) {
02075         s++; valpoin = value = s;
02076         while ( *s != '"' ) {
02077             if ( *s == '\\' ) {
02078                 if ( s[1] == 'n' ) { *valpoin++ = LINEFEED; s += 2; }
02079                 else if ( s[1] == '"' ) { *valpoin++ = '"'; s += 2; }
02080                 else if ( s[1] == 0 ) goto illval;
02081                 else { *valpoin++ = *s++; *valpoin++ = *s++; }
02082             }
02083             else *valpoin++ = *s++;
02084         }
02085         *valpoin = 0;
02086         if ( PutPreVar(name,value,args,mode) < 0 ) return(-1);
02087     }
02088     else {
02089         MesPrint("@Illegal string for preprocessor variable %s. Forgotten double quotes (\") ?",name);
02090         return(-1);
02091     }
02092     return(0);
02093 illname:;
02094     MesPrint("@Illegally formed name of preprocessor variable");
02095     return(-1);
02096 illarg:;
02097     MesPrint("@Illegally formed name of argument of preprocessor definition");
02098     return(-1);
02099 illargs:;
02100     MesPrint("@Illegally formed arguments of preprocessor definition");
02101     return(-1);
02102 illval:;
02103     MesPrint("@Illegal valpoin for preprocessor variable %s",name);
02104     return(-1);
02105 }
02106
02107 /*
02108         #] TheDefine :
02109         #[ DoCommentChar :
```

```
02110 */
02111
02112 int DoCommentChar(UBYTE *s)
02113 {
02114     UBYTE c;
02115     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
02116     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
02117     while ( *s == ' ' || *s == '\t' ) s++;
02118     if ( *s == 0 || *s == '\n' ) {
02119         MesPrint("@No valid comment character specified");
02120         return(-1);
02121     }
02122     c = *s++;
02123     while ( *s == ' ' || *s == '\t' ) s++;
02124     if ( *s != 0 && *s != '\n' ) {
02125         MesPrint("@Comment character should be a single valid character");
02126         return(-1);
02127     }
02128     AP.ComChar = c;
02129     return(0);
02130 }
02131
02132 /*
02133         #] DoCommentChar :
02134         #[ DoPreAssign :
02135
02136         Routine assigns a 'value' to a $variable.
02137         Syntax: #assign
02138             next line(s) a statement of the type
02139             $name = expression;
02140         Note: at the moment of the assign there cannot be an 'open' statement.
02141 */
02142
02143 int DoPreAssign(UBYTE *s)
02144 {
02145     int error = 0;
02146     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) {
02147         return(0);
02148     }
02149     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) {
02150         return(0);
02151     }
02152     if ( *s ) {
02153         MesPrint("@Illegal characters in %#assign instruction");
02154         error = 1;
02155     }
02156     PUSHPREASSIGNLEVEL;
02157     AP.PreAssignFlag = 1;
02158 /*
02159     if ( AP.PreContinuation ) {
02160         MesPrint("@Assign instructions cannot occur inside statements");
02161         MesPrint("@Missing ; ?");
02162         AP.PreContinuation = 0;
02163         error = 1;
02164     }
02165 */
02166     return(error);
02167 }
02168
02169 /*
02170         #] DoPreAssign :
02171         #[ DoDefine :
02172 */
02173
02174 int DoDefine(UBYTE *s)
02175 {
02176     return(TheDefine(s,0));
02177 }
02178
02179 /*
02180         #] DoDefine :
02181         #[ DoRedefine :
02182 */
02183
02184 int DoRedefine(UBYTE *s)
02185 {
02186     return(TheDefine(s,1));
02187 }
02188
02189 /*
02190         #] DoRedefine :
02191         #[ ClearMacro :
02192
02193         Undefines the arguments of a macro after its use.
02194 */
02195
02196 int ClearMacro(UBYTE *name)
```

```
02197 {
02198     int i;
02199     PREVAR *p;
02200     UBYTE *s;
02201     for ( i = NumPre-1, p = &(PreVar[NumPre-1]); i >= 0; i--, p-- ) {
02202         if ( StrCmp(name,p->name) == 0 ) break;
02203     }
02204     if ( i < 0 ) return(-1);
02205     if ( p->nargs <= 0 ) return(0);
02206     s = p->argnames;
02207     for ( i = 0; i < p->nargs; i++ ) {
02208         TheUndefine(s);
02209         while ( *s ) s++;
02210         s++;
02211     }
02212     return(0);
02213 }
02214
02215 /*
02216         #] ClearMacro :
02217         #[ TheUndefine :
02218
02219         There is a complication here. If there are redefine statements
02220         they will be pointing at the wrong variable if their number is
02221         greater than the number of the variable we pop.
02222 */
02223
02224 int TheUndefine(UBYTE *name)
02225 {
02226     int i, inum, error = 0;
02227     PREVAR *p;
02228     for ( i = NumPre-1, p = &(PreVar[NumPre-1]); i >= 0; i--, p-- ) {
02229         if ( StrCmp(name,p->name) == 0 ) {
02230             M_free(p->name,"undefining PreVar");
02231             NumPre--;
02232             inum = i;
02233             while ( i < NumPre ) {
02234                 p->name  = p[1].name;
02235                 p->value = p[1].value;
02236                 p++; i++;
02237             }
02238             p->name = 0; p->value = 0;
02239             {
02240                 CBUF *CC = cbuf + AC.cbufnum;
02241                 int j, k;
02242                 for ( j = 1; j <= CC->numlhs; j++ ) {
02243                     if ( CC->lhs[j][0] == TYPEREDEFPRE ) {
02244                         if ( CC->lhs[j][2] > inum ) CC->lhs[j][2]--;
02245                         else if ( CC->lhs[j][2] == inum ) {
02246                             for ( k = inum - 1; k >= 0; k-- )
02247                                 if ( StrCmp(name, PreVar[k].name) == 0 ) break;
02248                             if ( k >= 0 ) CC->lhs[j][2] = k;
02249                             else {
02250                                 MesPrint("@Conflict between undefining a preprocessor variable and a
    redefine statement");
02251                                 error = 1;
02252                             }
02253                         }
02254                     }
02255                 }
02256 #ifdef PARALLELCODE
02257                 for ( j = 0; j < AC.numpfirstnum; j++ ) {
02258                     if ( AC.pfirstnum[j] > inum ) AC.pfirstnum[j]--;
02259                     else if ( AC.pfirstnum[j] == inum ) {
02260                         for ( k = inum - 1; k >= 0; k-- )
02261                             if ( StrCmp(name, PreVar[k].name) == 0 ) break;
02262                         if ( k >= 0 ) AC.pfirstnum[j] = k;
02263                     }
02264                 }
02265 #endif
02266             }
02267             break;
02268         }
02269     }
02270     return(error);
02271 }
02272
02273 /*
02274         #] TheUndefine :
02275         #[ DoUndefine :
02276 */
02277
02278 int DoUndefine(UBYTE *s)
02279 {
02280     UBYTE *name, *t;
02281     int error = 0, retval;
02282 /*
```

```
02283     int i;
02284     PREVAR *p;
02285 */
02286     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
02287     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
02288     name = s;
02289     if ( chartype[*s] != 0 ) goto illname;
02290     s++;
02291     while ( chartype[*s] <= 1 ) s++;
02292     t = s;
02293     if ( *s && *s != ' ' && *s != '\t' ) goto illname;
02294     while ( *s == ' ' || *s == '\t' ) s++;
02295     if ( *s ) {
02296         MesPrint("@Undefine should just have a variable name");
02297         error = -1;
02298     }
02299     *t = 0;
02300     if ( ( retval = TheUndefine(name) ) != 0 ) {
02301         if ( error == 0 ) return(retval);
02302         if ( error > 0 ) error = retval;
02303     }
02304 /*
02305     for ( i = NumPre-1, p = &(PreVar[NumPre-1]); i >= 0; i--, p-- ) {
02306         if ( StrCmp(name,p->name) == 0 ) {
02307             M_free(p->name,"undefining PreVar");
02308             NumPre--;
02309             while ( i < NumPre ) {
02310                 p->name  = p[1].name;
02311                 p->value = p[1].value;
02312                 p++; i++;
02313             }
02314             p->name = 0; p->value = 0;
02315             break;
02316         }
02317     }
02318 */
02319     return(error);
02320 illname:;
02321     MesPrint("@Illegally formed name of preprocessor variable");
02322     return(-1);
02323 }
02324
02325 /*
02326         #] DoUndefine :
02327         #[ DoInclude :
02328 */
02329
02330 int DoInclude(UBYTE *s) { return(Include(s,FILESTREAM)); }
02331
02332 /*
02333         #] DoInclude :
02334         #[ DoReverseInclude :
02335 */
02336
02337 int DoReverseInclude(UBYTE *s) { return(Include(s,REVERSEFILESTREAM)); }
02338
02339 /*
02340         #] DoReverseInclude :
02341         #[ Include :
02342 */
02343
02344 int Include(UBYTE *s, int type)
02345 {
02346     UBYTE *name = s, *fold, *t, c, c1 = 0, c2 = 0, c3 = 0;
02347     int str1offset, withnolist = AC.NoShowInput;
02348     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
02349     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
02350     if ( *s == '-' || *s == '+' ) {
02351         if ( *s == '-' ) withnolist = 1;
02352         else             withnolist = 0;
02353         s++;
02354         while ( *s == ' ' || *s == '\t' ) s++;
02355         name = s;
02356     }
02357     if ( *s == '"' ) {
02358         while ( *s && *s != '"' ) {
02359             if ( *s == '\\' ) s++;
02360             s++;
02361         }
02362         t = s++;
02363     }
02364     else {
02365         while ( *s && *s != ' ' && *s != '\t' ) {
02366             if ( *s == '\\' ) s++;
02367             s++;
02368         }
02369         t = s;
```

```
02370     }
02371     while ( *s == ' ' || *s == '\t' ) s++;
02372     if ( *s == '#' ) {
02373         *t = 0;
02374         s++;
02375         while ( *s == ' ' || *s == '\t' ) s++;
02376         fold = s;
02377         if ( *s == 0 ) {
02378             MesPrint("@Empty fold name");
02379             return(-1);
02380         }
02381 continue_fold:
02382         while ( *s && *s != ' ' && *s != '\t' ) {
02383             if ( *s == '\\' ) s++;
02384             s++;
02385         }
02386         t = s;
02387         while ( *s == ' ' || *s == '\t' ) s++;
02388         if ( *s ) {
02389             /*
02390              * A non-whitespace character is found. Continue parsing the fold.
02391              */
02392             goto continue_fold;
02393         }
02394     }
02395     else if ( *s == 0 ) {
02396         fold = 0;
02397     }
02398     else {
02399         MesPrint("@Improper syntax for file name");
02400         return(-1);
02401     }
02402     *t = 0;
02403     if ( fold ) {
02404         fold = strDup1(fold,"foldname");
02405     }
02406 /*
02407     We have the name of the file in 'name' and the fold in 'fold' (or NULL)
02408 */
02409     if ( OpenStream(name,type,0,PRENOACTION) == 0 ) {
02410         if ( fold ) { M_free(fold,"foldname"); fold = 0; }
02411         return(-1);
02412     }
02413     if ( fold ) {
02414         LONG position = -1;
02415         int foldopen = 0;
02416         LONG linenum = 0, prevline = 0;
02417         name = strDup1(name,"name of include file");
02418         AC.CurrentStream->FoldName = strDup1(fold,"name of fold");
02419         AC.NoShowInput++;
02420         for(;;) {
02421             c = GetFromStream(AC.CurrentStream);
02422             if ( c == ENDOFSTREAM ) {
02423                 AC.CurrentStream = CloseStream(AC.CurrentStream);
02424                 goto nofold;
02425             }
02426             if ( c == AP.ComChar ) {
02427                 strloffset = AC.CurrentStream-AC.Streams;
02428                 LoadInstruction(1);
02429                 if ( AC.CurrentStream != strloffset+AC.Streams ) {
02430                     c = ENDOFSTREAM;
02431                 }
02432                 else {
02433                     t = AP.preStart;
02434                     if ( t[2] == '#' && ( ( t[3] == '[' && !foldopen )
02435                     || ( t[3] == ']' && foldopen ) ) ) {
02436                         t += 4;
02437                         while ( *t == ' ' || *t == '\t' ) t++;
02438                         s = AC.CurrentStream->FoldName;
02439                         while ( *s == *t ) { s++; t++; }
02440                         if ( *s == 0 && ( *t == ' ' || *t == '\t'
02441                         || *t == ':' ) ) {
02442                             while ( *t == ' ' || *t == '\t' ) t++;
02443                             if ( *t == ':' ) {
02444                                 if ( foldopen == 0 ) {
02445                                     foldopen = 1;
02446                                     position = GetStreamPosition(AC.CurrentStream);
02447                                     linenum = AC.CurrentStream->linenumber;
02448                                     prevline = AC.CurrentStream->prevline;
02449                                     c3 = AC.CurrentStream->isnextchar;
02450                                     c1 = AC.CurrentStream->nextchar[0];
02451                                     c2 = AC.CurrentStream->nextchar[1];
02452                                 }
02453                                 else {
02454                                     foldopen = 0;
02455                                     PositionStream(AC.CurrentStream,position);
02456                                     AC.CurrentStream->linenumber = linenum;
```

```
02457                                   AC.CurrentStream->prevline = prevline;
02458                                   AC.CurrentStream->eqnum = 1;
02459                                   AC.NoShowInput--;
02460                                   AC.CurrentStream->isnextchar = c3;
02461                                   AC.CurrentStream->nextchar[0] = c1;
02462                                   AC.CurrentStream->nextchar[1] = c2;
02463                                   break;
02464                               }
02465                           }
02466                       }
02467                   }
02468               }
02469           }
02470           else {
02471               while ( c != LINEFEED && c != ENDOFSTREAM ) {
02472                   c = GetFromStream(AC.CurrentStream);
02473                   if ( c == ENDOFSTREAM ) {
02474                       AC.CurrentStream = CloseStream(AC.CurrentStream);
02475                       break;
02476                   }
02477               }
02478           }
02479           if ( c == ENDOFSTREAM ) {
02480 nofold:
02481               MesPrint("@Cannot find fold %s in file %s",fold,name);
02482               UngetChar(c);
02483               AC.NoShowInput--;
02484               M_free(name,"name of include file");
02485               Terminate(-1);
02486           }
02487       }
02488       M_free(name,"name of include file");
02489   }
02490   AC.NoShowInput = withnolist;
02491   if ( fold ) { M_free(fold,"foldname"); fold = 0; }
02492   return(0);
02493 }
02494
02495 /*
02496       #] Include :
02497       #[ DoPreExchange :
02498
02499       Exchanges the names of expressions or the contents of dollars
02500       Syntax:
02501           #exchange expr1,expr2
02502           #exchange $var1,$var2
02503 */
02504
02505 int DoPreExchange(UBYTE *s)
02506 {
02507   int error = 0;
02508   UBYTE *s1, *s2;
02509   WORD num1, num2;
02510   if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
02511   if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
02512   while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
02513   if ( *s == '$' ) {
02514       s++; s1 = s; while ( FG.cTable[*s] <= 1 ) s++;
02515       if ( *s != ',' && *s != ' ' && *s != '\t' ) goto syntax;
02516       *s++ = 0;
02517       while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
02518       if ( *s != '$' ) goto syntax;
02519       s++; s2 = s; while ( FG.cTable[*s] <= 1 ) s++;
02520       if ( *s != 0 && *s != ';' ) goto syntax;
02521       *s = 0;
02522       if ( ( num1 = GetDollar(s1) ) <= 0 ) {
02523           MesPrint("@$%s has not been defined (yet)",s1);
02524           error = 1;
02525       }
02526       if ( ( num2 = GetDollar(s2) ) <= 0 ) {
02527           MesPrint("@$%s has not been defined (yet)",s2);
02528           error = 1;
02529       }
02530       if ( error == 0 ) {
02531           ExchangeDollars((int)num1,(int)num2);
02532       }
02533   }
02534   else {
02535       s1 = s; s = SkipAName(s);
02536       if ( *s != ',' && *s != ' ' && *s != '\t' ) goto syntax;
02537       *s++ = 0;
02538       while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
02539       if ( FG.cTable[*s] != 0 && *s != '[' ) goto syntax;
02540       s2 = s; s = SkipAName(s);
02541       if ( *s != 0 && *s != ';' ) goto syntax;
02542       *s = 0;
02543       if ( GetName(AC.exprnames,s1,&num1,NOAUTO) != CEXPRESSION ) {
```

```
02544             MesPrint("@%s is not an expression",s1);
02545             error = 1;
02546         }
02547         if ( GetName(AC.exprnames,s2,&num2,NOAUTO) != CEXPRESSION ) {
02548             MesPrint("@%s is not an expression",s2);
02549             error = 1;
02550         }
02551         if ( error == 0 ) {
02552             ExchangeExpressions((int)num1,(int)num2);
02553         }
02554     }
02555     return(error);
02556 syntax:
02557     MesPrint("@Proper syntax: %#exchange expr1,expr2 or %#exchange $var1,$var2");
02558     return(1);
02559 }
02560
02561 /*
02562         #] DoPreExchange :
02563         #[ DoCall :
02564 */
02565
02566 int DoCall(UBYTE *s)
02567 {
02568     UBYTE *t, *u, *v, *name, c, cp, *args1, *args2, *t1, *t2, *wild = 0;
02569     int bratype = 0, wildargs = 0, inwildargs = 0, nwildargs = 0;
02570     PROCEDURE *p;
02571     int streamoffset;
02572     int i, namesize, narg1, narg2, bralevel, numpre;
02573     LONG i1, i2;
02574     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
02575     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
02576 /*
02577     1:  Get the name of the procedure.
02578     2:  Locate the procedure.
02579 */
02580     name = s; s = EndOfToken(s); c = *s; *s = 0;
02581     for ( i = NumProcedures-1; i >= 0; i-- ) {
02582         if ( StrCmp(Procedures[i].name,name) == 0 ) break;
02583     }
02584     // AP.ProcList.num = NumProcedures is incremented inside FromList
02585     p = (PROCEDURE *)FromList(&AP.ProcList);
02586     if ( i < 0 ) {  /* Try to find a file */
02587         namesize = 0;
02588         t = name;
02589         while ( *t ) { t++; namesize++; }
02590         t = AP.procedureExtension;
02591         while ( *t ) { t++; namesize++; }
02592         t = p->name = (UBYTE *)Malloc1(namesize+2,"procedure");
02593         u = name;
02594         while ( *u ) *t++ = *u++;
02595         *t++ = '.';
02596         v = AP.procedureExtension;
02597         while ( *v ) *t++ = *v++;
02598         *t = 0;
02599         p->loadmode = 0;    /* buffer should be freed at end */
02600         p->mustfree = 1;
02601         p->p.buffer = LoadInputFile(p->name,PROCEDUREFILE);
02602         if ( p->p.buffer == 0 ) return(-1);
02603         t[-4] = 0;
02604     }
02605     else {
02606         p->p.buffer = Procedures[i].p.buffer;
02607         p->name = Procedures[i].name;
02608         p->loadmode = 1;
02609         p->mustfree = 0; // this is just a copy of pointers to a permanently stored procedure
02610     }
02611     t = p->p.buffer;
02612     SKIPBLANKS(t)
02613     if ( *t++ != '#' ) goto wrongfile;
02614     SKIPBLANKS(t)
02615     t += 9;
02616     SKIPBLANKS(t)
02617     u = EndOfToken(t);
02618     cp = *u; *u = 0;
02619     if ( StrCmp(t,name) != 0 ) goto wrongfile;
02620     *u = cp;
02621     *s = c;
02622 /*
02623     The pointer p points to the contents of the procedure (in memory)
02624     Now we have to match the arguments. u points to after the name
02625     in the 'file', s to after the name in the call statement.
02626 */
02627     bralevel = narg1 = narg2 = 0; args2 = u;
02628     SKIPBLANKS(u)
02629     if ( *u == '(' ) {
02630         u++; SKIPBLANKS(u)
```

```
02631         args2 = u;
02632         while ( *u != ')' ) {
02633             if ( *u == '?' ) { wildargs++; u++; nwildargs = narg2+1; }
02634             narg2++; u = EndOfToken(u); SKIPBLANKS(u)
02635             if ( *u == ',' ) { u++; SKIPBLANKS(u) }
02636             else if ( *u != ')' || ( wildargs > 1 ) ) {
02637                 MesPrint("@Illegal argument field in procedure %s",p->name);
02638                 return(-1);
02639             }
02640         }
02641     }
02642     while ( *u != LINEFEED ) u++;
02643     SKIPBLANKS(s)
02644     args1 = s+1;
02645     if ( *s == '(' ) bratype = 1;
02646     do {
02647         if ( *s == '{' && bratype == 0 ) bralevel++;
02648         else if ( *s == '(' && bratype == 1 ) bralevel++;
02649         else if ( *s == '}' && bratype == 0 ) {
02650             bralevel--;
02651             if ( bralevel == 0 ) {
02652                 *s = 0; narg1++;
02653                 if ( wildargs && narg1 == nwildargs ) wild = s;
02654             }
02655         }
02656         else if ( *s == ')' && bratype == 1 ) {
02657             bralevel--;
02658             if ( bralevel == 0 ) {
02659                 *s = 0; narg1++;
02660                 if ( wildargs && narg1 == nwildargs ) wild = s;
02661             }
02662         }
02663         /*[12dec2003 mt]:*/
02664         /*else if ( *s == ',' || *s == '|' ) {*/
02665         else if (set_in(*s,AC.separators)) {/*Function set_in see in
02666                                                 file tools.c*/
02667         /*:[12dec2003 mt]*/
02668             *s = 0; narg1++;
02669             if ( wildargs && narg1 == nwildargs ) wild = s;
02670         }
02671         else if ( *s == '\\' ) s++;
02672         s++;
02673     } while ( bralevel > 0 );
02674     if ( wildargs && narg1 >= narg2-1 ) {
02675         inwildargs = narg1-narg2+1;
02676         if ( inwildargs == 0 ) nwildargs = 0;
02677         else {
02678             while ( inwildargs > 1 ) {
02679                 *wild = ',';
02680                 while ( *wild ) wild++;
02681                 inwildargs--;
02682             }
02683         }
02684     }
02685     else if ( narg1 != narg2 && ( narg2 != 0 || narg1 != 1 || *args1 != 0 ) ) {
02686         MesPrint("@Arguments of procedure %s are not matching",p->name);
02687         return(-1);
02688     }
02689     numpre = -NumPre-1; /* For the stream */
02690     for ( i = 0; i < narg2; i++ ) {
02691         t = args2;
02692         if ( *t == '?' ) {
02693             args2++;
02694         }
02695         if ( *t == '?' && inwildargs == 0 ) {
02696             args2 = EndOfToken(args2); c = *args2; *args2 = 0;
02697             if ( PutPreVar(t,(UBYTE *)"",0,0) < 0 ) return(-1);
02698         }
02699         else {
02700             args2 = EndOfToken(args2); c = *args2; *args2 = 0;
02701             t1 = t2 = args1;
02702             while ( *t1 ) {
02703                 if ( *t1 == '\\' ) t1++;
02704                 if ( t1 != t2 ) *t2 = *t1;
02705                 t2++; t1++;
02706             }
02707             *t2 = 0;
02708             if ( PutPreVar(t,args1,0,0) < 0 ) return(-1);
02709             args1 = t1+1;                   /* Next argument */
02710         }
02711         *args2 = c; SKIPBLANKS(args2)  /* skip to next name */
02712         args2++; SKIPBLANKS(args2)
02713     }
02714     streamoffset = AC.CurrentStream - AC.Streams;
02715     args1 = AC.CurrentStream->name;
02716     AC.CurrentStream->name = p->name;
02717     i1 = AC.CurrentStream->linenumber;
```

```
02718     i2 = AC.CurrentStream->prevline;
02719     AC.CurrentStream->prevline   =
02720     AC.CurrentStream->linenumber = 2;
02721     OpenStream(u+1,PREREADSTREAM3,numpre,PRENOACTION);
02722     AC.Streams[streamoffset].name = args1;
02723     AC.Streams[streamoffset].linenumber = i1;
02724     AC.Streams[streamoffset].prevline   = i2;
02725     AddToPreTypes(PRETYPEPROCEDURE);
02726     return(0);
02727 wrongfile:;
02728     if ( i < 0 ) MesPrint("@File %s is not a proper procedure",p->name);
02729     else {
02730 /* INTERNAL_ERROR_EXCL_START */
02731         MesPrint("!>Internal error with procedure names: %s",name);
02732 /* INTERNAL_ERROR_EXCL_STOP */
02733     }
02734     return(-1);
02735 }
02736
02737 /*
02738         #] DoCall :
02739         #[ DoDebug :
02740 */
02741
02742 int DoDebug(UBYTE *s)
02743 {
02744     int x;
02745     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
02746     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
02747     NeedNumber(x,s,nonumber)
02748     if ( x < 0 || x >(PREPROONLY
02749                     | DUMPTOCOMPILER
02750                     | DUMPOUTTERMS
02751                     | DUMPINTERMS
02752                     | DUMPTOSORT
02753                     | DUMPTOPARALLEL
02754 #ifdef WITHPTHREADS
02755                     | THREADSDEBUG
02756 #endif
02757             ) ) goto nonumber;
02758     AP.PreDebug = 0;
02759     if ( ( x & PREPROONLY     ) != 0 ) AP.PreDebug |= PREPROONLY;     /* 1  */
02760     if ( ( x & DUMPTOCOMPILER ) != 0 ) AP.PreDebug |= DUMPTOCOMPILER; /* 2  */
02761     if ( ( x & DUMPOUTTERMS   ) != 0 ) AP.PreDebug |= DUMPOUTTERMS;   /* 4  */
02762     if ( ( x & DUMPINTERMS    ) != 0 ) AP.PreDebug |= DUMPINTERMS;    /* 8  */
02763     if ( ( x & DUMPTOSORT     ) != 0 ) AP.PreDebug |= DUMPTOSORT;     /* 16 */
02764     if ( ( x & DUMPTOPARALLEL ) != 0 ) AP.PreDebug |= DUMPTOPARALLEL; /* 32 */
02765 #ifdef WITHPTHREADS
02766     if ( ( x & THREADSDEBUG   ) != 0 ) AP.PreDebug |= THREADSDEBUG;   /* 64 */
02767 #endif
02768     return(0);
02769 nonumber:
02770     MesPrint("@Illegal argument for debug instruction");
02771     return(1);
02772 }
02773
02774 /*
02775         #] DoDebug :
02776         #[ DoTerminate :
02777 */
02778
02779 int DoTerminate(UBYTE *s)
02780 {
02781     int x;
02782     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
02783     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
02784     if ( *s ) {
02785         NeedNumber(x,s,nonumber)
02786         Terminate(x);
02787     }
02788     else {
02789         Terminate(-1);
02790     }
02791     return(0);
02792 nonumber:
02793     MesPrint("@Illegal argument for terminate instruction");
02794     return(1);
02795 }
02796
02797 /*
02798         #] DoTerminate :
02799         #[ DoContinueDo :
02800 */
02801
02813 int DoContinueDo(UBYTE *s)
02814 {
02815     DOLOOP *loop;
```

```
02816     WORD levels;
02817     int result;
02818
02819     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
02820     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
02821
02822     if ( NumDoLoops <= 0 ) {
02823         MesPrint("@%#continuedo without %#do");
02824         return(1);
02825     }
02826
02827     SkipSpaces(&s);
02828     if ( *s == 0 ) {
02829         levels = 1;
02830     }
02831     else if ( FG.cTable[*s] == 1 ) {
02832         ParseNumber(levels,s);
02833         SkipSpaces(&s);
02834         if ( *s != 0 ) goto improper;
02835     }
02836     else {
02837 improper:
02838         MesPrint("@Improper syntax of %#continuedo instruction");
02839         return(1);
02840     }
02841
02842     if ( levels > NumDoLoops ) {
02843         MesPrint("@Too many loop levels requested in %#continuedo instruction");
02844         return(1);
02845     }
02846
02847     result = ExitDoLoops(levels-1,"continuedo");
02848     if ( result != 0 ) return(result);
02849
02850     if ( levels <= 0 ) return(0);
02851
02852     if ( AC.CurrentStream->type == PREREADSTREAM3
02853         || AP.PreTypes[AP.NumPreTypes] == PRETYPEPROCEDURE ) {
02854             MesPrint("@Trying to jump out of a procedure with a %#continuedo instruction");
02855             return(1);
02856         }
02857
02858     loop = &(DoLoops[NumDoLoops-1]);
02859     AP.NumPreTypes = loop->NumPreTypes+1;
02860     AP.PreIfLevel = loop->PreIfLevel;
02861     AP.PreSwitchLevel = loop->PreSwitchLevel;
02862
02863     return(DoEnddo(s));
02864 }
02865
02866 /*
02867         #] DoContinueDo :
02868         #[ DoDo :
02869
02870         The do loop has three varieties:
02871         #do i = num1,num2 [,num3]
02872         #do i = {string1,string2,....,stringn}
02873             The | as separator is also allowed for backwards compatibility
02874         #do i = expression      One by one all terms of the expression
02875 */
02876
02877 int DoDo(UBYTE *s)
02878 {
02879     GETIDENTITY
02880     UBYTE *t, c, *u, *uu;
02881     DOLOOP *loop;
02882     WORD expnum;
02883     LONG linenum  = AC.CurrentStream->linenumber;
02884     int oldNoShowInput = AC.NoShowInput, i, oldpreassignflag;
02885
02886     if ( ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH )
02887     || ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) ) {
02888         if ( PreSkip((UBYTE *)"do",(UBYTE *)"enddo",1) ) return(-1);
02889         return(0);
02890     }
02891
02892 /*
02893     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
02894     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
02895 */
02896     AddToPreTypes(PRETYPEDO);
02897
02898     loop = (DOLOOP *)FromList(&AP.LoopList);
02899     loop->firstdollar = loop->lastdollar = loop->incdollar = -1;
02900     loop->NumPreTypes = AP.NumPreTypes-1;
02901     loop->PreIfLevel = AP.PreIfLevel;
02902     loop->PreSwitchLevel = AP.PreSwitchLevel;
```

```
02903     AC.NoShowInput = 1;
02904     if ( PreLoad(&(loop->p),(UBYTE *)"do",(UBYTE *)"enddo",1,"doloop") ) return(-1);
02905     AC.NoShowInput = oldNoShowInput;
02906     loop->NoShowInput = AC.NoShowInput;
02907 /*
02908     Get now the name. We have to take great care when the name is terminated!
02909 */
02910     s = loop->p.buffer + (s - AP.preStart);
02911     SKIPBLANKS(s)
02912     loop->name = s;
02913     if ( chartype[*s] != 0 ) goto illname;
02914     s++;
02915     while ( chartype[*s] <= 1 ) s++;
02916     t = s;
02917     while ( *s == ' ' || *s == '\t' ) s++;
02918     if ( *s != '=' ) goto illdo;
02919     s++;
02920     while ( *s == ' ' || *s == '\t' ) s++;
02921     *t = 0;
02922
02923     if ( *s == '{' ) {
02924         loop->type = LISTEDLOOP;
02925         s++; loop->vars = s;
02926         loop->lastnum = 0;
02927         while ( *s != '}' && *s != 0 ) {
02928             if ( set_in(*s,AC.separators) ) { *s = 0; loop->lastnum++; }
02929             else if ( *s == '\\' ) s++;
02930             s++;
02931         }
02932         if ( *s == 0 ) goto illdo;
02933         *s++ = 0;
02934         loop->lastnum++;
02935         loop->firstnum = 0;
02936         loop->contents = s;
02937     }
02938     else if ( *s == '-' || *s == '+' || chartype[*s] == 1 || *s == '$' ) {
02939         loop->type = NUMERICALLOOP;
02940         t = s;
02941         while ( *s && *s != ',' ) s++;
02942         if ( *s == 0 ) goto illdo;
02943         if ( *t == '$' ) {
02944             c = *s; *s = 0;
02945             if ( GetName(AC.dollarnames,t+1,&loop->firstdollar,NOAUTO) != CDOLLAR ) {
02946                 MesPrint("@%s is undefined in first parameter in %#do instruction",t);
02947                 return(-1);
02948             }
02949             loop->firstnum = DolToLong(BHEAD loop->firstdollar);
02950             if ( AN.ErrorInDollar ) {
02951                 MesPrint("@%s does not evaluate into a valid loop parameter",t);
02952                 return(-1);
02953             }
02954             *s++ = c;
02955         }
02956         else {
02957             *s = '}';
02958             if ( PreEval(t,&loop->firstnum) == 0 ) goto illdo;
02959             *s++ = ',';
02960         }
02961         t = s;
02962         while ( *s && *s != ',' && *s != ';' && *s != LINEFEED ) s++;
02963         c = *s;
02964         if ( *t == '$' ) {
02965             *s = 0;
02966             if ( GetName(AC.dollarnames,t+1,&loop->lastdollar,NOAUTO) != CDOLLAR ) {
02967                 MesPrint("@%s is undefined in second parameter in %#do instruction",t);
02968                 return(-1);
02969             }
02970             loop->lastnum = DolToLong(BHEAD loop->lastdollar);
02971             if ( AN.ErrorInDollar ) {
02972                 MesPrint("@%s does not evaluate into a valid loop parameter",t);
02973                 return(-1);
02974             }
02975             *s++ = c;
02976         }
02977         else {
02978             *s = '}';
02979             if ( PreEval(t,&loop->lastnum) == 0 ) goto illdo;
02980             *s++ = c;
02981         }
02982         if ( c == ',' ) {
02983             t = s;
02984             while ( *s && *s != ';' && *s != LINEFEED ) s++;
02985             if ( *t == '$' ) {
02986                 c = *s; *s = 0;
02987                 if ( GetName(AC.dollarnames,t+1,&loop->incdollar,NOAUTO) != CDOLLAR ) {
02988                     MesPrint("@%s is undefined in third parameter in %#do instruction",t);
02989                     return(-1);
```

```
02990                 }
02991                 loop->incnum = DolToLong(BHEAD loop->incdollar);
02992                 if ( AN.ErrorInDollar ) {
02993                     MesPrint("@%s does not evaluate into a valid loop parameter",t);
02994                     return(-1);
02995                 }
02996                 *s++ = c;
02997             }
02998             else {
02999                 c = *s; *s = '}';
03000                 if ( PreEval(t,&loop->incnum) == 0 ) goto illdo;
03001                 *s++ = c;
03002             }
03003         }
03004         else loop->incnum = 1;
03005         loop->contents = s;
03006     }
03007     else if ( ( chartype[*s] == 0 ) || ( *s == '[' ) ) {
03008         int oldNumPotModdollars = NumPotModdollars;
03009 #ifdef WITHMPI
03010         WORD oldRhsExprInModuleFlag = AC.RhsExprInModuleFlag;
03011         AC.RhsExprInModuleFlag = 0;
03012 #endif
03013         t = s;
03014         if ( ( s = SkipAName(s) ) == 0 ) goto illdo;
03015         c = *s; *s = 0;
03016         if ( GetName(AC.exprnames,t,&expnum,NOAUTO) == CEXPRESSION ) {
03017             loop->type = ONEEXPRESSION;
03018 /*
03019             We should remember the expression by name for when it gets
03020             renumbered!!! If it gets deleted there will be a crash or at
03021             least the loop terminates.
03022 */
03023             loop->vars = t;
03024         }
03025         else goto illdo;
03026         if ( c == ',' || c == '\t' || c == ';' ) { s++; }
03027         else if ( c != 0 && c != '\n' ) goto illdo;
03028         while ( *s == ',' || *s == '\t' || *s == ';' ) s++;
03029         if ( *s != 0 && *s != '\n' ) goto illdo;
03030         loop->firstnum = 0;
03031         s++;
03032         loop->contents = s;
03033         loop->incnum = 0;
03034 /*
03035         Next determine size of statement and allocate space
03036 */
03037         while ( *t ) t++;
03038         i = t - loop->vars;
03039         t = loop->name;
03040         while ( *t ) { t++; i++; }
03041         i += 4;
03042         loop->dollarname = Malloc1((LONG)i,"do-loop instruction");
03043 /*
03044         Construct the statement
03045 */
03046         u = loop->dollarname;
03047         *u++ = '$'; t = loop->name; while ( *t ) *u++ = *t++;
03048         *u++ = '_'; uu = u; *u++ = '='; t = loop->vars;
03049         while ( *t ) *u++ = *t++;
03050         *t = 0; *u = 0;
03051 /*
03052         Compile and put in dollar variable.
03053         Note that we remember the dollar by name and that this name ends in _
03054 */
03055         oldpreassignflag = AP.PreAssignFlag;
03056         AP.PreAssignFlag = 2;
03057         CompileStatement(loop->dollarname);
03058         if ( CatchDollar(0) ) {
03059             MesPrint("@Cannot load expression in do loop");
03060             return(-1);
03061         }
03062         AP.PreAssignFlag = oldpreassignflag;
03063         NumPotModdollars = oldNumPotModdollars;
03064 #ifdef WITHMPI
03065         AC.RhsExprInModuleFlag = oldRhsExprInModuleFlag;
03066 #endif
03067         *uu = 0;
03068     }
03069     else goto illdo; /* Syntax problems */
03070     loop->errorsinloop = 0;
03071 /*  loop->startlinenumber = linenum+1; 5-oct-2000 One too much? */
03072     loop->startlinenumber = linenum;
03073     PutPreVar(loop->name,(UBYTE *)"0",0,0);
03074     loop->firstloopcall = 1;
03075     return(DoEnddo(s));
03076 illname:;
```

```
03077     MesPrint("@Improper name for do loop variable");
03078     return(-1);
03079 illdo:;
03080     MesPrint("@Improper syntax in do loop instruction");
03081     return(-1);
03082 }
03083
03084 /*
03085         #] DoDo :
03086         #[ DoBreakDo :
03087
03088         #breakdo [num]
03089         jumps out of num #do-loops (if there are that many) (default is 1)
03090 */
03091
03092 int DoBreakDo(UBYTE *s)
03093 {
03094     WORD levels;
03095
03096     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03097     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03098
03099     if ( NumDoLoops <= 0 ) {
03100         MesPrint("@%#breakdo without %#do");
03101         return(1);
03102     }
03103 /*
03104     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEDO ) { MessPreNesting(4); return(-1); }
03105 */
03106     while ( *s && ( *s == ',' || *s == ' ' || *s == '\t' ) ) s++;
03107     if ( *s == 0 ) {
03108         levels = 1;
03109     }
03110     else if ( FG.cTable[*s] == 1 ) {
03111         levels = 0;
03112         while ( *s >= '0' && *s <= '9' ) { levels = 10*levels + *s++ - '0'; }
03113         if ( *s != 0 ) goto improper;
03114     }
03115     else {
03116 improper:
03117         MesPrint("@Improper syntax of %#breakdo instruction");
03118         return(1);
03119     }
03120     if ( levels > NumDoLoops ) {
03121         MesPrint("@Too many loop levels requested in %#breakdo instruction");
03122         Terminate(-1);
03123     }
03124     return(ExitDoLoops(levels,"breakdo"));
03125 }
03126
03134 static int ExitDoLoops(int levels, const char *instruction)
03135 {
03136     DOLOOP *loop;
03137     while ( levels > 0 ) {
03138         while ( AC.CurrentStream->type != PREREADSTREAM
03139           && AC.CurrentStream->type != PREREADSTREAM2
03140           && AC.CurrentStream->type != PREREADSTREAM3 ) {
03141             AC.CurrentStream = CloseStream(AC.CurrentStream);
03142         }
03143         while ( AP.PreTypes[AP.NumPreTypes] != PRETYPEDO
03144         && AP.PreTypes[AP.NumPreTypes] != PRETYPEPROCEDURE ) AP.NumPreTypes--;
03145         if ( AC.CurrentStream->type == PREREADSTREAM3
03146         || AP.PreTypes[AP.NumPreTypes] == PRETYPEPROCEDURE ) {
03147             MesPrint("@Trying to jump out of a procedure with a %#%s instruction",instruction);
03148             return(1);
03149         }
03150         loop = &(DoLoops[NumDoLoops-1]);
03151         AP.NumPreTypes = loop->NumPreTypes;
03152         AP.PreIfLevel = loop->PreIfLevel;
03153         AP.PreSwitchLevel = loop->PreSwitchLevel;
03154         NumDoLoops--;
03155         DoUndefine(loop->name);
03156         M_free(loop->p.buffer,"loop->p.buffer");
03157         loop->firstloopcall = 0;
03158
03159         AC.CurrentStream = CloseStream(AC.CurrentStream);
03160         levels--;
03161     }
03162     return(0);
03163 }
03164
03165 /*
03166         #] DoBreakDo :
03167         #[ DoElse :
03168 */
03169
03170 int DoElse(UBYTE *s)
```

```
03171 {
03172     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEIF ) {
03173         if ( AP.PreIfLevel <= 0 ) MesPrint("@%#else without corresponding %#if");
03174         else MessPreNesting(1);
03175         return(-1);
03176     }
03177     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03178     while ( *s == ' ' ) s++;
03179     if ( tolower(*s) == 'i' && tolower(s[1]) == 'f' && s[2]
03180         && FG.cTable[s[2]] > 1 && s[2] != '_' ) {
03181         s += 2;
03182         while ( *s == ' ' ) s++;
03183         return(DoElseif(s));
03184     }
03185     if ( AP.PreIfLevel <= 0 ) {
03186         MesPrint("@%#else without corresponding %#if");
03187         return(-1);
03188     }
03189     switch ( AP.PreIfStack[AP.PreIfLevel] ) {
03190         case EXECUTINGIF:
03191             AP.PreIfStack[AP.PreIfLevel] = LOOKINGFORENDIF;
03192             break;
03193         case LOOKINGFORELSE:
03194             AP.PreIfStack[AP.PreIfLevel] = EXECUTINGIF;
03195             break;
03196         case LOOKINGFORENDIF:
03197             break;
03198     }
03199     return(0);
03200 }
03201
03202 /*
03203         #] DoElse :
03204         #[ DoElseif :
03205 */
03206
03207 int DoElseif(UBYTE *s)
03208 {
03209     int condition;
03210     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEIF ) {
03211         if ( AP.PreIfLevel <= 0 ) MesPrint("@%#elseif without corresponding %#if");
03212         else MessPreNesting(2);
03213         return(-1);
03214     }
03215     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03216     if ( AP.PreIfLevel <= 0 ) {
03217         MesPrint("@%#elseif without corresponding %#if");
03218         return(-1);
03219     }
03220     switch ( AP.PreIfStack[AP.PreIfLevel] ) {
03221         case EXECUTINGIF:
03222             AP.PreIfStack[AP.PreIfLevel] = LOOKINGFORENDIF;
03223             break;
03224         case LOOKINGFORELSE:
03225             if ( ( condition = EvalPreIf(s) ) < 0 ) return(-1);
03226             AP.PreIfStack[AP.PreIfLevel] = condition;
03227             break;
03228         case LOOKINGFORENDIF:
03229             break;
03230     }
03231     return(0);
03232 }
03233
03234 /*
03235         #] DoElseif :
03236         #[ DoEnddo :
03237
03238         At the first call there is no stream yet.
03239         After that we have to close the stream and start a new one.
03240 */
03241
03242 int DoEnddo(UBYTE *s)
03243 {
03244     GETIDENTITY
03245     DOLOOP *loop;
03246     UBYTE *t, *tt, *value, numstr[16];
03247     LONG xval;
03248     int xsign, retval;
03249     DUMMYUSE(s);
03250     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03251     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03252 /*
03253     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ||
03254         AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) {
03255         if ( AP.PreTypes[AP.NumPreTypes] == PRETYPEDO ) AP.NumPreTypes--;
03256         else { MessPreNesting(3); return(-1); }
03257         return(0);
```

```
03258     }
03259 */
03260     if ( NumDoLoops <= 0 ) {
03261         MesPrint("@%#enddo without %#do");
03262         return(1);
03263     }
03264     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEDO ) { MessPreNesting(4); return(-1); }
03265     loop = &(DoLoops[NumDoLoops-1]);
03266     if ( !loop->firstloopcall ) AC.CurrentStream = CloseStream(AC.CurrentStream);
03267
03268     if ( loop->errorsinloop ) {
03269         MesPrint("++++Errors in Loop");
03270         goto finish;
03271     }
03272     if ( loop->type == LISTEDLOOP ) {
03273         if ( loop->firstnum >= loop->lastnum ) goto finish;
03274         loop->firstnum++;
03275         t = value = loop->vars;
03276         while ( *value ) value++;
03277         value++;
03278         loop->vars = value;
03279         value = tt = t;
03280         while ( *value ) {
03281             if ( *value == '\\' ) value++;
03282             *tt++ = *value++;
03283         }
03284         *tt = 0;
03285         PutPreVar(loop->name,t,0,1);    /* We overwrite the definition */
03286     }
03287     else if ( loop->type == NUMERICALLOOP ) {
03288
03289         if ( !loop->firstloopcall ) {
03290 /*
03291             Test whether the variable was changed inside the loop into
03292             a different numerical value. If so, adjust.
03293 */
03294             t = GetPreVar(loop->name,WITHOUTERROR);
03295             if ( t ) {
03296                 value = t;
03297                 xsign = 1;
03298                 while ( *value && ( *value == ' '
03299                         || *value == '-' || *value == '+' ) ) {
03300                     if ( *value == '-' ) xsign = -xsign;
03301                     value++;
03302                 }
03303                 t = value; xval = 0;
03304                 while ( *value >= '0' && *value <= '9' ) xval = 10*xval + *value++ - '0';
03305                 while ( *value && *value == ' ' ) value++;
03306                 if ( *value == 0 ) {
03307 /*
03308                     Now we may substitute the loopvalue.
03309 */
03310                     if ( xsign < 0 ) xval = -xval;
03311                     if ( loop->incdollar >= 0 ) {
03312                         loop->incnum = DolToLong(BHEAD loop->incdollar);
03313                         if ( AN.ErrorInDollar ) {
03314                             MesPrint("@%s does not evaluate into a valid third loop
      parameter",DOLLARNAME(Dollars,loop->incdollar));
03315                             return(-1);
03316                         }
03317                     }
03318                     loop->firstnum = xval + loop->incnum;
03319                 }
03320             }
03321             if ( loop->lastdollar >= 0 ) {
03322                 loop->lastnum = DolToLong(BHEAD loop->lastdollar);
03323                 if ( AN.ErrorInDollar ) {
03324                     MesPrint("@%s does not evaluate into a valid second loop
      parameter",DOLLARNAME(Dollars,loop->lastdollar));
03325                     return(-1);
03326                 }
03327             }
03328         }
03329         if ( ( loop->incnum > 0 && loop->firstnum > loop->lastnum )
03330         || ( loop->incnum < 0 && loop->firstnum < loop->lastnum ) ) goto finish;
03331         NumToStr(numstr,loop->firstnum);
03332         t = numstr;
03333         loop->firstnum += loop->incnum;
03334         PutPreVar(loop->name,t,0,1);    /* We overwrite the definition */
03335     }
03336     else if ( loop->type == ONEEXPRESSION ) {
03337 /*
03338         Find the dollar expression
03339 */
03340         WORD numdollar = GetDollar(loop->dollarname+1);
03341         DOLLARS d = Dollars + numdollar;
03342         WORD *w, *dw, v, *ww;
```

```
03343         if ( (d->where) == 0 ) {
03344             d->type = DOLUNDEFINED;
03345             M_free(loop->dollarname,"do-loop instruction");
03346             goto finish;
03347         }
03348         w = d->where + loop->incnum;
03349         if ( *w == 0 ) {
03350             M_free(d->where,"dollar");
03351             d->where = 0;
03352             d->type = DOLUNDEFINED;
03353             M_free(loop->dollarname,"do-loop instruction");
03354             goto finish;
03355         }
03356         loop->incnum += *w;
03357 /*
03358         Now the term has to be converted to text.
03359 */
03360         ww = w + *w; v = *ww; *ww = 0;
03361         dw = d->where; d->where = w;
03362         t = WriteDollarToBuffer(numdollar,1);
03363         d->where = dw; *ww = v;
03364         PutPreVar(loop->name,t,0,1);   /* We overwrite the definition */
03365         M_free(t,"dollar");
03366     }
03367     if ( loop->firstloopcall ) OpenStream(loop->contents,PREREADSTREAM2,0,PRENOACTION);
03368     else OpenStream(loop->contents,PREREADSTREAM,0,PRENOACTION);
03369     AC.CurrentStream->prevline   =
03370     AC.CurrentStream->linenumber = loop->startlinenumber;
03371     AC.CurrentStream->eqnum = 0;
03372     loop->firstloopcall = 0;
03373     return(0);
03374 finish:;
03375     NumDoLoops--;
03376     retval = DoUndefine(loop->name);
03377     M_free(loop->p.buffer,"loop->p.buffer");
03378     loop->firstloopcall = 0;
03379     AP.NumPreTypes--;
03380     return(retval);
03381 }
03382
03383 /*
03384         #] DoEnddo :
03385         #[ DoEndif :
03386 */
03387
03388 int DoEndif(UBYTE *s)
03389 {
03390     DUMMYUSE(s);
03391     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEIF ) {
03392         if ( AP.PreIfLevel <= 0 ) MesPrint("@%#endif without corresponding %#if");
03393         else MessPreNesting(5);
03394         return(-1);
03395     }
03396     AP.NumPreTypes--;
03397     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03398     if ( AP.PreIfLevel <= 0 ) {
03399         MesPrint("@%#endif without corresponding %#if");
03400         return(-1);
03401     }
03402     AP.PreIfLevel--;
03403     return(0);
03404 }
03405
03406 /*
03407         #] DoEndif :
03408         #[ DoEndprocedure :
03409
03410         Action is simple: close the current stream if it is still
03411         the stream from which the statement came.
03412         Then pop the current procedure and all its local derivatives.
03413         if loadmode > 1 the procedure was defined locally.
03414 */
03415
03416 int DoEndprocedure(UBYTE *s)
03417 {
03418     DUMMYUSE(s);
03419     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEPROCEDURE ) {
03420         MessPreNesting(6);
03421         return(-1);
03422     }
03423     AP.NumPreTypes--;
03424     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03425     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03426     AC.CurrentStream = CloseStream(AC.CurrentStream);
03427
03428     do {
03429         NumProcedures--;
```

```
03430         if ( Procedures[NumProcedures].mustfree == 1 ) {
03431             M_free(Procedures[NumProcedures].p.buffer,"procedures buffer");
03432             M_free(Procedures[NumProcedures].name,"procedures name");
03433         }
03434     } while ( Procedures[NumProcedures].loadmode > 1 );
03435     return(0);
03436 }
03437
03438 /*
03439         #] DoEndprocedure :
03440         #[ DoIf :
03441 */
03442
03443 int DoIf(UBYTE *s)
03444 {
03445     int condition;
03446     AddToPreTypes(PRETYPEIF);
03447     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03448     if ( AP.PreIfStack[AP.PreIfLevel] == EXECUTINGIF ) {
03449         condition = EvalPreIf(s);
03450         if ( condition < 0 ) return(-1);
03451     }
03452     else condition = LOOKINGFORENDIF;
03453     if ( AP.PreIfLevel+1 >= AP.MaxPreIfLevel ) {
03454         int **ppp = &AP.PreIfStack; /* To avoid a compiler warning */
03455         if ( DoubleList((void ***)ppp,&AP.MaxPreIfLevel,sizeof(int),
03456             "PreIfLevels") ) return(-1);
03457     }
03458     AP.PreIfStack[++AP.PreIfLevel] = condition;
03459     return(0);
03460 }
03461
03462 /*
03463         #] DoIf :
03464         #[ DoIfdef :
03465 */
03466
03467 int DoIfdef(UBYTE *s, int par)
03468 {
03469     int condition;
03470     AddToPreTypes(PRETYPEIF);
03471     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03472     if ( AP.PreIfStack[AP.PreIfLevel] == EXECUTINGIF ) {
03473         while ( *s == ' ' || *s == '\t' ) s++;
03474         if ( ( *s == 0 ) == ( par == 1 ) ) condition = LOOKINGFORELSE;
03475         else                               condition = EXECUTINGIF;
03476     }
03477     else condition = LOOKINGFORENDIF;
03478     if ( AP.PreIfLevel+1 >= AP.MaxPreIfLevel ) {
03479         int **ppp = &AP.PreIfStack; /* to avoid a compiler warning */
03480         if ( DoubleList((void ***)ppp,&AP.MaxPreIfLevel,sizeof(int),
03481             "PreIfLevels") ) return(-1);
03482     }
03483     AP.PreIfStack[++AP.PreIfLevel] = condition;
03484     return(0);
03485 }
03486
03487 /*
03488         #] DoIfdef :
03489         #[ DoIfydef :
03490 */
03491
03492 int DoIfydef(UBYTE *s)
03493 {
03494     return DoIfdef(s,1);
03495 }
03496
03497 /*
03498         #] DoIfydef :
03499         #[ DoIfndef :
03500 */
03501
03502 int DoIfndef(UBYTE *s)
03503 {
03504     return DoIfdef(s,2);
03505 }
03506
03507 /*
03508         #] DoIfndef :
03509         #[ DoInside :
03510
03511     #inside $var1,...,$varn
03512         statements without .sort
03513     #endinside
03514
03515     executes the statements on the contents of the $ variables as if they
03516     are a module. The results are put back in the dollar variables.
```

```
03517     To do this right we need a struct with
03518         old compiler buffer
03519         list of numbers of dollars
03520         length of the list
03521         length of the array containing the list
03522     Because we need to compose statements, the statement buffer must be
03523     empty. This means that we have to test for that. Same at the end. We
03524     must have a completed statement.
03525 */
03526
03527 int DoInside(UBYTE *s)
03528 {
03529     GETIDENTITY
03530     int numdol, error = 0;
03531     WORD *nb, newsize, i;
03532     UBYTE *name, c;
03533     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03534     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03535     if ( AP.PreInsideLevel != 0 ) {
03536         MesPrint("@Illegal nesting of %#inside/%#endinside instructions");
03537         return(-1);
03538     }
03539 /*
03540     if ( AP.PreContinuation ) {
03541         error = -1;
03542         MesPrint("@%#inside cannot be inside a regular statement");
03543     }
03544 */
03545     PUSHPREASSIGNLEVEL
03546 /*
03547     Now the dollars to do
03548 */
03549     AP.inside.numdollars = 0;
03550     for(;;) {
03551         while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
03552         if ( *s == 0 ) break;
03553         if ( *s != '$' ) {
03554             MesPrint("@%#inside instruction can have only $ variables for parameters");
03555             return(-1);
03556         }
03557         s++;
03558         name = s;
03559         while (chartype[*s] <= 1 ) s++;
03560         c = *s; *s = 0;
03561         if ( ( numdol = GetDollar(name) ) < 0 ) {
03562             MesPrint("@%#inside: $%s has not (yet) been defined",name);
03563             *s = c;
03564             error = -1;
03565         }
03566         else {
03567             *s = c;
03568             if ( AP.inside.numdollars >= AP.inside.size ) {
03569                 if ( AP.inside.buffer == 0 ) newsize = 20;
03570                 else                         newsize = 2*AP.inside.size;
03571                 nb = (WORD *)Malloc1(newsize*sizeof(WORD),"insidebuffer");
03572                 if ( AP.inside.buffer ) {
03573                     for ( i = 0; i < AP.inside.size; i++ ) nb[i] = AP.inside.buffer[i];
03574                     M_free(AP.inside.buffer,"insidebuffer");
03575                 }
03576                 AP.inside.buffer = nb;
03577                 AP.inside.size = newsize;
03578             }
03579             AP.inside.buffer[AP.inside.numdollars++] = numdol;
03580         }
03581     }
03582 /*
03583     We have to store the configuration of the compiler buffer, so that
03584     we know where to start executing and how to reset the buffer.
03585 */
03586     AP.inside.oldcompiletype = AC.compiletype;
03587     AP.inside.oldparallelflag = AC.mparallelflag;
03588     AP.inside.oldnumpotmoddollars = NumPotModdollars;
03589     AP.inside.oldcbuf = AC.cbufnum;
03590     AP.inside.oldrbuf = AM.rbufnum;
03591     AP.inside.oldcnumlhs = AR.Cnumlhs,
03592     AddToPreTypes(PRETYPEINSIDE);
03593     AP.PreInsideLevel = 1;
03594     AC.cbufnum = AP.inside.inscbuf;
03595     AM.rbufnum = AP.inside.inscbuf;
03596     clearcbuf(AC.cbufnum);
03597     AC.compiletype = 0;
03598     AC.mparallelflag = PARALLELFLAG;
03599 #ifdef WITHMPI
03600     /*
03601      * We use AC.RhsExprInModuleFlag, PotModdollars, and AC.pfirstnum
03602      * in order to check (1) whether there are expression names in RHS,
03603      * (2) which dollar variables can be modified, and (3) which
```

```
03604      * preprocessor variables can be redefined, in #inside.
03605      * We store the current values of them, and then reset them.
03606      */
03607     PF_StoreInsideInfo();
03608     AC.RhsExprInModuleFlag = 0;
03609     NumPotModdollars = 0;
03610     AC.numpfirstnum = 0;
03611 #endif
03612     return(error);
03613 }
03614
03615 /*
03616         #] DoInside :
03617         #[ DoEndInside :
03618 */
03619
03620 int DoEndInside(UBYTE *s)
03621 {
03622     GETIDENTITY
03623     WORD numdol, *oldworkpointer = AT.WorkPointer, *term, *t, j, i;
03624     DOLLARS d, nd;
03625     WORD oldbracketon = AR.BracketOn;
03626     WORD *oldcompresspointer = AR.CompressPointer;
03627     int oldmultithreaded = AS.MultiThreaded;
03628     /* int oldmparallelflag = AC.mparallelflag; */
03629     FILEHANDLE *f;
03630 #ifdef WITHMPI
03631     int error = 0;
03632 #endif
03633     DUMMYUSE(s);
03634     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03635     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03636     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEINSIDE ) {
03637         if ( AP.PreInsideLevel != 1 ) MesPrint("@%#endinside without corresponding %#inside");
03638         else MessPreNesting(11);
03639         return(-1);
03640     }
03641     AP.NumPreTypes--;
03642     if ( AP.PreInsideLevel != 1 ) {
03643         MesPrint("@%#endinside without corresponding %#inside");
03644         return(-1);
03645     }
03646     if ( AP.PreContinuation ) {
03647         MesPrint("@%#endinside: previous statement not terminated.");
03648         Terminate(-1);
03649     }
03650     AC.compiletype = AP.inside.oldcompiletype;
03651     AR.Cnumlhs = cbuf[AM.rbufnum].numlhs;
03652 #ifdef WITHMPI
03653     /*
03654      * If the #inside...#endinside contains expressions in RHS, only the master executes it
03655      * and then broadcasts the result to the all slaves. If not, the all processes execute
03656      * it and in this case no MPI interactions are needed.
03657      */
03658     if ( PF.me == MASTER || !AC.RhsExprInModuleFlag ) {
03659 #endif
03660     AR.BracketOn = 0;
03661     AS.MultiThreaded = 0;
03662     /* AC.mparallelflag = PARALLELFLAG; */
03663     if ( AR.CompressPointer == 0 ) AR.CompressPointer = AR.CompressBuffer;
03664     f = AR.infile; AR.infile = AR.outfile; AR.outfile = f;
03665 /*
03666     Now we have to execute the statements on the proper dollars.
03667 */
03668     for ( i = 0; i < AP.inside.numdollars; i++ ) {
03669         numdol = AP.inside.buffer[i];
03670         nd = d = Dollars + numdol;
03671         if ( d->type != DOLZERO ) {
03672             if ( d->type != DOLTERMS ) nd = DolToTerms(BHEAD numdol);
03673             term = nd->where;
03674             NewSort(BHEAD0);
03675             NewSort(BHEAD0);
03676             AR.MaxDum = AM.IndDum;
03677             while ( *term ) {
03678                 t = oldworkpointer; j = *term;
03679                 NCOPY(t,term,j);
03680                 AT.WorkPointer = t;
03681                 AN.IndDum = AM.IndDum;
03682                 AR.CurDum = ReNumber(BHEAD term);
03683                 if ( Generator(BHEAD oldworkpointer,0) ) {
03684                     MesPrint("@Called from %#endinside");
03685                     MesPrint("@Evaluating variable $%s",DOLLARNAME(Dollars,numdol));
03686                     Terminate(-1);
03687                 }
03688             }
03689             AT.WorkPointer = oldworkpointer;
03690             CleanDollarFactors(d);
```

```
03691             if ( d->where ) { M_free(d->where,"dollar contents"); d->where = 0; }
03692             EndSort(BHEAD (WORD *)((void *)(&(d->where))),2);
03693             LowerSortLevel();
03694             term = d->where; while ( *term ) term += *term;
03695             d->size = term - d->where;
03696             if ( nd != d ) M_free(nd,"Copy of dollar variable");
03697             if ( d->where[0] == 0 ) {
03698                 M_free(d->where,"dollar contents"); d->where = 0;
03699                 d->type = DOLZERO;
03700             }
03701         }
03702     }
03703 #ifdef WITHMPI
03704     }
03705     if ( AC.RhsExprInModuleFlag ) {
03706         /*
03707          * The only master executed the statements in #inside.
03708          * We need to broadcast the result to the all slaves.
03709          */
03710         for ( i = 0; i < AP.inside.numdollars; i++ ) {
03711             /*
03712              * Mark $-variables specified in the #inside instruction as modified
03713              * such that they will be broadcast.
03714              */
03715             AddPotModdollar(AP.inside.buffer[i]);
03716         }
03717         /* Now actual broadcast of modified variables. */
03718         if ( NumPotModdollars > 0 ) {
03719             error = PF_BroadcastModifiedDollars();
03720             if ( error ) goto cleanup;
03721         }
03722         if ( AC.numpfirstnum > 0 ) {
03723             error = PF_BroadcastRedefinedPreVars();
03724             if ( error ) goto cleanup;
03725         }
03726     }
03727 cleanup:
03728 #endif
03729     f = AR.infile; AR.infile = AR.outfile; AR.outfile = f;
03730     AC.cbufnum = AP.inside.oldcbuf;
03731     AM.rbufnum = AP.inside.oldrbuf;
03732     AR.Cnumlhs = AP.inside.oldcnumlhs;
03733     AR.BracketOn = oldbracketon;
03734     AP.PreInsideLevel = 0;
03735     AR.CompressPointer = oldcompresspointer;
03736     AS.MultiThreaded = oldmultithreaded;
03737     AC.mparallelflag = AP.inside.oldparallelflag;
03738     NumPotModdollars = AP.inside.oldnumpotmoddollars;
03739     POPPREASSIGNLEVEL
03740 #ifdef WITHMPI
03741     PF_RestoreInsideInfo();
03742     if ( error ) return error;
03743 #endif
03744     return(0);
03745 }
03746
03747 /*
03748         #] DoEndInside :
03749         #[ DoMessage :
03750 */
03751
03752 int DoMessage(UBYTE *s)
03753 {
03754     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03755     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03756     while ( *s == ' ' || *s == '\t' ) s++;
03757     MesPrint("~~~%s",s);
03758     return(0);
03759 }
03760
03761 /*
03762         #] DoMessage :
03763         #[ DoPipe :
03764 */
03765
03766 int DoPipe(UBYTE *s)
03767 {
03768 #ifndef WITHPIPE
03769     DUMMYUSE(s);
03770 #endif
03771     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03772     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03773 #ifdef WITHPIPE
03774     FLUSHCONSOLE;
03775     while ( *s == ' ' || *s == '\t' ) s++;
03776     if ( OpenStream(s,PIPESTREAM,0,PRENOACTION) == 0 ) return(-1);
03777     return(0);
```

```
03778 #else
03779     Error0("Pipes not implemented on this computer/system");
03780     return(-1);
03781 #endif
03782 }
03783
03784 /*
03785         #] DoPipe :
03786         #[ DoPrcExtension :
03787 */
03788
03789 int DoPrcExtension(UBYTE *s)
03790 {
03791     UBYTE *t, *u, c;
03792     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03793     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03794     while ( *s == ' ' || *s == '\t' ) s++;
03795     if ( *s == 0 || *s == '\n' ) {
03796         MesPrint("@No valid procedure extension specified");
03797         return(-1);
03798     }
03799     if ( FG.cTable[*s] != 0 ) {
03800         MesPrint("@Procedure extension should be a string starting with an alphabetic character. No
    whitespace.");
03801         return(-1);
03802     }
03803     t = s;
03804     while ( *s && *s != '\n' && *s != ' ' && *s != '\t' ) s++;
03805     u = s;
03806     while ( *s == ' ' || *s == '\t' ) s++;
03807     if ( *s != 0 && *s != '\n' ) {
03808         MesPrint("@Too many parameters in ProcedureExtension instruction");
03809         return(-1);
03810     }
03811     c = *u; *u = 0;
03812     if ( AP.procedureExtension ) M_free(AP.procedureExtension,"ProcedureExtension");
03813     AP.procedureExtension = strDup1(t,"ProcedureExtension");
03814     *u = c;
03815     return(0);
03816 }
03817
03818 /*
03819         #] DoPrcExtension :
03820         #[ DoPreOut :
03821 */
03822
03823 int DoPreOut(UBYTE *s)
03824 {
03825     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03826     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03827     if ( tolower(*s) == 'o' ) {
03828         if ( tolower(s[1]) == 'n' && s[2] == 0 ) {
03829             AP.PreOut = 1;
03830             return(0);
03831         }
03832         if ( tolower(s[1]) == 'f' && tolower(s[2]) == 'f' && s[3] == 0 ) {
03833             AP.PreOut = 0;
03834             return(0);
03835         }
03836     }
03837     MesPrint("@Illegal option in PreOut instruction");
03838     return(-1);
03839 }
03840
03841 /*
03842         #] DoPreOut :
03843         #[ DoPrePrintTimes :
03844 */
03845
03846 int DoPrePrintTimes(UBYTE *s)
03847 {
03848     DUMMYUSE(s);
03849     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03850     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03851     PrintRunningTime();
03852     return(0);
03853 }
03854
03855 /*
03856         #] DoPrePrintTimes :
03857         #[ DoPreSortReallocate :
03858 */
03859
03860 int DoPreSortReallocate(UBYTE *s)
03861 {
03862     DUMMYUSE(s);
03863     if ( AC.SortReallocateFlag == 0 ) {
```

```
03864         /* Currently off, so set to 2. Then the reallocation code knows the flag was
03865            set here, since "On sortreallocate;" sets it to 1. */
03866         AC.SortReallocateFlag = 2;
03867     }
03868     /* If the flag is already on, do nothing. */
03869     return(0);
03870 }
03871
03872 /*
03873         #] DoPreSortReallocate :
03874         #[ DoPreAppend :
03875
03876         Syntax:
03877         #append <filename>
03878 */
03879
03880 int DoPreAppend(UBYTE *s)
03881 {
03882     UBYTE *name, *to;
03883
03884     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03885     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03886     if ( AP.preError ) return(0);
03887     while ( *s == ' ' || *s == '\t' ) s++;
03888 /*
03889     Determine where to write
03890 */
03891     if ( *s == '<' ) {
03892         s++;
03893         name = to = s;
03894         while ( *s && *s != '>' ) {
03895             if ( *s == '\\' ) s++;
03896             *to++ = *s++;
03897         }
03898         if ( *s == 0 ) {
03899             MesPrint("@Improper termination of filename");
03900             return(-1);
03901         }
03902         s++;
03903         *to = 0;
03904         if ( *name ) { GetAppendChannel((char *)name); }
03905         else goto improper;
03906     }
03907     else {
03908 improper:
03909         MesPrint("@Proper syntax is: %#append <filename>");
03910         return(-1);
03911     }
03912     return(0);
03913 }
03914
03915 /*
03916         #] DoPreAppend :
03917         #[ DoPreCreate :
03918
03919         Syntax:
03920         #create <filename>
03921 */
03922
03923 int DoPreCreate(UBYTE *s)
03924 {
03925     UBYTE *name, *to;
03926
03927     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03928     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03929     if ( AP.preError ) return(0);
03930     while ( *s == ' ' || *s == '\t' ) s++;
03931 /*
03932     Determine where to write
03933 */
03934     if ( *s == '<' ) {
03935         s++;
03936         name = to = s;
03937         while ( *s && *s != '>' ) {
03938             if ( *s == '\\' ) s++;
03939             *to++ = *s++;
03940         }
03941         if ( *s == 0 ) {
03942             MesPrint("@Improper termination of filename");
03943             return(-1);
03944         }
03945         s++;
03946         *to = 0;
03947         if ( *name ) { GetChannel((char *)name,0); }
03948         else goto improper;
03949     }
03950     else {
```

```
03951 improper:
03952         MesPrint("@Proper syntax is: %#create <filename>");
03953         return(-1);
03954     }
03955     return(0);
03956 }
03957
03958 /*
03959         #] DoPreCreate :
03960         #[ DoPreRemove :
03961 */
03962
03963 int DoPreRemove(UBYTE *s)
03964 {
03965     UBYTE *name, *to;
03966     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03967     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03968     if ( AP.preError ) return(0);
03969     while ( *s == ' ' || *s == '\t' ) s++;
03970     if ( *s == '<' ) { s++; }
03971     else {
03972         MesPrint("@Proper syntax is: %#remove <filename>");
03973         return(-1);
03974     }
03975     name = to = s;
03976     while ( *s && *s != '>' ) {
03977         if ( *s == '\\' ) s++;
03978         *to++ = *s++;
03979     }
03980     if ( *s == 0 ) {
03981         MesPrint("@Improper filename");
03982         return(-1);
03983     }
03984     s++;
03985     *to = 0;
03986     CloseChannel((char *)name);
03987     remove((char *)name);
03988     return(0);
03989 }
03990
03991 /*
03992         #] DoPreRemove :
03993         #[ DoPreClose :
03994 */
03995
03996 int DoPreClose(UBYTE *s)
03997 {
03998     UBYTE *name, *to;
03999     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
04000     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
04001     if ( AP.preError ) return(0);
04002     while ( *s == ' ' || *s == '\t' ) s++;
04003     if ( *s == '<' ) { s++; }
04004     else {
04005         MesPrint("@Proper syntax is: %#close <filename>");
04006         return(-1);
04007     }
04008     name = to = s;
04009     while ( *s && *s != '>' ) {
04010         if ( *s == '\\' ) s++;
04011         *to++ = *s++;
04012     }
04013     if ( *s == 0 ) {
04014         MesPrint("@Improper filename");
04015         return(-1);
04016     }
04017     s++;
04018     *to = 0;
04019     return(CloseChannel((char *)name));
04020 }
04021
04022 /*
04023         #] DoPreClose :
04024         #[ DoPreWrite :
04025
04026         Syntax:
04027         #write [<filename>] "formatstring" [,objects]
04028         The format string can contain the following special objects/codes
04029         \n  newline
04030         \t  tab
04031         \!  if last entry in string: no linefeed at end
04032         \b  put \ in output
04033         %$  $-variable (to be found among the objects)
04034         %e  expression (name to be found among the objects)
04035         %E  expression without ; (name to be found among the objects)
04036         %s  string (to be found among the objects) (with or without "")
04037         %S  subterms (see PrintSubtermList)
```

```
04038 */
04039
04040 int DoPreWrite(UBYTE *s)
04041 {
04042     HANDLERS h;
04043
04044     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
04045     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
04046     if ( AP.preError ) return(0);
04047
04048 #ifdef WITHMPI
04049     if ( PF.me != MASTER ) return 0;
04050 #endif
04051
04052     h.oldsilent    = AM.silent;
04053     h.newlogonly   = h.oldlogonly = AM.FileOnlyFlag;
04054     h.newhandle    = h.oldhandle  = AC.LogHandle;
04055     h.oldprinttype = AO.PrintType;
04056
04057     while ( *s == ' ' || *s == '\t' ) s++;
04058 /*
04059     Determine where to write
04060 */
04061     if( (s=defineChannel(s,&h))==0 ) return(-1);
04062
04063     return(writeToChannel(WRITEOUT,s,&h));
04064 }
04065
04066 /*
04067         #] DoPreWrite :
04068         #[ DoProcedure :
04069
04070         We have to read this procedure into a buffer.
04071         The only complications are:
04072         1: we have to seek through the file to do this efficiently
04073            the file operations under VMS cannot do this properly
04074            (unless we use the proper ANSI structs?)
04075            This is the reason why we read whole input files under VMS.
04076         2: what to do when the same name is used twice.
04077         Note that we have to do the reading without substitution of
04078         preprocessor variables.
04079 */
04080
04081 int DoProcedure(UBYTE *s)
04082 {
04083     UBYTE c;
04084     PROCEDURE *p;
04085     LONG i;
04086     if ( ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH )
04087     || ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) ) {
04088         if ( PreSkip((UBYTE *)"procedure",(UBYTE *)"endprocedure",1) ) return(-1);
04089         return(0);
04090     }
04091     // AP.ProcList.num = NumProcedures is incremented inside FromList
04092     p = (PROCEDURE *)FromList(&AP.ProcList);
04093     if ( PreLoad(&(p->p),(UBYTE *)"procedure",(UBYTE *)"endprocedure"
04094         ,1,(char *)"procedure") ) return(-1);
04095
04096     // If the procedure below is not local (loadmode 0) or has been called (loadmode 1), this is a
04097     // nested procedure definition. We must free its allocations when we hit its DoEndprocedure.
04098     // If it seems local (loadmode 2) but is tagged as "mustfree", it is multiply nested and we
04099     // similarly must free its allocations.
04100     if ( NumProcedures >= 2 &&
04101         ( Procedures[NumProcedures-2].loadmode != 2 || Procedures[NumProcedures-2].mustfree ) ) {
04102         p->mustfree = 1;
04103     }
04104     // Otherwise, we are defining a local procedure at "ground level", and we keep the definition
04105     // on the procedure stack until FORM terminates.
04106     else {
04107         p->mustfree = 0;
04108     }
04109
04110     p->loadmode = 2;
04111     s = p->p.buffer + 10;
04112     while ( *s == ' ' || *s == LINEFEED ) s++;
04113     if ( chartype[*s] ) {
04114         MesPrint("@Illegal name for procedure");
04115         return(-1);
04116     }
04117     p->name = s++;
04118     while ( chartype[*s] == 0 || chartype[*s] == 1 ) s++;
04119     c = *s; *s = 0;
04120     p->name = strDup1(p->name,"procedure");
04121     *s = c;
04122 /*
04123     Check for double names
04124 */
```

```
04125     for ( i = NumProcedures-2; i >= 0; i-- ) {
04126         if ( StrCmp(Procedures[i].name,p->name) == 0 ) {
04127             Error1("Multiple occurrence of procedure name ",p->name);
04128         }
04129     }
04130     return(0);
04131 }
04132
04133 /*
04134         #] DoProcedure :
04135         #[ DoPreBreak :
04136 */
04137
04138 int DoPreBreak(UBYTE *s)
04139 {
04140     DUMMYUSE(s);
04141     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
04142     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPESWITCH ) {
04143         if ( AP.PreSwitchLevel <= 0 )
04144             MesPrint("@Break without corresponding Switch");
04145         else MessPreNesting(7);
04146         return(-1);
04147     }
04148     if ( AP.PreSwitchLevel <= 0 ) {
04149         MesPrint("@Break without corresponding Switch");
04150         return(-1);
04151     }
04152     if ( AP.PreSwitchModes[AP.PreSwitchLevel] == EXECUTINGPRESWITCH )
04153         AP.PreSwitchModes[AP.PreSwitchLevel] = SEARCHINGPREENDSWITCH;
04154     return(0);
04155 }
04156
04157 /*
04158         #] DoPreBreak :
04159         #[ DoPreCase :
04160 */
04161
04162 int DoPreCase(UBYTE *s)
04163 {
04164     UBYTE *t;
04165     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
04166     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPESWITCH ) {
04167         if ( AP.PreSwitchLevel <= 0 )
04168             MesPrint("@Case without corresponding Switch");
04169         else MessPreNesting(8);
04170         return(-1);
04171     }
04172     if ( AP.PreSwitchLevel <= 0 ) {
04173         MesPrint("@Case without corresponding Switch");
04174         return(-1);
04175     }
04176     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != SEARCHINGPRECASE ) return(0);
04177
04178     SKIPBLANKS(s)
04179     t = s;
04180     while ( *s ) { if ( *s == '\\' ) s++; s++; }
04181     while ( s > t && ( s[-1] == ' ' || s[-1] == '\t' ) && s[-2] != '\\' ) {
04182         if ( s[-2] == '\\' ) s--;
04183         s--;
04184     }
04185     if ( *t == '"' && s > t+1 && s[-1] == '"' && s[-2] != '\\' ) {
04186         t++; s--; *s = 0;
04187     }
04188     else *s = 0;
04189     s = AP.PreSwitchStrings[AP.PreSwitchLevel];
04190     while ( *t == *s && *t ) { s++; t++; }
04191     if ( *t || *s ) return(0);  /* case did not match */
04192     AP.PreSwitchModes[AP.PreSwitchLevel] = EXECUTINGPRESWITCH;
04193     return(0);
04194 }
04195
04196 /*
04197         #] DoPreCase :
04198         #[ DoPreDefault :
04199 */
04200
04201 int DoPreDefault(UBYTE *s)
04202 {
04203     DUMMYUSE(s);
04204     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
04205     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPESWITCH ) {
04206         if ( AP.PreSwitchLevel <= 0 )
04207             MesPrint("@Default without corresponding Switch");
04208         else MessPreNesting(9);
04209         return(-1);
04210     }
04211     if ( AP.PreSwitchLevel <= 0 ) {
```

```
04212         MesPrint("@Default without corresponding Switch");
04213         return(-1);
04214     }
04215     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != SEARCHINGPRECASE ) return(0);
04216     AP.PreSwitchModes[AP.PreSwitchLevel] = EXECUTINGPRESWITCH;
04217     return(0);
04218 }
04219
04220 /*
04221         #] DoPreDefault :
04222         #[ DoPreEndSwitch :
04223 */
04224
04225 int DoPreEndSwitch(UBYTE *s)
04226 {
04227     DUMMYUSE(s);
04228     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
04229     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPESWITCH ) {
04230         if ( AP.PreSwitchLevel <= 0 )
04231              MesPrint("@EndSwitch without corresponding Switch");
04232         else MessPreNesting(10);
04233         return(-1);
04234     }
04235     AP.NumPreTypes--;
04236     if ( AP.PreSwitchLevel <= 0 ) {
04237         MesPrint("@EndSwitch without corresponding Switch");
04238         return(-1);
04239     }
04240     M_free(AP.PreSwitchStrings[AP.PreSwitchLevel--],"pre switch string");
04241     return(0);
04242 }
04243
04244 /*
04245         #] DoPreEndSwitch :
04246         #[ DoPreSwitch :
04247
04248         There should be a string after this.
04249         We have to store it somewhere.
04250 */
04251
04252 int DoPreSwitch(UBYTE *s)
04253 {
04254     UBYTE *t, *switchstring, **newstrings;
04255     int newnum, i, *newmodes;
04256     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
04257     SKIPBLANKS(s)
04258     t = s;
04259     while ( *s ) { if ( *s == '\\' ) s++; s++; }
04260     while ( s > t && ( s[-1] == ' ' || s[-1] == '\t' ) && s[-2] != '\\' ) {
04261         if ( s[-2] == '\\' ) s--;
04262         s--;
04263     }
04264     if ( *t == '"' && s > t+1 && s[-1] == '"' && s[-2] != '\\' ) {
04265         t++; s--; *s = 0;
04266     }
04267     else *s = 0;
04268     switchstring = (UBYTE *)Malloc1((s-t)+1,"case string");
04269     s = switchstring;
04270     while ( *t ) {
04271         if ( *t == '\\' ) t++;
04272         *s++ = *t++;
04273     }
04274     *s = 0;
04275     if ( AP.PreSwitchLevel >= AP.NumPreSwitchStrings ) {
04276         newnum = 2*AP.NumPreSwitchStrings;
04277         newstrings = (UBYTE **)Malloc1(sizeof(UBYTE *)*(newnum+1),"case strings");
04278         newmodes   = (int *)Malloc1(sizeof(int)*(newnum+1),"case strings");
04279         for ( i = 0; i < AP.NumPreSwitchStrings; i++ )
04280             newstrings[i] = AP.PreSwitchStrings[i];
04281         M_free(AP.PreSwitchStrings,"AP.PreSwitchStrings");
04282         for ( i = 0; i <= AP.NumPreSwitchStrings; i++ )
04283             newmodes[i] = AP.PreSwitchModes[i];
04284         M_free(AP.PreSwitchModes,"AP.PreSwitchModes");
04285         AP.PreSwitchStrings = newstrings;
04286         AP.PreSwitchModes   = newmodes;
04287         AP.NumPreSwitchStrings = newnum;
04288     }
04289     AP.PreSwitchStrings[++AP.PreSwitchLevel] = switchstring;
04290     if ( ( AP.PreSwitchLevel > 1 )
04291      && ( AP.PreSwitchModes[AP.PreSwitchLevel-1] != EXECUTINGPRESWITCH ) )
04292         AP.PreSwitchModes[AP.PreSwitchLevel] = SEARCHINGPREENDSWITCH;
04293     else
04294         AP.PreSwitchModes[AP.PreSwitchLevel] = SEARCHINGPRECASE;
04295     AddToPreTypes(PRETYPESWITCH);
04296     return(0);
04297 }
04298
```

```
04299 /*
04300         #] DoPreSwitch :
04301         #[ DoPreShow :
04302 
04303         Print the contents of the preprocessor variables
04304 */
04305 
04306 int DoPreShow(UBYTE *s)
04307 {
04308     int i;
04309     UBYTE *name, c;
04310     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
04311     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
04312     while ( *s == ' ' || *s == '\t' ) s++;
04313     if ( *s == 0 ) {
04314         MesPrint("%#The preprocessor variables:");
04315         for ( i = 0; i < NumPre; i++ ) {
04316             MesPrint("%d: %s = \"%s\"",i,PreVar[i].name,PreVar[i].value);
04317         }
04318     }
04319     else {
04320         while ( *s ) {
04321             name = s; while ( *s && *s != ' ' && *s != '\t' && *s != ',' ) s++;
04322             c = *s; *s = 0;
04323             for ( i = 0; i < NumPre; i++ ) {
04324                 if ( StrCmp(PreVar[i].name,name) == 0 )
04325                     MesPrint("%d: %s = \"%s\"",i,PreVar[i].name,PreVar[i].value);
04326             }
04327             *s = c;
04328             while ( *s == ' ' || *s == '\t' ) s++;
04329         }
04330     }
04331     return(0);
04332 }
04333 
04334 /*
04335         #] DoPreShow :
04336         #[ DoSystem :
04337 */
04338 
04339 /*
04340  * A macro for translating the contents of `x' into a string after expanding.
04341  */
04342 #define STRINGIFY(x)  STRINGIFY__(x)
04343 #define STRINGIFY__(x) #x
04344 
04345 int DoSystem(UBYTE *s)
04346 {
04347     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
04348     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
04349     if ( AP.preError ) return(0);
04350 #ifdef WITHSYSTEM
04351     FLUSHCONSOLE;
04352     while ( *s == ' ' || *s == '\t' ) s++;
04353     if ( *s == '-' && s[1] == 'e' ) {
04354         LONG err;
04355         UBYTE str[24];
04356         s += 2;
04357         if ( *s != ' ' ) {
04358             MesPrint("@Syntax error in #system command.");
04359             return(-1);
04360         }
04361         while ( *s == ' ' || *s == '\t' ) s++;
04362         err = system((char *)s);
04363         NumToStr(str,err);
04364         PutPreVar((UBYTE *)"SYSTEMERROR_",str,0,1);
04365     }
04366     else if ( system((char *)s) ) {
04367         MesPrint("@System call returned with error condition");
04368         Terminate(-1);
04369     }
04370     return(0);
04371 #else
04372     Error0("External programs not implemented on this computer/system");
04373     return(-1);
04374 #endif
04375 }
04376 
04377 /*
04378         #] DoSystem :
04379         #[ PreLoad :
04380 
04381         Loads a loop or procedure into a special buffer.
04382         Note: The current instruction is already in the preStart buffer
04383 */
04384 
04385 int PreLoad(PRELOAD *p, UBYTE *start, UBYTE *stop, int mode, char *message)
```

```
04386 {
04387     UBYTE *s, *t, *top, *newbuffer, c;
04388     LONG i, ppsize, linenum = AC.CurrentStream->linenumber;
04389     int size1, size2, level, com=0, last=1, strng = 0;
04390     p->size = AP.pSize;
04391     p->buffer = (UBYTE *)Malloc1(p->size+1,message);
04392     top = p->buffer + p->size - 2;
04393     t = p->buffer; *t++ = '#';
04394     s = start; size1 = size2 = 0;
04395     while ( *s ) { s++; size1++; }
04396     s = stop; while ( *s ) { s++; size2++; }
04397     s = AP.preStart; while ( *s ) *t++ = *s++; *t++ = LINEFEED;
04398     level = 1;
04399     i = 100;
04400     for (;;) {
04401         c = GetInput();
04402         if ( c == ENDOFINPUT ) {
04403             MesPrint("@Missing %#%s, Should match line %l",stop,linenum);
04404             return(-1);
04405         }
04406         if ( c == AP.ComChar && last == 1 ) com = 1;
04407         if ( c == LINEFEED ) { last = 1; com = 0; }
04408         else last = 0;
04409
04410         if ( ( c == '"' ) && ( com == 0 ) ) { strng ^= 1; }
04411
04412         if ( ( c == '#' ) && ( com == 0 ) ) i = 0;
04413         else i++;
04414
04415         if ( t >= top ) {
04416             ppsize = t - p->buffer;
04417             p->size *= 2;
04418             newbuffer = (UBYTE *)Malloc1(p->size,message);
04419             t = newbuffer; s = p->buffer;
04420             while ( --ppsize >= 0 ) *t++ = *s++;
04421             M_free(p->buffer,"loading do loop");
04422             p->buffer = newbuffer;
04423             top = p->buffer + p->size - 2;
04424         }
04425         *t++ = c;
04426         if ( strng == 0 ) {
04427           if ( ( i == size2 ) && ( com == 0 ) ) {
04428             *t = 0;
04429             if ( StrICmp(t-size2,(UBYTE *)(stop)) == 0 ) {
04430                 while ( ( c = GetInput() ) != LINEFEED && c != ENDOFINPUT ) {}
04431                 level--;
04432                 if ( level <= 0 ) break;
04433                 if ( c == ENDOFINPUT ) Error1("Missing #",stop);
04434                 *t++ = LINEFEED; *t = 0; last = 1;
04435             }
04436           }
04437           if ( ( i == size1 ) && mode && ( com == 0 ) ) {
04438             *t = 0;
04439             if ( StrICmp(t-size1,(UBYTE *)(start)) == 0 ) {
04440 /*
04441                 while ( ( c = GetInput() ) != LINEFEED && c != ENDOFINPUT ) {}
04442                 if ( c == ENDOFINPUT ) Error1("Missing #",stop);
04443 */
04444                 level++;
04445             }
04446           }
04447           if ( i == 1 && t[-2] == LINEFEED ) {
04448             if ( c == '-' )      AC.NoShowInput = 1;
04449             else if ( c == '+' ) AC.NoShowInput = 0;
04450           }
04451         }
04452     }
04453     *t++ = LINEFEED;
04454     *t = 0;
04455     return(0);
04456 }
04457
04458 /*
04459         #] PreLoad :
04460         #[ PreSkip :
04461
04462         Skips a loop or procedure.
04463         Note: The current instruction is already in the preStart buffer
04464 */
04465
04466 #define SKIPBUFSIZE 20
04467
04468 int PreSkip(UBYTE *start, UBYTE *stop, int mode)
04469 {
04470     UBYTE *s, *t, buffer[SKIPBUFSIZE+2], c;
04471     LONG i, linenum = AC.CurrentStream->linenumber;
04472     int size1, size2, level, com=0, last=1;
```

```
04473
04474     t = buffer; *t++ = '#';
04475     s = start; size1 = size2 = 0;
04476     while ( *s ) { s++; size1++; }
04477     s = stop; while ( *s ) { s++; size2++; }
04478     level = 1;
04479     i = 0;
04480     for (;;) {
04481         c = GetInput();
04482         if ( c == ENDOFINPUT ) {
04483             MesPrint("@Missing %#%s, Should match line %l",stop,linenum);
04484             return(-1);
04485         }
04486         if ( c == AP.ComChar && last == 1 ) com = 1;
04487         if ( c == LINEFEED ) { last = 1; com = 0; i = 0; t = buffer; }
04488         else last = 0;
04489         if ( ( c == '#' ) && ( com == 0 ) ) { i = 0; t = buffer; }
04490         else i++;
04491
04492         if ( i < SKIPBUFSIZE ) *t++ = c;
04493         if ( ( i == size2 ) && ( com == 0 ) ) {
04494             *t = 0;
04495             if ( StrICmp(t-size2,(UBYTE *)(stop)) == 0 ) {
04496                 while ( ( c = GetInput() ) != LINEFEED && c != ENDOFINPUT ) {}
04497                 level--;
04498                 if ( level <= 0 ) {
04499                     pushbackchar = LINEFEED;
04500                     break;
04501                 }
04502                 if ( c == ENDOFINPUT ) Error1("Missing #",stop);
04503                 i = 0; t = buffer;
04504             }
04505         }
04506         if ( ( i == size1 ) && mode && ( com == 0 ) ) {
04507             *t = 0;
04508             if ( StrICmp(t-size1,(UBYTE *)(start)) == 0 ) {
04509                 while ( ( c = GetInput() ) != LINEFEED && c != ENDOFINPUT ) {}
04510                 level++;
04511                 i = 0; t = buffer;
04512             }
04513         }
04514     }
04515     return(0);
04516 }
04517
04518 /*
04519         #] PreSkip :
04520         #[ StartPrepro :
04521 */
04522
04523 void StartPrepro(void)
04524 {
04525     int **ppp;
04526     AP.MaxPreIfLevel = 2;
04527     ppp = &AP.PreIfStack;
04528     if ( DoubleList((void ***)ppp,&AP.MaxPreIfLevel,sizeof(int),
04529             "PreIfLevels") ) Terminate(-1);
04530     AP.PreIfLevel = 0; AP.PreIfStack[0] = EXECUTINGIF;
04531
04532     AP.NumPreSwitchStrings = 10;
04533     AP.PreSwitchStrings = (UBYTE **)Malloc1(sizeof(UBYTE *)*
04534                                 (AP.NumPreSwitchStrings+1),"case strings");
04535     AP.PreSwitchModes   = (int *)Malloc1(sizeof(int)*
04536                                 (AP.NumPreSwitchStrings+1),"case strings");
04537     AP.PreSwitchModes[0] = EXECUTINGPRESWITCH;
04538     AP.PreSwitchLevel = 0;
04539 }
04540
04541 /*
04542         #] StartPrepro :
04543         #[ EvalPreIf :
04544
04545         Evaluates the condition in an if instruction.
04546         The return value is EXECUTINGIF if the condition is true.
04547         If it is false the returnvalue is LOOKINGFORELSE.
04548         An error gives a return value of -1
04549 */
04550
04551 int EvalPreIf(UBYTE *s)
04552 {
04553     UBYTE *t, *u;
04554     int val;
04555     t = s;
04556     while ( *t ) t++;
04557     *t++ = ')';
04558     *t = 0;
04559     if ( ( u = PreIfEval(s,&val) ) == 0 ) return(-1);
```

```
04560     if ( u < t ) {
04561         MesPrint("@Unmatched parentheses in condition");
04562         return(-1);
04563     }
04564     if ( val ) return(EXECUTINGIF);
04565     else       return(LOOKINGFORELSE);
04566 }
04567
04568 /*
04569         #] EvalPreIf :
04570         #[ PreIfEval :
04571
04572         Used for recursions in the evaluation of a preprocessor if-condition.
04573         It determines whether the contents of () is true or false
04574         (or in error).
04575         The return value is the address of the first character after the
04576         closing parenthesis or null if there is an error.
04577         In value we find true(1) or false(0)
04578         We enter after the opening parenthesis.
04579         There are levels:
04580             0: orlevel:  a || b
04581             1: andlevel: a && b
04582             2: eqlevel:  a == b or a != b or a = b
04583             3: cmplevel: a > b or a >= b or a < b or a <= b or a >~ b etc
04584 */
04585
04586 UBYTE *PreIfEval(UBYTE *s, int *value)
04587 {
04588     int orlevel = 0, andlevel = 0, eqlevel = 0, cmplevel = 0;
04589     int type, val;
04590     LONG val2;
04591     int ortype, orval, cmptype, cmpval, eqtype, eqval, andtype, andval;
04592     UBYTE *t, *eqt, *cmpt, c;
04593     int eqop, cmpop;
04594     ortype = orval = cmptype = cmpval = eqtype = eqval = andtype = andval = 0;
04595     eqop = cmpop = 0;
04596     eqt = cmpt = 0;
04597     *value = 0;
04598     while ( *s != ')' ) {
04599         while ( *s == ' ' || *s == '\t' || *s == '\n' || *s == '\r' ) s++;
04600         t = s;
04601         s = pParseObject(s,&type,&val2);
04602         if ( s == 0 ) return(0);
04603         val = val2;
04604         c = *s;
04605         *s++ = 0;    /* in case the object is a string without " */
04606         while ( c == ' ' || c == '\t' || c == '\n' || c == '\r' ) {
04607             c = *s; *s++ = 0;
04608         }
04609         if ( *t == '"' ) t++;
04610         switch(c) {
04611             case '|':
04612                 if ( *s != '|' ) goto illoper;
04613                 s++;
04614                 /* fall through */
04615             case ')':
04616                 if ( cmplevel ) {
04617                     if ( type == 0 || cmptype == 0 ) goto illobject;
04618                     val = PreCmp(type,val,t,cmptype,cmpval,cmpt,cmpop);
04619                     type = 0;
04620                     cmplevel = 0;
04621                 }
04622                 if ( eqlevel ) {
04623                     val = PreEq(type,val,t,eqtype,eqval,eqt,eqop);
04624                     type = 0;
04625                     eqlevel = 0;
04626                 }
04627                 if ( andlevel ) {
04628                     if ( andtype != 0 || type != 0 ) goto illobject;
04629                     val &= andval;
04630                     andlevel = 0;
04631                 }
04632                 if ( orlevel ) {
04633                     if ( ortype != 0 || type != 0 ) goto illobject;
04634                     val |= orval;
04635                 }
04636                 if ( c == ')' ) {
04637                     *value = val;
04638                     return(s);
04639                 }
04640                 orlevel = 1;
04641                 orval = val;
04642                 ortype = type;
04643                 break;
04644             case '&':
04645                 if ( *s != '&' ) goto illoper;
04646                 s++;
```

```
04647                 if ( cmplevel ) {
04648                     if ( type == 0 || cmptype == 0 ) goto illobject;
04649                     val = PreCmp(type,val,t,cmptype,cmpval,cmpt,cmpop);
04650                     type = 0;
04651                     cmplevel = 0;
04652                 }
04653                 if ( eqlevel ) {
04654                     val = PreEq(type,val,t,eqtype,eqval,eqt,eqop);
04655                     type = 0;
04656                     eqlevel = 0;
04657                 }
04658                 if ( andlevel ) {
04659                     if ( andtype != 0 || type != 0 ) goto illobject;
04660                     val &= andval;
04661                 }
04662                 andlevel = 1;
04663                 andval = val;
04664                 andtype = type;
04665                 break;
04666             case '!':
04667             case '=':
04668                 if ( eqlevel ) goto illorder;
04669                 if ( cmplevel ) {
04670                     if ( type == 0 || cmptype == 0 ) goto illobject;
04671                     val = PreCmp(type,val,t,cmptype,cmpval,cmpt,cmpop);
04672                     type = 0;
04673                     cmplevel = 0;
04674                 }
04675                 if ( c == '!' && *s != '=' ) goto illoper;
04676                 if ( *s == '=' ) s++;
04677                 if ( c == '!' ) eqop = 1;
04678                 else eqop = 0;
04679                 eqlevel = 1; eqt = t; eqval = val; eqtype = type;
04680                 break;
04681             case '>':
04682             case '<':
04683                 if ( cmplevel ) goto illorder;
04684                 if ( c == '<' ) cmpop = -1;
04685                 else            cmpop = 1;
04686                 cmplevel = 1; cmpt = t; cmpval = val; cmptype = type;
04687                 if ( *s == '=' ) {
04688                     s++;
04689                     if ( *s == '~' ) { s++; cmpop *= 4; }
04690                     else cmpop *= 2;
04691                 }
04692                 else if ( *s == '~' ) { s++; cmpop *= 3; }
04693                 break;
04694             default:
04695                 goto illoper;
04696         }
04697     }
04698     return(s);
04699 illorder:
04700     MesPrint("@illegal order of operators");
04701     return(0);
04702 illobject:
04703     MesPrint("@illegal object for this operator");
04704     return(0);
04705 illoper:
04706     MesPrint("@illegal operator");
04707     return(0);
04708 }
04709
04710 /*
04711         #] PreIfEval :
04712         #[ PreCmp :
04713 */
04714
04715 int PreCmp(int type, int val, UBYTE *t, int type2, int val2, UBYTE *t2, int cmpop)
04716 {
04717     if ( type == 2 || type2 == 2 || cmpop < -2 || cmpop > 2 ) {
04718         if ( cmpop < 0 && cmpop > -3 ) cmpop -= 2;
04719         if ( cmpop > 0 && cmpop <  3 ) cmpop += 2;
04720             if ( cmpop ==  3 ) val = StrCmp(t2,t) >  0;
04721         else if ( cmpop ==  4 ) val = StrCmp(t2,t) >= 0;
04722         else if ( cmpop == -3 ) val = StrCmp(t2,t) <  0;
04723         else if ( cmpop == -4 ) val = StrCmp(t2,t) <= 0;
04724     }
04725     else {
04726             if ( cmpop ==  1 ) val = ( val2 >  val );
04727         else if ( cmpop ==  2 ) val = ( val2 >= val );
04728         else if ( cmpop == -1 ) val = ( val2 <  val );
04729         else if ( cmpop == -2 ) val = ( val2 <= val );
04730     }
04731     return(val);
04732 }
04733
```

```
04734 /*
04735         #] PreCmp :
04736         #[ PreEq :
04737 */
04738
04739 int PreEq(int type, int val, UBYTE *t, int type2, int val2, UBYTE *t2, int eqop)
04740 {
04741     UBYTE str[20];
04742     if ( type == 2 || type2 == 2 ) {
04743         if ( type  != 2 ) { NumToStr(str,val ); t  = str; }
04744         if ( type2 != 2 ) { NumToStr(str,val2); t2 = str; }
04745         if ( eqop == 1 ) val = StrCmp(t,t2) != 0;
04746         else             val = StrCmp(t,t2) == 0;
04747     }
04748     else {
04749         if ( eqop ) val = val != val2;
04750         else        val = val == val2;
04751     }
04752     return(val);
04753 }
04754
04755 /*
04756         #] PreEq :
04757         #[ pParseObject :
04758
04759         Parses a preprocessor object. We can have:
04760         1: a number  (type = 1)
04761         2: a string  (type = 2)
04762         3: an expression between parentheses (type = 0)
04763         4: a special function (type = 3)
04764         If the object is not a number, an expression or a special operator
04765         we try to interpret it as a string.
04766 */
04767
04768 UBYTE *pParseObject(UBYTE *s, int *type, LONG *val2)
04769 {
04770     UBYTE *t, c;
04771     int sign, val = 0;
04772     LONG x;
04773     while ( *s == ' ' || *s == '\t' ) s++;
04774     if ( *s == '(' ) {
04775         s++;
04776         while ( *s == ' ' || *s == '\t' || *s == '\n' || *s == '\r' ) s++;
04777         s = PreIfEval(s,&val);
04778         *type = 0;
04779         *val2 = val;
04780         return(s);
04781     }
04782     else if ( *s == '$' && s[1] == '(' ) {
04783         s += 2;
04784         while ( *s == ' ' || *s == '\t' || *s == '\n' || *s == '\r' ) s++;
04785         s = PreIfDollarEval(s,&val);
04786         *type = 0; *val2 = val;
04787         return(s);
04788     }
04789     if ( *s == 0 ) {
04790 illend:
04791         MesPrint("@illegal end of condition");
04792         return(0);
04793     }
04794     if ( *s == '"' ) {
04795         s++;
04796         while ( *s && *s != '"' ) {
04797             if ( *s == '\\' ) s++;
04798             s++;
04799         }
04800         if ( *s == 0 ) goto illend;
04801         else *s = 0;
04802         *type = 2;
04803         s++;
04804
04805         while ( *s == ' ' || *s == '\t' || *s == '\n' || *s == '\r' ) s++;
04806
04807         return(s);
04808     }
04809     t = s; sign = 1; x = 0;
04810     if ( chartype[*t] == 0 ) {  /* Special operators and strings without "" */
04811         do { t++; } while ( chartype[*t] <= 1 );
04812         if ( *t == '(' ) {
04813             WORD ttype;
04814             c = *t; *t = 0;
04815             if ( StrICmp(s,(UBYTE *)"termsin") == 0 ) {
04816                 UBYTE *tt;
04817                 WORD numdol, numexp;
04818                 ttype = 0;
04819 together:
04820                 *t++ = c;
```

```
04821                 while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
04822                 if ( *t == '$' ) {
04823                     t++; tt = t; while (chartype[*tt] <= 1 ) tt++;
04824                     c = *tt; *tt = 0;
04825                     if ( ( numdol = GetDollar(t) ) > 0 ) {
04826                         *tt = c;
04827                         if ( ttype == 1 ) {
04828                             x = SizeOfDollar(numdol);
04829                         }
04830                         else {
04831                             x = TermsInDollar(numdol);
04832                         }
04833                     }
04834                     else {
04835                         MesPrint("@$%s has not (yet) been defined",t);
04836                         *tt = c;
04837                         Terminate(-1);
04838                     }
04839                 }
04840                 else {
04841                     tt = SkipAName(t);
04842                     c = *tt; *tt = 0;
04843                     if ( GetName(AC.exprnames,t,&numexp,NOAUTO) == NAMENOTFOUND ) {
04844                         MesPrint("@%s has not (yet) been defined",t);
04845                         *tt = c;
04846                         Terminate(-1);
04847                     }
04848                     else {
04849                         *tt = c;
04850                         if ( ttype == 1 ) {
04851                             x = SizeOfExpression(numexp);
04852                         }
04853                         else {
04854                             x = TermsInExpression(numexp);
04855                         }
04856                     }
04857                 }
04858                 while ( *tt == ' ' || *tt == '\t'
04859                          || *tt == '\n' || *tt == '\r' ) tt++;
04860                 if ( *tt != ')' ) {
04861                     MesPrint("@Improper use of terms($var) or terms(expr)");
04862                     Terminate(-1);
04863                 }
04864                 *type = 3;
04865                 s = tt+1;
04866                 *val2 = x;
04867                 return(s);
04868             }
04869             else if ( StrICmp(s,(UBYTE *)"sizeof") == 0 ) {
04870                 ttype = 1;
04871                 goto together;
04872             }
04873             else if ( StrICmp(s,(UBYTE *)"exists") == 0 ) {
04874                 UBYTE *tt;
04875                 WORD numdol, numexp;
04876                 *t++ = c;
04877                 while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
04878                 if ( *t == '$' ) {
04879                     t++; tt = t; while (chartype[*tt] <= 1 ) tt++;
04880                     c = *tt; *tt = 0;
04881                     if ( ( numdol = GetDollar(t) ) >= 0 ) { x = 1; }
04882                     else                                  { x = 0; }
04883                     *tt = c;
04884                 }
04885                 else if ( *t == '"' ) { /* see whether a file exists */
04886 /*                  UBYTE *name, *oldname; */
04887                     t++; tt = t;
04888                     for (;;) {
04889                         if ( *tt == '\\' ) tt++;
04890                         else if ( *tt == '"' ) break;
04891                         tt++;
04892                     }
04893                     c = *tt; *tt = 0;
04894 /*
04895                     Try to open the file. If possible, return 1.
04896                     Afterwards close it.
04897                     We do have to run through the FORMPATH. Hence we use LocateFile.
04898                     This routine may change the name to the full name.
04899
04900                     oldname = name = strDup1(t,"name in exists");
04901                     x = LocateFile(&name,-1);
04902 */
04903                     x = OpenFile((char *)t);
04904                     if ( x >= 0 ) {
04905                         CloseFile(x);
04906                         x = 1;
04907 /*
```

```
04908                       if ( name != oldname ) M_free(name,"name from LocateFile");
04909 */
04910                   }
04911                   else x = 0;
04912 /*
04913                   M_free(oldname,"name in exists");
04914 */
04915                   *tt++ = c;
04916               }
04917               else {
04918                   tt = SkipAName(t);
04919                   c = *tt; *tt = 0;
04920                   if ( GetName(AC.exprnames,t,&numexp,NOAUTO) == NAMENOTFOUND ) { x = 0; }
04921                   else { x = 1; }
04922                   *tt = c;
04923               }
04924               while ( *tt == ' ' || *tt == '\t'
04925                       || *tt == '\n' || *tt == '\r' ) tt++;
04926               if ( *tt != ')' ) {
04927                   MesPrint("@Improper use of exists($var) or exists(expr)");
04928                   Terminate(-1);
04929               }
04930               *type = 3;
04931               s = tt+1;
04932               *val2 = x;
04933               return(s);
04934           }
04935           else if ( StrICmp(s,(UBYTE *)"isnumerical") == 0 ) {
04936               GETIDENTITY
04937               UBYTE *tt;
04938               WORD numdol, numexp;
04939               *t++ = c;
04940               while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
04941               if ( *t == '$' ) {
04942                   t++; tt = t; while (chartype[*tt] <= 1 ) tt++;
04943                   c = *tt; *tt = 0;
04944                   if ( ( numdol = GetDollar(t) ) < 0 ) {
04945                       MesPrint("@$ variable in isnumerical(%s) does not exist",t);
04946                       Terminate(-1);
04947                   }
04948                   x = DolToLong(BHEAD numdol);
04949                   if ( AN.ErrorInDollar ) {
04950                       DOLLARS d = Dollars + numdol;
04951                       x = 0;
04952                       if ( d->type == DOLNUMBER || d->type == DOLTERMS ) {
04953                           if ( d->where[0] == 0 ) x = 1;
04954                           else if ( d->where[d->where[0]] == 0 ) {
04955                               if ( ABS(d->where[d->where[0]-1]) == d->where[0]-1 )
04956                                   x = 1;
04957                           }
04958                       }
04959                   }
04960                   else x = 1;
04961                   *tt = c;
04962               }
04963               else {
04964                   tt = SkipAName(t);
04965                   c = *tt; *tt = 0;
04966                   if ( GetName(AC.exprnames,t,&numexp,NOAUTO) == NAMENOTFOUND ) {
04967                       MesPrint("@expression in isnumerical(%s) does not exist",t);
04968                       Terminate(-1);
04969                   }
04970                   x = TermsInExpression(numexp);
04971                   if ( x != 1 ) x = 0;
04972                   else {
04973                       WORD *term = AT.WorkPointer;
04974                       if ( GetFirstTerm(term,numexp,1) < 0 ) {
04975                           MesPrint("@error reading expression in isnumerical(%s)",t);
04976                           Terminate(-1);
04977                       }
04978                       if ( *term == ABS(term[*term-1])+1 ) x = 1;
04979                       else                                 x = 0;
04980                   }
04981                   *tt = c;
04982               }
04983               while ( *tt == ' ' || *tt == '\t'
04984                       || *tt == '\n' || *tt == '\r' ) tt++;
04985               if ( *tt != ')' ) {
04986                   MesPrint("@Improper use of isnumerical($var) or numerical(expr)");
04987                   Terminate(-1);
04988               }
04989               *type = 3;
04990               s = tt+1;
04991               *val2 = x;
04992               return(s);
04993           }
04994           else if ( StrICmp(s,(UBYTE *)("maxpowerof")) == 0 ) {
```

```
04995                 UBYTE *tt;
04996                 WORD numsym;
04997                 int stype;
04998                 *t++ = c;
04999                 while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
05000                 tt = SkipAName(t);
05001                 c = *tt; *tt = 0;
05002                 if ( ( stype = GetName(AC.varnames,t,&numsym,NOAUTO) ) == NAMENOTFOUND ) {
05003                     MesPrint("@%s has not (yet) been defined",t);
05004                     *tt = c;
05005                     Terminate(-1);
05006                 }
05007                 else if ( stype != CSYMBOL ) {
05008                     MesPrint("@%s should be a symbol",t);
05009                     *tt = c;
05010                     Terminate(-1);
05011                 }
05012                 else {
05013                     *tt = c;
05014                     x = symbols[numsym].maxpower;
05015                 }
05016                 while ( *tt == ' ' || *tt == '\t'
05017                         || *tt == '\n' || *tt == '\r' ) tt++;
05018                 if ( *tt != ')' ) {
05019                     MesPrint("@Improper use of maxpowerof(symbol)");
05020                     Terminate(-1);
05021                 }
05022                 *type = 3;
05023                 s = tt+1;
05024                 *val2 = x;
05025                 return(s);
05026             }
05027             else if ( StrICmp(s,(UBYTE *)("minpowerof")) == 0 ) {
05028                 UBYTE *tt;
05029                 WORD numsym;
05030                 int stype;
05031                 *t++ = c;
05032                 while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
05033                 tt = SkipAName(t);
05034                 c = *tt; *tt = 0;
05035                 if ( ( stype = GetName(AC.varnames,t,&numsym,NOAUTO) ) == NAMENOTFOUND ) {
05036                     MesPrint("@%s has not (yet) been defined",t);
05037                     *tt = c;
05038                     Terminate(-1);
05039                 }
05040                 else if ( stype != CSYMBOL ) {
05041                     MesPrint("@%s should be a symbol",t);
05042                     *tt = c;
05043                     Terminate(-1);
05044                 }
05045                 else {
05046                     *tt = c;
05047                     x = symbols[numsym].minpower;
05048                 }
05049                 while ( *tt == ' ' || *tt == '\t'
05050                         || *tt == '\n' || *tt == '\r' ) tt++;
05051                 if ( *tt != ')' ) {
05052                     MesPrint("@Improper use of minpowerof(symbol)");
05053                     Terminate(-1);
05054                 }
05055                 *type = 3;
05056                 s = tt+1;
05057                 *val2 = x;
05058                 return(s);
05059             }
05060             else if ( StrICmp(s,(UBYTE *)"isfactorized") == 0 ) {
05061                 UBYTE *tt;
05062                 WORD numdol, numexp;
05063                 *t++ = c;
05064                 while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
05065                 if ( *t == '$' ) {
05066                     t++; tt = t; while (chartype[*tt] <= 1 ) tt++;
05067                     c = *tt; *tt = 0;
05068                     if ( ( numdol = GetDollar(t) ) > 0 ) {
05069                         if ( Dollars[numdol].factors != 0 ) x = 1;
05070                         else x = 0;
05071                     }
05072                     else {
05073                         MesPrint("@ %s should be the name of an expression or a $ variable",t-1);
05074                         Terminate(-1);
05075                     }
05076                     *tt = c;
05077                 }
05078                 else {
05079                     tt = SkipAName(t);
05080                     c = *tt; *tt = 0;
05081                     if ( GetName(AC.exprnames,t,&numexp,NOAUTO) == NAMENOTFOUND ) {
```

```
05082                       MesPrint("@ %s should be the name of an expression or a $ variable",t);
05083                       Terminate(-1);
05084                   }
05085                   else {
05086                       if ( ( Expressions[numexp].vflags & ISFACTORIZED ) != 0 ) x = 1;
05087                       else x = 0;
05088                   }
05089                   *tt = c;
05090               }
05091               while ( *tt == ' ' || *tt == '\t'
05092                       || *tt == '\n' || *tt == '\r' ) tt++;
05093               if ( *tt != ')' ) {
05094                   MesPrint("@Improper use of isfactorized($var) or isfactorized(expr)");
05095                   Terminate(-1);
05096               }
05097               *type = 3;
05098               s = tt+1;
05099               *val2 = x;
05100               return(s);
05101           }
05102           else if ( StrICmp(s,(UBYTE *)"isdefined") == 0 ) {
05103               UBYTE *tt;
05104               *t++ = c;
05105               while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
05106               tt = SkipAName(t);
05107               c = *tt; *tt = 0;
05108               if ( GetPreVar(t,WITHOUTERROR) != 0 ) x = 1;
05109               else x = 0;
05110               *tt = c;
05111               while ( *tt == ' ' || *tt == '\t'
05112                       || *tt == '\n' || *tt == '\r' ) tt++;
05113               if ( *tt != ')' ) {
05114                   MesPrint("@Improper use of isdefined(var)");
05115                   Terminate(-1);
05116               }
05117               *type = 3;
05118               s = tt+1;
05119               *val2 = x;
05120               return(s);
05121           }
05122           else if ( StrICmp(s,(UBYTE *)"flag") == 0 ) {
05123               UBYTE *tt;
05124               WORD x = 0, numexp;
05125               {
05126               *t++ = c;
05127               while ( *t == ' ' || *t == '\t' ) t++;
05128               if ( FG.cTable[*t] != 1 ) goto flagerror;
05129               while ( FG.cTable[*t] == 1 ) x = 10*x + (*t++ - '0');
05130               if ( x < 1 || x > BITSINWORD ) {
05131                   MesPrint("@Illegal number %d for flag in flag condition",x);
05132                   goto flagerror;
05133               }
05134               while ( *t == ' ' || *t == '\t' ) t++;
05135               if ( *t != ',' ) goto flagerror;
05136               t++;
05137               while ( *t == ' ' || *t == '\t' ) t++;
05138               tt = SkipAName(t);
05139               c = *tt; *tt = 0;
05140               if ( GetName(AC.exprnames,t,&numexp,NOAUTO) == NAMENOTFOUND ) {
05141                   MesPrint("@ %s should be the name of an expression",t);
05142                   goto flagerror;
05143               }
05144               *tt = c;
05145               while ( *t == ' ' || *t == '\t' ) t++;
05146               if ( *tt != ')' ) {
05147 flagerror:
05148                   MesPrint("@Improper use of flag(num,expr)");
05149                   Terminate(-1);
05150                   return(0);
05151               }
05152               if ( ( Expressions[numexp].uflags & ( 1 << (x-1) ) ) != 0 )
05153                   *val2 = 1;
05154               else *val2 = 0;
05155               *type = 3;
05156               s = tt+1;
05157               return(s);
05158               }
05159           }
05160           else *t = c;
05161       }
05162       else if ( *t == '=' || *t == '<' || *t == '>' || *t == '!'
05163       || *t == ')' || *t == ' ' || *t == '\t' || *t == 0 || *t == '\n' ) {
05164           *val2 = 0;
05165           *type = 2;
05166           return(t);
05167       }
05168       else {
```

```
05169             MesPrint("@Illegal use of string in preprocessor condition: %s",s);
05170             Terminate(-1);
05171         }
05172     }
05173     while ( *t == '-' || *t == '+' || *t == ' ' || *t == '\t' ) {
05174         if ( *t == '-' ) sign = -sign;
05175         t++;
05176     }
05177     while ( chartype[*t] == 1 ) { x = 10*x + *t++ - '0'; }
05178     while ( *t == ' ' || *t == '\t' ) t++;
05179     if ( chartype[*t] == 8 || *t == ')' || *t == '=' || *t == 0 ) {
05180         *val2 = sign > 0 ? x: -x;
05181         *type = 1;
05182         return(t);
05183     }
05184     while ( chartype[*t] != 8 && *t != ')' && *t != '=' && *t ) t++;
05185     while ( ( t > s ) && ( t[-1] == ' ' || t[-1] == '\t' ) ) t--;
05186     *type = 2;
05187     *val2 = val;
05188     return(t);
05189 }
05190
05191 /*
05192         #] pParseObject :
05193         #[ PreCalc :
05194
05195         To be called when a { is encountered.
05196         Action: read first till matching }. This is to be stored.
05197         Next we look whether this is a set or whether it can be
05198         evaluated. If it is a set we consider it as a new stream.
05199         The stream will have to be deallocated when read completely.
05200         If it is to be evaluated we do that and put the result in
05201         a stream.
05202 */
05203
05204 UBYTE *PreCalc(void)
05205 {
05206     UBYTE *buff, *s = 0, *t, *newb, c;
05207     int size, i, n, parlevel = 0, bralevel = 0;
05208     LONG answer;
05209     ULONG uanswer;
05210     size = n = 0;
05211     buff = 0; c = '{';
05212     for (;;) {
05213         if ( n >= size ) {
05214             if ( size == 0 ) size = 72;
05215             else size *= 2;
05216             if ( ( newb = (UBYTE *)Malloc1(size+2,"{}") ) == 0 ) return(0);
05217             s = newb;
05218             if ( buff ) {
05219                 i = n;
05220                 t = buff;
05221                 NCOPYB(s,t,i);
05222                 M_free(buff,"pre calc buffer");
05223             }
05224             else s = newb;
05225             buff = newb;
05226         }
05227         *s++ = c; n++;
05228         c = GetChar(0);
05229         if ( c == 0 ) {
05230             Error0("Unmatched {}");
05231             M_free(buff,"precalc buffer");
05232             return(0);
05233         }
05234         else if ( c == '{' ) { bralevel++; }
05235         else if ( c == '}' ) {
05236             if ( --bralevel < 0 ) { *s++ = c; *s = 0; break; }
05237         }
05238         else if ( c == '(' ) { parlevel++; }
05239         else if ( c == ')' ) {
05240             if ( --parlevel < 0 ) { *s++ = c; *s = 0; goto setstring; }
05241         }
05242         else if ( chartype[c] != 1 && chartype[c] != 5
05243         && chartype[c] != 6 && c != '!' && c != '&'
05244         && c != '|' && c != '\\' ) { *s++ = c; *s = 0; goto setstring; }
05245     }
05246     if ( parlevel > 0 ) goto setstring;
05247 /*
05248     Try now to evaluate the string.
05249     If it works, copy the resulting value back into buff as a string.
05250 */
05251     answer = 0;
05252     if ( PreEval(buff+1,&answer) == 0 ) goto setstring;
05253     t = buff + size;
05254     s = buff;
05255     if ( answer < 0 ) { *s++ = '-'; }
```

```
05256     uanswer = LongAbs(answer);
05257     n = 0;
05258     do {
05259         *--t = ( uanswer % 10 ) + '0';
05260         uanswer /= 10;
05261         n++;
05262     } while ( uanswer > 0 );
05263     NCOPYB(s,t,n);
05264     *s = 0;
05265 setstring:;
05266 /*
05267     Open a stream that contains the current string.
05268     Mark it to be removed after termination.
05269 */
05270     if ( OpenStream(buff,PRECALCSTREAM,0,PRENOACTION) == 0 ) return(0);
05271     return(buff);
05272 }
05273
05274 /*
05275         #] PreCalc :
05276         #[ PreEval :
05277
05278         Operations are:
05279         +, -, *, /, %, &, |, ^, !,  ^% (postfix 2log), ^/ (postfix sqrt)
05280 */
05281
05282 UBYTE *PreEval(UBYTE *s, LONG *x)
05283 {
05284     LONG y, z, a;
05285     int tobemultiplied, tobeadded = 1, expsign, i;
05286     UBYTE *t;
05287     *x = 0; a = 1;
05288     while ( *s == ' ' || *s == '\t' ) s++;
05289     for(;;){
05290         if ( *s == '+' || *s == '-' ) {
05291             if ( *s == '-' ) tobeadded = -1;
05292             else tobeadded = 1;
05293             s++;
05294             while ( *s == '-' || *s == '+' || *s == ' ' || *s == '\t' ) {
05295                 if ( *s == '-' ) tobeadded = -tobeadded;
05296                 s++;
05297             }
05298         }
05299         tobemultiplied = 0;
05300         for(;;){
05301             while ( *s == ' ' || *s == '\t' ) s++;
05302             if ( *s <= '9' && *s >= '0' ) {
05303                 ULONG uy;
05304                 ParseNumber(uy,s)
05305                 y = uy;  /* may cause an implementation-defined behaviour */
05306             }
05307             else if ( *s == '(' || *s == '{' ) {
05308                 if ( ( t = PreEval(s+1,&y) ) == 0 ) return(0);
05309                 s = t;
05310             }
05311             else return(0);
05312             while ( *s == ' ' || *s == '\t' ) s++;
05313             expsign = 1;
05314             while ( *s == '^' || *s == '!' ) {
05315                 s++;
05316                 if ( s[-1] == '!' ) {   /* factorial of course */
05317                     while ( *s == ' ' || *s == '\t' ) s++;
05318                     if ( y < 0 ) {
05319                         MesPrint("@Negative value in preprocessor factorial: %l",y);
05320                         return(0);
05321                     }
05322                     else if ( y == 0 ) y = 1;
05323                     else if ( y > 1 ) {
05324                         z = y-1;
05325                         while ( z > 0 ) { y = y*z; z--; }
05326                     }
05327                     continue;
05328                 }
05329                 else if ( *s == '%' ) { /* ^% is postfix 2log */
05330                     s++;
05331                     while ( *s == ' ' || *s == '\t' ) s++;
05332                     z = y;
05333                     if ( z <= 0 ) {
05334                         MesPrint("@Illegal value in preprocessor logarithm: %l",z);
05335                         return(0);
05336                     }
05337                     y = 0; z >>= 1;
05338                     while ( z ) { y++; z >>= 1; }
05339                     continue;
05340                 }
05341                 else if ( *s == '/' ) { /* ^/ is postfix sqrt */
05342                     LONG yy, zz;
```

```
05343                   s++;
05344                   while ( *s == ' ' || *s == '\t' ) s++;
05345                   z = y;
05346                   if ( z <= 0 ) {
05347                       MesPrint("@Illegal value in preprocessor square root: %l",z);
05348                       return(0);
05349                   }
05350                   if ( z > 8 ) {      /* Very crude integer square root */
05351                       zz = z;
05352                       yy = 0; zz >>= 1;
05353                       while ( zz ) { yy++; zz >>= 1; }
05354                       zz = z >> (yy/2); i = 10; y = 0;
05355                       do {
05356                           yy = zz/2 + z/(2*zz); i--;
05357                           if ( y == yy ) break;
05358                           y = zz; zz = yy;
05359                       } while ( y != yy && i > 0 );
05360                       while ( y*y < z ) y++;
05361                       while ( y*y > z ) y--;
05362                   }
05363                   else if ( z >= 4 ) y = 2;
05364                   else if ( z == 0 ) y = 0;
05365                   else y = 1;
05366                   continue;
05367               }
05368               while ( *s == ' ' || *s == '\t' ) s++;
05369               while ( *s == '-' || *s == '+' || *s == ' ' || *s == '\t' ) {
05370                   if ( *s == '-' ) expsign = -expsign;
05371               }
05372               if ( *s <= '9' && *s >= '0' ) {
05373                   ParseNumber(z,s)
05374               }
05375               else if ( *s == '(' || *s == '{' ) {
05376                   if ( ( t = PreEval(s+1,&z) ) == 0 ) return(0);
05377                   s = t;
05378               }
05379               else return(0);
05380               while ( *s == ' ' || *s == '\t' ) s++;
05381               y = iexp(y,(int)z);
05382           }
05383           if ( tobemultiplied == 0 ) {
05384               if ( expsign < 0 ) a = 1/y;
05385               else a = y;
05386           }
05387           else {
05388               if ( tobemultiplied > 2 && expsign != 1 ) {
05389                   MesPrint("&Incorrect use of ^ with & or |. Use brackets!");
05390                   Terminate(-1);
05391               }
05392               tobemultiplied *= expsign;
05393               if ( tobemultiplied == 1 ) a *= y;
05394               else if ( tobemultiplied == 3 ) a &= y;
05395               else if ( tobemultiplied == 4 ) a |= y;
05396               else {
05397                   if ( y == 0 || tobemultiplied == -2 ) {
05398                       MesPrint("@Division by zero in preprocessor calculator");
05399                       Terminate(-1);
05400                   }
05401                   if ( tobemultiplied == 2 ) a %= y;
05402                   else a /= y;
05403               }
05404           }
05405           if ( *s == '%' ) tobemultiplied = 2;
05406           else if ( *s == '*' ) tobemultiplied = 1;
05407           else if ( *s == '/' ) tobemultiplied = -1;
05408           else if ( *s == '&' ) tobemultiplied = 3;
05409           else if ( *s == '|' ) tobemultiplied = 4;
05410           else {
05411               ULONG ux, ua;
05412               ux = *x;
05413               ua = a;
05414               if ( tobeadded >= 0 ) ux += ua;
05415               else ux -= ua;
05416               *x = ULongToLong(ux);
05417               if ( *s == ')' || *s == '}' ) return(s+1);
05418               else if ( *s == '-' || *s == '+' ) { tobeadded = 1; break; }
05419               else return(0);
05420           }
05421           s++;
05422       }
05423   }
05424 /*  return(0); */
05425 }
05426
05427 /*
05428         #] PreEval :
05429         #[ AddToPreTypes :
```

```
05430 */
05431
05432 void AddToPreTypes(int type)
05433 {
05434     if ( AP.NumPreTypes >= AP.MaxPreTypes ) {
05435         int i, *newlist = (int *)Malloc1(sizeof(int)*(2*AP.MaxPreTypes+1)
05436                         ,"preprocessor type lists");
05437         for ( i = 0; i <= AP.MaxPreTypes; i++ ) newlist[i] = AP.PreTypes[i];
05438         M_free(AP.PreTypes,"preprocessor type lists");
05439         AP.PreTypes = newlist;
05440         AP.MaxPreTypes = 2*AP.MaxPreTypes;
05441     }
05442     AP.PreTypes[++AP.NumPreTypes] = type;
05443 }
05444
05445 /*
05446         #] AddToPreTypes :
05447         #[ MessPreNesting :
05448 */
05449
05450 void MessPreNesting(int par)
05451 {
05452     MesPrint("@(%d)Illegal nesting of %#if, %#do, %#procedure and/or %#switch",par);
05453 }
05454
05455 /*
05456         #] MessPreNesting :
05457         #[ DoPreAddSeparator :
05458
05459         Preprocessor directives "addseparator" and "rmseparator" add/remove
05460         separator characters used to separate function arguments.
05461         Example:
05462
05463             #define QQ "a|g|a"
05464             #addseparator %
05465             *Comma must be quoted!:
05466             #rmseparator ","
05467             #rmseparator |
05468             #call H(a,a%'QQ')
05469
05470         Characters ' ', '\t' and '"' are ignored!
05471 */
05472
05473 int DoPreAddSeparator(UBYTE *s)
05474 {
05475     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
05476     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
05477     for(;*s != '\0';s++){
05478         while ( *s == ' ' || *s == '\t' || *s == '"') s++;
05479         /* Todo:
05480         if ( set_in(*s,invalidseparators) ) {
05481             MesPrint("@Invalid separator specified");
05482             return(-1);
05483         }
05484         */
05485         set_set(*s,AC.separators);
05486     }
05487     return(0);
05488 }
05489
05490 /*
05491         #] DoPreAddSeparator :
05492         #[ DoPreRmSeparator :
05493
05494         See commentary with DoPreAddSeparator
05495
05496         Characters ' ', '\t' and '"' are ignored!
05497 */
05498 int DoPreRmSeparator(UBYTE *s)
05499 {
05500     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
05501     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
05502     for(;*s != '\0';s++){
05503         while ( *s == ' ' || *s == '\t' || *s == '"') s++;
05504         set_del(*s,AC.separators);
05505     }
05506     return(0);
05507 }
05508
05509 /*
05510         #] DoPreRmSeparator :
05511         #[ DoExternal:
05512
05513         #external ["prevar"] command
05514 */
05515 int DoExternal(UBYTE *s)
05516 {
```

```
05517 #ifdef WITHEXTERNALCHANNEL
05518     UBYTE *prevar=0;
05519     int externalD= 0;
05520 #else
05521     DUMMYUSE(s);
05522 #endif
05523     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
05524     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
05525     if ( AP.preError ) return(0);
05526 
05527 #ifdef WITHEXTERNALCHANNEL
05528     while ( *s == ' ' || *s == '\t' ) s++;
05529     if(*s == '"'){/*prevar to store the descriptor is defined*/
05530         prevar=++s;
05531 
05532         if ( chartype[*s] == 0 )for(;*s != '"'; s++)switch(chartype[*s]){
05533             case 10:/*'\0' fits here*/
05534                 MesPrint("@Can't finde closing \"");
05535                 Terminate(-1);
05536             case 0:case 1: continue;
05537             default:
05538                 break;
05539         }
05540         if(*s != '"'){
05541                 MesPrint("@Illegal name of preprocessor variable to store external channel");
05542                 return(-1);
05543       }
05544       *s='\0';
05545         for(s++; *s == ' ' || *s == '\t'; s++);
05546     }
05547 
05548     if(*s == '\0'){
05549         MesPrint("@Illegal external command");
05550         return(-1);
05551     }
05552     /*here s is a command*/
05553    /*See the file extcmd.c*/
05554     /*[08may2006 mt]:*/
05555     externalD=openExternalChannel(
05556                 s,
05557                 AX.daemonize,
05558                 AX.shellname,
05559                 AX.stderrname);
05560     /*:[08may2006 mt]*/
05561     if(externalD<1){/*error?*/
05562         /*Not quite correct - terminate the program on error:*/
05563         Error1("Can't start external program",s);
05564         return(-1);
05565     }
05566    /*Now external command runs.*/
05567 
05568    if(prevar){/*Store the external channel descriptor in the provided variable:*/
05569         UBYTE buf[21];/* 64/Log_2[10] = 19.3, so this is enough forever...*/
05570         NumToStr(buf,externalD);
05571         if ( PutPreVar(prevar,buf,0,1) < 0 ) return(-1);
05572     }
05573 
05574     AX.currentExternalChannel=externalD;
05575     /*[08may2006 mt]:*/
05576     if(AX.currentPrompt!=0){/*Change default terminator*/
05577         if(setTerminatorForExternalChannel(  (char *)AX.currentPrompt)){
05578             MesPrint("@Prompt is too long");
05579             return(-1);
05580         }
05581     }
05582     setKillModeForExternalChannel(AX.killSignal,AX.killWholeGroup);
05583     /*:[08may2006 mt]*/
05584     return(0);
05585 #else /*ifdef WITHEXTERNALCHANNEL*/
05586     Error0("External channel: not implemented on this computer/system");
05587     return(-1);
05588 #endif /*ifdef WITHEXTERNALCHANNEL ... else*/
05589 }
05590 
05591 /*
05592         #] DoExternal:
05593         #[ DoPrompt:
05594             #prompt string
05595 */
05596 
05597 int DoPrompt(UBYTE *s)
05598 {
05599 #ifndef WITHEXTERNALCHANNEL
05600     DUMMYUSE(s);
05601 #endif
05602     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
05603     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
```

```
05604
05605 #ifdef WITHEXTERNALCHANNEL
05606     while ( *s == ' ' || *s == '\t' ) s++;
05607     if ( AX.currentPrompt )
05608           M_free(AX.currentPrompt,"external channel prompt");
05609     if ( *s == '\0' )
05610         AX.currentPrompt = (UBYTE *)strDup1((UBYTE *)"","external channel prompt");
05611     else
05612         AX.currentPrompt = strDup1(s,"external channel prompt");
05613     if(  setTerminatorForExternalChannel( (char *)AX.currentPrompt) > 0  ){
05614         MesPrint("@Prompt is too long");
05615         return(-1);
05616     }
05617     /*else: if 0, ok; if -1, there is no current channel-ok, just prompt is stored.*/
05618     return(0);
05619 #else /*ifdef WITHEXTERNALCHANNEL*/
05620     Error0("External channel: not implemented on this computer/system");
05621     return(-1);
05622 #endif /*ifdef WITHEXTERNALCHANNEL ... else*/
05623 }
05624 /*
05625         #] DoPrompt:
05626         #[ DoSetExternal:
05627             #setexternal n
05628 */
05629
05630 int DoSetExternal(UBYTE *s)
05631 {
05632 #ifdef WITHEXTERNALCHANNEL
05633     int n=0;
05634 #else
05635     DUMMYUSE(s);
05636 #endif
05637     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
05638     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
05639     if ( AP.preError ) return(0);
05640
05641 #ifdef WITHEXTERNALCHANNEL
05642     while ( *s == ' ' || *s == '\t' ) s++;
05643     while ( chartype[*s] == 1 ) { n = 10*n + *s++ - '0'; }
05644     while ( *s == ' ' || *s == '\t' ) s++;
05645     if(*s!='\0'){
05646         MesPrint("@setexternal: number expected");
05647         return(-1);
05648     }
05649     if(selectExternalChannel(n)<0){
05650         MesPrint("@setexternal: invalid number");
05651         return(-1);
05652     }
05653     AX.currentExternalChannel=n;
05654     return(0);
05655 #else /*ifdef WITHEXTERNALCHANNEL*/
05656     Error0("External channel: not implemented on this computer/system");
05657     return(-1);
05658 #endif /*ifdef WITHEXTERNALCHANNEL ... else*/
05659 }
05660 /*
05661         #] DoSetExternal:
05662         #[ DoSetExternalAttr:
05663 */
05664
05665 static FORM_INLINE UBYTE *pickupword(UBYTE *s)
05666 {
05667
05668     for(;*s>' ';s++)switch(*s){
05669         case '=':
05670         case ',':
05671         case ';':
05672             return(s);
05673     }/*for(;*s>' ';s++)switch(*s)*/
05674     return(s);
05675 }
05676 /*Returns 0 if the first string (case insensitively) equal to
05677   the beginning of the second string (of length n):
05678 */
05679 static inline int strINCmp(UBYTE *a, UBYTE *b, int n)
05680 {
05681     for(;n>0;n--)if(tolower(*a++)!=tolower(*b++))
05682         return(1);
05683     return(*a != '\0');
05684 }
05685
05686 #define KILL "kill"
05687 #define KILLALL "killall"
05688 #define DAEMON "daemon"
05689 #define SHELL "shell"
05690 #define STDERR "stderr"
```

```
05691
05692 #define TRUE_EXPR "true"
05693 #define FALSE_EXPR "false"
05694 #define NOSHELL "noshell"
05695 #define TERMINAL "terminal"
05696
05697 /*
05698     Expects comma-separated list of pairs name=value
05699 */
05700 int DoSetExternalAttr(UBYTE *s)
05701 {
05702 #ifdef WITHEXTERNALCHANNEL
05703     int lnam,lval;
05704     UBYTE *nam,*val;
05705 #else
05706     DUMMYUSE(s);
05707 #endif
05708     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
05709     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
05710     if ( AP.preError ) return(0);
05711
05712 #ifdef WITHEXTERNALCHANNEL
05713     do{
05714         /*Read the name:*/
05715         while ( *s == ' ' || *s == '\t' ) s++;
05716         s=pickupword(nam=s);
05717         lnam=s-nam;
05718         while ( *s == ' ' || *s == '\t' ) s++;
05719         if(*s++!='='){
05720             MesPrint("@External channel:'=' expected instead of %s",s-1);
05721             return(-1);
05722         }
05723         /*Read the value:*/
05724         while ( *s == ' ' || *s == '\t' ) s++;
05725         val=s;
05726
05727         for(;;){
05728             UBYTE *m;
05729             s=pickupword(s);
05730             m=s;
05731             while ( *s == ' ' || *s == '\t' ) s++;
05732          if( (*s == ',')||(*s == '\n')||(*s == ';')||(*s == '\0') ){
05733                 s=m;
05734                 break;
05735             }
05736         }/*for(;;)*/
05737
05738         lval=s-val;
05739         while ( *s == ' ' || *s == '\t' ) s++;
05740
05741         if(strINCmp((UBYTE *)SHELL,nam,lnam)==0){
05742             if(AX.shellname!=NULL)
05743                 M_free(AX.shellname,"external channel shellname");
05744             if(strINCmp((UBYTE *)NOSHELL,val,lval)==0)
05745                 AX.shellname=NULL;
05746             else{
05747                 UBYTE *ch,*b;
05748                 b=ch=AX.shellname=Malloc1(lval+1,"external channel shellname");
05749                 while(ch-b<lval)
05750                     *ch++=*val++;
05751                 *ch='\0';
05752             }
05753         }else if(strINCmp((UBYTE *)DAEMON,nam,lnam)==0){
05754             if(strINCmp((UBYTE *)TRUE_EXPR,val,lval)==0)
05755                 AX.daemonize = 1;
05756             else if(strINCmp((UBYTE *)FALSE_EXPR,val,lval)==0)
05757                 AX.daemonize = 0;
05758             else{
05759                 MesPrint("@External channel:true or false expected for %s",DAEMON);
05760                 return(-1);
05761             }
05762         }else   if(strINCmp((UBYTE *)KILLALL,nam,lnam)==0){
05763             if(strINCmp((UBYTE *)TRUE_EXPR,val,lval)==0)
05764                 AX.killWholeGroup = 1;
05765             else if(strINCmp((UBYTE *)FALSE_EXPR,val,lval)==0)
05766                 AX.killWholeGroup = 0;
05767             else{
05768                 MesPrint("@External channel: true or false expected for %s",KILLALL);
05769                 return(-1);
05770             }
05771         }else   if(strINCmp((UBYTE *)KILL,nam,lnam)==0){
05772             int i,n=0;
05773             for(i=0;i<lval;i++) {
05774                 if( *val>='0' && *val<= '9' )
05775                     n = 10*n + *val++  - '0';
05776                 else{
05777                     MesPrint("@External channel: number expected for %s",KILL);
```

```
05778                     return(-1);
05779                 }
05780             }
05781             AX.killSignal=n;
05782         }else   if(strINCmp((UBYTE *)STDERR,nam,lnam)==0){
05783             if( AX.stderrname != NULL ) {
05784                 M_free(AX.stderrname,"external channel stderrname");
05785             }
05786             if(strINCmp((UBYTE *)TERMINAL,val,lval)==0)
05787                 AX.stderrname = NULL;
05788             else{
05789                 UBYTE *ch,*b;
05790                 b=ch=AX.stderrname=Malloc1(lval+1,"external channel stderrname");
05791                 while(ch-b<lval)
05792                     *ch++=*val++;
05793                 *ch='\0';
05794             }
05795         }else{
05796             nam[lnam+1]='\0';
05797             MesPrint("@External channel: unrecognized attribute",nam);
05798             return(-1);
05799         }
05800     }while(*s++ == ',');
05801     if(  (*(s-1)>' ')&&(*(s-1)!=';')  ){
05802         MesPrint("@External channel: syntax error: %s",s-1);
05803         return(-1);
05804     }
05805    return(0);
05806 #else /*ifdef WITHEXTERNALCHANNEL*/
05807     Error0("External channel: not implemented on this computer/system");
05808     return(-1);
05809 #endif /*ifdef WITHEXTERNALCHANNEL ... else*/
05810 }
05811 /*
05812         #] DoSetExternalAttr:
05813         #[ DoRmExternal:
05814             #rmexternal [n] (if 0, close all)
05815 */
05816
05817 int DoRmExternal(UBYTE *s)
05818 {
05819 #ifdef WITHEXTERNALCHANNEL
05820     int n = -1;
05821 #else
05822     DUMMYUSE(s);
05823 #endif
05824     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
05825     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
05826     if ( AP.preError ) return(0);
05827
05828 #ifdef WITHEXTERNALCHANNEL
05829     while ( *s == ' ' || *s == '\t' ) s++;
05830     if( chartype[*s] == 1 ){
05831         for(n=0; chartype[*s] == 1 ; s++) { n = 10*n + *s - '0'; }
05832         while ( *s == ' ' || *s == '\t' ) s++;
05833     }
05834     if(*s!='\0'){
05835         MesPrint("@rmexternal: invalid number");
05836         return(-1);
05837     }
05838     switch(n){
05839         case 0:/*Close all opened channels*/
05840             closeAllExternalChannels();
05841             AX.currentExternalChannel=0;
05842             /*Do not clean AX.currentPrompt!*/
05843             return(0);
05844         case -1:/*number is not specified - try current*/
05845             n=AX.currentExternalChannel;
05846             /* fall through */
05847         default:
05848             closeExternalChannel(n);/*No reaction for possible error*/
05849     }
05850     if (n == AX.currentExternalChannel)/*cleaned up by closeExternalChannel()*/
05851         AX.currentExternalChannel=0;
05852     return(0);
05853 #else /*ifdef WITHEXTERNALCHANNEL*/
05854     Error0("External channel: not implemented on this computer/system");
05855     return(-1);
05856 #endif /*ifdef WITHEXTERNALCHANNEL ... else*/
05857
05858 }
05859 /*
05860         #] DoRmExternal:
05861         #[ DoFromExternal :
05862                 #fromexternal
05863                     is used to read the text from the running external
05864                     program, the syntax is similar to the #include
```

```
05865                   directive.
05866               #fromexternal "varname"
05867                   is used to read the text from the running external
05868                   program into the preprocessor variable varname.
05869                   directive.
05870               #fromexternal "varname" maxlength
05871                   is used to read the text from the running external
05872                   program into the preprocessor variable varname.
05873                   directive. Only first maxlength characters are
05874                   stored.
05875
05876                   FORM continues to read the running external
05877                   program output until the external program outputs a
05878                   prompt.
05879
05880 */
05881
05882 int DoFromExternal(UBYTE *s)
05883 {
05884 #ifdef WITHEXTERNALCHANNEL
05885     UBYTE *prevar=0;
05886     int lbuf=-1;
05887     int withNoList=AC.NoShowInput;
05888     int oldpreassignflag;
05889 #else
05890     DUMMYUSE(s);
05891 #endif
05892     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
05893     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
05894     if ( AP.preError ) return(0);
05895 #ifdef WITHEXTERNALCHANNEL
05896
05897     FLUSHCONSOLE;
05898
05899     while ( *s == ' ' || *s == '\t' ) s++;
05900     /*[17may2006 mt]:*/
05901     if ( *s == '-' || *s == '+' ) {
05902         if ( *s == '-' )
05903             withNoList = 1;
05904         else
05905             withNoList = 0;
05906         s++;
05907         while ( *s == ' ' || *s == '\t' ) s++;
05908     }/*if ( *s == '-' || *s == '+' )*/
05909     /*:[17may2006 mt]*/
05910     /*[02feb2006 mt]:*/
05911     if(*s == '"'){/*prevar to store the output is defined*/
05912         prevar=++s;
05913
05914         if ( *s=='$' || chartype[*s] == 0 )for(;*s != '"'; s++)switch(chartype[*s]){
05915             case 10:/*'\0' fits here*/
05916                 MesPrint("@Can't finde closing \"");
05917                 Terminate(-1);
05918             case 0:case 1: continue;
05919             default:
05920                 break;
05921         }
05922         if(*s != '"'){
05923                 MesPrint("@Illegal name to store output of external channel");
05924                 return(-1);
05925       }
05926       *s='\0';
05927         for(s++; *s == ' ' || *s == '\t'; s++);
05928     }/*if(*s == '"')*/
05929
05930     if(*s != '\0'){
05931         if( chartype[*s] == 1 ){
05932             for(lbuf=0; chartype[*s] == 1 ; s++) { lbuf = 10*lbuf + *s - '0'; }
05933             while ( *s == ' ' || *s == '\t' ) s++;
05934         }
05935         if( (*s!='\0')||(lbuf<0) ){
05936             MesPrint("@Illegal buffer length in fromexternal");
05937             return(-1);
05938         }
05939     }/*if(*s != '\0')*/
05940     /*:[02feb20006 mt]*/
05941     if(getCurrentExternalChannel()!=AX.currentExternalChannel)
05942         /*[08may20006 mt]:*/
05943         /*selectExternalChannel(AX.currentExternalChannel);*/
05944         if(selectExternalChannel(AX.currentExternalChannel)){
05945             MesPrint("@No current external channel");
05946             return(-1);
05947         }
05948         /*:[08may20006 mt]*/
05949
05950     /*[02feb2006 mt]:*/
05951     if(prevar!=0){/*The result must be stored into preprovar*/
```

```
05952       UBYTE *buf;
05953         int cc = 0;
05954         if(lbuf == -1){/*Unlimited buffer, everything must be stored*/
05955             int i;
05956             buf=Malloc1( (lbuf=255)+1,"Fromexternal");
05957             /*[18may20006 mt]:*/
05958             /*for(i=0;(cc=getcFromExtChannel())!=EOF;i++){*/
05959             /* May 2006: now getcFromExtChannelOk returns EOF while
05960                 getcFromExtChannelFailure returns -2 (see comments in
05961                 exctcmd.c):*/
05962             for(i=0;(cc=getcFromExtChannel())>0;i++){
05963             /*:[18may20006 mt]*/
05964                 if(i==lbuf){
05965                     int j;
05966                     UBYTE *tmp=Malloc1( (lbuf*=2)+1,"Fromexternal");
05967                     for(j=0;j<i;j++)tmp[j]=buf[j];
05968                     M_free(buf,"Fromexternal");
05969                     buf=tmp;
05970                 }
05971                 buf[i]=(UBYTE)(cc);
05972             }/*for(i=0;(cc=getcFromExtChannel())>0;i++)*/
05973             /*[18may20006 mt]:*/
05974          if(cc == -2){
05975                 MesPrint("@No current external channel");
05976                 return(-1);
05977             }
05978             lbuf=i;
05979             /*:[18may20006 mt]*/
05980             buf[i]='\0';
05981         }else{/*Fixed buffer, only lbuf chars must be stored*/
05982             int i;
05983             buf=Malloc1(lbuf+1,"Fromexternal");
05984             for(i=0; i<lbuf;i++){
05985             /*[18may20006 mt]:*/
05986                 /*if( (cc=getcFromExtChannel())==EOF )*/
05987                 /* May 2006: now getcFromExtChannelOk returns EOF while
05988                     getcFromExtChannelFailure returns -2 (see comments in
05989                     exctcmd.c):*/
05990                 if( (cc=getcFromExtChannel())<1 )
05991             /*:[18may20006 mt]*/
05992                     break;
05993                 buf[i]=(UBYTE)(cc);
05994             }
05995             buf[i]='\0';
05996             /*[18may20006 mt]:*/
05997             /*if(cc!=EOF)
05998                 while(getcFromExtChannel()!=EOF);*//*Eat the rest*/
05999             /* May 2006: now getcFromExtChannelOk returns EOF while
06000                 getcFromExtChannelFailure returns -2 (see comments in
06001                 exctcmd.c):*/
06002             if(cc>0)
06003                 while(getcFromExtChannel()>0);/*Eat the rest*/
06004             else if(cc == -2){
06005                 MesPrint("@No current external channel");
06006                 return(-1);
06007             }
06008             /*:[18may20006 mt]*/
06009         }
06010         /*[18may20006 mt]:*/
06011         if(*prevar == '$'){/*Put the answer to the dollar variable*/
06012             int oldNumPotModdollars = NumPotModdollars;
06013 #ifdef WITHMPI
06014             WORD oldRhsExprInModuleFlag = AC.RhsExprInModuleFlag;
06015             AC.RhsExprInModuleFlag = 0;
06016 #endif
06017             /*Here lbuf is the actual length of buf!*/
06018             /*"prevar=buf'\0'":*/
06019             UBYTE *pbuf=Malloc1(StrLen(prevar)+1+lbuf+1,"Fromexternal to dollar");
06020             UBYTE *c=pbuf;
06021             UBYTE *b=prevar;
06022             while(*b!='\0'){*c++ = *b++;}
06023             *c++='=';
06024             b=buf;
06025             while(  (*c++=*b++)!='\0'  );
06026             oldpreassignflag = AP.PreAssignFlag;
06027             AP.PreAssignFlag = 1;
06028             if ( ( cc = CompileStatement(pbuf) ) || ( cc = CatchDollar(0) ) ) {
06029                 Error1("External channel: can't asign output to dollar variable ",prevar);
06030             }
06031             AP.PreAssignFlag = oldpreassignflag;
06032             NumPotModdollars = oldNumPotModdollars;
06033 #ifdef WITHMPI
06034             AC.RhsExprInModuleFlag = oldRhsExprInModuleFlag;
06035 #endif
06036             M_free(pbuf,"Fromexternal to dollar");
06037         }else{
06038             cc = PutPreVar(prevar, buf, 0, 1) < 0;
```

```
06039         }
06040         /*:[18may20006 mt]*/
06041         M_free(buf,"Fromexternal");
06042         if ( cc ) return(-1);
06043         return(0);
06044     }
06045     /*:[02feb2006 mt]*/
06046     if ( OpenStream(s,EXTERNALCHANNELSTREAM,0,PRENOACTION) == 0 ) return(-1);
06047     /*[17may2006 mt]:*/
06048     AC.NoShowInput = withNoList;
06049     /*:[17may2006 mt]*/
06050     return(0);
06051 #else
06052     Error0("External channel: not implemented on this computer/system");
06053     return(-1);
06054 #endif
06055 }
06056
06057 /*
06058         #] DoFromExternal :
06059         #[ DoToExternal :
06060             #toexetrnal
06061 */
06062
06063 #ifdef WITHEXTERNALCHANNEL
06064
06065 /*A wrapper to writeBufToExtChannel, see the file extcmd.c:*/
06066 LONG WriteToExternalChannel(int handle, UBYTE *buffer, LONG size)
06067 {
06068     /*ATT! handle is not used! Actual output is performed to
06069         the current external channel, see extcmd.c!*/
06070     DUMMYUSE(handle);
06071     if(writeBufToExtChannel((char*)buffer,size))
06072         return(-1);
06073     return(size);
06074 }
06075 #endif /*ifdef WITHEXTERNALCHANNEL*/
06076
06077 int DoToExternal(UBYTE *s)
06078 {
06079 #ifdef WITHEXTERNALCHANNEL
06080    HANDLERS h;
06081    LONG (*OldWrite)(int handle, UBYTE *buffer, LONG size) = WriteFile;
06082     int ret=-1;
06083 #else
06084     DUMMYUSE(s);
06085 #endif
06086     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
06087     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
06088     if ( AP.preError ) return(0);
06089 #ifdef WITHEXTERNALCHANNEL
06090
06091     h.oldsilent=AM.silent;
06092     h.newlogonly = h.oldlogonly = AM.FileOnlyFlag;
06093     h.newhandle = h.oldhandle = AC.LogHandle;
06094     h.oldprinttype = AO.PrintType;
06095
06096     WriteFile=&WriteToExternalChannel;
06097
06098     while ( *s == ' ' || *s == '\t' ) s++;
06099
06100     if(AX.currentExternalChannel==0){
06101         MesPrint("@No current external channel");
06102         goto DoToExternalReady;
06103     }
06104
06105     if(getCurrentExternalChannel()!=AX.currentExternalChannel)
06106         selectExternalChannel(AX.currentExternalChannel);
06107
06108     ret=writeToChannel(EXTERNALCHANNELOUT,s,&h);
06109     DoToExternalReady:
06110         WriteFile=OldWrite;
06111         return(ret);
06112 #else /*ifdef WITHEXTERNALCHANNEL*/
06113     Error0("External channel: not implemented on this computer/system");
06114     return(-1);
06115 #endif /*ifdef WITHEXTERNALCHANNEL ... else*/
06116
06117 }
06118
06119 /*
06120         #] DoToExternal :
06121         #[ defineChannel :
06122 */
06123
06124 UBYTE *defineChannel(UBYTE *s, HANDLERS *h)
06125 {
```

```
06126     UBYTE *name,*to;
06127
06128     if ( *s != '<' )
06129         return(s);
06130
06131     s++;
06132     name = to = s;
06133     while ( *s && *s != '>' ) {
06134         if ( *s == '\\' ) s++;
06135         *to++ = *s++;
06136     }
06137     if ( *s == 0 ) {
06138         MesPrint("@Improper termination of filename");
06139         return(0);
06140     }
06141     s++;
06142     *to = 0;
06143     if ( *name ) {
06144         h->newhandle = GetChannel((char *)name,0);
06145         h->newlogonly = 1;
06146     }
06147     else if ( AC.LogHandle >= 0 ) {
06148         h->newhandle = AC.LogHandle;
06149         h->newlogonly = 1;
06150     }
06151     return(s);
06152 }
06153
06154 /*
06155         #] defineChannel :
06156         #[ writeToChannel :
06157 */
06158
06159 int writeToChannel(int wtype, UBYTE *s, HANDLERS *h)
06160 {
06161     UBYTE *to, *fstring, *ss, *sss, *s1, c, c1;
06162     WORD  num, number, nfac;
06163     WORD oldOptimizationLevel;
06164     UBYTE Out[MAXLINELENGTH+14], *stopper;
06165     int nosemi, i;
06166     int plus = 0;
06167
06168 /*
06169     Now determine the format string
06170 */
06171     while ( *s == ',' || *s == ' ' ) s++;
06172     if ( *s != '"' ) {
06173         MesPrint("@No format string present");
06174         return(-1);
06175     }
06176     s++; fstring = to = s;
06177     while ( *s ) {
06178         if ( *s == '\\' ) {
06179             s++;
06180             if ( *s == '\\' ) {
06181                 *to++ = *s++;
06182                 if ( *s == '\\' ) *to++ = *s++;
06183             }
06184             else if ( *s == '"' ) *to++ = *s++;
06185             else { *to++ = '\\'; *to++ = *s++; }
06186         }
06187         else if ( *s == '"' ) break;
06188         else *to++ = *s++;
06189     }
06190     if ( *s != '"' ) {
06191         MesPrint("@No closing \" in format string");
06192         return(-1);
06193     }
06194     *to = 0; s++;
06195     if ( AC.LineLength > 20 && AC.LineLength <= MAXLINELENGTH ) stopper = Out + AC.LineLength;
06196     else stopper = Out + MAXLINELENGTH;
06197     to = Out;
06198 /*
06199     s points now at the list of objects (if any)
06200     we can start executing the format string.
06201 */
06202     AM.silent = 0;
06203     AC.LogHandle = h->newhandle;
06204     AM.FileOnlyFlag = h->newlogonly;
06205     if ( h->newhandle >= 0 ) {
06206         AO.PrintType |= PRINTLFILE;
06207     }
06208     while ( *fstring ) {
06209         if ( to >= stopper ) {
06210             if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90 ) {
06211                 *to++ = '&';
06212             }
```

```
06213             num = to - Out;
06214             WriteString(wtype,Out,num);
06215             to = Out;
06216             if ( AC.OutputMode == FORTRANMODE
06217              || AC.OutputMode == PFORTRANMODE ) {
06218                 number = 7;
06219                 for ( i = 0; i < number; i++ ) *to++ = ' ';
06220                 to[-2] = '&';
06221             }
06222         }
06223         if ( *fstring == '\\' ) {
06224             fstring++;
06225             if ( *fstring == 'n' ) {
06226                 num = to - Out;
06227                 WriteString(wtype,Out,num);
06228                 to = Out;
06229                 fstring++;
06230             }
06231             else if ( *fstring == 't' ) { *to++ = '\t'; fstring++; }
06232             else if ( *fstring == 'b' ) { *to++ = '\\'; fstring++; }
06233             else *to++ = *fstring++;
06234         }
06235         else if ( *fstring == '%' ) {
06236             plus = 0;
06237 retry:
06238             fstring++;
06239             if ( *fstring == 'd' ) {
06240                 int sign,dig;
06241                 number = -1;
06242 donumber:
06243                 while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
06244                 sign = 1;
06245                 while ( *s == '+' || *s == '-' ) {
06246                     if ( *s == '-' ) sign = -sign;
06247                     s++;
06248                 }
06249                 dig = 0; ss = s; if ( sign < 0 ) { ss--; *ss = '-'; dig++; }
06250                 while ( *s >= '0' && *s <= '9' ) { s++; dig++; }
06251                 if ( number < 0 ) {
06252                     while ( ss < s ) {
06253                         if ( to >= stopper ) {
06254                             num = to - Out;
06255                             WriteString(wtype,Out,num);
06256                             to = Out;
06257                         }
06258                         if ( *ss == '\\' ) ss++;
06259                         *to++ = *ss++;
06260                     }
06261                 }
06262                 else {
06263                     if ( number < dig ) { dig = number; ss = s - dig; }
06264                     while ( number > dig ) {
06265                         if ( to >= stopper ) {
06266                             num = to - Out;
06267                             WriteString(wtype,Out,num);
06268                             to = Out;
06269                         }
06270                         *to++ = ' '; number--;
06271                     }
06272                     while ( ss < s ) {
06273                         if ( to >= stopper ) {
06274                             num = to - Out;
06275                             WriteString(wtype,Out,num);
06276                             to = Out;
06277                         }
06278                         if ( *ss == '\\' ) ss++;
06279                         *to++ = *ss++;
06280                     }
06281                 }
06282                 fstring++;
06283             }
06284             else if ( *fstring == '$' ) {
06285                 UBYTE *dolalloc;
06286                 number = AO.OutSkip;
06287 dodollar:
06288                 while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
06289                 if ( AC.OutputMode == FORTRANMODE
06290                  || AC.OutputMode == PFORTRANMODE ) {
06291                     number = 7;
06292                 }
06293                 if ( *s != '$' ) {
06294 nodollar:           MesPrint("@$-variable expected in #write instruction");
06295                     AM.FileOnlyFlag = h->oldlogonly;
06296                     AC.LogHandle = h->oldhandle;
06297                     AO.PrintType = h->oldprinttype;
06298                     AM.silent = h->oldsilent;
06299                     return(-1);
```

```
06300                 }
06301                 s++; ss = s;
06302                 while ( chartype[*s] <= 1 ) s++;
06303                 if ( s == ss ) goto nodollar;
06304                 c = *s; *s = 0;
06305                 num = GetDollar(ss);
06306                 if ( num < 0 ) {
06307                     MesPrint("@#write instruction: $%s has not been defined",ss);
06308                     AM.FileOnlyFlag = h->oldlogonly;
06309                     AC.LogHandle = h->oldhandle;
06310                     AO.PrintType = h->oldprinttype;
06311                     AM.silent = h->oldsilent;
06312                     return(-1);
06313                 }
06314                 *s = c;
06315                 if ( *s == '[' ) {
06316                     if ( Dollars[num].nfactors <= 0 ) {
06317                         *s = 0;
06318                         MesPrint("@#write instruction: $%s has not been factorized",ss);
06319                         AM.FileOnlyFlag = h->oldlogonly;
06320                         AC.LogHandle = h->oldhandle;
06321                         AO.PrintType = h->oldprinttype;
06322                         AM.silent = h->oldsilent;
06323                         return(-1);
06324                     }
06325 /*
06326                     Now get the number between the []
06327 */
06328                     nfac = GetDollarNumber(&s,Dollars+num);
06329 
06330                     if ( Dollars[num].nfactors == 1 && nfac == 1 ) goto writewhole;
06331 
06332                     if ( ( dolalloc = WriteDollarFactorToBuffer(num,nfac,0) ) == 0 ) {
06333                         AM.FileOnlyFlag = h->oldlogonly;
06334                         AC.LogHandle = h->oldhandle;
06335                         AO.PrintType = h->oldprinttype;
06336                         AM.silent = h->oldsilent;
06337                         return(-1);
06338                     }
06339                     goto writealloc;
06340                 }
06341                 else if ( *s && *s != ' ' && *s != ',' && *s != '\t' ) {
06342                     MesPrint("@#write instruction: illegal characters after $-variable");
06343                     AM.FileOnlyFlag = h->oldlogonly;
06344                     AC.LogHandle = h->oldhandle;
06345                     AO.PrintType = h->oldprinttype;
06346                     AM.silent = h->oldsilent;
06347                     return(-1);
06348                 }
06349                 else {
06350 writewhole:
06351                   if ( ( dolalloc = WriteDollarToBuffer(num,0) ) == 0 ) {
06352                     AM.FileOnlyFlag = h->oldlogonly;
06353                     AC.LogHandle = h->oldhandle;
06354                     AO.PrintType = h->oldprinttype;
06355                     AM.silent = h->oldsilent;
06356                     return(-1);
06357                   }
06358                   else {
06359 writealloc:
06360                     ss = dolalloc;
06361                     while ( *ss ) {
06362                         if ( to >= stopper ) {
06363                             if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90 ) {
06364                                 *to++ = '&';
06365                             }
06366                             num = to - Out;
06367                             WriteString(wtype,Out,num);
06368                             to = Out;
06369                             for ( i = 0; i < number; i++ ) *to++ = ' ';
06370                             if ( AC.OutputMode == FORTRANMODE
06371                              || AC.OutputMode == PFORTRANMODE ) to[-2] = '&';
06372                         }
06373                         if ( chartype[*ss] > 3 ) { *to++ = *ss++; }
06374                         else {
06375                             sss = ss; while ( chartype[*ss] <= 3 ) ss++;
06376                             if ( ( to + (ss-sss) ) >= stopper ) {
06377                                 if ( (ss-sss) >= (stopper-Out) ) {
06378                                     if ( ( to - stopper ) < 10 ) {
06379                                         if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 ==
      ISFORTRAN90 ) {
06380                                             *to++ = '&';
06381                                         }
06382                                         num = to - Out;
06383                                         WriteString(wtype,Out,num);
06384                                         to = Out;
06385                                         for ( i = 0; i < number; i++ ) *to++ = ' ';
```

```
06386                                       if ( AC.OutputMode == FORTRANMODE
06387                                        || AC.OutputMode == PFORTRANMODE ) to[-2] = '&';
06388                                   }
06389                                   while ( (ss-sss) >= (stopper-Out) ) {
06390                                       while ( to < stopper-1 ) {
06391                                           *to++ = *sss++;
06392                                       }
06393                                       if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 ==
    ISFORTRAN90 ) {
06394                                           *to++ = '&';
06395                                       }
06396                                       else {
06397                                           *to++ = '\\';
06398                                       }
06399                                       num = to - Out;
06400                                       WriteString(wtype,Out,num);
06401                                       to = Out;
06402                                       if ( AC.OutputMode == FORTRANMODE
06403                                        || AC.OutputMode == PFORTRANMODE ) {
06404                                           for ( i = 0; i < number; i++ ) *to++ = ' ';
06405                                           to[-2] = '&';
06406                                       }
06407                                   }
06408                               }
06409                               else {
06410                                   if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90
    ) {
06411                                       *to++ = '&';
06412                                   }
06413                                   num = to - Out;
06414                                   WriteString(wtype,Out,num);
06415                                   to = Out;
06416                                   for ( i = 0; i < number; i++ ) *to++ = ' ';
06417                                   if ( AC.OutputMode == FORTRANMODE
06418                                    || AC.OutputMode == PFORTRANMODE ) to[-2] = '&';
06419                               }
06420                           }
06421                           while ( sss < ss ) *to++ = *sss++;
06422                       }
06423                   }
06424                 }
06425                 M_free(dolalloc,"written dollar");
06426                 fstring++;
06427               }
06428           }
06429           else if ( *fstring == 's' ) {
06430               fstring++;
06431               while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
06432               if ( *s == '"' ) {
06433                   s++; ss = s;
06434                   while ( *s ) {
06435                       if ( *s == '\\' ) s++;
06436                       else if ( *s == '"' ) break;
06437                       s++;
06438                   }
06439                   if ( *s == 0 ) {
06440                       MesPrint("@#write instruction: Missing \" in string");
06441                       AM.FileOnlyFlag = h->oldlogonly;
06442                       AC.LogHandle = h->oldhandle;
06443                       AO.PrintType = h->oldprinttype;
06444                       AM.silent = h->oldsilent;
06445                       return(-1);
06446                   }
06447                   while ( ss < s ) {
06448                       if ( to >= stopper ) {
06449                           num = to - Out;
06450                           WriteString(wtype,Out,num);
06451                           to = Out;
06452                       }
06453                       if ( *ss == '\\' ) ss++;
06454                       *to++ = *ss++;
06455                   }
06456                   s++;
06457               }
06458               else {
06459                   sss = ss = s;
06460                   while ( *s && *s != ',' ) {
06461                       if ( *s == '\\' ) { s++; sss = s+1; }
06462                       s++;
06463                   }
06464                   while ( s > sss+1 && ( s[-1] == ' ' || s[-1] == '\t' ) ) s--;
06465                   while ( ss < s ) {
06466                       if ( to >= stopper ) {
06467                           num = to - Out;
06468                           WriteString(wtype,Out,num);
06469                           to = Out;
06470                       }
```

```
06471                           if ( *ss == '\\' ) ss++;
06472                           *to++ = *ss++;
06473                       }
06474                   }
06475               }
06476               else if ( *fstring == 'X' ) {
06477                   fstring++;
06478                   if ( cbuf[AM.sbufnum].numrhs > 0 ) {
06479 /*
06480                       This should be only to the value of AM.oldnumextrasymbols
06481 */
06482                       UBYTE *s = GetPreVar(AM.oldnumextrasymbols,0);
06483                       WORD x = 0;
06484                       while ( *s >= '0' && *s <= '9' ) x = 10*x + *s++ - '0';
06485                       if ( x > 0 )
06486                           PrintSubtermList(1,x);
06487                       else
06488                           PrintSubtermList(1,cbuf[AM.sbufnum].numrhs);
06489                   }
06490               }
06491               else if ( *fstring == 'O' ) {
06492                   number = AO.OutSkip;
06493 dooptim:
06494                   fstring++;
06495 /*
06496                   First test whether there is an optimization buffer
06497 */
06498                   if ( AO.OptimizeResult.code == NULL && AO.OptimizationLevel != 0 ) {
06499                       MesPrint("@In #write instruction: no optimization results available!");
06500                       return(-1);
06501                   }
06502                   num = to - Out;
06503                   WriteString(wtype,Out,num);
06504                   to = Out;
06505                   if ( AO.OptimizationLevel != 0 ) {
06506                       WORD oldoutskip = AO.OutSkip;
06507                       AO.OutSkip = number;
06508                       optimize_print_code(0);
06509                       AO.OutSkip = oldoutskip;
06510                   }
06511               }
06512               else if ( *fstring == 'e' || *fstring == 'E' ) {
06513                   if ( *fstring == 'E'
06514                       || AC.OutputMode == FORTRANMODE
06515                       || AC.OutputMode == PFORTRANMODE ) nosemi = 1;
06516                   else nosemi = 0;
06517                   fstring++;
06518                   while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
06519                   if ( chartype[*s] != 0 && *s != '[' ) {
06520 noexpr:               MesPrint("@expression name expected in #write instruction");
06521                       AM.FileOnlyFlag = h->oldlogonly;
06522                       AC.LogHandle = h->oldhandle;
06523                       AO.PrintType = h->oldprinttype;
06524                       AM.silent = h->oldsilent;
06525                       return(-1);
06526                   }
06527                   ss = s;
06528                   if ( ( s = SkipAName(ss) ) == 0 || s[-1] == '_' ) goto noexpr;
06529                   s1 = s; c = c1 = *s1;
06530                   if ( c1 == '(' ) {
06531                       SKIPBRA3(s)
06532                       if ( *s == ')' ) {
06533                           AO.CurBufWrt = s1+1;
06534                           c = *s; *s = 0;
06535                       }
06536                       else {
06537                           MesPrint("@Illegal () specifier in expression name in #write");
06538                           AM.FileOnlyFlag = h->oldlogonly;
06539                           AC.LogHandle = h->oldhandle;
06540                           AO.PrintType = h->oldprinttype;
06541                           AM.silent = h->oldsilent;
06542                           return(-1);
06543                       }
06544                   }
06545                   else AO.CurBufWrt = (UBYTE *)underscore;
06546                   *s1 = 0;
06547                   num = to - Out;
06548                   if ( num > 0 ) WriteUnfinString(wtype,Out,num);
06549                   to = Out;
06550                   oldOptimizationLevel = AO.OptimizationLevel;
06551                   AO.OptimizationLevel = 0;
06552                   if ( WriteOne(ss,(int)num,nosemi,plus) < 0 ) {
06553                       AM.FileOnlyFlag = h->oldlogonly;
06554                       AC.LogHandle = h->oldhandle;
06555                       AO.PrintType = h->oldprinttype;
06556                       AM.silent = h->oldsilent;
06557                       return(-1);
```

```
06558                 }
06559                 AO.OptimizationLevel = oldOptimizationLevel;
06560                 *s1 = c1;
06561                 if ( s > s1 ) *s++ = c;
06562             }
06563 /*
06564             File content
06565 */
06566             else if ( ( *fstring == 'f' ) || ( *fstring == 'F' ) ) {
06567                 LONG n;
06568                 while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
06569                 ss = s;
06570                 while ( *s && *s != ',' ) {
06571                     if ( *s == '\\' ) s++;
06572                     s++;
06573                 }
06574                 c = *s; *s = 0;
06575                 s1 = LoadInputFile(ss,HEADERFILE);
06576                 *s = c;
06577 /*
06578                 There should have been a way to pass the file size.
06579                 Also there should be conversions for \r\n etc.
06580 */
06581                 if ( s1 ) {
06582                     ss = s1; while ( *ss ) ss++;
06583                     n = ss-s1;
06584                     WriteString(wtype,s1,n);
06585                     M_free(s1,"copy file");
06586                 }
06587                 else if ( *fstring == 'F' ) {
06588                     *s = 0;
06589                     MesPrint("@Error in #write: could not open file %s",ss);
06590                     *s = c;
06591                     goto ReturnWithError;
06592                 }
06593                 fstring++;
06594             }
06595             else if ( *fstring == '%' ) {
06596                 *to++ = *fstring++;
06597             }
06598             else if ( FG.cTable[*fstring] == 1 ) {  /* %#S */
06599                 number = 0;
06600                 while ( FG.cTable[*fstring] == 1 ) {
06601                     number = 10*number + *fstring++ - '0';
06602                 }
06603                 if ( *fstring == 'O' ) goto dooptim;
06604                 else if ( *fstring == 'd' ) goto donumber;
06605                 else if ( *fstring == '$' ) goto dodollar;
06606                 else if ( *fstring == 't' ) {   /* `tab' position */
06607                     if ( number < (WORD)(stopper-Out) ) {
06608                         while ( (WORD)(to-Out) < number ) *to++ = ' ';
06609                     }
06610                     fstring++;
06611                 }
06612                 else if ( *fstring == 'X' || *fstring == 'x' ) {
06613                     if ( number > 0 && number <= cbuf[AM.sbufnum].numrhs ) {
06614                         UBYTE buffer[80], *out, *old1, *old2, *old3;
06615                         WORD *term, first;
06616                         if ( *fstring == 'X' ) {
06617                             out = StrCopy((UBYTE *)AC.extrasym,buffer);
06618                             if ( AC.extrasymbols == 0 ) {
06619                                 out = NumCopy(number,out);
06620                                 out = StrCopy((UBYTE *)"_",out);
06621                             }
06622                             else if ( AC.extrasymbols == 1 ) {
06623                                 if ( AC.OutputMode == CMODE ) {
06624                                     out = StrCopy((UBYTE *)"[",out);
06625                                     out = NumCopy(number,out);
06626                                     out = StrCopy((UBYTE *)"]",out);
06627                                 }
06628                                 else {
06629                                     out = StrCopy((UBYTE *)"(",out);
06630                                     out = NumCopy(number,out);
06631                                     out = StrCopy((UBYTE *)")",out);
06632                                 }
06633                             }
06634                             out = StrCopy((UBYTE *)"=",out);
06635                             ss = buffer;
06636                             while ( ss < out ) {
06637                                 if ( to >= stopper ) {
06638                                     num = to - Out;
06639                                     WriteString(wtype,Out,num);
06640                                     to = Out;
06641                                 }
06642                                 *to++ = *ss++;
06643                             }
06644                         }
```

```
06645                           term = cbuf[AM.sbufnum].rhs[number];
06646                           first = 1;
06647                           if ( *term == 0 ) {
06648                               *to++ = '0';
06649                           }
06650                           else {
06651                               old1 = AO.OutFill;
06652                               old2 = AO.OutputLine;
06653                               old3 = AO.OutStop;
06654                               AO.OutFill = to;
06655                               AO.OutputLine = Out;
06656                               AO.OutStop = Out + AC.LineLength;
06657                               while ( *term ) {
06658                                   if ( WriteInnerTerm(term,first) ) Terminate(-1);
06659                                   term += *term;
06660                                   first = 0;
06661                               }
06662                               to = Out + (AO.OutFill-AO.OutputLine);
06663                               AO.OutFill = old1;
06664                               AO.OutputLine = old2;
06665                               AO.OutStop = old3;
06666                           }
06667                       }
06668                       fstring++;
06669                   }
06670                   else {
06671                       goto IllegControlSequence;
06672                   }
06673               }
06674               else if ( *fstring == '+' ) {
06675                   plus = 1; goto retry;
06676               }
06677               else if ( *fstring == 0 ) {
06678                   *to++ = 0;
06679               }
06680               else {
06681 IllegControlSequence:
06682                   MesPrint("@Illegal control sequence in format string in #write instruction");
06683 ReturnWithError:
06684                   AM.FileOnlyFlag = h->oldlogonly;
06685                   AC.LogHandle = h->oldhandle;
06686                   AO.PrintType = h->oldprinttype;
06687                   AM.silent = h->oldsilent;
06688                   return(-1);
06689               }
06690           }
06691           else {
06692               *to++ = *fstring++;
06693           }
06694       }
06695 /*
06696       Now flush the output
06697 */
06698       num = to - Out;
06699       /*[15apr2004 mt]:*/
06700       if(wtype==EXTERNALCHANNELOUT){
06701           if(num!=0)
06702               WriteUnfinString(wtype,Out,num);
06703       }else
06704       /*:[15apr2004 mt]*/
06705       WriteString(wtype,Out,num);
06706 /*
06707       and restore original parameters
06708 */
06709       AM.FileOnlyFlag = h->oldlogonly;
06710       AC.LogHandle = h->oldhandle;
06711       AO.PrintType = h->oldprinttype;
06712       AM.silent = h->oldsilent;
06713       return(0);
06714 }
06715
06716 /*
06717           #] writeToChannel :
06718           #[ DoFactDollar :
06719
06720           Executes the #factdollar $var
06721                 instruction
06722 */
06723
06724 int DoFactDollar(UBYTE *s)
06725 {
06726       GETIDENTITY
06727       WORD numdollar, *oldworkpointer;
06728
06729       if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
06730       if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
06731       while ( *s == ' ' || *s == '\t' ) s++;
```

```
06732     if ( *s == '$' ) {
06733         if ( GetName(AC.dollarnames,s+1,&numdollar,NOAUTO) != CDOLLAR ) {
06734             MesPrint("@%s is undefined",s);
06735             return(-1);
06736         }
06737         s = SkipAName(s+1);
06738         if ( *s != 0 ) {
06739             MesPrint("@#FactDollar should have a single $variable for its argument");
06740             return(-1);
06741         }
06742         NewSort(BHEAD0);
06743         oldworkpointer = AT.WorkPointer;
06744         if ( DollarFactorize(BHEAD numdollar) ) return(-1);
06745         AT.WorkPointer = oldworkpointer;
06746         LowerSortLevel();
06747         return(0);
06748     }
06749     else if ( ParenthesesTest(s) ) return(-1);
06750     else {
06751         MesPrint("@#FactDollar should have a single $variable for its argument");
06752         return -1;
06753     }
06754 }
06755
06756 /*
06757         #] DoFactDollar :
06758         #[ GetDollarNumber :
06759 */
06760
06761 WORD GetDollarNumber(UBYTE **inp, DOLLARS d)
06762 {
06763     UBYTE *s = *inp, c, *name;
06764     WORD number, nfac, *w;
06765     DOLLARS dd;
06766     s++;
06767     if ( *s == '$' ) {
06768         s++; name = s;
06769         while ( FG.cTable[*s] < 2 ) s++;
06770         c = *s; *s = 0;
06771         if ( GetName(AC.dollarnames,name,&number,NOAUTO) == NAMENOTFOUND ) {
06772             MesPrint("@dollar in #write should have been defined previously");
06773             Terminate(-1);
06774         }
06775         *s = c;
06776         dd = Dollars + number;
06777         if ( c == '[' ) {
06778             *inp = s;
06779             nfac = GetDollarNumber(inp,dd);
06780             s = *inp;
06781             if ( *s != ']' ) {
06782                 MesPrint("@Illegal factor for dollar variable");
06783                 Terminate(-1);
06784             }
06785             *inp = s+1;
06786             if ( nfac == 0 ) {
06787                 if ( dd->nfactors > d->nfactors ) {
06788 TooBig:
06789                     MesPrint("@Factor number for dollar variable too large");
06790                     Terminate(-1);
06791                 }
06792                 return(dd->nfactors);
06793             }
06794             w = dd->factors[nfac-1].where;
06795             if ( w == 0 ) {
06796                 if ( dd->factors[nfac-1].value > d->nfactors ||
06797                      dd->factors[nfac-1].value < 0 ) goto TooBig;
06798                 return(dd->factors[nfac-1].value);
06799             }
06800             if ( *w == 4 && w[4] == 0 && w[3] == 3 && w[2] == 1
06801               && w[1] <= d->nfactors ) return(w[1]);
06802             if ( w[*w] == 0 && w[*w-1] == *w-1 ) goto TooBig;
06803 IllNum:
06804             MesPrint("@Illegal factor number for dollar variable");
06805             Terminate(-1);
06806         }
06807         else {  /* The dollar should be a number */
06808             if ( dd->type == DOLZERO ) {
06809                 return(0);
06810             }
06811             else if ( dd->type == DOLTERMS || dd->type == DOLNUMBER ) {
06812                 w = dd->where;
06813                 if ( *w == 4 && w[4] == 0 && w[3] == 3 && w[2] == 1
06814                   && w[1] <= d->nfactors ) return(w[1]);
06815                 if ( w[*w] == 0 && w[*w-1] == *w-1 ) goto TooBig;
06816                 goto IllNum;
06817             }
06818             else goto IllNum;
```

```
06819         }
06820     }
06821     else if ( FG.cTable[*s] == 1 ) {
06822         WORD x = *s++ - '0';
06823         while ( FG.cTable[*s] == 1 ) {
06824             x = 10*x + *s++ - '0';
06825             if ( x > d->nfactors ) {
06826                 MesPrint("@Factor number %d for dollar variable too large",x);
06827                 Terminate(-1);
06828             }
06829         }
06830         if ( *s != ']' ) {
06831             MesPrint("@Illegal factor number for dollar variable");
06832             Terminate(-1);
06833         }
06834         s++; *inp = s;
06835         return(x);
06836     }
06837     else {
06838         MesPrint("@Illegal factor indicator for dollar variable");
06839         Terminate(-1);
06840     }
06841     return(-1);
06842 }
06843
06844 /*
06845         #] GetDollarNumber :
06846         #[ DoSetRandom :
06847
06848         Executes the #SetRandom number
06849 */
06850
06851 int DoSetRandom(UBYTE *s)
06852 {
06853     ULONG x;
06854     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
06855     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
06856     while ( *s == ' ' || *s == '\t' ) s++;
06857     x = 0;
06858     while ( FG.cTable[*s] == 1 ) {
06859         x = 10*x + (*s++-'0');
06860     }
06861     while ( *s == ' ' || *s == '\t' ) s++;
06862     if ( *s  == 0 ) {
06863 #ifdef WITHPTHREADS
06864 #ifdef WITHSORTBOTS
06865         int id, totnum = MaX(2*AM.totalnumberofthreads-3,AM.totalnumberofthreads);
06866 #else
06867         int id, totnum = AM.totalnumberofthreads;
06868 #endif
06869         for ( id = 0; id < totnum; id++ ) {
06870             AB[id]->R.wranfseed = x;
06871             if ( AB[id]->R.wranfia ) M_free(AB[id]->R.wranfia,"wranf");
06872             AB[id]->R.wranfia = 0;
06873         }
06874 #else
06875         AR.wranfseed = x;
06876         if ( AR.wranfia ) M_free(AR.wranfia,"wranf");
06877         AR.wranfia = 0;
06878 #endif
06879         return(0);
06880     }
06881     else {
06882         MesPrint("@proper syntax is #SetRandom number");
06883         return(-1);
06884     }
06885 }
06886
06887 /*
06888         #] DoSetRandom :
06889         #[ DoOptimize :
06890
06891         Executes the #Optimize(expr) instruction.
06892 */
06893
06894 int DoOptimize(UBYTE *s)
06895 {
06896     GETIDENTITY
06897     UBYTE *exprname;
06898     WORD numexpr;
06899     int error = 0, i;
06900     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
06901     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
06902     DUMMYUSE(*s)
06903     exprname = s; s = SkipAName(s);
06904     if ( *s != 0 && *s != ';' ) {
06905         MesPrint("@proper syntax is #Optimize,expression");
```

```
06906         return(-1);
06907     }
06908     *s = 0;
06909     if ( GetName(AC.exprnames,exprname,&numexpr,NOAUTO) != CEXPRESSION ) {
06910         MesPrint("@%s is not an expression",exprname);
06911         error = 1;
06912     }
06913     else if ( AP.preError == 0 ) {
06914         EXPRESSIONS e = Expressions + numexpr;
06915         POSITION position;
06916         int firstterm;
06917         WORD *term = AT.WorkPointer;
06918         ClearOptimize();
06919         if ( AO.OptimizationLevel == 0 ) return(0);
06920         switch ( e->status ) {
06921             case LOCALEXPRESSION:
06922             case GLOBALEXPRESSION:
06923                 break;
06924             default:
06925                 MesPrint("@Expression %s is not an active unhidden local or global
      expression.",exprname);
06926                 Terminate(-1);
06927                 break;
06928         }
06929 #ifdef WITHMPI
06930         if ( PF.me == MASTER )
06931 #endif
06932         RevertScratch();
06933         for ( i = NumExpressions-1; i >= 0; i-- ) {
06934             AS.OldOnFile[i] = Expressions[i].onfile;
06935             AS.OldNumFactors[i] = Expressions[i].numfactors;
06936             AS.Oldvflags[i] = Expressions[i].vflags;
06937             Expressions[i].vflags &= ~(ISUNMODIFIED|ISZERO);
06938         }
06939         for ( i = 0; i < NumExpressions; i++ ) {
06940             if ( i == numexpr ) {
06941                 PutPreVar(AM.oldnumextrasymbols,
06942                         GetPreVar((UBYTE *)"EXTRASYMBOLS_",0),0,1);
06943                 Optimize(numexpr, 0);
06944                 AO.OptimizeResult.nameofexpr = strDup1(exprname,"optimize expression name");
06945                 continue;
06946             }
06947 #ifdef WITHMPI
06948             if ( PF.me == MASTER ) {
06949 #endif
06950             e = Expressions + i;
06951             switch ( e->status ) {
06952                 case LOCALEXPRESSION:
06953                 case SKIPLEXPRESSION:
06954                 case DROPLEXPRESSION:
06955                 case DROPPEDEXPRESSION:
06956                 case GLOBALEXPRESSION:
06957                 case SKIPGEXPRESSION:
06958                 case DROPGEXPRESSION:
06959                 case HIDELEXPRESSION:
06960                 case HIDEGEXPRESSION:
06961                 case DROPHLEXPRESSION:
06962                 case DROPHGEXPRESSION:
06963                 case INTOHIDELEXPRESSION:
06964                 case INTOHIDEGEXPRESSION:
06965                     break;
06966                 default:
06967                     continue;
06968             }
06969             AR.GetFile = 0;
06970             SetScratch(AR.infile,&(e->onfile));
06971             if ( GetTerm(BHEAD term) <= 0 ) {
06972 /* INTERNAL_ERROR_EXCL_START */
06973                 MesPrint("!>Expression %d has problems reading from scratchfile",i);
06974                 Terminate(-1);
06975 /* INTERNAL_ERROR_EXCL_STOP */
06976             }
06977             term[3] = i;
06978             AR.DeferFlag = 0;
06979             SeekScratch(AR.outfile,&position);
06980             e->onfile = position;
06981             *AM.S0->sBuffer = 0; firstterm = -1;
06982             do {
06983                 WORD *oldipointer = AR.CompressPointer;
06984                 WORD *comprtop = AR.ComprTop;
06985                 AR.ComprTop = AM.S0->sTop;
06986                 AR.CompressPointer = AM.S0->sBuffer;
06987                 if ( firstterm > 0 ) {
06988                     if ( PutOut(BHEAD term,&position,AR.outfile,1) < 0 ) goto DoSerr;
06989                 }
06990                 else if ( firstterm < 0 ) {
06991                     if ( PutOut(BHEAD term,&position,AR.outfile,0) < 0 ) goto DoSerr;
```

```
06992                   firstterm++;
06993               }
06994               else {
06995                   if ( PutOut(BHEAD term,&position,AR.outfile,-1) < 0 ) goto DoSerr;
06996                   firstterm++;
06997               }
06998               AR.CompressPointer = oldipointer;
06999               AR.ComprTop = comprtop;
07000           } while ( GetTerm(BHEAD term) );
07001           if ( FlushOut(&position,AR.outfile,1) ) {
07002 DoSerr:
07003 /* INTERNAL_ERROR_EXCL_START */
07004               MesPrint("!>Expression %d has problems writing to scratchfile",i);
07005               Terminate(-1);
07006 /* INTERNAL_ERROR_EXCL_STOP */
07007           }
07008 #ifdef WITHMPI
07009           }
07010 #endif
07011       }
07012 /*
07013       Now some administration and we are done
07014 */
07015       UpdateMaxSize();
07016   }
07017   else {
07018       ClearOptimize();
07019   }
07020   return(error);
07021
07022 }
07023
07024 /*
07025       #] DoOptimize :
07026       #[ DoClearOptimize :
07027
07028       Clears all relevant buffers of the output optimization
07029 */
07030
07031 int DoClearOptimize(UBYTE *s)
07032 {
07033   if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
07034   if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
07035   DUMMYUSE(*s);
07036   return(ClearOptimize());
07037 }
07038
07039 /*
07040       #] DoClearOptimize :
07041       #[ DoSkipExtraSymbols :
07042
07043       Adds the intermediate variables of the previous optimization
07044       to the list of extra symbols, provided it has not yet been erased
07045       by a #clearoptimize
07046       To remove them again one needs to use the 'delete extrasymbols;'
07047       or the 'delete extrasymbols>num;' statement in which num is the
07048       old number of extra symbols.
07049 */
07050
07051 int DoSkipExtraSymbols(UBYTE *s)
07052 {
07053   CBUF *C = cbuf + AM.sbufnum;
07054   WORD tt = 0, j = 0, oldval = AO.OptimizeResult.minvar;
07055   if ( AO.OptimizeResult.code == NULL ) return(0);
07056   if ( AO.OptimizationLevel == 0 ) return(0);
07057   while ( *s == ',' ) s++;
07058   if ( *s == 0 ) {
07059       AO.OptimizeResult.minvar = AO.OptimizeResult.maxvar+1;
07060   }
07061   else {
07062       while ( *s <= '9' && *s >= '0' ) j = 10*j + *s++ - '0';
07063       if ( *s ) {
07064           MesPrint("@Illegal use of #SkipExtraSymbols instruction");
07065           Terminate(-1);
07066       }
07067       AO.OptimizeResult.minvar += j;
07068       if ( AO.OptimizeResult.minvar > AO.OptimizeResult.maxvar )
07069           AO.OptimizeResult.minvar = AO.OptimizeResult.maxvar+1;
07070   }
07071   j = AO.OptimizeResult.minvar - oldval;
07072   while ( j > 0 ) {
07073       AddRHS(AM.sbufnum,1);
07074       AddNtoC(AM.sbufnum,1,&tt,16);
07075       AddToCB(C,0)
07076       InsTree(AM.sbufnum,C->numrhs);
07077       j--;
07078   }
```

```
07079     return(0);
07080 }
07081
07082 /*
07083         #] DoSkipExtraSymbols :
07084         #[ DoPreReset :
07085
07086         Does a reset of variables.
07087         Currently only the timer (stopwatch) of `timer_'
07088 */
07089
07090 int DoPreReset(UBYTE *s)
07091 {
07092     UBYTE *ss, c;
07093     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
07094     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
07095     while ( *s == ' ' || *s == '\t' ) s++;
07096     if ( *s == 0 ) {
07097         MesPrint("@proper syntax is #Reset variable");
07098         return(-1);
07099     }
07100     ss = s;
07101     while ( FG.cTable[*s] == 0 ) s++;
07102     c = *s; *s = 0;
07103     if ( ( StrICmp(ss,(UBYTE *)"timer") == 0 )
07104       || ( StrICmp(ss,(UBYTE *)"stopwatch") == 0 ) ) {
07105         *s = c;
07106         AP.StopWatchZero = GetRunningTime();
07107         return(0);
07108     }
07109     else {
07110         *s = c;
07111         MesPrint("@proper syntax is #Reset variable");
07112         return(-1);
07113     }
07114 }
07115
07116 /*
07117         #] DoPreReset :
07118         #[ DoPreAppendPath :
07119 */
07120
07121 static int DoAddPath(UBYTE *s, int bPrepend)
07122 {
07123     /* NOTE: this doesn't support some file systems, e.g., 0x5c with CP932. */
07124
07125     UBYTE *path, *path_end, *current_dir, *current_dir_end, *NewPath, *t;
07126     int bRelative, n;
07127
07128     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
07129     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
07130
07131     /* Parse the path in the input. */
07132     while ( *s == ' ' || *s == '\t' ) s++;  /* skip spaces */
07133     if ( *s == '"' ) {  /* the path is given by "..." */
07134         path = ++s;
07135         while ( *s && *s != '"' ) {
07136             if ( SEPARATOR != '\\' && *s == '\\' ) {  /* escape character, e.g., "\\"" */
07137                 if ( !s[1] ) goto ImproperPath;
07138                 s++;
07139             }
07140             s++;
07141         }
07142         if ( *s != '"' ) goto ImproperPath;
07143         path_end = s++;
07144     }
07145     else {
07146         path = s;
07147         while ( *s && *s != ' ' && *s != '\t' ) {
07148             if ( SEPARATOR != '\\' && *s == '\\' ) {  /* escape character, e.g., "\\ " */
07149                 if ( !s[1] ) goto ImproperPath;
07150                 s++;
07151             }
07152             s++;
07153         }
07154         path_end = s;
07155     }
07156     if ( path == path_end ) goto ImproperPath;  /* empty path */
07157     while ( *s == ' ' || *s == '\t' ) s++;  /* skip spaces */
07158     if ( *s ) goto ImproperPath;  /* extra tokens found */
07159
07160     /* Check if the path is an absolute path. */
07161     bRelative = 1;
07162     if ( path[0] == SEPARATOR ) {  /* starts with the directory separator */
07163         bRelative = 0;
07164     }
07165 #ifdef WINDOWS
```

```
07166     else if ( chartype[path[0]] == 0 && path[1] == ':' ) {  /* starts with (drive letter): */
07167         bRelative = 0;
07168     }
07169 #endif
07170
07171     /* Get the current file directory when a relative path is given. */
07172     if ( bRelative ) {
07173         if ( !AC.CurrentStream ) goto FileNameUnavailable;
07174         if ( AC.CurrentStream->type != FILESTREAM && AC.CurrentStream->type != REVERSEFILESTREAM )
      goto FileNameUnavailable;
07175         if ( !AC.CurrentStream->name ) goto FileNameUnavailable;
07176         s = current_dir = current_dir_end = AC.CurrentStream->name;
07177         while ( *s ) {
07178             if ( SEPARATOR != '\\' && *s == '\\' && s[1] ) {  /* escape character, e.g., "\\\"" */
07179                 s += 2;
07180                 continue;
07181             }
07182             if ( *s == SEPARATOR ) {
07183                 current_dir_end = s;
07184             }
07185             s++;
07186         }
07187     }
07188     else {
07189         current_dir = current_dir_end = NULL;
07190     }
07191
07192     /* Allocate a buffer for new AM.Path. */
07193     n = path_end - path;
07194     if ( AM.Path ) n += StrLen(AM.Path) + 1;
07195     if ( current_dir != current_dir_end ) n+= current_dir_end - current_dir + 1;
07196     s = NewPath = (UBYTE *)Malloc1(n + 1,"add path");
07197
07198     /* Construct new FORM path. */
07199     if ( bPrepend ) {
07200         if ( current_dir != current_dir_end ) {
07201             t = current_dir;
07202             while ( t != current_dir_end ) *s++ = *t++;
07203             *s++ = SEPARATOR;
07204         }
07205         t = path;
07206         while ( t != path_end ) *s++ = *t++;
07207         if ( AM.Path ) *s++ = PATHSEPARATOR;
07208     }
07209     if ( AM.Path ) {
07210         t = AM.Path;
07211         while ( *t ) *s++ = *t++;
07212     }
07213     if ( !bPrepend ) {
07214         if ( AM.Path ) *s++ = PATHSEPARATOR;
07215         if ( current_dir != current_dir_end ) {
07216             t = current_dir;
07217             while ( t != current_dir_end ) *s++ = *t++;
07218             *s++ = SEPARATOR;
07219         }
07220         t = path;
07221         while ( t != path_end ) *s++ = *t++;
07222     }
07223     *s = '\0';
07224
07225     /* Update AM.Path. */
07226     if ( AM.Path ) M_free(AM.Path,"add path");
07227     AM.Path = NewPath;
07228
07229     return(0);
07230
07231 ImproperPath:
07232     MesPrint("@Improper syntax for %#%sPath", bPrepend ? "Prepend" : "Append");
07233     return(-1);
07234
07235 FileNameUnavailable:
07236     /* This may be improved in future. */
07237     MesPrint("@Sorry, %#%sPath can't resolve the current file name from here", bPrepend ? "Prepend" :
    "Append");
07238     return(-1);
07239 }
07240
07248 int DoPreAppendPath(UBYTE *s)
07249 {
07250     return DoAddPath(s, 0);
07251 }
07252
07253 /*
07254         #] DoPreAppendPath :
07255         #[ DoPrePrependPath :
07256 */
07257
```

```
07265 int DoPrePrependPath(UBYTE *s)
07266 {
07267     return DoAddPath(s, 1);
07268 }
07269
07270 /*
07271         #] DoPrePrependPath :
07272         #[ DoTimeOutAfter :
07273
07274         Executes the #timeoutafter number
07275 */
07276
07277 int DoTimeOutAfter(UBYTE *s)
07278 {
07279 #ifdef WITH_ALARM
07280     ULONG x;
07281 #else
07282     DUMMYUSE(s);
07283 #endif
07284     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
07285     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
07286 #ifdef WITH_ALARM
07287     while ( *s == ' ' || *s == '\t' ) s++;
07288     x = 0;
07289     while ( FG.cTable[*s] == 1 ) {
07290         x = 10*x + (*s++-'0');
07291     }
07292     while ( *s == ' ' || *s == '\t' ) s++;
07293     if ( *s  == 0 ) {
07294         alarm(x);
07295         return(0);
07296     }
07297     else {
07298         MesPrint("@proper syntax is #TimeoutAfter number");
07299         return(-1);
07300     }
07301 #else
07302     Error0("#timeoutafter not implemented on this computer/system");
07303     return(-1);
07304 #endif
07305 }
07306
07307 /*
07308         #] DoTimeOutAfter :
07309         #[ DoNamespace :
07310
07311         Syntax:
07312             #Namespace name
07313                 .....
07314             #use variables
07315                 .....
07316             #EndNamespace
07317         Effect:
07318             All variables/expressions defined inside the range of the
07319             namespace get name_ prepended.
07320             This holds also for $-variables, names of procedures and
07321             names of files.
07322         Namespaces can be used in a nested way. cf this_is_deep_x
07323         A leading _ takes the role of what is super:: in some other languages.
07324         Remarks:
07325             Names of preprocessor variables are excluded!
07326             Names of built in objects are excluded! (like sum_, d_ etc.)
07327 */
07328 /* UNFINISHED_FEATURE_EXCL_START */
07329 int DoNamespace(UBYTE *s)
07330 {
07331     UBYTE *s1, *s2, c;
07332     NAMESPACE *namespace;
07333     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
07334     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
07335     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
07336     if ( FG.cTable[*s] != 0 ) {
07337         MesPrint("@Illegal name in #namespace instruction: %s",s);
07338         return(-1);
07339     }
07340     s1 = s;
07341     while ( FG.cTable[*s1] <= 1 ) s1++;
07342     s2 = s1;
07343     while ( *s2 == ' ' || *s2 == ',' || *s2 == '\t' ) s2++;
07344     if ( *s2 != 0 ) {
07345         MesPrint("@A #namespace instruction can only have one name with only alphanumeric
    characters.");
07346         return(-1);
07347     }
07348     c = *s1; *s1 = 0;
07349 /*
07350     Now we have the name and the statement is legal.
```

```
07351     We can proceed creating the namespace and its use tree.
07352 */
07353     namespace = (NAMESPACE *)Malloc1(sizeof(NAMESPACE),"namespace");
07354     namespace->name = strDup1(s,"namespace_name");
07355     namespace->usenames = MakeNameTree();
07356     if ( AP.firstnamespace == 0 ) {
07357         namespace->previous = 0;
07358         namespace->next = 0;
07359         AP.firstnamespace = namespace;
07360         AP.lastnamespace = namespace;
07361     }
07362     else {
07363         AP.lastnamespace->next = namespace;
07364         namespace->next = 0;
07365         namespace->previous = AP.lastnamespace;
07366         AP.lastnamespace = namespace;
07367     }
07368     *s1 = c;
07369     return(0);
07370 }
07371 /* UNFINISHED_FEATURE_EXCL_STOP */
07372 /*
07373         #] DoNamespace :
07374         #[ DoEndNamespace :
07375 */
07376 /* UNFINISHED_FEATURE_EXCL_START */
07377 int DoEndNamespace(UBYTE *s)
07378 {
07379     NAMESPACE *namespace;
07380     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
07381     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
07382     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
07383     if ( *s != 0 ) {
07384         MesPrint("@Illegal #endnamespace instruction");
07385         return(-1);
07386     }
07387     namespace = AP.lastnamespace;
07388     AP.lastnamespace = namespace->previous;
07389     M_free(namespace->name,"namespace_name");
07390     FreeNameTree(namespace->usenames);
07391     M_free(namespace,"namespace");
07392     return(0);
07393 }
07394 /* UNFINISHED_FEATURE_EXCL_STOP */
07395 /*
07396         #] DoEndNamespace :
07397         #[ SkipName :
07398 */
07399
07400 UBYTE *SkipName(UBYTE *s)
07401 {
07402     UBYTE *t = s, *s1, c;
07403     int num = 0, block = 0;
07404     if ( *s == '[' ) {
07405 straight:
07406         SKIPBRA1(s)
07407         if ( *s == 0 ) {
07408             MesPrint("&Illegal name: '%s'",t);
07409             return(0);
07410         }
07411         s++; s1 = s;
07412         while ( FG.cTable[*s] <= 1 || *s == '_' ) s++;
07413         if ( s1 != s ) goto witherror;
07414     }
07415     else if ( *s == '$' ) {
07416         s++;
07417         while ( *s == '_' ) { s++; num++; }
07418         block = 1;
07419         while ( FG.cTable[*s] <= 1 || *s == '_' ) {
07420             if ( FG.cTable[*s] != 0 && block == 1 ) {
07421 blocked:
07422                 MesPrint("&Illegally formed name: %s",t);
07423                 return(0);
07424             }
07425             if ( *s == '_' ) { num++; block = 1; }
07426             else block = 0;
07427             s++;
07428         }
07429         if ( s[-1] == '_' && num > 1 ) goto built;
07430     }
07431     else if ( FG.cTable[*s] == 0 ) {
07432 regular:
07433         while ( FG.cTable[*s] <= 1 || *s == '_' ) {
07434             if ( FG.cTable[*s] != 0 && block == 1 ) goto blocked;
07435             if ( *s == '_' ) { block = 1; num++; }
07436             else block = 0;
07437             s++;
```

```
07438         }
07439         if ( *s == '[' ) goto straight;
07440         if ( s[-1] == '_' && num > 1 ) {
07441 built:
07442             c = *s; *s = 0;
07443             MesPrint("&Built in objects cannot be part of namespaces: %s",t);
07444             *s = c;
07445             return(0);
07446         }
07447     }
07448     else if ( *s == '_' ) {
07449         while ( *s == '_' ) { s++; num++; }
07450         if ( FG.cTable[*s] == 0 ) { block = 0; goto regular; }
07451         goto witherror;
07452     }
07453     else if ( *s == '@' ) {
07454         s++; block = 1;
07455         if ( *s == '_' || FG.cTable[*s] == 1 ) {
07456             MesPrint("@Illegally formed name: %s",s-1);
07457         }
07458         goto regular;
07459     }
07460     else if ( *s == '#' ) { /* name of a procedure */
07461         s++;
07462         while ( *s == '_' ) { s++; num++; }
07463         block = 1;
07464         while ( FG.cTable[*s] <= 1 || *s == '_' ) {
07465             if ( FG.cTable[*s] != 0 && block == 1 ) goto blocked;
07466             if ( *s == '_' ) { num++; block = 1; }
07467             else block = 0;
07468             s++;
07469         }
07470         if ( s[-1] == '_' ) {
07471 witherror:
07472             c = *s; *s = 0;
07473             MesPrint("&Illegally formed name: %s",t);
07474             *s = c;
07475             return(0);
07476         }
07477     }
07478     else if ( *s == '<' ) { /* name of a file. Can be anything (more or less) */
07479         s++;
07480         while ( *s && *s != '>' ) s++;
07481         if ( *s != '>' ) goto witherror;
07482         s++;
07483     }
07484     return(s);
07485 }
07486
07487 /*
07488         #] SkipName :
07489         #[ ConstructName :
07490
07491         Routine gets a 'raw' name and modifies it if the namespace
07492         settings ask for it. It puts the new name in a buffer that
07493         may be expanded if the names become rather long.
07494         Note that eventually that name needs to be copied, because
07495         we do not allocate new buffers for each name.
07496
07497         type tells what kind of name we look for
07498 */
07499
07500 UBYTE *ConstructName(UBYTE *s,UBYTE type)
07501 {
07502     int len;
07503     UBYTE *t, *u;
07504     WORD number;
07505     NAMESPACE *namespace;
07506     if ( AP.lastnamespace == 0 ) return(s);
07507     if ( *s == '@' ) return(s+1);
07508     if ( GetName(AP.lastnamespace->usenames,s,&number,NOAUTO) !=
07509             NAMENOTFOUND ) return(s);
07510 /*
07511     Now the real stuff
07512     First we have to compute the size of the new name.
07513 */
07514     len = StrLen(s) + 1;
07515     namespace = AP.firstnamespace;
07516     while ( namespace ) {
07517         len += StrLen(namespace->name)+1;
07518         namespace = namespace->previous;
07519     }
07520     if ( len > AP.fullnamesize ) {
07521         while ( len > AP.fullnamesize ) AP.fullnamesize *= 2;
07522         M_free(AP.fullname,"AP.fullname");
07523         AP.fullname = (UBYTE *)Malloc1(AP.fullnamesize*sizeof(UBYTE *),"AP.fullname");
07524     }
```

```
07525     namespace = AP.firstnamespace;
07526     t = AP.fullname;
07527     switch ( type ) {
07528         case ' ':
07529         case 0:
07530             while ( namespace ) {
07531                 u = namespace->name;
07532                 while ( *u ) *t++ = *u++;
07533                 *t++ = '_';
07534                 namespace = namespace->previous;
07535             }
07536             while ( *s ) *t++ = *s++;
07537             *t = 0;
07538             break;
07539         case '$':
07540         case '#':
07541             *t++ = type;
07542             while ( namespace ) {
07543                 u = namespace->name;
07544                 while ( *u ) *t++ = *u++;
07545                 *t++ = '_';
07546                 namespace = namespace->previous;
07547             }
07548             if ( type == '$' ) s++;
07549             while ( *s ) *t++ = *s++;
07550             *t = 0;
07551             break;
07552         case '<':
07553             while ( namespace ) {
07554                 u = namespace->name;
07555                 while ( *u ) *t++ = *u++;
07556                 *t++ = '_';
07557                 namespace = namespace->previous;
07558             }
07559             s++;
07560             while ( *s ) *t++ = *s++;
07561             t--;  /* strip the '>' */
07562             *t = 0;
07563             break;
07564         default:
07565 /* INTERNAL_ERROR_EXCL_START */
07566             MesPrint("!>Unrecognized datatype in ConstructName");
07567             *t = 0;
07568             break;
07569 /* INTERNAL_ERROR_EXCL_STOP */
07570     }
07571     return(AP.fullname);
07572 }
07573
07574 /*
07575         #] ConstructName :
07576         #[ DoUse :
07577
07578         Routine makes (inside the confines of the current namespace)
07579         a list of variables that are excluded from the namespace.
07580         Once the namespace is ended, the list is removed.
07581         The list can include names of variables, dollars and procedures.
07582         Preprocessor variables are excluded from the namespace. Their
07583         inclusion would be too complicated for the input streams.
07584         Note that the preprocessor variables that are arguments in a #do
07585         or in a procedure are on a stack and should not cause problems.
07586         Names of variables can be just that.
07587         Names of $-variables are also straightforward.
07588         Names of procedures should be preceeded by a # character.
07589         Names of files (like in #include) should be enclosed by <>.
07590         The names are stored in a balanced tree. Each namespace may have
07591         its own tree. The toplevel (no namespace) does not allow a #use.
07592 */
07593 /* UNFINISHED_FEATURE_EXCL_START */
07594 int DoUse(UBYTE *s)
07595 {
07596     NAMESPACE *namespace;
07597     UBYTE *t, c;
07598     int number;
07599     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
07600     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
07601     if ( AP.lastnamespace == 0 ) {
07602         MesPrint("@It is not allowed to use #use outside the scope of a namespace.");
07603         return(-1);
07604     }
07605     namespace = AP.lastnamespace;
07606     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
07607     while ( *s ) {
07608         t = s;
07609         if ( ( s = SkipName(t) ) == 0 ) return(-1);
07610         if ( s == t ) {
07611             MesPrint("@Unrecognized object in #use instruction: %s",t);
```

```
07612             return(-1);
07613         }
07614         c = *s; *s = 0;
07615 /*
07616         In usenames we only need the names to know whether they are 'protected'.
07617         We need to keep the $, # and <> to avoid potential double names.
07618 */
07619         AddName(namespace->usenames,t,0,0,&number);
07620         *s = c;
07621         while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
07622     }
07623     return(0);
07624 }
07625 /* UNFINISHED_FEATURE_EXCL_STOP */
07626 /*
07627         #] DoUse :
07628         #[ UserFlags :
07629
07630     Syntax:
07631         #ClearFlag number(s),expression(s)
07632         #ClearFlag number(s)
07633         #ClearFlag expression(s)
07634         #ClearFlag
07635         #SetFlag number(s),expression(s)
07636         #SetFlag number(s)
07637         #SetFlag expression(s)
07638         #SetFlag
07639         par == 0: Clear, par == 1: Set.
07640 */
07641 /* UNFINISHED_FEATURE_EXCL_START */
07642 int UserFlags(UBYTE *s,int par)
07643 {
07644     int mask = 0, error = 0, i;
07645     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
07646     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
07647     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
07648     if ( *s == 0 ) {    /* Treat all flags in all active expressions */
07649 allexpr:
07650         for ( i = 0; i < NumExpressions; i++ ) {
07651             switch ( Expressions[i].status ) {
07652                 case UNHIDELEXPRESSION:
07653                 case UNHIDEGEXPRESSION:
07654                 case INTOHIDELEXPRESSION:
07655                 case INTOHIDEGEXPRESSION:
07656                 case LOCALEXPRESSION:
07657                 case GLOBALEXPRESSION:
07658                 case SKIPLEXPRESSION:
07659                 case SKIPGEXPRESSION:
07660                 case HIDELEXPRESSION:
07661                 case HIDEGEXPRESSION:
07662                     if ( par == 1 ) Expressions[i].uflags |= ~mask;
07663                     else            Expressions[i].uflags &= mask;
07664                     break;
07665                 case DROPPEDEXPRESSION:
07666                 case DROPLEXPRESSION:
07667                 case DROPGEXPRESSION:
07668                 case DROPHLEXPRESSION:
07669                 case DROPHGEXPRESSION:
07670                 case STOREDEXPRESSION:
07671                 case HIDDENLEXPRESSION:
07672                 case HIDDENGEXPRESSION:
07673                 case SPECTATOREXPRESSION:
07674                 default:
07675                     break;
07676             }
07677         }
07678     }
07679     else if ( FG.cTable[*s] == 1 ) {
07680         mask = (int)WORDMASK;
07681         while ( FG.cTable[*s] == 1 ) {
07682             int x = 0;
07683             while ( FG.cTable[*s] == 1 ) { x = 10*x + (*s++-'0'); }
07684             if ( x < 1 || x > BITSINWORD ) {
07685                 MesPrint("@Illegal number %d for flag in #...Flag instruction",x);
07686                 return(1);
07687             }
07688             mask ^= (1<<(x-1));
07689             while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
07690         }
07691         if ( *s == 0 ) goto allexpr;
07692     }
07693     else {  /* Clear all flags in all expressions that are specified */
07694         mask = (int)WORDMASK;
07695     }
07696     while ( *s ) { /* now read the expressions */
07697         UBYTE *s1, c;
07698         WORD num1;
```

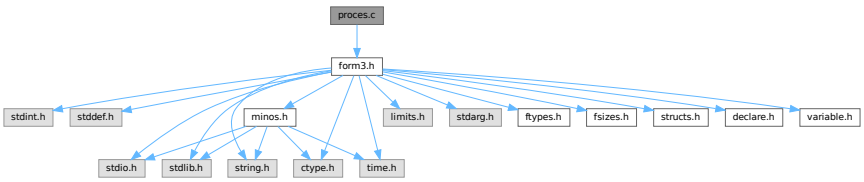
```
07699         if ( FG.cTable[*s] != 0 && *s != '[' ) goto syntax;
07700         s1 = s; s = SkipAName(s);
07701         c = *s; *s = 0;
07702         if ( GetName(AC.exprnames,s1,&num1,NOAUTO) != CEXPRESSION ) {
07703             MesPrint("@%s is not an active expression",s1);
07704             error = 1;
07705             return(error);
07706         }
07707         switch ( Expressions[num1].status ) {
07708             case UNHIDELEXPRESSION:
07709             case UNHIDEGEXPRESSION:
07710             case INTOHIDELEXPRESSION:
07711             case INTOHIDEGEXPRESSION:
07712             case LOCALEXPRESSION:
07713             case GLOBALEXPRESSION:
07714             case SKIPLEXPRESSION:
07715             case SKIPGEXPRESSION:
07716             case HIDELEXPRESSION:
07717             case HIDEGEXPRESSION:
07718                 if ( par == 1 ) Expressions[num1].uflags |= ~mask;
07719                 else            Expressions[num1].uflags &= mask;
07720                 break;
07721             case DROPPEDEXPRESSION:
07722             case DROPLEXPRESSION:
07723             case DROPGEXPRESSION:
07724             case DROPHLEXPRESSION:
07725             case DROPHGEXPRESSION:
07726             case STOREDEXPRESSION:
07727             case HIDDENLEXPRESSION:
07728             case HIDDENGEXPRESSION:
07729             case SPECTATOREXPRESSION:
07730             default:
07731                 MesPrint("@%s is not an active expression",s1);
07732                 error = 1;
07733                 break;
07734         }
07735         *s = c;
07736         while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
07737     }
07738     return(error);
07739 syntax:
07740     MesPrint("@Illegal name in #...Flag instruction.");
07741     return(1);
07742 }
07743 /* UNFINISHED_FEATURE_EXCL_STOP */
07744 /*
07745         #] UserFlags :
07746         #[ DoClearUserFlag :
07747 */
07748 /* UNFINISHED_FEATURE_EXCL_START */
07749 int DoClearUserFlag(UBYTE *s)
07750 {
07751     return(UserFlags(s,0));
07752 }
07753 /* UNFINISHED_FEATURE_EXCL_STOP */
07754 /*
07755         #] DoClearUserFlag :
07756         #[ DoSetUserFlag :
07757 */
07758 /* UNFINISHED_FEATURE_EXCL_START */
07759 int DoSetUserFlag(UBYTE *s)
07760 {
07761     return(UserFlags(s,1));
07762 }
07763 /* UNFINISHED_FEATURE_EXCL_STOP */
07764 /*
07765         #] DoSetUserFlag :
07766         #[ DoStartFloat :
07767
07768         If there is a number follwing, it will be the new default precision.
07769         If float has been started before, the old one will be removed first.
07770         If there are two numbers, the second one is the maximum weight for
07771         MZV's.
07772 */
07773 #ifdef WITHFLOAT
07774
07775 int DoStartFloat(UBYTE *s)
07776 {
07777     GETIDENTITY
07778     int error = 0;
07779     LONG x;
07780     UBYTE *ss;
07781     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
07782     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
07783     if ( AR.PolyFun != 0 ) {
07784         MesPrint("@Simultaneous use of Poly(Rat)Fun and float_ is not allowed.");
07785         error = 1;
```

```
07786     }
07787     if ( AC.ncmod != 0 ) {
07788         MesPrint("@Simultaneous use of floating point and modulus arithmetic makes no sense.");
07789         error = 1;
07790     }
07791     if ( AT.aux_ ) { // First, we clean up any previous floating point system.
07792         ClearfFloat();
07793         ClearMZVTables();
07794     }
07795     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
07796 /*
07797     The first parameter is the float precision
07798 */
07799     ss = s;
07800     if ( *s >= '0' && *s <= '9' ) {
07801         x = 0;
07802         do {
07803             x = 10*x + (*s++-'0');
07804         } while ( *s >= '0' && *s <= '9' );
07805 /*
07806         The precision can either be in digits or bits.
07807         AC.DefaultPrecision is always in bits.
07808 */
07809         if ( tolower(*s) == 'd' ) { AC.tDefaultPrecision = (LONG)ceil(x*log2(10.0)); s++; }
07810         else if ( tolower(*s) == 'b' ) { AC.tDefaultPrecision = x; s++; }
07811         else goto IllPar;
07812         while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
07813 /*
07814         The second parameter is either absent, which implies zero MZV weight,
07815         or of the form MZV = <weight>
07816 */
07817         if ( tolower(*s) == 'm' && tolower(s[1]) == 'z' && tolower(s[2]) == 'v') {
07818             s+=3;
07819             while ( *s == ' ' || *s == '\t' ) s++;
07820             if ( *s != '=') goto IllPar;
07821             s++;
07822             while ( *s == ' ' || *s == '\t' ) s++;
07823             if ( *s >= '0' && *s <= '9' ) {
07824                 x = 0;
07825                 do {
07826                     x = 10*x + (*s++ - '0');
07827                 } while ( *s >= '0' && *s <= '9' );
07828                 AC.tMaxWeight = x;
07829                 while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
07830             }
07831             else goto IllPar;
07832         }
07833         else {
07834             AC.tMaxWeight = 0;
07835         }
07836         if ( *s ) goto IllPar;
07837     }
07838     else if ( *s != 0 ) {
07839 IllPar:
07840         MesPrint("@Illegal parameter in %#StartFloat: %s ",ss);
07841         error = 1;
07842     }
07843     if ( error == 0 ) {
07844         if ( AC.tDefaultPrecision && ( AC.tDefaultPrecision != AC.DefaultPrecision
07845         || AT.aux_ == 0 ) ) {
07846             AC.DefaultPrecision = AC.tDefaultPrecision;
07847             AC.tDefaultPrecision = 0;
07848         }
07849         if ( AC.tMaxWeight && ( AC.tMaxWeight != AC.MaxWeight
07850         || AT.aux_ == 0 ) ) {
07851             AC.MaxWeight = AC.tMaxWeight;
07852             AC.tMaxWeight = 0;
07853         }
07854         SetFloatPrecision(AC.DefaultPrecision+AC.MaxWeight+1);
07855         SetupMPFTables();
07856         if ( AC.MaxWeight > 0 ) SetupMZVTables();
07857         SetfFloatPrecision(AC.DefaultPrecision);
07858     }
07859     else {
07860         AC.tDefaultPrecision = 0;
07861         AC.tMaxWeight = 0;
07862     }
07863     return(error);
07864 }
07865
07866 #endif
07867 /*
07868         #] DoStartFloat :
07869         #[ DoEndFloat :
07870 */
07871 #ifdef WITHFLOAT
07872
```

```
07873 int DoEndFloat(UBYTE *s)
07874 {
07875     int error = 0;
07876     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
07877     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
07878     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
07879     if ( *s != 0 ) {
07880         MesPrint("@Illegal parameter in %#EndFloat instruction: %s ",s);
07881         error = 1;
07882     }
07883     if ( error == 0 ) {
07884         ClearfFloat();
07885         ClearMZVTables();
07886     }
07887     return(error);
07888 }
07889
07890 #endif
07891 /*
07892         #] DoEndFloat :
07893     # ] PreProcessor :
07894 */
```

4.117 proces.c File Reference

`#include "form3.h"`
Include dependency graph for proces.c:



Macros

- #define DONE(x) { retvalue = x; goto Done; }

Functions

- int Processor (void)
- WORD TestSub (PHEAD WORD *term, WORD level)
- int InFunction (PHEAD WORD *term, WORD *termout)
- int InsertTerm (PHEAD WORD *term, WORD replac, WORD extractbuff, WORD *position, WORD *termout, WORD tepos)
- LONG PasteFile (PHEAD WORD number, WORD *accum, POSITION *position, WORD **accfill, RENUMBER renumber, WORD *freeze, WORD nexpr)
- WORD * PasteTerm (PHEAD WORD number, WORD *accum, WORD *position, WORD times, WORD divby)
- int FiniTerm (PHEAD WORD *term, WORD *accum, WORD *termout, WORD number, WORD tepos)
- int Generator (PHEAD WORD *term, WORD level)
- int DoOnePow (PHEAD WORD *term, WORD power, WORD nexp, WORD *accum, WORD *aa, WORD level, WORD *freeze)
- int Deferred (PHEAD WORD *term, WORD level)
- int PrepPoly (PHEAD WORD *term, WORD par)
- int PolyFunMul (PHEAD WORD *term)

Variables

- WORD printscratch [2]

4.117.1 Detailed Description

Contains the central terms processor routines. This is the core of the virtual machine. All other files are to help these routines.

Definition in file proces.c.

4.117.2 Macro Definition Documentation

4.117.2.1 DONE

```
#define DONE(
              x ) { retvalue = x; goto Done; }
```

TestSub hunts for subexpression pointers. If one is found its power is given in AN.TeSuOut. and the returnvalue is 'expressionnumber'. If the expression number is negative it is an expression on disk.

In addition this routine tries to locate subexpression pointers in functions. It also notices that action must be taken with any of the special functions.

Parameters

term	The term in which TestSub hunts for potential action
level	The number of the 'level' in the compiler buffer.

Returns

The number of the (sub)expression that was encountered.

Other values that are returned are in AN.TeSuOut, AR.TePos, AT.TMbuff, AN.TeInFun, AN.Frozen, AT.TMaddr

The level in the compiler buffer is more or less the number of the statement in the module. Hence it refers to the element in the lhs array.

This routine is one of the most important routines in FORM.

Definition at line 701 of file proces.c.

4.117.3 Function Documentation

4.117.3.1 Processor()

```
int Processor (
             void  )
```

This is the central processor. It accepts a stream of Expressions which is accessed by calls to GetTerm. The expressions reside either in AR.infile or AR.hidefile The definitions of an expression are seen as an id-statement, so the primary Expressions should be written to the system of scratch files as single terms with an expression pointer. Each expression is terminated with a zero and the whole is terminated by two zeroes.

The routine DoExecute should determine whether results are to be printed, should revert the scratch I/O directions etc. In principle it is DoExecute that calls Processor.

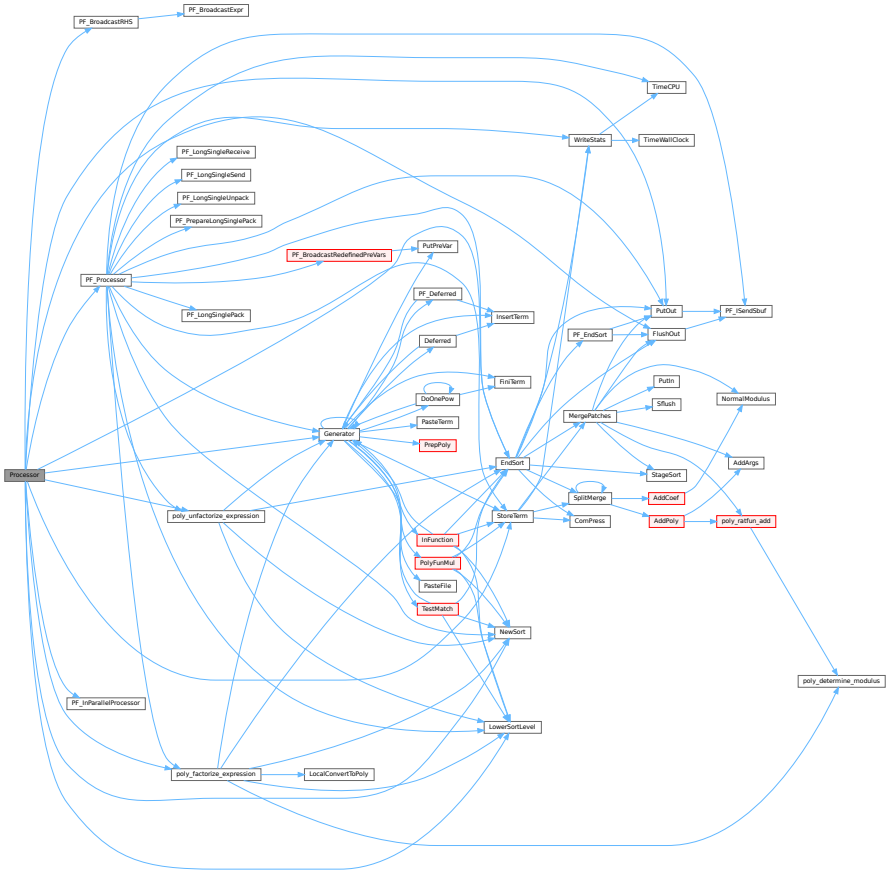
Returns

if everything OK: 0. Otherwise error. The preprocessor may continue with compilation though. Really fatal errors should return on the spot by calling Terminate.

Definition at line 64 of file proces.c.

References CbUf::Buffer, EndSort(), FlushOut(), Generator(), CbUf::lhs, LowerSortLevel(), NewSort(), PF_BroadcastRHS(), PF_InParallelProcessor(), PF_Processor(), CbUf::Pointer, poly_factorize_expression(), poly_unfactorize_expression(), PutOut(), CbUf::rhs, and StoreTerm().

Here is the call graph for this function:



4.117.3.2 TestSub()

```
WORD TestSub (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 703 of file proces.c.

4.117.3.3 InFunction()

```
int InFunction (
              PHEAD WORD * term,
              WORD * termout )
```

Makes the replacement of the subexpression with the number 'replac' in a function argument. Additional information is passed in some of the AR, AN, AT variables.

Parameters

term	The input term
termout	The output term

Returns

0: everything is fine, Negative: fatal, Positive: error.

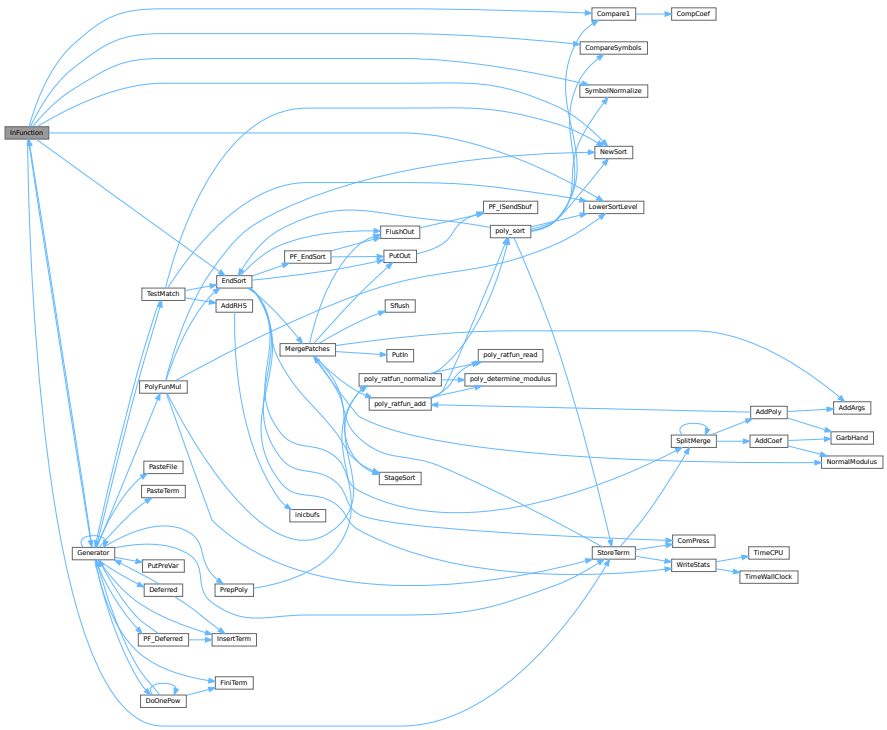
Special attention should be given to nested functions!

Definition at line 2180 of file proces.c.

References Compare1(), CompareSymbols(), EndSort(), Generator(), LowerSortLevel(), NewSort(), StoreTerm(), and SymbolNormalize().

Referenced by Generator().

Here is the call graph for this function:



4.117.3.4 InsertTerm()

```
int InsertTerm (
              PHEAD WORD * term,
              WORD replac,
              WORD extractbuff,
              WORD * position,
              WORD * termout,
              WORD tepos )
```

Puts the terms 'term' and 'position' together into a single legal term in termout. replac is the number of the subexpression that should be replaced. It must be a positive term. When action is needed in the argument of a function all terms in that argument are dealt with recursively. The subexpression is sorted. Only one subexpression is done at a time this way.

Parameters

term	the input term
replac	number of the subexpression pointer to replace
extractbuff	number of the compiler buffer replac refers to
position	position from where to take the term in the compiler buffer
termout	the output term
tepos	offset in term where the subexpression is.

Returns

Normal conventions (OK = 0).

Definition at line 2749 of file proces.c.

Referenced by Deferred(), Generator(), and PF_Deferred().

4.117.3.5 PasteFile()

```
LONG PasteFile (
              PHEAD WORD number,
              WORD * accum,
              POSITION * position,
              WORD ** accfill,
              RENUMBER renumber,
              WORD * freeze,
              WORD nexpr )
```

Gets a term from stored expression expr and puts it in the accumulator at position number. It returns the length of the term that came from file.

Parameters

number	number of partial terms to skip in accum
accum	the accumulator
position	file position from where to get the stored term
accfill	returns tail position in accum
renumber	the renumber struct for the variables in the stored expression
freeze	information about if we need only the contents of a bracket
nexpr	the number of the stored expression

Returns

Normal conventions (OK = 0).

Definition at line 2885 of file proces.c.

Referenced by Generator().

4.117.3.6 PasteTerm()

```
WORD * PasteTerm (
              PHEAD WORD number,
              WORD * accum,
              WORD * position,
              WORD times,
              WORD divby )
```

Puts the term at position in the accumulator accum at position 'number+1'. if times $>$ 0 the coefficient of this term is multiplied by times/divby.

Parameters

number	The number of term fragments in accum that should be skipped
accum	The accumulator of term fragments
position	A position in (typically) a compiler buffer from where a (piece of a) term comes.
times	Multiply the result by this
divby	Divide the result by this.

This routine is typically used when we have to replace a (sub)expression pointer by a power of a (sub)expression. This uses mostly a binomial expansion and the new term is the old term multiplied one by one by terms of the new expression. The factors times and divby keep track of the binomial coefficient. Once this is complete, the routine FiniTerm will make the contents of the accumulator into a proper term that still needs to be normalized.

Definition at line 3009 of file proces.c.

Referenced by Generator().

4.117.3.7 FiniTerm()

```
int FiniTerm (
              PHEAD WORD * term,
              WORD * accum,
              WORD * termout,
              WORD number,
              WORD tepos )
```

Concatenates the contents of the accumulator into a single legal term, which replaces the subexpression pointer

Parameters

term	the input term with the (sub)expression subterm
accum	the accumulator with the term fragments
termout	the location where the output should be written
number	the number of term fragments in the accumulator
tepos	the position of the subterm in term to be replaced

Definition at line 3076 of file proces.c.

Referenced by DoOnePow(), and Generator().

4.117.3.8 Generator()

```
int Generator (
              PHEAD WORD * term,
              WORD level )
```

The heart of the program. Here the expansion tree is set up in one giant recursion

Parameters

term	the input term. may be overwritten
level	the level in the compiler buffer (number of statement)

Returns

Normal conventions (OK = 0).

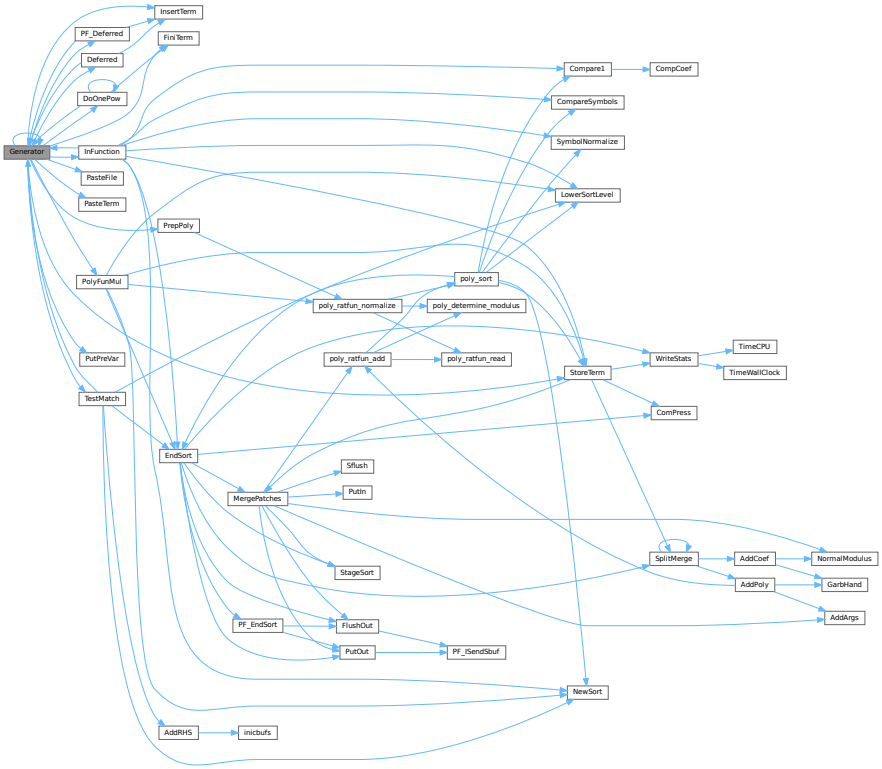
The routine looks first whether there are unsubstituted (sub)expressions. If so, one of them gets inserted term by term and the new term is used in a renewed call to Generator. If there are no (sub)expressions, the term is normalized, the compiler level is raised (next statement) and the program looks what type of statement this is. If this is a special statement it is either treated on the spot or the appropriate routine is called. If it is a substitution, the pattern matcher is called (TestMatch) which tells whether there was a match. If so we need to call TestSub again to test for (sub)expressions. If we run out of levels, the term receives a final treatment for modulus calculus and/or brackets and is then sent off to the sorting routines.

Definition at line 3275 of file proces.c.

References Deferred(), DoOnePow(), FiniTerm(), Generator(), InFunction(), InsertTerm(), CbUf::lhs, VaRrEnUm::lo, PasteFile(), PasteTerm(), PF_Deferred(), CbUf::Pointer, PolyFunMul(), PrepPoly(), PutPreVar(), CbUf::rhs, StoreTerm(), ReNuMbEr::symb, and TestMatch().

Referenced by Deferred(), DoOnePow(), generate_expression(), Generator(), InFunction(), MakeDollarInteger(), MakeDollarMod(), PF_CollectModifiedDollars(), PF_Deferred(), PF_Processor(), poly_factorize_expression(), poly_unfactorize_expression(), Processor(), and TestMatch().

Here is the call graph for this function:



4.117.3.9 DoOnePow()

```
int DoOnePow (
            PHEAD WORD * term,
            WORD power,
            WORD nexp,
            WORD * accum,
            WORD * aa,
            WORD level,
            WORD * freeze )
```

Routine gets one power of an expression in the scratch system. If there are more powers needed there will be a recursion.

No attempt is made to use binomials because we have no information about commuting properties.

There is a searching for the contents of brackets if needed. This searching may be rather slow because of the single links.

Parameters

term	is the term we are adding to.
power	is the power of the expression that we need.
nexp	is the number of the expression.
accum	is the accumulator of terms. It accepts the termfragments that are made into a proper term in FiniTerm

Parameters

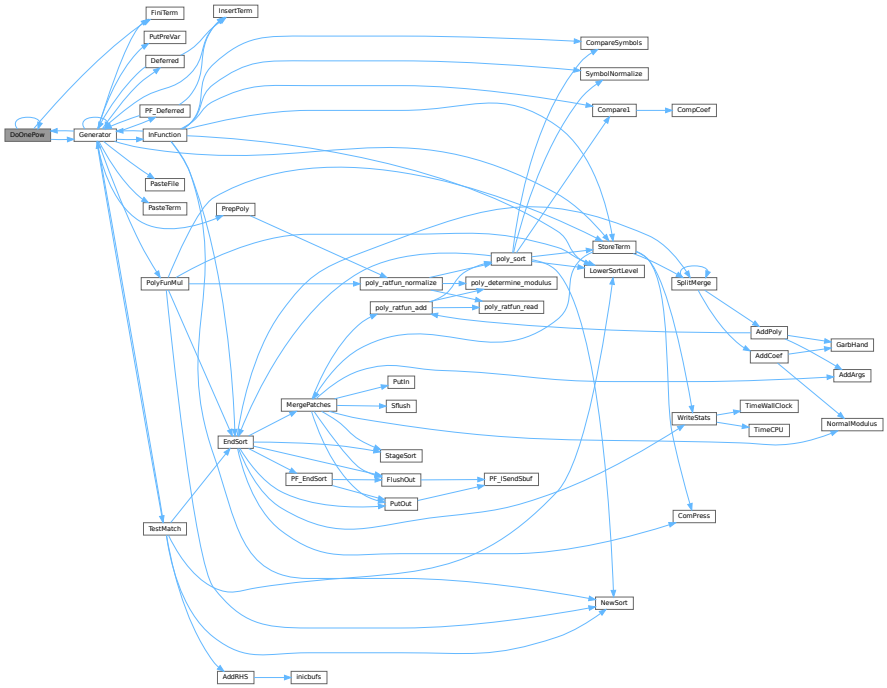
aa	points to the start of the entire accumulator. In ∗aa we store the number of term fragments that are in the accumulator.
level	is the current depth in the tree of statements. It is needed to continue to the next operation/substitution with each generated term
freeze	is the pointer to the bracket information that should be matched.

Definition at line 4653 of file proces.c.

References DoOnePow(), FiniTerm(), Generator(), and FiLe::handle.

Referenced by DoOnePow(), and Generator().

Here is the call graph for this function:



4.117.3.10 Deferred()

```
int Deferred (
            PHEAD WORD * term,
            WORD level )
```

Picks up the deferred brackets. These are the bracket contents of which we postpone the reading when we use the 'Keep Brackets' statement. These contents are multiplying the terms just before they are sent to the sorting system. Special attention goes to having it thread-safe We have to lock positioning the file and reading it in a thread specific buffer.

Parameters

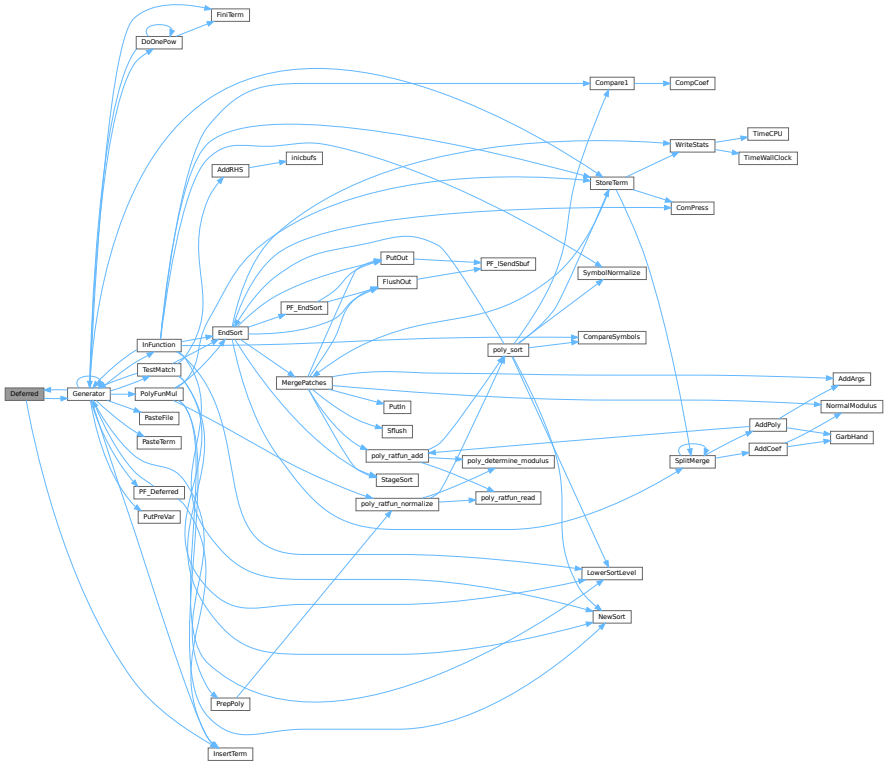
term	The term that must be multiplied by the contents of the current bracket
level	The compiler level. This is needed because after multiplying term by term we call Generator again.

Definition at line 4876 of file proces.c.

References Generator(), and InsertTerm().

Referenced by Generator().

Here is the call graph for this function:



4.117.3.11 PrepPoly()

```
int PrepPoly (
            PHEAD WORD * term,
            WORD par )
```

Routine checks whether the count of function AR.PolyFun is zero or one. If it is one and it has one scalarlike argument the coefficient of the term is pulled inside the argument. If the count is zero a new function is made with the coefficient as its only argument. The function should be placed at its proper position.

When this function is active it places the PolyFun as last object before the coefficient. This is needed because otherwise the compress algorithm has problems in MergePatches.

The bracket routine should also place the PolyFun at a comparable spot. The compression should then stop at the PolyFun. It doesn't really have to stop when writing the final result but this may be too complicated.

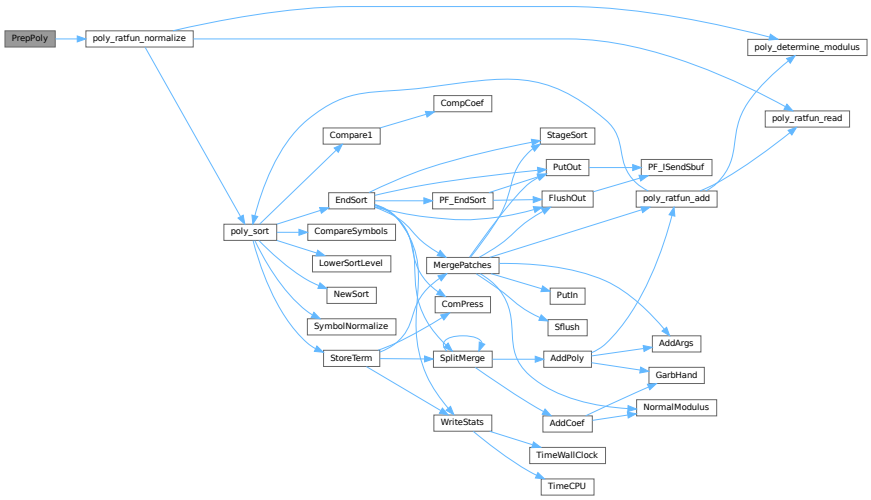
The parameter par tells whether we are at groundlevel or inside a function or dollar variable.

Definition at line 5004 of file proces.c.

References poly_ratfun_normalize().

Referenced by Generator().

Here is the call graph for this function:



4.117.3.12 PolyFunMul()

```
int PolyFunMul (
            PHEAD WORD * term )
```

Multiplies the arguments of multiple occurrences of the polyfun. In this routine we do the original PolyFun with one argument only. The PolyRatFun (PolyFunType = 2) is done in a dedicated routine in the file polywrap.cc The new result is written over the old result.

Parameters

term	It contains the input term and later the output.

Returns

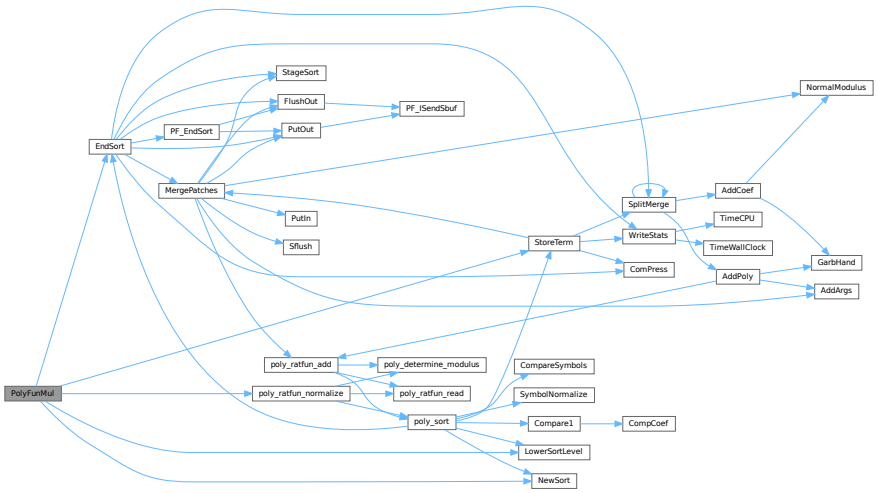
Normal conventions (OK = 0).

Definition at line 5394 of file proces.c.

References EndSort(), LowerSortLevel(), NewSort(), poly_ratfun_normalize(), and StoreTerm().

Referenced by Generator().

Here is the call graph for this function:



4.117.4 Variable Documentation

4.117.4.1 printscratch

```
WORD printscratch[2]
```

Definition at line 39 of file proces.c.

4.118 proces.c

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032 /*
00033 #define HIDEDEBUG
```

```
00034     #[ Includes : proces.c
00035 */
00036
00037 #include "form3.h"
00038
00039 WORD printscratch[2];
00040
00041 /*
00042     #] Includes :
00043     #[ Processor :
00044         #[ Processor :          WORD Processor()
00045 */
00064 int Processor(void)
00065 {
00066     GETIDENTITY
00067     WORD *term, *t, i, size;
00068     int retval = 0;
00069     EXPRESSIONS e;
00070     POSITION position;
00071     WORD last, LastExpression, fromspectator;
00072     LONG dd = 0;
00073     CBUF *C = cbuf+AC.cbufnum;
00074     int firstterm;
00075     CBUF *CC = cbuf+AT.ebufnum;
00076     WORD **w, *cpo, *cbo;
00077     FILEHANDLE *curfile, *oldoutfile = AR.outfile;
00078     WORD oldBracketOn = AR.BracketOn;
00079     WORD *oldBrackBuf = AT.BrackBuf;
00080     WORD oldbracketindexflag = AT.bracketindexflag;
00081 #ifdef WITHPTHREADS
00082     int OldMultiThreaded = AS.MultiThreaded, Oldmparallelflag = AC.mparallelflag;
00083 #endif
00084     if ( CC->numrhs > 0 || CC->numlhs > 0 ) {
00085         if ( CC->rhs ) {
00086             w = CC->rhs; i = CC->numrhs;
00087             do { *w++ = 0; } while ( --i > 0 );
00088         }
00089         if ( CC->lhs ) {
00090             w = CC->lhs; i = CC->numlhs;
00091             do { *w++ = 0; } while ( --i > 0 );
00092         }
00093         CC->numlhs = CC->numrhs = 0;
00094         ClearTree(AT.ebufnum);
00095         CC->Pointer = CC->Buffer;
00096     }
00097
00098     if ( NumExpressions == 0 ) return(0);
00099     AR.expflags = 0;
00100     AR.CompressPointer = AR.CompressBuffer;
00101     AR.NoCompress = AC.NoCompress;
00102     term = AT.WorkPointer;
00103     if ( ( (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer) ) > AT.WorkTop ) {
00104         MesWork();
00105     }
00106     UpdatePositions();
00107     C->rhs[C->numrhs+1] = C->Pointer;
00108     AR.KeptInHold = 0;
00109     if ( AC.CollectFun ) AR.DeferFlag = 0;
00110     AR.outtohide = 0;
00111     AN.PolyFunTodo = 0;
00112 #ifdef HIDEDEBUG
00113     MesPrint("Status at the start of Processor (HideLevel = %d)",AC.HideLevel);
00114     MesPrint("File %s POfill %l POfull %l POsize %l", AR.infile->name   ,AR.infile->POfill
    -AR.infile->PObuffer  ,AR.infile->POfull  -AR.infile->PObuffer  ,AR.infile->POsize/sizeof(WORD)  );
00115     MesPrint("File %s POfill %l POfull %l POsize %l", AR.outfile->name  ,AR.outfile->POfill
    -AR.outfile->PObuffer ,AR.outfile->POfull -AR.outfile->PObuffer ,AR.outfile->POsize/sizeof(WORD) );
00116     MesPrint("File %s POfill %l POfull %l POsize %l", AR.hidefile->name
    ,AR.hidefile->POfill-AR.hidefile->PObuffer,AR.hidefile->POfull-AR.hidefile->PObuffer,AR.hidefile->POsize/sizeof(WORD));
00117     for ( i = 0; i < NumExpressions; i++ ) {
00118         e = Expressions+i;
00119         ExprStatus(e);
00120     }
00121 #endif
00122 /*
00123         Next determine the last expression. This is used for removing the
00124         input file when the final stage of the sort of this expression is
00125         reached. That can save up to 1/3 in disk space.
00126 */
00127     for ( i = NumExpressions-1; i >= 0; i-- ) {
00128         e = Expressions+i;
00129         if ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION
00130         || e->status == HIDELEXPRESSION || e->status == HIDEGEXPRESSION
00131         || e->status == SKIPLEXPRESSION || e->status == SKIPGEXPRESSION
00132         || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION
00133         || e->status == INTOHIDELEXPRESSION || e->status == INTOHIDEGEXPRESSION
00134         ) break;
00135     }
```

```
00136     last = i;
00137     for ( i = NumExpressions-1; i >= 0; i-- ) {
00138         AS.OldOnFile[i] = Expressions[i].onfile;
00139         AS.OldNumFactors[i] = Expressions[i].numfactors;
00140 /*      AS.Oldvflags[i] = e[i].vflags; */
00141         AS.Oldvflags[i] = Expressions[i].vflags;
00142         AS.Olduflags[i] = Expressions[i].uflags;
00143         Expressions[i].vflags &= ~(ISUNMODIFIED|ISZERO);
00144     }
00145 #ifdef WITHPTHREADS
00146 /*
00147         When we run with threads we have to make sure that all local input
00148         buffers are pointed correctly. Of course this isn't needed if we
00149         run on a single thread only.
00150 */
00151     if ( AC.partodoflag && AM.totalnumberofthreads > 1 ) {
00152         AS.MultiThreaded = 1; AC.mparallelflag = PARALLELFLAG;
00153     }
00154     if ( AS.MultiThreaded && AC.mparallelflag == PARALLELFLAG ) {
00155         SetWorkerFiles();
00156     }
00157 /*
00158         We start with running the expressions with expr->partodo in parallel.
00159         The current model is: give each worker an expression. Wait for
00160         workers to finish and tell them where to write.
00161         Then give them a new expression. Workers may have to wait for each
00162         other. This is also the case with the last one.
00163 */
00164     if ( AS.MultiThreaded && AC.mparallelflag == PARALLELFLAG ) {
00165         if ( InParallelProcessor() ) {
00166             retval = 1;
00167         }
00168         AS.MultiThreaded = OldMultiThreaded;
00169         AC.mparallelflag = Oldmparallelflag;
00170     }
00171 #endif
00172 #ifdef WITHMPI
00173     if ( AC.RhsExprInModuleFlag && PF.rhsInParallel && (AC.mparallelflag == PARALLELFLAG ||
      AC.partodoflag) ) {
00174         if ( PF_BroadcastRHS() ) {
00175             retval = -1;
00176         }
00177     }
00178     PF.exprtodo = -1; /* This means, the slave does not perform inparallel */
00179     if ( AC.partodoflag > 0 ) {
00180         if ( PF_InParallelProcessor() ) {
00181             retval = -1;
00182         }
00183     }
00184 #endif
00185     for ( i = 0; i < NumExpressions; i++ ) {
00186 #ifdef INNERTEST
00187         if ( AC.InnerTest ) {
00188             if ( StrCmp(AC.TestValue,(UBYTE *)INNERTEST) == 0 ) {
00189                 MesPrint("Testing(Processor): value = %s",AC.TestValue);
00190             }
00191         }
00192 #endif
00193         e = Expressions+i;
00194 #ifdef WITHPTHREADS
00195         if ( AC.partodoflag > 0 && e->partodo > 0 && AM.totalnumberofthreads > 2 ) {
00196             e->partodo = 0;
00197             continue;
00198         }
00199 #endif
00200 #ifdef WITHMPI
00201         if ( AC.partodoflag > 0 && e->partodo > 0 && PF.numtasks > 2 ) {
00202             e->partodo = 0;
00203             continue;
00204         }
00205 #endif
00206         AS.CollectOverFlag = 0;
00207         AR.expchanged = 0;
00208         if ( i == last ) LastExpression = 1;
00209         else             LastExpression = 0;
00210         if ( e->inmem ) {
00211 /*
00212             #[ in memory : Memory allocated by poly.c only thusfar.
00213                            Here GetTerm cannot work.
00214                            For the moment we ignore this for parallelization.
00215 */
00216             WORD j;
00217
00218             AR.GetFile = 0;
00219             SetScratch(AR.infile,&(e->onfile));
00220             if ( GetTerm(BHEAD term) <= 0 ) {
00221 /* INTERNAL_ERROR_EXCL_START */
```

```
00222                 MesPrint("!>(1) Expression %d has problems in scratchfile",i);
00223                 retval = -1;
00224                 break;
00225 /* INTERNAL_ERROR_EXCL_STOP */
00226             }
00227             term[3] = i;
00228             AR.CurExpr = i;
00229             SeekScratch(AR.outfile,&position);
00230             e->onfile = position;
00231             if ( PutOut(BHEAD term,&position,AR.outfile,0) < 0 ) goto ProcErr;
00232             AR.DeferFlag = AC.ComDefer;
00233             NewSort(BHEAD0);
00234             AN.ninterms = 0;
00235             t = e->inmem;
00236             while ( *t ) {
00237                 for ( j = 0; j < *t; j++ ) term[j] = t[j];
00238                 t += *t;
00239                 AN.ninterms++; dd = AN.deferskipped;
00240                 if ( AC.CollectFun && *term <= (AM.MaxTer/(2*(LONG)(sizeof(WORD)))) ) {
00241                     if ( GetMoreFromMem(term,&t) ) {
00242                         LowerSortLevel(); goto ProcErr;
00243                     }
00244                 }
00245                 AT.WorkPointer = term + *term;
00246                 AN.RepPoint = AT.RepCount + 1;
00247                 AN.IndDum = AM.IndDum;
00248                 AR.CurDum = ReNumber(BHEAD term);
00249                 if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
00250                 if ( AN.ncmod ) {
00251                     if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
00252                     else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
00253                 }
00254                 else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
00255                 if ( Generator(BHEAD term,0) ) {
00256                     LowerSortLevel(); goto ProcErr;
00257                 }
00258                 AN.ninterms += dd;
00259             }
00260             AN.ninterms += dd;
00261             if ( EndSort(BHEAD AM.S0->sBuffer,0) < 0 ) goto ProcErr;
00262             if ( AM.S0->TermsLeft ) e->vflags &= ~ISZERO;
00263             else e->vflags |= ISZERO;
00264             if ( AR.expchanged == 0 ) e->vflags |= ISUNMODIFIED;
00265                 if ( AM.S0->TermsLeft ) AR.expflags |= ISZERO;
00266             if ( AR.expchanged ) AR.expflags |= ISUNMODIFIED;
00267             AR.GetFile = 0;
00268 /*
00269             #] in memory :
00270 */
00271         }
00272         else {
00273             AR.CurExpr = i;
00274             switch ( e->status ) {
00275             case UNHIDELEXPRESSION:
00276             case UNHIDEGEXPRESSION:
00277                 AR.GetFile = 2;
00278 #ifdef WITHMPI
00279                 if ( PF.me == MASTER ) SetScratch(AR.hidefile,&(e->onfile));
00280 #else
00281                 SetScratch(AR.hidefile,&(e->onfile));
00282                 AR.InHiBuf = AR.hidefile->POfull-AR.hidefile->POfill;
00283 #ifdef HIDEDEBUG
00284                 MesPrint("Hidefile: onfile: %15p, POposition: %15p, filesize: %15p",&(e->onfile)
00285                 ,&(AR.hidefile->POposition),&(AR.hidefile->filesize));
00286                 MesPrint("Set hidefile to buffer position %l/%l; AR.InHiBuf = %l"
00287                     ,(AR.hidefile->POfill-AR.hidefile->PObuffer)*sizeof(WORD)
00288                     ,(AR.hidefile->POfull-AR.hidefile->PObuffer)*sizeof(WORD)
00289                     ,AR.InHiBuf
00290                 );
00291 #endif
00292 #endif
00293                 curfile = AR.hidefile;
00294                 goto commonread;
00295             case INTOHIDELEXPRESSION:
00296             case INTOHIDEGEXPRESSION:
00297                 AR.outtohide = 1;
00298 /*
00299                 BugFix 12-feb-2016
00300                 This may not work when the file is open and we move around.
00301                 AR.hidefile->POfill = AR.hidefile->POfull;
00302 */
00303                 SetEndHScratch(AR.hidefile,&position);
00304                 /* fall through */
00305             case LOCALEXPRESSION:
00306             case GLOBALEXPRESSION:
00307                 AR.GetFile = 0;
00308 /*[20oct2009 mt]:*/
```

```
00309 #ifdef WITHMPI
00310                 if( ( PF.me == MASTER ) || (PF.mkSlaveInfile) )
00311 #endif
00312                 SetScratch(AR.infile,&(e->onfile));
00313 /*:[20oct2009 mt]*/
00314                 curfile = AR.infile;
00315 commonread:;
00316 #ifdef WITHMPI
00317                 if ( PF_Processor(e,i,LastExpression) ) {
00318                     MesPrint("Error in PF_Processor");
00319                     goto ProcErr;
00320                 }
00321 /*[20oct2009 mt]:*/
00322                 if ( AC.mparallelflag != PARALLELFLAG ){
00323                     if(PF.me != MASTER)
00324                         break;
00325 #endif
00326 /*:[20oct2009 mt]*/
00327                 if ( GetTerm(BHEAD term) <= 0 ) {
00328 #ifdef HIDEDEBUG
00329                     MesPrint("Error condition 1a");
00330                     ExprStatus(e);
00331 #endif
00332 /* INTERNAL_ERROR_EXCL_START */
00333                     MesPrint("!>(2) Expression %d has problems in scratchfile(process)",i);
00334                     retval = -1;
00335                     break;
00336 /* INTERNAL_ERROR_EXCL_STOP */
00337                 }
00338                 term[3] = i;
00339                 if ( term[5] < 0 ) {    /* Fill with spectator */
00340                     fromspectator = -term[5];
00341                     PUTZERO(AM.SpectatorFiles[fromspectator-1].readpos);
00342                     term[5] = AC.cbufnum;
00343                 }
00344                 else fromspectator = 0;
00345                 if ( AR.outtohide ) {
00346                     SeekScratch(AR.hidefile,&position);
00347                     e->onfile = position;
00348                     if ( PutOut(BHEAD term,&position,AR.hidefile,0) < 0 ) goto ProcErr;
00349                 }
00350                 else {
00351                     SeekScratch(AR.outfile,&position);
00352                     e->onfile = position;
00353                     if ( PutOut(BHEAD term,&position,AR.outfile,0) < 0 ) goto ProcErr;
00354                 }
00355                 AR.DeferFlag = AC.ComDefer;
00356                 AR.Eside = RHSIDE;
00357                 if ( ( e->vflags & ISFACTORIZED ) != 0 ) {
00358                     AR.BracketOn = 1;
00359                     AT.BrackBuf = AM.BracketFactors;
00360                     AT.bracketindexflag = 1;
00361                 }
00362                 if ( AT.bracketindexflag > 0 ) OpenBracketIndex(i);
00363 #ifdef WITHPTHREADS
00364                 if ( AS.MultiThreaded && AC.mparallelflag == PARALLELFLAG ) {
00365                     if ( ThreadsProcessor(e,LastExpression,fromspectator) ) {
00366 /* INTERNAL_ERROR_EXCL_START */
00367                         MesPrint("!>Error in ThreadsProcessor");
00368                         goto ProcErr;
00369 /* INTERNAL_ERROR_EXCL_STOP */
00370                     }
00371                     if ( AR.outtohide ) {
00372                         AR.outfile = oldoutfile;
00373                         AR.hidefile->POfull = AR.hidefile->POfill;
00374                     }
00375                 }
00376                 else
00377 #endif
00378                 {
00379                     NewSort(BHEAD0);
00380                     AR.MaxDum = AM.IndDum;
00381                     AN.ninterms = 0;
00382                     for(;;) {
00383                      if ( fromspectator ) size = GetFromSpectator(term,fromspectator-1);
00384                      else                 size = GetTerm(BHEAD term);
00385                      if ( size <= 0 ) break;
00386                      SeekScratch(curfile,&position);
00387                      if ( ( e->vflags & ISFACTORIZED ) != 0 && term[1] == HAAKJE ) {
00388                       StoreTerm(BHEAD term);
00389                      }
00390                      else {
00391                       AN.ninterms++; dd = AN.deferskipped;
00392                       if ( AC.CollectFun && *term <= (AM.MaxTer/(2*(LONG)(sizeof(WORD)))) ) {
00393                         if ( GetMoreTerms(term) < 0 ) {
00394                           LowerSortLevel(); goto ProcErr;
00395                         }
```

```
00396                       SeekScratch(curfile,&position);
00397                     }
00398                     AT.WorkPointer = term + *term;
00399                     AN.RepPoint = AT.RepCount + 1;
00400                     if ( AR.DeferFlag ) {
00401                       AN.IndDum = Expressions[AR.CurExpr].numdummies + AM.IndDum;
00402                       AR.CurDum = AN.IndDum;
00403                     }
00404                     else {
00405                       AN.IndDum = AM.IndDum;
00406                       AR.CurDum = ReNumber(BHEAD term);
00407                     }
00408                     if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
00409                     if ( AN.ncmod ) {
00410                       if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
00411                       else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
00412                     }
00413                     else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
00414                     if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
00415                       && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
00416                       PolyFunClean(BHEAD term);
00417                     }
00418                     if ( Generator(BHEAD term,0) ) {
00419                       LowerSortLevel(); goto ProcErr;
00420                     }
00421                     AN.ninterms += dd;
00422                    }
00423                    SetScratch(curfile,&position);
00424                    if ( AR.GetFile == 2 ) {
00425                       AR.InHiBuf = (curfile->POfull-curfile->PObuffer)
00426                           -DIFBASE(position,curfile->POposition)/sizeof(WORD);
00427                    }
00428                    else {
00429                       AR.InInBuf = (curfile->POfull-curfile->PObuffer)
00430                           -DIFBASE(position,curfile->POposition)/sizeof(WORD);
00431                    }
00432                   }
00433                   AN.ninterms += dd;
00434                   if ( LastExpression ) {
00435                       UpdateMaxSize();
00436                       if ( AR.infile->handle >= 0 ) {
00437                           CloseFile(AR.infile->handle);
00438                           AR.infile->handle = -1;
00439                           remove(AR.infile->name);
00440                           PUTZERO(AR.infile->POposition);
00441                       }
00442                       AR.infile->POfill = AR.infile->POfull = AR.infile->PObuffer;
00443                   }
00444                   if ( AR.outtohide ) AR.outfile = AR.hidefile;
00445                   if ( EndSort(BHEAD AM.S0->sBuffer,0) < 0 ) goto ProcErr;
00446                   if ( AR.outtohide ) {
00447                       AR.outfile = oldoutfile;
00448                       AR.hidefile->POfull = AR.hidefile->POfill;
00449                   }
00450                   e->numdummies = AR.MaxDum - AM.IndDum;
00451                   UpdateMaxSize();
00452                 }
00453                 AR.BracketOn = oldBracketOn;
00454                 AT.BrackBuf = oldBrackBuf;
00455                 if ( ( e->vflags & TOBEFACTORED ) != 0 ) {
00456                     poly_factorize_expression(e);
00457                 }
00458                 else if ( ( ( e->vflags & TOBEUNFACTORED ) != 0 )
00459                  && ( ( e->vflags & ISFACTORIZED ) != 0 ) ) {
00460                     poly_unfactorize_expression(e);
00461                 }
00462                 AT.bracketindexflag = oldbracketindexflag;
00463                 if ( AM.S0->TermsLeft )   e->vflags &= ~ISZERO;
00464                 else                      e->vflags |= ISZERO;
00465                 if ( AR.expchanged == 0 ) e->vflags |= ISUNMODIFIED;
00466                 if ( AM.S0->TermsLeft ) AR.expflags |= ISZERO;
00467                 if ( AR.expchanged )    AR.expflags |= ISUNMODIFIED;
00468                 AR.GetFile = 0;
00469                 AR.outtohide = 0;
00470 /*[20oct2009 mt]:*/
00471 #ifdef WITHMPI
00472                 }
00473 #endif
00474 #ifdef WITHPTHREADS
00475                 if ( e->status == INTOHIDELEXPRESSION ||
00476                      e->status == INTOHIDEGEXPRESSION ) {
00477                     SetHideFiles();
00478                 }
00479 #endif
00480                 break;
00481             case SKIPLEXPRESSION:
00482             case SKIPGEXPRESSION:
```

```
00483 /*
00484                 This can be greatly improved of course by file-to-file copy.
00485 */
00486 #ifdef WITHMPI
00487                 if ( PF.me != MASTER ) break;
00488 #endif
00489                 AR.GetFile = 0;
00490                 SetScratch(AR.infile,&(e->onfile));
00491                 if ( GetTerm(BHEAD term) <= 0 ) {
00492 #ifdef HIDEDEBUG
00493                     MesPrint("Error condition 1b");
00494                     ExprStatus(e);
00495 #endif
00496 /* INTERNAL_ERROR_EXCL_START */
00497                     MesPrint("!>(3) Expression %d has problems in scratchfile",i);
00498                     retval = -1;
00499                     break;
00500 /* INTERNAL_ERROR_EXCL_STOP */
00501                 }
00502                 term[3] = i;
00503                 AR.DeferFlag = 0;
00504                 SeekScratch(AR.outfile,&position);
00505                 e->onfile = position;
00506                 *AM.S0->sBuffer = 0; firstterm = -1;
00507                 do {
00508                     WORD *oldipointer = AR.CompressPointer;
00509                     WORD *comprtop = AR.ComprTop;
00510                     AR.ComprTop = AM.S0->sTop;
00511                     AR.CompressPointer = AM.S0->sBuffer;
00512                     if ( firstterm > 0 ) {
00513                         if ( PutOut(BHEAD term,&position,AR.outfile,1) < 0 ) goto ProcErr;
00514                     }
00515                     else if ( firstterm < 0 ) {
00516                         if ( PutOut(BHEAD term,&position,AR.outfile,0) < 0 ) goto ProcErr;
00517                         firstterm++;
00518                     }
00519                     else {
00520                         if ( PutOut(BHEAD term,&position,AR.outfile,-1) < 0 ) goto ProcErr;
00521                         firstterm++;
00522                     }
00523                     AR.CompressPointer = oldipointer;
00524                     AR.ComprTop = comprtop;
00525                 } while ( GetTerm(BHEAD term) );
00526                 if ( FlushOut(&position,AR.outfile,1) ) goto ProcErr;
00527                 UpdateMaxSize();
00528                 break;
00529             case HIDELEXPRESSION:
00530             case HIDEGEXPRESSION:
00531 #ifdef WITHMPI
00532                 if ( PF.me != MASTER ) break;
00533 #endif
00534                 AR.GetFile = 0;
00535                 SetScratch(AR.infile,&(e->onfile));
00536                 if ( GetTerm(BHEAD term) <= 0 ) {
00537 #ifdef HIDEDEBUG
00538                     MesPrint("Error condition 1c");
00539                     ExprStatus(e);
00540 #endif
00541 /* INTERNAL_ERROR_EXCL_START */
00542                     MesPrint("!>(4) Expression %d has problems in scratchfile",i);
00543                     retval = -1;
00544                     break;
00545 /* INTERNAL_ERROR_EXCL_STOP */
00546                 }
00547                 term[3] = i;
00548                 AR.DeferFlag = 0;
00549                 SetEndHScratch(AR.hidefile,&position);
00550                 e->onfile = position;
00551 #ifdef HIDEDEBUG
00552                 if ( AR.hidefile->handle >= 0 ) {
00553                     POSITION possize,pos;
00554                     PUTZERO(possize);
00555                     PUTZERO(pos);
00556                     SeekFile(AR.hidefile->handle,&pos,SEEK_CUR);
00557                     SeekFile(AR.hidefile->handle,&possize,SEEK_END);
00558                     SeekFile(AR.hidefile->handle,&pos,SEEK_SET);
00559                     MesPrint("Processor Hide1: filesize(th) = %12p, filesize(ex) = %12p",&(position),
00560                             &(possize));
00561                     MesPrint("    in buffer:
      %l",(AR.hidefile->POfill-AR.hidefile->PObuffer)*sizeof(WORD));
00562                 }
00563 #endif
00564                 *AM.S0->sBuffer = 0; firstterm = -1;
00565                 cbo = cpo = AM.S0->sBuffer;
00566                 do {
00567                     WORD *oldipointer = AR.CompressPointer;
00568                     WORD *oldibuffer = AR.CompressBuffer;
```

```
00569                     WORD *comprtop = AR.ComprTop;
00570                     AR.ComprTop = AM.S0->sTop;
00571                     AR.CompressPointer = cpo;
00572                     AR.CompressBuffer = cbo;
00573                     if ( firstterm > 0 ) {
00574                         if ( PutOut(BHEAD term,&position,AR.hidefile,1) < 0 ) goto ProcErr;
00575                     }
00576                     else if ( firstterm < 0 ) {
00577                         if ( PutOut(BHEAD term,&position,AR.hidefile,0) < 0 ) goto ProcErr;
00578                         firstterm++;
00579                     }
00580                     else {
00581                         if ( PutOut(BHEAD term,&position,AR.hidefile,-1) < 0 ) goto ProcErr;
00582                         firstterm++;
00583                     }
00584                     cpo = AR.CompressPointer;
00585                     cbo = AR.CompressBuffer;
00586                     AR.CompressPointer = oldipointer;
00587                     AR.CompressBuffer = oldibuffer;
00588                     AR.ComprTop = comprtop;
00589                 } while ( GetTerm(BHEAD term) );
00590 #ifdef HIDEDEBUG
00591                 if ( AR.hidefile->handle >= 0 ) {
00592                     POSITION possize,pos;
00593                     PUTZERO(possize);
00594                     PUTZERO(pos);
00595                     SeekFile(AR.hidefile->handle,&pos,SEEK_CUR);
00596                     SeekFile(AR.hidefile->handle,&possize,SEEK_END);
00597                     SeekFile(AR.hidefile->handle,&pos,SEEK_SET);
00598                     MesPrint("Processor Hide2: filesize(th) = %12p, filesize(ex) = %12p",&(position),
00599                             &(possize));
00600                     MesPrint("    in buffer:
      %l",(AR.hidefile->POfill-AR.hidefile->PObuffer)*sizeof(WORD));
00601                 }
00602 #endif
00603                 if ( FlushOut(&position,AR.hidefile,1) ) goto ProcErr;
00604                 AR.hidefile->POfull = AR.hidefile->POfill;
00605 #ifdef HIDEDEBUG
00606                 if ( AR.hidefile->handle >= 0 ) {
00607                     POSITION possize,pos;
00608                     PUTZERO(possize);
00609                     PUTZERO(pos);
00610                     SeekFile(AR.hidefile->handle,&pos,SEEK_CUR);
00611                     SeekFile(AR.hidefile->handle,&possize,SEEK_END);
00612                     SeekFile(AR.hidefile->handle,&pos,SEEK_SET);
00613                     MesPrint("Processor Hide3: filesize(th) = %12p, filesize(ex) = %12p",&(position),
00614                             &(possize));
00615                     MesPrint("    in buffer:
      %l",(AR.hidefile->POfill-AR.hidefile->PObuffer)*sizeof(WORD));
00616                 }
00617 #endif
00618 /*
00619                 Because we direct the e->onfile already to the hide file, we
00620                 need to change the status of the expression. Otherwise the use
00621                 of parts (or the whole) of the expression looks in the infile
00622                 while the position is that of the hide file.
00623                 We choose to get everything from the hide file. On average that
00624                 should give least file activity.
00625 */
00626                 if ( e->status == HIDELEXPRESSION ) {
00627                     e->status = HIDDENLEXPRESSION;
00628                     AS.OldOnFile[i] = e->onfile;
00629                     AS.OldNumFactors[i] = Expressions[i].numfactors;
00630                 }
00631                 if ( e->status == HIDEGEXPRESSION ) {
00632                     e->status = HIDDENGEXPRESSION;
00633                     AS.OldOnFile[i] = e->onfile;
00634                     AS.OldNumFactors[i] = Expressions[i].numfactors;
00635                 }
00636 #ifdef WITHPTHREADS
00637                 SetHideFiles();
00638 #endif
00639                 UpdateMaxSize();
00640                 break;
00641             case DROPPEDEXPRESSION:
00642             case DROPLEXPRESSION:
00643             case DROPGEXPRESSION:
00644             case DROPHLEXPRESSION:
00645             case DROPHGEXPRESSION:
00646             case STOREDEXPRESSION:
00647             case HIDDENLEXPRESSION:
00648             case HIDDENGEXPRESSION:
00649             case SPECTATOREXPRESSION:
00650             default:
00651                 break;
00652         }
00653         }
```

```
00654         AR.KeptInHold = 0;
00655     }
00656     AR.DeferFlag = 0;
00657     AT.WorkPointer = term;
00658 #ifdef HIDEDEBUG
00659     MesPrint("Status at the end of Processor (HideLevel = %d)",AC.HideLevel);
00660     MesPrint("File %s POfill %l POfull %l POsize %l", AR.infile->name   ,AR.infile->POfill
      -AR.infile->PObuffer  ,AR.infile->POfull  -AR.infile->PObuffer  ,AR.infile->POsize/sizeof(WORD)  );
00661     MesPrint("File %s POfill %l POfull %l POsize %l", AR.outfile->name  ,AR.outfile->POfill
      -AR.outfile->PObuffer ,AR.outfile->POfull -AR.outfile->PObuffer ,AR.outfile->POsize/sizeof(WORD) );
00662     MesPrint("File %s POfill %l POfull %l POsize %l", AR.hidefile->name
      ,AR.hidefile->POfill-AR.hidefile->PObuffer,AR.hidefile->POfull-AR.hidefile->PObuffer,AR.hidefile->POsize/sizeof(WORD));
00663     for ( i = 0; i < NumExpressions; i++ ) {
00664         e = Expressions+i;
00665         ExprStatus(e);
00666     }
00667 #endif
00668     return(retval);
00669 ProcErr:
00670     AT.WorkPointer = term;
00671     if ( AM.tracebackflag ) MesCall("Processor");
00672     return(-1);
00673 }
00674 /*
00675         #] Processor :
00676         #[ TestSub :            WORD TestSub(term,level)
00677 */
00701 #define DONE(x) { retvalue = x; goto Done; }
00702
00703 WORD TestSub(PHEAD WORD *term, WORD level)
00704 {
00705     GETBIDENTITY
00706     WORD *m, *t, *r, retvalue = 0, funflag, j, oldncmod, nexpr, *Tpattern = 0;
00707     WORD *stop, *t1, *t2, funnum, wilds, tbufnum, stilldirty = 0;
00708     NESTING n;
00709     CBUF *C = cbuf+AT.ebufnum;
00710     LONG isp, i;
00711     TABLES T;
00712     COMPARE oldcompareroutine = (COMPARE)(AR.CompareRoutine);
00713     WORD oldsorttype = AR.SortType;
00714 ReStart:
00715     tbufnum = 0; i = 0; retvalue = 0;
00716     AT.TMbuff = AM.rbufnum;
00717     funflag = 0;
00718     t = term;
00719     r = t + *t - 1;
00720     m = r - ABS(*r) + 1;
00721     t++;
00722     if ( t < m ) do {
00723         if ( *t == SUBEXPRESSION ) {
00724             /*
00725                 Subexpression encountered
00726                 There may be more than one.
00727                 The old strategy was to take the last.
00728                 A newer strategy was to take the lowest power first.
00729                 The current strategy is that we compute the number of terms
00730                 generated by this subexpression and take the minimum of that.
00731             */
00732
00733 #ifdef WHICHSUBEXPRESSION
00734
00735             WORD *tmin = t, AN.nbino;
00736 /*          LONG minval = MAXLONG; */
00737             LONG minval = -1;
00738             LONG mm, mnum1 = 1;
00739             if ( AN.BinoScrat == 0 ) {
00740                 AN.BinoScrat = (UWORD *)Malloc1((AM.MaxTal+2)*sizeof(UWORD),"GetBinoScrat");
00741             }
00742 #endif
00743             if ( t[3] ) {
00744                 r = t + t[1];
00745                 while ( AN.subsubveto == 0 &&
00746                             *r == SUBEXPRESSION && r < m && r[3] ) {
00747 #ifdef WHICHSUBEXPRESSION
00748                     mnum1++;
00749 #endif
00750                     if ( r[1] == t[1] && r[2] == t[2] && r[4] == t[4] ) {
00751                         j = t[1] - SUBEXPSIZE;
00752                         t1 = t + SUBEXPSIZE;
00753                         t2 = r + SUBEXPSIZE;
00754                         while ( j > 0 && *t1++ == *t2++ ) j--;
00755                         if ( j <= 0 ) {
00756                             t[3] += r[3];
00757                             if ( t[3] == 0 ) {
00758                                 t1 = r + r[1];
00759                                 t2 = term + *term;
00760                                 *term -= r[1]+t[1];
```

```
00761                                   r = t;
00762                                   while ( t1 < t2 ) *r++ = *t1++;
00763                                   goto ReStart;
00764                               }
00765                               else {
00766                                   t1 = r + r[1];
00767                                   t2 = term + *term;
00768                                   *term -= r[1];
00769                                   m -= r[1];
00770                                   while ( t1 < t2 ) *r++ = *t1++;
00771                                   r = t;
00772                               }
00773                           }
00774                       }
00775 #ifdef WHICHSUBEXPRESSION
00776
00777                       else if ( t[2] >= 0 ) {
00778 /*
00779                           Compute Binom(numterms+power-1,power-1)
00780                           We need potentially long arithmetic.
00781                           That is why we had to allocate AN.BinoScrat
00782 */
00783                           if ( AN.last1 == t[3] && AN.last2 == cbuf[t[4]].NumTerms[t[2]] + t[3] - 1 ) {
00784                               if ( AN.last3 > minval ) {
00785                                   minval = AN.last3; tmin = t;
00786                               }
00787                           }
00788                           else {
00789                           AN.last1 = t[3]; mm = AN.last2 = cbuf[t[4]].NumTerms[t[2]] + t[3] - 1;
00790                           if ( t[3] == 1 ) {
00791                               if ( mm > minval ) {
00792                                   minval = mm; tmin = t;
00793                               }
00794                           }
00795                           else if ( t[3] > 0 ) {
00796                               if ( mm > MAXPOSITIVE ) goto TooMuch;
00797                               GetBinom(AN.BinoScrat,&AN.nbino,(WORD)mm,t[3]);
00798                               if ( AN.nbino > 2 ) goto TooMuch;
00799                               if ( AN.nbino == 2 ) {
00800                                   mm = AN.BinoScrat[1];
00801                                   mm = ( mm « BITSINWORD ) + AN.BinoScrat[0];
00802                               }
00803                               else if ( AN.nbino == 1 ) mm = AN.BinoScrat[0];
00804                               else mm = 0;
00805                               if ( mm > minval ) {
00806                                   minval = mm; tmin = t;
00807                               }
00808                           }
00809                           AN.last3 = mm;
00810                           }
00811                       }
00812 #endif
00813                       t = r;
00814                       r += r[1];
00815                   }
00816 #ifdef WHICHSUBEXPRESSION
00817                   if ( mnum1 > 1 && t[2] >= 0 ) {
00818 /*
00819                       To keep the flowcontrol simple we duplicate some code here
00820 */
00821                       if ( AN.last1 == t[3] && AN.last2 == cbuf[t[4]].NumTerms[t[2]] + t[3] - 1 ) {
00822                           if ( AN.last3 > minval ) {
00823                               minval = AN.last3; tmin = t;
00824                           }
00825                       }
00826                       else {
00827                       AN.last1 = t[3]; mm = AN.last2 = cbuf[t[4]].NumTerms[t[2]] + t[3] - 1;
00828                       if ( t[3] == 1 ) {
00829                           if ( mm > minval ) {
00830                               minval = mm; tmin = t;
00831                           }
00832                       }
00833                       else if ( t[3] > 0 ) {
00834                           if ( mm > MAXPOSITIVE ) {
00835 /*
00836                               We will generate more terms than we can count
00837 */
00838 TooMuch:;
00839 /* INTERNAL_ERROR_EXCL_START */
00840                               MLOCK(ErrorMessageLock);
00841                               MesPrint("!>Attempt to generate more terms than FORM can count");
00842                               MUNLOCK(ErrorMessageLock);
00843                               Terminate(-1);
00844 /* INTERNAL_ERROR_EXCL_STOP */
00845                           }
00846                           GetBinom(AN.BinoScrat,&AN.nbino,(WORD)mm,t[3]);
00847                           if ( AN.nbino > 2 ) goto TooMuch;
```

```
00848                         if ( AN.nbino == 2 ) {
00849                             mm = AN.BinoScrat[1];
00850                             mm = ( mm « BITSINWORD ) + AN.BinoScrat[0];
00851                         }
00852                         else if ( AN.nbino == 1 ) mm = AN.BinoScrat[0];
00853                         else mm = 0;
00854                         if ( mm > minval ) {
00855                             minval = mm; tmin = t;
00856                         }
00857                     }
00858                     AN.last3 = mm;
00859                     }
00860                 }
00861                 t = tmin;
00862 #endif
00863 /*              AR.TePos = 0;  */
00864                 AR.TePos = WORDDIF(t,term);
00865                 AT.TMbuff = t[4];
00866                 if ( t[4] == AM.dbufnum && (t+t[1]) < m && t[t[1]] == DOLLAREXPR2 ) {
00867                     if ( t[t[1]+2] < 0 ) AT.TMdolfac = -t[t[1]+2];
00868                     else {  /* resolve the element number */
00869                         AT.TMdolfac = GetDolNum(BHEAD t+t[1],m)+1;
00870                     }
00871                 }
00872                 else AT.TMdolfac = 0;
00873                 if ( t[3] < 0 ) {
00874                     AN.TeInFun = 1;
00875                     AR.TePos = WORDDIF(t,term);
00876                     DONE(t[2])
00877                 }
00878                 else {
00879                     AN.TeInFun = 0;
00880                     AN.TeSuOut = t[3];
00881                 }
00882                 if ( t[2] < 0 ) {
00883                     AN.TeSuOut = -t[3];
00884                     DONE(-t[2])
00885                 }
00886                 DONE(t[2])
00887             }
00888         }
00889         else if ( *t == EXPRESSION ) {
00890             WORD *toTMaddr;
00891             i = -t[2] - 1;
00892             if ( t[3] < 0 ) {
00893                 AN.TeInFun = 1;
00894                 AR.TePos = WORDDIF(t,term);
00895                 DONE(i)
00896             }
00897             nexpr = t[3];
00898             toTMaddr = m = AT.WorkPointer;
00899             AN.Frozen = 0;
00900 /*
00901             We have to be very careful with respect to setting variables
00902             like AN.TeInFun, because we may still call Generator and that
00903             may change those variables. That is why we set them at the
00904             last moment only.
00905 */
00906             j = t[1];
00907             AT.WorkPointer += j;
00908             r = t;
00909             NCOPY(m,r,j);
00910             r = t + t[1];
00911             t += SUBEXPSIZE;
00912             while ( t < r ) {
00913                 if ( *t == FROMBRAC ) {
00914                     WORD *ttstop,*tttstop;
00915 /*
00916                     Note: Convention is that wildcards are done
00917                     after the expression has been picked up. So
00918                     no wildcard substitutions are needed here.
00919 */
00920                     t += 2;
00921                     AN.Frozen = m = AT.WorkPointer;
00922 /*
00923                     We should check now for subexpressions and if necessary
00924                     we substitute them. Keep in mind: only one term allowed!
00925
00926                     In retrospect (26-jan-2010): take also functions that
00927                     have a dirty flag on
00928 */
00929                     j = *t; tttstop = t + j;
00930                     GETSTOP(t,ttstop);
00931                     *m++ = j; t++;
00932                     while ( t < ttstop ) {
00933                         if ( *t == SUBEXPRESSION ) break;
00934                         if ( *t >= FUNCTION && ( ( t[2] & DIRTYFLAG ) == DIRTYFLAG ) ) break;
```

```
00935                       j = t[1]; NCOPY(m,t,j);
00936                   }
00937                   if ( t < ttstop ) {
00938 /*
00939                       We ran into a subexpression or a function with a
00940                       'dirty' argument. It could also be a $ or
00941                       just e[(a^2)*b]. In all cases we should evaluate
00942 */
00943                       while ( t < tttstop ) *m++ = *t++;
00944                       *AT.WorkPointer = m-AT.WorkPointer;
00945                       m = AT.WorkPointer;
00946                       AT.WorkPointer = m + *m;
00947                       NewSort(BHEAD0);
00948                       if ( Generator(BHEAD m,AR.Cnumlhs) ) {
00949                           LowerSortLevel(); goto EndTest;
00950                       }
00951                       if ( EndSort(BHEAD m,0) < 0 ) goto EndTest;
00952                       AN.Frozen = m;
00953                       if ( *m == 0 ) {
00954                           *m++ = 4; *m++ = 1; *m++ = 1; *m++ = 3;
00955                       }
00956                       else if ( m[*m] != 0 ) {
00957                           MLOCK(ErrorMessageLock);
00958                           MesPrint("Bracket specification in expression should be one single term");
00959                           MUNLOCK(ErrorMessageLock);
00960                           Terminate(-1);
00961                       }
00962                       else {
00963                           m += *m;
00964                           m -= ABS(m[-1]);
00965                           *m++ = 1; *m++ = 1; *m++ = 3;
00966                           *AN.Frozen = m - AN.Frozen;
00967                       }
00968                   }
00969                   else {
00970                       while ( t < tttstop ) *m++ = *t++;
00971                       *AT.WorkPointer = m-AT.WorkPointer;
00972                       m = AT.WorkPointer;
00973                       AT.WorkPointer = m + *m;
00974                       if ( Normalize(BHEAD m) ) {
00975 /* INTERNAL_ERROR_EXCL_START */
00976                           MLOCK(ErrorMessageLock);
00977                           MesPrint("!>Error while picking up contents of bracket");
00978                           MUNLOCK(ErrorMessageLock);
00979                           Terminate(-1);
00980 /* INTERNAL_ERROR_EXCL_STOP */
00981                       }
00982                       if ( !*m ) {
00983                           *m++ = 4; *m++ = 1; *m++ = 1; *m++ = 3;
00984                       }
00985                       else m += *m;
00986                   }
00987                   AT.WorkPointer = m;
00988                   break;
00989               }
00990               t += t[1];
00991           }
00992           AN.TeInFun = 0;
00993           AR.TePos = 0;
00994           AN.TeSuOut = nexpr;
00995           AT.TMaddr = toTMaddr;
00996           DONE(i)
00997       }
00998       else if ( *t >= FUNCTION ) {
00999           if ( t[0] == EXPONENT ) {
01000               if ( t[1] == FUNHEAD+4 && t[FUNHEAD] == -SYMBOL &&
01001               t[FUNHEAD+2] == -SNUMBER && t[FUNHEAD+3] < MAXPOWER
01002               && t[FUNHEAD+3] > -MAXPOWER ) {
01003                   t[0] = SYMBOL;
01004                   t[1] = 4;
01005                   t[2] = t[FUNHEAD+1];
01006                   t[3] = t[FUNHEAD+3];
01007                   r = term + *term;
01008                   m = t + FUNHEAD+4;
01009                   t += 4;
01010                   while ( m < r ) *t++ = *m++;
01011                   *term = WORDDIF(t,term);
01012                   goto ReStart;
01013               }
01014               else if ( t[1] == FUNHEAD+ARGHEAD+11 && t[FUNHEAD] == ARGHEAD+9
01015                && t[FUNHEAD+ARGHEAD] == 9 && t[FUNHEAD+ARGHEAD+1] == DOTPRODUCT
01016                && t[FUNHEAD+ARGHEAD+8] == 3
01017                && t[FUNHEAD+ARGHEAD+7] == 1
01018                && t[FUNHEAD+ARGHEAD+6] == 1
01019                && t[FUNHEAD+ARGHEAD+5] == 1
01020                && t[FUNHEAD+ARGHEAD+9] == -SNUMBER
01021                && t[FUNHEAD+ARGHEAD+10] < MAXPOWER
```

```
01022                  && t[FUNHEAD+ARGHEAD+10] > -MAXPOWER ) {
01023                     t[0] = DOTPRODUCT;
01024                     t[1] = 5;
01025                     t[2] = t[FUNHEAD+ARGHEAD+3];
01026                     t[3] = t[FUNHEAD+ARGHEAD+4];
01027                     t[4] = t[FUNHEAD+ARGHEAD+10];
01028                     r = term + *term;
01029                     m = t + FUNHEAD+ARGHEAD+11;
01030                     t += 5;
01031                     while ( m < r ) *t++ = *m++;
01032                     *term = WORDDIF(t,term);
01033                     goto ReStart;
01034                  }
01035               }
01036               funnum = *t;
01037               if ( *t >= FUNCTION + WILDOFFSET ) funnum -= WILDOFFSET;
01038               if ( *t == EXPONENT ) {
01039 /*
01040                   Test whether the second argument is an integer
01041 */
01042                   r = t+FUNHEAD;
01043                   NEXTARG(r)
01044                   if ( *r == -SNUMBER && r[1] < MAXPOWER && r+2 == t+t[1] &&
01045                   t[FUNHEAD] > -FUNCTION && ( t[FUNHEAD] != -SNUMBER
01046                   || t[FUNHEAD+1] != 0 ) && t[FUNHEAD] != ARGHEAD ) {
01047                     if ( r[1] == 0 ) {
01048                       if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] == 0 ) {
01049                           MLOCK(ErrorMessageLock);
01050                           MesPrint("Encountered 0^0. Fatal error.");
01051                           MUNLOCK(ErrorMessageLock);
01052                           SETERROR(-1);
01053                       }
01054                       *t = DUMMYFUN;
01055 /*
01056                           Now mark it clean to avoid further interference.
01057                           Normalize will remove this object.
01058 */
01059                       t[2] = 0;
01060                     }
01061                     else {
01062                       /* Note that the case 0^ is treated in Normalize */
01063
01064                       t1 = AddRHS(AT.ebufnum,1);
01065                       m = t + FUNHEAD;
01066                       if ( *m > 0 ) {
01067                           m += ARGHEAD;
01068                           i = t[FUNHEAD] - ARGHEAD;
01069                           while ( (t1 + i + 10) > C->Top )
01070                               t1 = DoubleCbuffer(AT.ebufnum,t1,9);
01071                           while ( --i >= 0 ) *t1++ = *m++;
01072                       }
01073                       else {
01074                           if ( (t1 + 20) > C->Top )
01075                               t1 = DoubleCbuffer(AT.ebufnum,t1,10);
01076                           ToGeneral(m,t1,1);
01077                           t1 += *t1;
01078                       }
01079                       *t1++ = 0;
01080                       C->rhs[C->numrhs+1] = t1;
01081                       C->Pointer = t1;
01082
01083                       /* No provisions yet for commuting objects */
01084
01085                       C->CanCommu[C->numrhs] = 1;
01086                       *t++ = SUBEXPRESSION;
01087                       *t++ = SUBEXPSIZE;
01088                       *t++ = C->numrhs;
01089                       *t++ = r[1];
01090                       *t++ = AT.ebufnum;
01091 #if SUBEXPSIZE > 5
01092 Important: we may not have enough spots here
01093 #endif
01094                       FILLSUB(t)  /* Important: We have maybe only 5 spots! */
01095                       r += 2;
01096                       m = term + *term;
01097                       do { *t++ = *r++; } while ( r < m );
01098                       *term -= WORDDIF(r,t);
01099                       goto ReStart;
01100                     }
01101                   }
01102               }
01103               else if ( *t == SUMF1 || *t == SUMF2 ) {
01104 /*
01105                   What we are looking for is:
01106                   1-st argument:  Single symbol or index.
01107                   2-nd argument:  Number.
01108                   3-rd argument:  Number.
```

```
01109                 (4-th argument):Number.
01110                 One more argument.
01111                 This would activate the summation procedure.
01112                 Note that the initiated recursion here can be done
01113                 without upsetting the regular procedures.
01114 */
01115                 WORD *tstop, lcounter, lcmin, lcmax, lcinc;
01116                 tstop = t + t[1];
01117                 r = t+FUNHEAD;
01118                 if ( r+6 < tstop && r[2] == -SNUMBER && r[4] == -SNUMBER
01119                 && ( r[0] == -SYMBOL )
01120                 || ( r[0] == -INDEX && r[1] >= AM.OffsetIndex
01121                 && r[3] >= 0 && r[3] < AM.OffsetIndex
01122                 && r[5] >= 0 && r[5] < AM.OffsetIndex ) ) ) {
01123                     lcounter = r[0] == -INDEX ? -r[1]: r[1]; /* The loop counter */
01124                     lcmin = r[3];
01125                     lcmax = r[5];
01126                     r += 6;
01127                     if ( *r == -SNUMBER && r+2 < tstop ) {
01128                         lcinc = r[1];
01129                         r += 2;
01130                     }
01131                     else lcinc = 1;
01132                     if ( r < tstop && ( ( *r > 0 && (r+*r) == tstop )
01133                     || ( *r <= -FUNCTION && r+1 == tstop )
01134                     || ( *r > -FUNCTION && *r < 0 && r+2 == tstop ) ) ) {
01135                         m = AddRHS(AT.ebufnum,1);
01136                         if ( *r > 0 ) {
01137                             i = *r - ARGHEAD;
01138                             r += ARGHEAD;
01139                             while ( (m + i + 10) > C->Top )
01140                                 m = DoubleCbuffer(AT.ebufnum,m,11);
01141                             while ( --i >= 0 ) *m++ = *r++;
01142                         }
01143                         else {
01144                             while ( (m + 20) > C->Top )
01145                                 m = DoubleCbuffer(AT.ebufnum,m,12);
01146                             ToGeneral(r,m,1);
01147                             m += *m;
01148                         }
01149                         *m++ = 0;
01150                         C->rhs[C->numrhs+1] = m;
01151                         C->Pointer = m;
01152                         m = AT.TMout;
01153                         *m++ = 6;
01154                         if ( *t == SUMF1 ) *m++ = SUMNUM1;
01155                         else               *m++ = SUMNUM2;
01156                         *m++ = lcounter;
01157                         *m++ = lcmin;
01158                         *m++ = lcmax;
01159                         *m++ = lcinc;
01160                         m = t + t[1];
01161                         r = C->rhs[C->numrhs];
01162 /*
01163                         Test now if the argument was already evaluated.
01164                         In that case it needs a new subexpression prototype.
01165                         In either case we replace the function now by a
01166                         subexpression prototype.
01167 */
01168                         if ( *r >= (SUBEXPSIZE+4)
01169                         && ABS(*(r+*r-1)) < (*r - 1)
01170                         && r[1] == SUBEXPRESSION ) {
01171                             r++;
01172                             i = r[1] - 5;
01173                             *t++ = *r++; *t++ = *r++; *t++ = C->numrhs;
01174                             r++; *t++ = *r++; *t++ = AT.ebufnum; r++;
01175                             while ( --i >= 0 ) *t++ = *r++;
01176                         }
01177                         else {
01178                             *t++ = SUBEXPRESSION;
01179                             *t++ = 4+SUBEXPSIZE;
01180                             *t++ = C->numrhs;
01181                             *t++ = 1;
01182                             *t++ = AT.ebufnum;
01183                             FILLSUB(t)
01184                             if ( lcounter < 0 ) {
01185                                 *t++ = INDTOIND;
01186                                 *t++ = 4;
01187                                 *t++ = -lcounter;
01188                             }
01189                             else {
01190                                 *t++ = SYMTONUM;
01191                                 *t++ = 4;
01192                                 *t++ = lcounter;
01193                             }
01194                             *t++ = lcmin;
01195                         }
```

```
01196                       t2 = term + *term;
01197                       while ( m < t2 ) *t++ = *m++;
01198                       *term = WORDDIF(t,term);
01199                       AN.TeInFun = -C->numrhs;
01200                       AR.TePos = 0;
01201                       AN.TeSuOut = 0;
01202                       AT.TMbuff = AT.ebufnum;
01203                       DONE(C->numrhs)
01204                   }
01205               }
01206           }
01207           else if ( *t == TOPOLOGIES ) {
01208               MesPrint("&The topologies_ function was removed in FORM 5.0.");
01209               MesPrint("&See the TopologiesOnly_ option of diagrams_.");
01210               Terminate(-1);
01211           }
01212           else if ( *t == DIAGRAMS ) {
01213 /*
01214               Syntax:
01215               diagrams_(model,setinparticles,setoutparticles,
01216                   setextmomenta,setintmomenta,couplings or loops)
01217 */
01218               if ( AC.nummodels > 0 ) { /* No model, no diagrams */
01219                 t1 = t+FUNHEAD; t2 = t+t[1];
01220                 if (
01221                 t1[0] == -SETSET && Sets[t1[1]].type == CMODEL &&
01222                 t1[2] == -SETSET && ( Sets[t1[3]].type == CFUNCTION
01223                   || ( Sets[t1[3]].type == ANYTYPE && ( Sets[t1[3]].first == Sets[t1[3]].last ) ) )
      &&
01224                 t1[4] == -SETSET && ( Sets[t1[5]].type == CFUNCTION
01225                   || ( Sets[t1[5]].type == ANYTYPE && ( Sets[t1[5]].first == Sets[t1[5]].last ) ) )
      &&
01226                 t1[6] == -SETSET && ( Sets[t1[7]].type == CVECTOR
01227                   || ( Sets[t1[7]].type == ANYTYPE && ( Sets[t1[7]].first == Sets[t1[7]].last ) ) )
      &&
01228                 t1[8] == -SETSET && ( Sets[t1[9]].type == CVECTOR
01229                   || ( Sets[t1[9]].type == ANYTYPE && ( Sets[t1[9]].first == Sets[t1[9]].last ) ) )
      &&
01230                 t1+12 <= t2 ) {
01231 /*
01232                   Test that the sets of particles correspond to particles
01233                   of the set model.
01234 */
01235                   MODEL *m = AC.models[SetElements[Sets[t1[1]].first]];
01236                   int nn0,nn1,nn2;
01237                   for ( nn0 = 3; nn0 <= 5; nn0 += 2 ) {
01238                     for ( nn1 = Sets[t1[nn0]].first; nn1 < Sets[t1[nn0]].last; nn1++ ) {
01239                       for ( nn2 = 0; nn2 < m->nparticles; nn2++ ) {
01240                           if ( m->vertices[nn2]->particles[0].number == SetElements[nn1]
01241                             || m->vertices[nn2]->particles[1].number == SetElements[nn1] ) break;
01242                       }
01243                       if ( nn2 >= m->nparticles ) goto doesnotwork;
01244                     }
01245                   }
01246 /*
01247                   Now test for a single argument indicating the order
01248                   in perturbation theory.
01249 */
01250                   if ( ( t1[10] == -SNUMBER && t1[11] >= 0 && t1+12 == t2 )
01251                   || ( t1[10] == -SYMBOL && t1+12 == t2 )
01252                   || ( t1+10+t1[10] == t2 && t1+10+ARGHEAD+t1[10+ARGHEAD] == t2
01253                   && t1[11+ARGHEAD] == SYMBOL ) ) {
01254 /*
01255                       Now test that all symbols are valid coupling constants.
01256 */
01257                       if ( t1+12 > t2 ) {
01258                           WORD *tt1 = t1+13+ARGHEAD, im;
01259                           t2 -=  ABS(t2[-1]);
01260                           while ( tt1 < t2 ) {
01261                               for ( im = 0; im < m->ncouplings; im++ ) {
01262                                   if ( *tt1 == m->couplings[im] ) break;
01263                               }
01264                               if ( im >= m->ncouplings ) goto doesnotwork;
01265                               tt1 += 2;
01266                           }
01267                       }
01268                       AN.TeInFun = -15;
01269                       AN.TeSuOut = 0;
01270                       AR.TePos = -1;
01271                       AR.funoffset = t - term;
01272                       DONE(1)
01273                   }
01274                   else if ( ( ( t1[10] == -SNUMBER && t1[11] >= 0 && t1+12 == t2-2 )
01275                   || ( t1[10] == -SYMBOL && t1+12 == t2-2 )
01276                   || ( t1+10+t1[10] == t2-2 && t1+10+ARGHEAD+t1[10+ARGHEAD] == t2-2
01277                   && t1[11+ARGHEAD] == SYMBOL ) ) && t2[-2] == -SNUMBER ) {
01278 /*
```

```
01279                         With options at t2[-2],t2[-1]
01280                         Now test that all symbols are valid coupling constants.
01281 */
01282                         t2 -= 2;
01283                         if ( t1+12 > t2 ) {
01284                             WORD *tt1 = t1+13+ARGHEAD, im;
01285                             t2 -=  ABS(t2[-1]);
01286                             while ( tt1 < t2 ) {
01287                                 for ( im = 0; im < m->ncouplings; im++ ) {
01288                                     if ( *tt1 == m->couplings[im] ) break;
01289                                 }
01290                                 if ( im >= m->ncouplings ) goto doesnotwork;
01291                                 tt1 += 2;
01292                             }
01293                         }
01294                         AN.TeInFun = -15;
01295                         AN.TeSuOut = 0;
01296                         AR.TePos = -1;
01297                         AR.funoffset = t - term;
01298                         DONE(1)
01299                     }
01300 doesnotwork:;
01301                   }
01302                 }
01303             }
01304             if ( functions[funnum-FUNCTION].spec <= 0
01305                 || ( t[2] & (DIRTYFLAG|MUSTCLEANPRF) ) != 0 ) {
01306                 funflag = 1;
01307             }
01308             if ( *t <= MAXBUILTINFUNCTION ) {
01309               if ( *t <= DELTAP && *t >= THETA ) { /* Speeds up by 2 or 3 compares */
01310               if ( *t == THETA || *t == THETA2 ) {
01311                 WORD *tstop, *tt2, kk;
01312                 tstop = t + t[1];
01313                 tt2 = t + FUNHEAD;
01314                 while ( tt2 < tstop ) {
01315                     if ( *tt2 > 0 && tt2[1] != 0 ) goto DoSpec;
01316                     NEXTARG(tt2)
01317                 }
01318                 if ( !AT.RecFlag ) {
01319                     if ( ( kk = DoTheta(BHEAD t) ) == 0 ) {
01320                         *term = 0;
01321                         DONE(0)
01322                     }
01323                     else if ( kk > 0 ) {
01324                         m = t + t[1];
01325                         r = term + *term;
01326                         while ( m < r ) *t++ = *m++;
01327                         *term = WORDDIF(t,term);
01328                         goto ReStart;
01329                     }
01330                 }
01331               }
01332               else if ( *t == DELTA2 || *t == DELTAP ) {
01333                 WORD *tstop, *tt2, kk;
01334                 tstop = t + t[1];
01335                 tt2 = t + FUNHEAD;
01336                 while ( tt2 < tstop ) {
01337                     if ( *tt2 > 0 && tt2[1] != 0 ) goto DoSpec;
01338                     NEXTARG(tt2)
01339                 }
01340                 if ( !AT.RecFlag ) {
01341                     if ( ( kk = DoDelta(t) ) == 0 ) {
01342                         *term = 0;
01343                         DONE(0)
01344                     }
01345                     else if ( kk > 0 ) {
01346                         m = t + t[1];
01347                         r = term + *term;
01348                         while ( m < r ) *t++ = *m++;
01349                         *term = WORDDIF(t,term);
01350                         goto ReStart;
01351                     }
01352                 }
01353               } }
01354               else if ( *t == DISTRIBUTION && t[FUNHEAD] == -SNUMBER
01355               && t[FUNHEAD+1] >= -2 && t[FUNHEAD+1] <= 2
01356               && t[FUNHEAD+2] == -SNUMBER
01357               && t[FUNHEAD+4] <= -FUNCTION
01358               && t[FUNHEAD+5] <= -FUNCTION ) {
01359                 WORD *ttt = t+FUNHEAD+6, *tttstop = t+t[1];
01360                 while ( ttt < tttstop ) {
01361                     if ( *ttt == -DOLLAREXPRESSION ) break;
01362                     NEXTARG(ttt);
01363                 }
01364                 if ( ttt >= tttstop ) {
01365                     AN.TeInFun = -1;
```

```
01366                   AN.TeSuOut = 0;
01367                   AR.TePos = -1;
01368                   DONE(1)
01369                 }
01370               }
01371               else if ( *t == DELTA3 && ((t[1]-FUNHEAD) & 1 ) == 0 ) {
01372                 AN.TeInFun = -2;
01373                 AN.TeSuOut = 0;
01374                 AR.TePos = -1;
01375                 DONE(1)
01376               }
01377               else if ( ( *t == TABLEFUNCTION ) && ( t[FUNHEAD] <= -FUNCTION )
01378               && ( T = functions[-t[FUNHEAD]-FUNCTION].tabl ) != 0
01379               && ( t[1] >= FUNHEAD+1+2*ABS(T->numind) )
01380               && ( t[FUNHEAD+1] == -SYMBOL ) ) {
01381 /*
01382                 The case of table_(tab,sym1,...,symn)
01383 */
01384                 for ( isp = 0; isp < ABS(T->numind); isp++ ) {
01385                   if ( t[FUNHEAD+1+2*isp] != -SYMBOL ) break;
01386                 }
01387                 if ( isp >= ABS(T->numind) ) {
01388                   AN.TeInFun = -3;
01389                   AN.TeSuOut = 0;
01390                   AR.TePos = -1;
01391                   DONE(1)
01392                 }
01393               }
01394               else if ( *t == TABLEFUNCTION && t[FUNHEAD] <= -FUNCTION
01395               && ( T = functions[-t[FUNHEAD]-FUNCTION].tabl ) != 0
01396               && ( t[1] == FUNHEAD+2 )
01397               && ( t[FUNHEAD+1] <= -FUNCTION ) ) {
01398 /*
01399                 The case of table_(tab,fun)
01400 */
01401                 AN.TeInFun = -3;
01402                 AN.TeSuOut = 0;
01403                 AR.TePos = -1;
01404                 DONE(1)
01405               }
01406               else if ( *t == FACTORIN ) {
01407                 if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -DOLLAREXPRESSION ) {
01408                   AN.TeInFun = -4;
01409                   AN.TeSuOut = 0;
01410                   AR.TePos = -1;
01411                   DONE(1)
01412                 }
01413                 else if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -EXPRESSION ) {
01414                   AN.TeInFun = -5;
01415                   AN.TeSuOut = 0;
01416                   AR.TePos = -1;
01417                   DONE(1)
01418                 }
01419               }
01420               else if ( *t == TERMSINBRACKET ) {
01421                 if ( t[1] == FUNHEAD || (
01422                      t[1] == FUNHEAD+2
01423                   && t[FUNHEAD] == -SNUMBER
01424                   && t[FUNHEAD+1] == 0
01425                  ) ) {
01426                   AN.TeInFun = -6;
01427                   AN.TeSuOut = 0;
01428                   AR.TePos = -1;
01429                   DONE(1)
01430                 }
01431 /*
01432                 The other cases have not yet been implemented
01433                 We still have to add the case of short arguments
01434                 First the different bracket in same expression
01435
01436                 else if ( t[1] > FUNHEAD+ARGHEAD
01437                         && t[FUNHEAD] == t[1]-FUNHEAD
01438                         && t[FUNHEAD+ARGHEAD] == t[1]-FUNHEAD-ARGHEAD
01439                         && t[t[1]-1] == 3
01440                         && t[t[1]-2] == 1
01441                         && t[t[1]-3] == 1 ) {
01442                   AN.TeInFun = -6;
01443                   AN.TeSuOut = 0;
01444                   AR.TePos = -1;
01445                   DONE(1)
01446                 }
01447
01448                 Next the bracket in an other expression
01449
01450                 else if ( t[1] > FUNHEAD+ARGHEAD+2
01451                         && t[FUNHEAD] == -EXPRESSION
01452                         && t[FUNHEAD+2] == t[1]-FUNHEAD-2
```

```
01453                       && t[FUNHEAD+ARGHEAD+2] == t[1]-FUNHEAD-ARGHEAD-2
01454                       && t[t[1]-1] == 3
01455                       && t[t[1]-2] == 1
01456                       && t[t[1]-3] == 1 ) {
01457                   AN.TeInFun = -6;
01458                   AN.TeSuOut = 0;
01459                   AR.TePos = -1;
01460                   DONE(1)
01461               }
01462 */
01463           }
01464           else if ( *t == EXTRASYMFUN ) {
01465               if ( t[1] == FUNHEAD+2 && (
01466                   ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] <= cbuf[AM.sbufnum].numrhs
01467                           && t[FUNHEAD+1] > 0 ) ||
01468                   ( t[FUNHEAD] == -SYMBOL && t[FUNHEAD+1] < MAXVARIABLES
01469                           && t[FUNHEAD+1] >= MAXVARIABLES-cbuf[AM.sbufnum].numrhs ) ) ) {
01470                   AN.TeInFun = -7;
01471                   AN.TeSuOut = 0;
01472                   AR.TePos = -1;
01473                   DONE(1)
01474               }
01475               else if ( t[1] == FUNHEAD ) {
01476                   AN.TeInFun = -7;
01477                   AN.TeSuOut = 0;
01478                   AR.TePos = -1;
01479                   DONE(1)
01480               }
01481           }
01482           else if ( *t == DIVFUNCTION || *t == REMFUNCTION
01483                     || *t == INVERSEFUNCTION || *t == MULFUNCTION
01484                     || *t == GCDFUNCTION ) {
01485               WORD *tf;
01486               int todo = 1, numargs = 0;
01487               tf = t + FUNHEAD;
01488               while ( tf < t + t[1] ) {
01489                   DOLLARS d;
01490                   if ( *tf == -DOLLAREXPRESSION ) {
01491                       d = Dollars + tf[1];
01492                       if ( d->type == DOLWILDARGS ) {
01493                           WORD *tterm = AT.WorkPointer, *tw;
01494                           WORD *ta = term, *tb = tterm, *tc, *td = term + *term;
01495                           while ( ta < t ) *tb++ = *ta++;
01496                           tc = tb;
01497                           while ( ta < tf ) *tb++ = *ta++;
01498                           tw = d->where+1;
01499                           while ( *tw ) {
01500                               if ( *tw < 0 ) {
01501                                   if ( *tw > -FUNCTION ) *tb++ = *tw++;
01502                                   *tb++ = *tw++;
01503                               }
01504                               else {
01505                                   int ia;
01506                                   for ( ia = 0; ia < *tw; ia++ ) *tb++ = *tw++;
01507                               }
01508                           }
01509                           NEXTARG(ta)
01510                           while ( ta < t+t[1] ) *tb++ = *ta++;
01511                           tc[1] = tb-tc;
01512                           while ( ta < td ) *tb++ = *ta++;
01513                           *tterm = tb - tterm;
01514                           {
01515                               int ia, na = *tterm;
01516                               ta = tterm; tb  = term;
01517                               for ( ia = 0; ia < na; ia++ ) *tb++ = *ta++;
01518                           }
01519                           if ( tb > AT.WorkTop ) {
01520                               MLOCK(ErrorMessageLock);
01521                               MesWork();
01522                               goto EndTest2;
01523                           }
01524                           AT.WorkPointer = tb;
01525                           goto ReStart;
01526                       }
01527                   }
01528                   NEXTARG(tf);
01529               }
01530               tf = t + FUNHEAD;
01531               while ( tf < t + t[1] ) {
01532                   numargs++;
01533                   if ( *tf > 0 && tf[1] != 0 ) todo = 0;
01534                   NEXTARG(tf);
01535               }
01536               if ( todo && numargs == 2 ) {
01537                   if ( *t == DIVFUNCTION ) AN.TeInFun = -9;
01538                   else if ( *t == REMFUNCTION ) AN.TeInFun = -10;
01539                   else if ( *t == INVERSEFUNCTION ) AN.TeInFun = -11;
```

```
01540                   else if ( *t == MULFUNCTION ) AN.TeInFun = -14;
01541                   else if ( *t == GCDFUNCTION ) AN.TeInFun = -8;
01542                   AN.TeSuOut = 0;
01543                   AR.TePos = -1;
01544                   DONE(1)
01545                 }
01546                 else if ( todo && numargs == 3 ) {
01547                   if ( *t == DIVFUNCTION ) AN.TeInFun = -9;
01548                   else if ( *t == REMFUNCTION ) AN.TeInFun = -10;
01549                   else if ( *t == GCDFUNCTION ) AN.TeInFun = -8;
01550                   AN.TeSuOut = 0;
01551                   AR.TePos = -1;
01552                   DONE(1)
01553                 }
01554                 else if ( todo && *t == GCDFUNCTION ) {
01555                   AN.TeInFun = -8;
01556                   AN.TeSuOut = 0;
01557                   AR.TePos = -1;
01558                   DONE(1)
01559                 }
01560               }
01561               else if ( *t == PERMUTATIONS && ( ( t[1] >= FUNHEAD+1
01562                 && t[FUNHEAD] <= -FUNCTION ) || ( t[1] >= FUNHEAD+3
01563                 && t[FUNHEAD] == -SNUMBER && t[FUNHEAD+2] <= -FUNCTION ) ) ) {
01564                 AN.TeInFun = -12;
01565                 AN.TeSuOut = 0;
01566                 AR.TePos = -1;
01567                 DONE(1)
01568               }
01569               else if ( *t == PARTITIONS ) {
01570                 if ( TestPartitions(t,&(AT.partitions)) ) {
01571                   AT.partitions.where = t-term;
01572                   AN.TeInFun = -13;
01573                   AN.TeSuOut = 0;
01574                   AR.TePos = -1;
01575                   DONE(1)
01576                 }
01577               }
01578             }
01579         }
01580         t += t[1];
01581     } while ( t < m );
01582     if ( funflag ) {    /* Search in functions */
01583 DoSpec:
01584         t = term;
01585         AT.NestPoin->termsize = t;
01586         if ( AT.NestPoin == AT.Nest ) AN.EndNest = t + *t;
01587         t++;
01588         oldncmod = AN.ncmod;
01589         if ( t < m ) do {
01590             if ( *t < FUNCTION ) {
01591                 t += t[1]; continue;
01592             }
01593             if ( AN.ncmod && ( ( AC.modmode & ALSOFUNARGS ) == 0 ) ) {
01594                 if ( *t != AR.PolyFun ) AN.ncmod = 0;
01595                 else                    AN.ncmod = oldncmod;
01596             }
01597             r = t + t[1];
01598             funnum = *t;
01599             if ( *t >= FUNCTION + WILDOFFSET ) funnum -= WILDOFFSET;
01600             if ( ( *t == NUMFACTORS || *t == FIRSTTERM || *t == CONTENTTERM )
01601              && t[1] == FUNHEAD+2 &&
01602             ( t[FUNHEAD] == -EXPRESSION || t[FUNHEAD] == -DOLLAREXPRESSION ) ) {
01603 /*
01604                 if ( *t == NUMFACTORS ) {
01605                     This we leave for Normalize
01606                 }
01607 */
01608             }
01609             else if ( functions[funnum-FUNCTION].spec <= 0 ) {
01610                 AT.NestPoin->funsize = t + 1;
01611                 t1 = t;
01612                 t += FUNHEAD;
01613                 while ( t < r ) {   /* Sum over arguments */
01614                     if ( *t > 0 && t[1] ) { /* Argument is dirty  */
01615                         AT.NestPoin->argsize = t;
01616                         AT.NestPoin++;
01617 /*                      stop = t + *t; */
01618                         t2 = t;
01619                         t += ARGHEAD;
01620                         while ( t < AT.NestPoin[-1].argsize+*(AT.NestPoin[-1].argsize) ) {
01621                                         /* Sum over terms */
01622                             AT.RecFlag++;
01623                             i = *t;
01624                             AN.subsubveto = 1;
01625 /*
01626                             AN.subsubveto repairs a bug that became apparent
```

```
01627                               in an example by York Schroeder:
01628                                   f(k1.k1)*replace_(k1,2*k2)
01629                               Is it possible to repair the counting of the various
01630                               length indicators? (JV 1-jun-2010)
01631 */
01632                               if ( ( retvalue = TestSub(BHEAD t,level) ) != 0 ) {
01633 /*
01634                                   Possible size changes:
01635                                   Note defs at 471,467,460,400,425,328
01636 */
01637 redosize:
01638                                   if ( i > *t ) {
01639 /*
01640                                       i -= *t;
01641                                       *t2 -= i;
01642                                       t1[1] -= i;
01643                                       t += *t;
01644                                       r = t + i;
01645                                       m = term + *term;
01646                                       while ( r < m ) *t++ = *r++;
01647                                       *term -= i;
01648 */
01649                                       i -= *t;
01650                                       t += *t;
01651                                       r = t + i;
01652                                       m = AN.EndNest;
01653                                       while ( r < m ) *t++ = *r++;
01654                                       t = AT.NestPoin[-1].argsize + ARGHEAD;
01655                                       n = AT.Nest;
01656                                       while ( n < AT.NestPoin ) {
01657                                           *(n->argsize) -= i;
01658                                           *(n->funsize) -= i;
01659                                           *(n->termsize) -= i;
01660                                           n++;
01661                                       }
01662                                       AN.EndNest -= i;
01663                                   }
01664                                   AN.subsubveto = 0;
01665                                   t1[2] = 1;
01666                                   if ( *t1 == AR.PolyFun && AR.PolyFunType == 2 )
01667                                       t1[2] |= MUSTCLEANPRF;
01668                                   AT.RecFlag--;
01669                                   AT.NestPoin--;
01670                                   AN.TeInFun++;
01671                                   AR.TePos = 0;
01672                                   AN.ncmod = oldncmod;
01673                                   DONE(retvalue)
01674                               }
01675                               else {
01676                                   /*
01677                                    * Somehow the next line fixes Issue #106.
01678                                    */
01679                                   i = *t;
01680                                   Normalize(BHEAD t);
01681 /*                                if ( i > *t ) { retvalue = 1; goto redosize; } */
01682                                   /*
01683                                    * Experimentally, the next line fixes Issue #105.
01684                                    */
01685                                   if ( *t == 0 ) { retvalue = 1; goto redosize; }
01686                                   {
01687                                       WORD *tend = t + *t, *tt = t+1;
01688                                       stilldirty = 0;
01689                                       tend -= ABS(tend[-1]);
01690                                       while ( tt < tend ) {
01691                                           if ( *tt == SUBEXPRESSION || *tt == EXPRESSION ) {
01692                                               stilldirty = 1; break;
01693                                           }
01694                                           tt += tt[1];
01695                                       }
01696                                   }
01697                                   if ( i > *t ) {
01698 /*
01699                                       We should not forget to correct the Nest
01700                                       stack. That caused trouble in the past.
01701 */
01702                                       retvalue = 1;
01703                                       i -= *t;
01704                                       t += *t;
01705                                       r = t + i;
01706                                       m = AN.EndNest;
01707                                       while ( r < m ) *t++ = *r++;
01708                                       t = AT.NestPoin[-1].argsize + ARGHEAD;
01709                                       n = AT.Nest;
01710                                       while ( n < AT.NestPoin ) {
01711                                           *(n->argsize) -= i;
01712                                           *(n->funsize) -= i;
01713                                           *(n->termsize) -= i;
```

```
01714                                   n++;
01715                               }
01716                               AN.EndNest -= i;
01717                           }
01718                       }
01719                       AN.subsubveto = 0;
01720                       AT.RecFlag--;
01721                       t += *t;
01722                   }
01723                   AT.NestPoin--;
01724 /*
01725                   Argument contains no subexpressions.
01726                   It should be normalized and sorted.
01727                   The main problem is the storage.
01728 */
01729                   t = AT.NestPoin->argsize;
01730                   j = *t;
01731                   t += ARGHEAD;
01732                   NewSort(BHEAD0);
01733                   if ( *t1 == AR.PolyFun && AR.PolyFunType == 2 ) {
01734                       AR.CompareRoutine = (COMPAREDUMMY)(&CompareSymbols);
01735                       AR.SortType = SORTHIGHFIRST;
01736                   }
01737                   if ( AT.WorkPointer < term + *term )
01738                       AT.WorkPointer = term + *term;
01739
01740                   while ( t < AT.NestPoin->argsize+*(AT.NestPoin->argsize) ) {
01741                       m = AT.WorkPointer;
01742                       r = t + *t;
01743                       do { *m++ = *t++; } while ( t < r );
01744                       r = AT.WorkPointer;
01745                       AT.WorkPointer = r + *r;
01746                       if ( Normalize(BHEAD r) ) {
01747                           if ( *t1 == AR.PolyFun && AR.PolyFunType == 2 ) {
01748                               AR.SortType = oldsorttype;
01749                               AR.CompareRoutine = (COMPAREDUMMY)oldcompareroutine;
01750                               t1[2] |= MUSTCLEANPRF;
01751                           }
01752                           LowerSortLevel(); goto EndTest;
01753                       }
01754                       if ( AN.ncmod != 0 ) {
01755                           if ( *r ) {
01756                               if ( Modulus(r) ) {
01757                                   LowerSortLevel();
01758                                   AT.WorkPointer = r;
01759                                   if ( *t1 == AR.PolyFun && AR.PolyFunType == 2 ) {
01760                                       AR.SortType = oldsorttype;
01761                                       AR.CompareRoutine = (COMPAREDUMMY)oldcompareroutine;
01762                                       t1[2] |= MUSTCLEANPRF;
01763                                   }
01764                                   goto EndTest;
01765                               }
01766                           }
01767                       }
01768                       if ( AR.PolyFun > 0 ) {
01769                           if ( PrepPoly(BHEAD r,1) != 0 ) goto EndTest;
01770                       }
01771                       if ( *r ) StoreTerm(BHEAD r);
01772                       AT.WorkPointer = r;
01773                   }
01774                   if ( EndSort(BHEAD AT.WorkPointer+ARGHEAD,1) < 0 ) goto EndTest;
01775                   m = AT.WorkPointer+ARGHEAD;
01776                   if ( *t1 == AR.PolyFun && AR.PolyFunType == 2 ) {
01777                       AR.SortType = oldsorttype;
01778                       AR.CompareRoutine = (COMPAREDUMMY)oldcompareroutine;
01779                       t1[2] |= MUSTCLEANPRF;
01780                   }
01781                   while ( *m ) m += *m;
01782                   i = WORDDIF(m,AT.WorkPointer);
01783                   *AT.WorkPointer = i;
01784                   AT.WorkPointer[1] = stilldirty;
01785                   if ( ToFast(AT.WorkPointer,AT.WorkPointer) ) {
01786                       m = AT.WorkPointer;
01787                       if ( *m <= -FUNCTION ) { m++; i = 1; }
01788                       else { m += 2; i = 2; }
01789                   }
01790                   j = i - j;
01791                   if ( j > 0 ) {
01792                       r = m + j;
01793                       if ( r > AT.WorkTop ) {
01794                           MLOCK(ErrorMessageLock);
01795                           MesWork();
01796                           goto EndTest2;
01797                       }
01798                       do { *--r = *--m; } while ( m > AT.WorkPointer );
01799                       AT.WorkPointer = r;
01800                       m = AN.EndNest;
```

```
01801                           r = m + j;
01802                           stop = AT.NestPoin->argsize+*(AT.NestPoin->argsize);
01803                           do { *--r = *--m; } while ( m >= stop );
01804                       }
01805                       else if ( j < 0 ) {
01806                           m = AT.NestPoin->argsize+*(AT.NestPoin->argsize);
01807                           r = m + j;
01808                           do { *r++ = *m++; } while ( m < AN.EndNest );
01809                       }
01810                       m = AT.NestPoin->argsize;
01811                       r = AT.WorkPointer;
01812                       while ( --i >= 0 ) *m++ = *r++;
01813                       n = AT.Nest;
01814                       while ( n <= AT.NestPoin ) {
01815                           if ( *(n->argsize) > 0 && n != AT.NestPoin )
01816                               *(n->argsize) += j;
01817                           *(n->funsize) += j;
01818                           *(n->termsize) += j;
01819                           n++;
01820                       }
01821                       AN.EndNest += j;
01822 /*                    (AT.NestPoin->argsize)[1] = 0;  */
01823                       if ( funnum == DENOMINATOR || funnum == EXPONENT ) {
01824                           if ( Normalize(BHEAD term) ) {
01825 /*
01826                               In this case something has been substituted
01827                               Either a $ or a replace_?????
01828                               Originally we had here:
01829
01830                               goto EndTest;
01831
01832                               It seems better to restart.
01833 */
01834                               AN.ncmod = oldncmod;
01835                               goto ReStart;
01836                           }
01837 /*
01838                           And size changes here?????
01839 */
01840                       }
01841                       AN.ncmod = oldncmod;
01842                       goto ReStart;
01843                   }
01844                   else if ( *t == -DOLLAREXPRESSION ) {
01845                       if ( ( *t1 == TERMSINEXPR || *t1 == SIZEOFFUNCTION )
01846                                && t1[1] == FUNHEAD+2 ) {}
01847                       else {
01848                           if ( AR.Eside != LHSIDE ) {
01849                               AN.TeInFun = 1; AR.TePos = 0;
01850                               AT.TMbuff = AM.dbufnum; t1[2] |= DIRTYFLAG;
01851                               AN.ncmod = oldncmod;
01852                               DONE(1)
01853                           }
01854                           AC.lhdollarflag = 1;
01855                       }
01856                   }
01857                   else if ( *t == -TERMSINBRACKET ) {
01858                       if ( AR.Eside != LHSIDE ) {
01859                           AN.TeInFun = 1; AR.TePos = 0;
01860                           t1[2] |= DIRTYFLAG;
01861                           AN.ncmod = oldncmod;
01862                           DONE(1)
01863                       }
01864                   }
01865                   else if ( AN.ncmod != 0 && *t == -SNUMBER ) {
01866                       if ( AN.ncmod == 1 || AN.ncmod == -1 ) {
01867                           isp = (UWORD)(AC.cmod[0]);
01868                           isp = t[1] % isp;
01869                           if ( ( AC.modmode & POSNEG ) != 0 ) {
01870                               if ( isp > (UWORD)(AC.cmod[0])/2 ) isp = isp - (UWORD)(AC.cmod[0]);
01871                               else if ( -isp > (UWORD)(AC.cmod[0])/2 ) isp = isp +
    (UWORD)(AC.cmod[0]);
01872                           }
01873                           else {
01874                               if ( isp < 0 ) isp += (UWORD)(AC.cmod[0]);
01875                           }
01876                           if ( isp <= MAXPOSITIVE && isp >= -MAXPOSITIVE ) {
01877                               t[1] = isp;
01878                           }
01879                       }
01880                   }
01881                   NEXTARG(t)
01882               }
01883               if ( funnum >= FUNCTION && functions[funnum-FUNCTION].tabl ) {
01884 /*
01885                   Test whether the table catches
01886                   Test 1: index arguments and range. i will be the number
```

```
01887                         of the element in the table.
01888 */
01889                     WORD rhsnumber, *oldwork = AT.WorkPointer;
01890                     WORD ii, *p, *pp, *ppstop;
01891                     MINMAX *mm;
01892                     T = functions[funnum-FUNCTION].tabl;
01893 /*
01894                     Because of tables with a variable number of indices
01895                     we need to make a copy of the pattern.
01896                     If we do this in the WorkSpace we get problems with EndNest.
01897                     This is why we use TermMalloc.
01898                     Now Tpattern is a copy that can be modified.
01899 */
01900 #ifdef WITHPTHREADS
01901                     pp = T->pattern[AT.identity];
01902 #else
01903                     pp = T->pattern;
01904 #endif
01905                     if ( Tpattern == 0 ) Tpattern = TermMalloc("Tpattern");
01906                     p = Tpattern;
01907                     ii = pp[1];
01908                     for ( i = 0; i < ii; i++ ) *p++ = *pp++;
01909
01910                     p = Tpattern + FUNHEAD+1;
01911                     mm = T->mm;
01912                     if ( T->sparse ) {
01913                         WORD xx;
01914                         t = t1+FUNHEAD;
01915                         if ( T->numind == 0 ) { isp = 0; xx = 0; }
01916                         else {
01917                             if ( T->numind < 0 ) {
01918                               if ( *t != -SNUMBER && t[2] != -SNUMBER ) {
01919                                 xx = ABS(T->numind);
01920                                 goto teststrict;
01921                               }
01922                               xx = t[1]+1;
01923                               if ( xx < 2 || xx > -T->numind ) goto teststrict;
01924                             }
01925                             else xx = T->numind;
01926                             for ( i = 0; i < xx; i++, t += 2 ) {
01927                                 if ( *t != -SNUMBER ) break;
01928                             }
01929                             if ( i < xx ) goto teststrict;
01930                             isp = FindTableTree(T,t1+FUNHEAD,2);
01931                         }
01932                         if ( isp < 0 ) {
01933 teststrict:                 if ( T->strict == -2 ) {
01934                                 rhsnumber = AM.zerorhs;
01935                                 tbufnum = AM.zbufnum;
01936                             }
01937                             else if ( T->strict == -3 ) {
01938                                 rhsnumber = AM.onerhs;
01939                                 tbufnum = AM.zbufnum;
01940                             }
01941                             else if ( T->strict < 0 ) goto NextFun;
01942                             else {
01943                                 MLOCK(ErrorMessageLock);
01944                                 MesPrint("Element in table is undefined");
01945                                 if ( Tpattern ) {
01946                                     TermFree(Tpattern,"Tpattern");
01947                                     Tpattern = 0;
01948                                 }
01949                                 goto showtable;
01950                             }
01951 /*
01952                             Copy the indices;
01953 */
01954                             t = t1+FUNHEAD+1;
01955                             for ( i = 0; i < xx; i++ ) {
01956                                 *p = *t; p+=2; t+=2;
01957                             }
01958                         }
01959                         else {
01960                             rhsnumber = T->tablepointers[isp+ABS(T->numind)];
01961 #if ( TABLEEXTENSION == 2 )
01962                             tbufnum = T->bufnum;
01963 #else
01964                             tbufnum = T->tablepointers[isp+ABS(T->numind)+1];
01965 #endif
01966                             t = t1+FUNHEAD+1;
01967                             ii = xx;
01968                             while ( --ii >= 0 ) {
01969                                 *p = *t; t += 2; p += 2;
01970                             }
01971                         }
01972                         if ( xx < ABS(T->numind) ) {
01973                             p--; ppstop = Tpattern+Tpattern[1];
```

```
01974                               pp = p+2*(-T->numind-xx);
01975                               while ( pp < ppstop ) *p++ = *pp++;
01976                               Tpattern[1] = p - Tpattern;
01977                           }
01978                           goto caughttable;
01979                       }
01980                       else {
01981                           i = 0;
01982                           t = t1 + FUNHEAD;
01983                           j = T->numind;
01984                           while ( --j >= 0 ) {
01985                               if ( *t != -SNUMBER ) goto NextFun;
01986                               t++;
01987                               if ( *t < mm->mini || *t > mm->maxi ) {
01988                                   if ( T->bounds ) {
01989                                       MLOCK(ErrorMessageLock);
01990                                       MesPrint("Table boundary check. Argument %d",
01991                                       T->numind-j);
01992 showtable:                            AO.OutFill = AO.OutputLine = (UBYTE *)m;
01993                                       AO.OutSkip = 8;
01994                                       IniLine(0);
01995                                       WriteSubTerm(t1,1);
01996                                       FiniLine();
01997                                       MUNLOCK(ErrorMessageLock);
01998                                       if ( Tpattern ) {
01999                                           TermFree(Tpattern,"Tpattern");
02000                                           Tpattern = 0;
02001                                       }
02002                                       SETERROR(-1)
02003                                   }
02004                                   if ( Tpattern ) {
02005                                       TermFree(Tpattern,"Tpattern");
02006                                       Tpattern = 0;
02007                                   }
02008                                   goto NextFun;
02009                               }
02010                               i += ( *t - mm->mini ) * (LONG)(mm->size);
02011                               *p = *t++;
02012                               p += 2;
02013                               mm++;
02014                           }
02015 /*
02016                           Test now whether the entry exists.
02017 */
02018                           i *= TABLEEXTENSION;
02019                           if ( T->tablepointers[i] == -1 ) {
02020                               if ( T->strict == -2 ) {
02021                                   rhsnumber = AM.zerorhs;
02022                                   tbufnum = AM.zbufnum;
02023                               }
02024                               else if ( T->strict == -3 ) {
02025                                   rhsnumber = AM.onerhs;
02026                                   tbufnum = AM.zbufnum;
02027                               }
02028                               else if ( T->strict < 0 ) {
02029                                   if ( Tpattern ) {
02030                                       TermFree(Tpattern,"Tpattern");
02031                                       Tpattern = 0;
02032                                   }
02033                                   goto NextFun;
02034                               }
02035                               else {
02036                                   MLOCK(ErrorMessageLock);
02037                                   MesPrint("Element in table is undefined");
02038                                   if ( Tpattern ) {
02039                                       TermFree(Tpattern,"Tpattern");
02040                                       Tpattern = 0;
02041                                   }
02042                                   goto showtable;
02043                               }
02044                           }
02045                           else {
02046                               rhsnumber = T->tablepointers[i];
02047 #if ( TABLEEXTENSION == 2 )
02048                               tbufnum = T->bufnum;
02049 #else
02050                               tbufnum = T->tablepointers[i+1];
02051 #endif
02052                           }
02053                       }
02054 /*
02055                       If there are more arguments we have to do some
02056                       pattern matching. This should be easy. We adapted the
02057                       pattern, so that the array indices match already.
02058                       Note that if there is no match the program will become
02059                       very slow.
02060 */
```

```
02061 caughttable:
02062 #ifdef WITHPTHREADS
02063                     AN.FullProto = T->prototype[AT.identity];
02064 #else
02065                     AN.FullProto = T->prototype;
02066 #endif
02067                     AN.WildValue = AN.FullProto + SUBEXPSIZE;
02068                     AN.WildStop = AN.FullProto+AN.FullProto[1];
02069                     ClearWild(BHEAD0);
02070                     AN.RepFunNum = 0;
02071                     AN.RepFunList = AN.EndNest;
02072                     AT.WorkPointer = (WORD *)(((UBYTE *)(AN.EndNest)) + AM.MaxTer/2);
02073                     if ( AT.WorkPointer >= AT.WorkTop ) {
02074                         MLOCK(ErrorMessageLock);
02075                         MesWork();
02076                         MUNLOCK(ErrorMessageLock);
02077                     }
02078                     wilds = 0;
02079                     if ( MatchFunction(BHEAD Tpattern,t1,&wilds) > 0 ) {
02080                         AT.WorkPointer = oldwork;
02081                         if ( AT.NestPoin != AT.Nest ) {
02082                             AN.ncmod = oldncmod;
02083                             if ( Tpattern ) {
02084                                 TermFree(Tpattern,"Tpattern");
02085                                 Tpattern = 0;
02086                             }
02087                             DONE(1)
02088                         }
02089
02090                         m = AN.FullProto;
02091                         retvalue = m[2] = rhsnumber;
02092                         m[4] = tbufnum;
02093                         t = t1;
02094                         j = t[1];
02095                         i = m[1];
02096                         if ( j > i ) {
02097                             j = i - j;
02098                             NCOPY(t,m,i);
02099                             m = term + *term;
02100                             while ( r < m ) *t++ = *r++;
02101                             *term += j;
02102                         }
02103                         else if ( j < i ) {
02104                             j = i-j;
02105                             t = term + *term;
02106                             while ( t >= r ) { t[j] = *t; t--; }
02107                             t = t1;
02108                             NCOPY(t,m,i);
02109                             *term += j;
02110                         }
02111                         else {
02112                             NCOPY(t,m,j);
02113                         }
02114                         AN.TeInFun = 0;
02115                         AR.TePos = 0;
02116                         AN.TeSuOut = -1;
02117                         if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
02118                         AT.TMbuff = tbufnum;
02119                         AN.ncmod = oldncmod;
02120                         DONE(retvalue);
02121                     }
02122                     AT.WorkPointer = oldwork;
02123                     if ( Tpattern ) {
02124                         TermFree(Tpattern,"Tpattern");
02125                         Tpattern = 0;
02126                     }
02127                 }
02128 NextFun:;
02129             }
02130             else if ( ( t[2] & DIRTYFLAG ) != 0 ) {
02131                 t += FUNHEAD;
02132                 while ( t < r ) {
02133                     if ( *t == FUNNYDOLLAR ) {
02134                         if ( AR.Eside != LHSIDE ) {
02135                             AN.TeInFun = 1;
02136                             AR.TePos = 0;
02137                             AT.TMbuff = AM.dbufnum;
02138                             AN.ncmod = oldncmod;
02139                             DONE(1)
02140                         }
02141                         AC.lhdollarflag = 1;
02142                     }
02143                     t++;
02144                 }
02145             }
02146             t = r;
02147             AN.ncmod = oldncmod;
```

```
02148         } while ( t < m );
02149     }
02150 Done:
02151     if ( Tpattern ) {
02152         TermFree(Tpattern,"Tpattern");
02153         Tpattern = 0;
02154     }
02155     return(retvalue);
02156 EndTest:;
02157     MLOCK(ErrorMessageLock);
02158 EndTest2:;
02159     MesCall("TestSub");
02160     MUNLOCK(ErrorMessageLock);
02161     SETERROR(-1)
02162 }
02163
02164 /*
02165         #] TestSub :
02166         #[ InFunction :         WORD InFunction(term,termout)
02167 */
02180 int InFunction(PHEAD WORD *term, WORD *termout)
02181 {
02182     GETBIDENTITY
02183     WORD *m, *t, *r, *rr, sign = 1, oldncmod;
02184     WORD *u, *v, *w, *from, *to,
02185         ipp, olddefer = AR.DeferFlag, oldPolyFun = AR.PolyFun, i, j;
02186     LONG numterms;
02187     from = t = term;
02188     r = t + *t - 1;
02189     m = r - ABS(*r) + 1;
02190     t++;
02191     while ( t < m ) {
02192         if ( *t >= FUNCTION+WILDOFFSET ) ipp = *t - WILDOFFSET;
02193         else ipp = *t;
02194         if ( AR.TePos ) {
02195             if ( ( term + AR.TePos ) == t ) {
02196                 m = termout;
02197                 while ( from < t ) *m++ = *from++;
02198                 *m++ = DENOMINATOR;
02199                 *m++ = t[1] + 4 + FUNHEAD + ARGHEAD;
02200                 *m++ = DIRTYFLAG;
02201                 FILLFUN3(m)
02202                 *m++ = t[1] + 4 + ARGHEAD;
02203                 *m++ = 1;
02204                 FILLARG(m)
02205                 *m++ = t[1] + 4;
02206                 t[3] = -t[3];
02207                 v = t + t[1];
02208                 while ( t < v ) *m++ = *t++;
02209                 from[3] = -from[3];
02210                 *m++ = 1;
02211                 *m++ = 1;
02212                 *m++ = 3;
02213                 r = term + *term;
02214                 while ( t < r ) *m++ = *t++;
02215                 if ( (m-termout) > (LONG)(AM.MaxTer/sizeof(WORD)) ) {
02216                     MLOCK(ErrorMessageLock);
02217                     MesPrint("Output term too large (%d words) (MaxTermSize: %d words)", m-termout,
    AM.MaxTer/sizeof(WORD));
02218                     MUNLOCK(ErrorMessageLock);
02219                     goto TooLarge;
02220                 }
02221                 *termout = WORDDIF(m,termout);
02222                 return(0);
02223             }
02224         }
02225         else if ( ( *t >= FUNCTION && functions[ipp-FUNCTION].spec <= 0 )
02226         && ( t[2] & DIRTYFLAG ) == DIRTYFLAG ) {
02227             m = termout;
02228             r = t + t[1];
02229             u = t;
02230             t += FUNHEAD;
02231             oldncmod = AN.ncmod;
02232             while ( t < r ) {   /* t points at an argument */
02233                 if ( *t > 0 && t[1] ) { /* Argument has been modified */
02234                     WORD oldsorttype = AR.SortType;
02235                     /* This whole argument must be redone */
02236
02237                     if ( ( AN.ncmod != 0 )
02238                         && ( ( AC.modmode & ALSOFUNARGS ) == 0 )
02239                         && ( *u != AR.PolyFun ) ) { AN.ncmod = 0; }
02240                     AR.DeferFlag = 0;
02241                     v = t + *t;
02242                     t += ARGHEAD;       /* First term */
02243                     LONG copy = t - from;
02244                     const LONG size = t - u;
02245                     NCOPY(m, from, copy);
```

```
02246                     w = m - size;
02247                     to = m;
02248                     NewSort(BHEAD0);
02249                     if ( *u == AR.PolyFun && AR.PolyFunType == 2 ) {
02250                         AR.CompareRoutine = (COMPAREDUMMY)(&CompareSymbols);
02251                         AR.SortType = SORTHIGHFIRST;
02252                     }
02253 /*
02254                     AR.PolyFun = 0;
02255 */
02256                     while ( t < v ) {
02257                         i = *t;
02258                         NCOPY(m,t,i);
02259                         m = to;
02260                         if ( AT.WorkPointer < m+*m ) AT.WorkPointer = m + *m;
02261                         if ( Generator(BHEAD m,AR.Cnumlhs) ) {
02262                             AN.ncmod = oldncmod;
02263                             LowerSortLevel(); goto InFunc;
02264                         }
02265                     }
02266                     /* w = the function */
02267                     /* v = the next argument */
02268                     /* u = the function */
02269                     /* to is new argument */
02270
02271                     to -= ARGHEAD;
02272                     if ( EndSort(BHEAD m,1) < 0 ) {
02273                         AN.ncmod = oldncmod;
02274                         goto InFunc;
02275                     }
02276                     AR.PolyFun = oldPolyFun;
02277                     if ( *u == AR.PolyFun && AR.PolyFunType == 2 ) {
02278                         AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
02279                         AR.SortType = oldsorttype;
02280                     }
02281                     while ( *m ) m += *m;
02282                     *to = WORDDIF(m,to);
02283                     to[1] = 1;  /* ??????? or rather 0?. 24-mar-2006 JV */
02284                     if ( ToFast(to,to) ) {
02285                         if ( *to <= -FUNCTION ) m = to+1;
02286                         else m = to+2;
02287                     }
02288                     w[1] = WORDDIF(m,w) + WORDDIF(r,v);
02289                     r = term + *term;
02290                     t = v;
02291                     while ( t < r ) *m++ = *t++;
02292                     if ( (m-termout) > (LONG)(AM.MaxTer/sizeof(WORD)) ) {
02293                         MLOCK(ErrorMessageLock);
02294                         MesPrint("Output term too large (%d words) (MaxTermSize: %d words)",
      m-termout, AM.MaxTer/sizeof(WORD));
02295                         MUNLOCK(ErrorMessageLock);
02296                         goto TooLarge;
02297                     }
02298                     *termout = WORDDIF(m,termout);
02299                     AR.DeferFlag = olddefer;
02300                     AN.ncmod = oldncmod;
02301                     return(0);
02302                 }
02303                 else if ( *t == -DOLLAREXPRESSION ) {
02304                     if ( AR.Eside == LHSIDE ) {
02305                         NEXTARG(t)
02306                         AC.lhdollarflag = 1;
02307                     }
02308                     else {
02309 /*
02310                         This whole argument must be redone
02311 */
02312                         DOLLARS d = Dollars + t[1];
02313 #ifdef WITHPTHREADS
02314                         int nummodopt, dtype = -1;
02315                         if ( AS.MultiThreaded ) {
02316                             for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
02317                                 if ( t[1] == ModOptdollars[nummodopt].number ) break;
02318                             }
02319                             if ( nummodopt < NumModOptdollars ) {
02320                                 dtype = ModOptdollars[nummodopt].type;
02321                                 if ( DollarLocalCopy(dtype) ) {
02322                                     d = ModOptdollars[nummodopt].dstruct+AT.identity;
02323                                 }
02324                                 else {
02325                                     LOCK(d->pthreadslock);
02326                                 }
02327                             }
02328                         }
02329 #endif
02330                         oldncmod = AN.ncmod;
02331                         if ( ( AN.ncmod != 0 )
```

```
02332                       && ( ( AC.modmode & ALSOFUNARGS ) == 0 )
02333                       && ( *u != AR.PolyFun ) ) { AN.ncmod = 0; }
02334                   AR.DeferFlag = 0;
02335                   v = t + 2;
02336                   LONG copy = t - from;
02337                   const LONG size = t - u;
02338                   NCOPY(m, from, copy);
02339                   w = m - size;
02340                   to = m;
02341                   switch ( d->type ) {
02342                       case DOLINDEX:
02343                           if ( d->index >= 0 && d->index < AM.OffsetIndex ) {
02344                               *m++ = -SNUMBER; *m++ = d->index;
02345                           }
02346                           else { *m++ = -INDEX; *m++ = d->index; }
02347                           break;
02348                       case DOLZERO:
02349                           *m++ = -SNUMBER; *m++ = 0; break;
02350                       case DOLNUMBER:
02351                           if ( d->where[0] == 4 &&
02352                           ( d->where[1] & MAXPOSITIVE ) == d->where[1] ) {
02353                               *m++ = -SNUMBER;
02354                               if ( d->where[3] >= 0 ) *m++ = d->where[1];
02355                               else *m++ = -d->where[1];
02356                               break;
02357                           }
02358                           /* fall through */
02359                       case DOLTERMS:
02360 /*
02361                           Here we have the special case of the PolyRatFun
02362                           That function may have a different sort of the
02363                           terms in the argument.
02364 */
02365                           to = m; r = d->where;
02366                           *m++ = 0; *m++ = 1;
02367                           FILLARG(m)
02368                           while ( *r ) {
02369                               i = *r; NCOPY(m,r,i)
02370                           }
02371                           *to = m-to;
02372                           if ( ToFast(to,to) ) {
02373                               if ( *to <= -FUNCTION ) m = to+1;
02374                               else m = to+2;
02375                           }
02376                           else if ( *u == AR.PolyFun && AR.PolyFunType == 2 ) {
02377                               AR.PolyFun = 0;
02378                               NewSort(BHEAD0);
02379                               AR.CompareRoutine = (COMPAREDUMMY)(&CompareSymbols);
02380                               r = to + ARGHEAD;
02381                               while ( r < m ) {
02382                                   rr = r; r += *r;
02383                                   if ( SymbolNormalize(rr) ) goto InFunc;
02384                                   if ( StoreTerm(BHEAD rr) ) {
02385                                       AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
02386                                       LowerSortLevel();
02387                                       Terminate(-1);
02388                                   }
02389                               }
02390                               if ( EndSort(BHEAD to+ARGHEAD,1) < 0 ) goto InFunc;
02391                               AR.PolyFun = oldPolyFun;
02392                               AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
02393                               m = to+ARGHEAD;
02394                               if ( *m == 0 ) {
02395                                   *to = -SNUMBER;
02396                                   to[1] = 0;
02397                                   m = to + 2;
02398                               }
02399                               else {
02400                                   while ( *m ) m += *m;
02401                                   *t = m - to;
02402                                   if ( ToFast(to,to) ) {
02403                                       if ( *to <= -FUNCTION ) m = to+1;
02404                                       else m = to+2;
02405                                   }
02406                               }
02407                           }
02408                           w[1] = w[1] - 2 + (m-to);
02409                           break;
02410                       case DOLSUBTERM:
02411                           to = m; r = d->where;
02412                           i = r[1];
02413                           *m++ = i+4+ARGHEAD; *m++ = 1;
02414                           FILLARG(m)
02415                           *m++ = i+4;
02416                           while ( --i >= 0 ) *m++ = *r++;
02417                           *m++ = 1; *m++ = 1; *m++ = 3;
02418                           if ( ToFast(to,to) ) {
```

```
02419                               if ( *to <= -FUNCTION ) m = to+1;
02420                               else m = to+2;
02421                           }
02422                           w[1] = w[1] - 2 + (m-to);
02423                           break;
02424                       case DOLARGUMENT:
02425                           to = m; r = d->where;
02426                           if ( *r > 0 ) {
02427                               i = *r - 2;
02428                               *m++ = *r++; *m++ = 1; r++;
02429                               while ( --i >= 0 ) *m++ = *r++;
02430                           }
02431                           else if ( *r <= -FUNCTION ) *m++ = *r++;
02432                           else { *m++ = *r++; *m++ = *r++; }
02433                           w[1] = w[1] - 2 + (m-to);
02434                           break;
02435                       case DOLWILDARGS:
02436                           to = m; r = d->where;
02437                           if ( *r > 0 ) { /* Tensor arguments */
02438                               i = *r++;
02439                               while ( --i >= 0 ) {
02440                                   if ( *r < 0 ) {
02441                                       *m++ = -VECTOR; *m++ = *r++;
02442                                   }
02443                                   else if ( *r >= AM.OffsetIndex ) {
02444                                       *m++ = -INDEX; *m++ = *r++;
02445                                   }
02446                                   else { *m++ = -SNUMBER; *m++ = *r++; }
02447                               }
02448                           }
02449                           else { /* Regular arguments */
02450                               r++;
02451                               while ( *r ) {
02452                                   if ( *r > 0 ) {
02453                                       i = *r - 2;
02454                                       *m++ = *r++; *m++ = 1; r++;
02455                                       while ( --i >= 0 ) *m++ = *r++;
02456                                   }
02457                                   else if ( *r <= -FUNCTION ) *m++ = *r++;
02458                                   else { *m++ = *r++; *m++ = *r++; }
02459                               }
02460                           }
02461                           w[1] = w[1] - 2 + (m-to);
02462                           break;
02463                       case DOLUNDEFINED:
02464                       default:
02465                           MLOCK(ErrorMessageLock);
02466                           MesPrint("!!!Undefined $-variable: $%s!!!",
02467                           AC.dollarnames->namebuffer+d->name);
02468                           MUNLOCK(ErrorMessageLock);
02469 #ifdef WITHPTHREADS
02470                           if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
02471 #endif
02472                           Terminate(-1);
02473                   }
02474 #ifdef WITHPTHREADS
02475                   if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
02476 #endif
02477                   r = term + *term;
02478                   t = v;
02479                   while ( t < r ) *m++ = *t++;
02480                   if ( (m-termout) > (LONG)(AM.MaxTer/sizeof(WORD)) ) {
02481                       MLOCK(ErrorMessageLock);
02482                       MesPrint("Output term too large (%d words) (MaxTermSize: %d words)",
     m-termout, AM.MaxTer/sizeof(WORD));
02483                       MUNLOCK(ErrorMessageLock);
02484                       goto TooLarge;
02485                   }
02486                   *termout = WORDDIF(m,termout);
02487                   AR.DeferFlag = olddefer;
02488                   AN.ncmod = oldncmod;
02489                   return(0);
02490               }
02491               }
02492               else if ( *t == -TERMSINBRACKET ) {
02493                   if ( AC.ComDefer ) numterms = CountTerms1(BHEAD0);
02494                   else               numterms = 1;
02495 /*
02496                   Compose the output term
02497                   First copy the part till this function argument
02498                   m points at the output term space
02499                   u points at the start of the function
02500                   t points at the start of the argument
02501 */
02502                   LONG copy = t - from;
02503                   const LONG size = t - u;
02504                   NCOPY(m, from, copy);
```

```
02505                     w = m - size;
02506                     if ( ( numterms & MAXPOSITIVE ) == numterms ) {
02507                         *m++ = -SNUMBER; *m++ =  numterms & MAXPOSITIVE;
02508                         w[1] += 1;
02509                     }
02510                     else if ( ( i = numterms >> BITSINWORD ) == 0 ) {
02511                         *m++ = ARGHEAD+4;
02512                         for ( j = 1; j < ARGHEAD; j++ ) *m++ = 0;
02513                         *m++ = 4; *m++ = numterms & WORDMASK; *m++ = 1; *m++ = 3;
02514                         w[1] += ARGHEAD+3;
02515                     }
02516                     else {
02517                         *m++ = ARGHEAD+6;
02518                         for ( j = 1; j < ARGHEAD; j++ ) *m++ = 0;
02519                         *m++ = 6; *m++ = numterms & WORDMASK;
02520                         *m++ = i; *m++ = 1; *m++ = 0; *m++ = 5;
02521                         w[1] += ARGHEAD+5;
02522                     }
02523                     from++;  /* Skip our function */
02524                     r = term + *term;
02525                     while ( from < r ) *m++ = *from++;
02526                     if ( (m-termout) > (LONG)(AM.MaxTer/sizeof(WORD)) ) {
02527                         MLOCK(ErrorMessageLock);
02528                         MesPrint("Output term too large (%d words) (MaxTermSize: %d words)",
      m-termout, AM.MaxTer/sizeof(WORD));
02529                         MUNLOCK(ErrorMessageLock);
02530                         goto TooLarge;
02531                     }
02532                     *termout = WORDDIF(m,termout);
02533                     return(0);
02534                 }
02535                 else { NEXTARG(t) }
02536             }
02537             t = u;
02538         }
02539         else if ( ( *t >= FUNCTION && functions[ipp-FUNCTION].spec > 0 )
02540         && ( t[2] & DIRTYFLAG ) == DIRTYFLAG ) { /* Could be FUNNYDOLLAR */
02541             u = t; v = t + t[1];
02542             t += FUNHEAD;
02543             while ( t < v ) {
02544                 if ( *t == FUNNYDOLLAR ) {
02545                     if ( AR.Eside != LHSIDE ) {
02546                     DOLLARS d = Dollars + t[1];
02547 #ifdef WITHPTHREADS
02548                     int nummodopt, dtype = -1;
02549                     if ( AS.MultiThreaded ) {
02550                         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
02551                             if ( t[1] == ModOptdollars[nummodopt].number ) break;
02552                         }
02553                         if ( nummodopt < NumModOptdollars ) {
02554                             dtype = ModOptdollars[nummodopt].type;
02555                             if ( DollarLocalCopy(dtype) ) {
02556                                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
02557                             }
02558                             else {
02559                                 LOCK(d->pthreadslock);
02560                             }
02561                         }
02562                     }
02563 #endif
02564                     oldncmod = AN.ncmod;
02565                     if ( ( AN.ncmod != 0 )
02566                         && ( ( AC.modmode & ALSOFUNARGS ) == 0 )
02567                         && ( *u != AR.PolyFun ) ) { AN.ncmod = 0; }
02568                     m = termout;
02569                     LONG copy = t - from;
02570                     const LONG size = t - u;
02571                     NCOPY(m, from, copy);
02572                     w = m - size;
02573                     to = m;
02574                     switch ( d->type ) {
02575                         case DOLINDEX:
02576                             *m++ = d->index; break;
02577                         case DOLZERO:
02578                             *m++ = 0; break;
02579                         case DOLNUMBER:
02580                         case DOLTERMS:
02581                             if ( d->where[0] == 4 && d->where[4] == 0
02582                             && d->where[3] == 3 && d->where[2] == 1
02583                             && d->where[1] < AM.OffsetIndex ) {
02584                                 *m++ = d->where[1];
02585                             }
02586                             else {
02587 wrongtype:;
02588 #ifdef WITHPTHREADS
02589                                 if ( dtype > 0 && ! DollarLocalCopy(dtype) ) {
      UNLOCK(d->pthreadslock); }
```

```
02590 #endif
02591                                 MLOCK(ErrorMessageLock);
02592                                 MesPrint("$%s has wrong type for tensor substitution",
02593                                 AC.dollarnames->namebuffer+d->name);
02594                                 MUNLOCK(ErrorMessageLock);
02595                                 AN.ncmod = oldncmod;
02596                                 return(-1);
02597                             }
02598                             break;
02599                         case DOLARGUMENT:
02600                             if ( d->where[0] == -INDEX ) {
02601                                 *m++ = d->where[1]; break;
02602                             }
02603                             else if ( d->where[0] == -VECTOR ) {
02604                                 *m++ = d->where[1]; break;
02605                             }
02606                             else if ( d->where[0] == -MINVECTOR ) {
02607                                 *m++ = d->where[1];
02608                                 sign = -sign;
02609                                 break;
02610                             }
02611                             else if ( d->where[0] == -SNUMBER ) {
02612                                 if ( d->where[1] >= 0
02613                                 && d->where[1] < AM.OffsetIndex ) {
02614                                     *m++ = d->where[1]; break;
02615                                 }
02616                             }
02617                             goto wrongtype;
02618                         case DOLWILDARGS:
02619                             if ( d->where[0] > 0 ) {
02620                                 r = d->where; i = *r++;
02621                                 while ( --i >= 0 ) *m++ = *r++;
02622                             }
02623                             else {
02624                                 r = d->where + 1;
02625                                 while ( *r ) {
02626                                     if ( *r == -INDEX ) {
02627                                         *m++ = r[1]; r += 2; continue;
02628                                     }
02629                                     else if ( *r == -VECTOR ) {
02630                                         *m++ = r[1]; r += 2; continue;
02631                                     }
02632                                     else if ( *r == -MINVECTOR ) {
02633                                         *m++ = r[1]; r += 2;
02634                                         sign = -sign; continue;
02635                                     }
02636                                     else if ( *r == -SNUMBER ) {
02637                                         if ( r[1] >= 0
02638                                         && r[1] < AM.OffsetIndex ) {
02639                                             *m++ = r[1]; r += 2; continue;
02640                                         }
02641                                     }
02642                                     goto wrongtype;
02643                                 }
02644                             }
02645                             break;
02646                         case DOLSUBTERM:
02647                             r = d->where;
02648                             if ( *r == INDEX && r[1] == 3 ) {
02649                                 *m++ = r[2];
02650                             }
02651                             else goto wrongtype;
02652                             break;
02653                         case DOLUNDEFINED:
02654 #ifdef WITHPTHREADS
02655                             if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
02656 #endif
02657                             MLOCK(ErrorMessageLock);
02658                             MesPrint("$%s is undefined in tensor substitution",
02659                             AC.dollarnames->namebuffer+d->name);
02660                             MUNLOCK(ErrorMessageLock);
02661                             AN.ncmod = oldncmod;
02662                             return(-1);
02663                     }
02664 #ifdef WITHPTHREADS
02665                     if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
02666 #endif
02667                     w[1] = w[1] - 2 + (m-to);
02668                     from += 2;
02669                     term += *term;
02670                     while ( from < term ) *m++ = *from++;
02671                     if ( sign < 0 ) m[-1] = -m[-1];
02672                     if ( (m-termout) > (LONG)(AM.MaxTer/sizeof(WORD)) ) {
02673                         MLOCK(ErrorMessageLock);
02674                         MesPrint("Output term too large (%d words) (MaxTermSize: %d words)",
     m-termout, AM.MaxTer/sizeof(WORD));
02675                         MUNLOCK(ErrorMessageLock);
```

```
02676                       goto TooLarge;
02677                   }
02678                   *termout = m - termout;
02679                   AN.ncmod = oldncmod;
02680                   return(0);
02681               }
02682               else {
02683                   AC.lhdollarflag = 1;
02684               }
02685               }
02686               t++;
02687           }
02688           t = u;
02689       }
02690       t += t[1];
02691   }
02692 /* INTERNAL_ERROR_EXCL_START */
02693   MLOCK(ErrorMessageLock);
02694   MesPrint("!>Internal error in InFunction: Function not encountered.");
02695   if ( AM.tracebackflag ) {
02696       MesPrint("%w: AR.TePos = %d",AR.TePos);
02697       MesPrint("%w: AN.TeInFun = %d",AN.TeInFun);
02698       termout = term;
02699       AO.OutFill = AO.OutputLine = (UBYTE *)AT.WorkPointer + AM.MaxTer;
02700       AO.OutSkip = 3;
02701       FiniLine();
02702       i = *termout;
02703       while ( --i >= 0 ) {
02704           TalToLine((UWORD)(*termout++));
02705           TokenToLine((UBYTE *)"  ");
02706       }
02707       AO.OutSkip = 0;
02708       FiniLine();
02709       MesCall("InFunction");
02710   }
02711   MUNLOCK(ErrorMessageLock);
02712   return(1);
02713 /* INTERNAL_ERROR_EXCL_STOP */
02714
02715 InFunc:
02716   MLOCK(ErrorMessageLock);
02717   MesCall("InFunction");
02718   MUNLOCK(ErrorMessageLock);
02719   SETERROR(-1)
02720
02721 TooLarge:
02722   MLOCK(ErrorMessageLock);
02723   MesCall("InFunction");
02724   MUNLOCK(ErrorMessageLock);
02725   SETERROR(-1)
02726 }
02727
02728 /*
02729       #] InFunction :
02730       #[ InsertTerm :         WORD InsertTerm(term,replac,extractbuff,position,termout)
02731 */
02749 int InsertTerm(PHEAD WORD *term, WORD replac, WORD extractbuff, WORD *position, WORD *termout,
02750               WORD tepos)
02751 {
02752   GETBIDENTITY
02753   WORD *m, *t, *r, i, l2, j;
02754   WORD *u, *v, l1, *coef;
02755   coef = AT.WorkPointer;
02756   if ( ( AT.WorkPointer = coef + 2*AM.MaxTal ) > AT.WorkTop ) {
02757       MLOCK(ErrorMessageLock);
02758       MesWork();
02759       MUNLOCK(ErrorMessageLock);
02760       return(-1);
02761   }
02762   t = term;
02763   r = t + *t;
02764   l1 = l2 = r[-1];
02765   m = r - ABS(l2);
02766   if ( tepos > 0 ) {
02767       t = term + tepos;
02768       goto foundit;
02769   }
02770   t++;
02771   while ( t < m ) {
02772       if ( *t == SUBEXPRESSION && t[2] == replac && t[3] && t[4] == extractbuff ) {
02773           r = t + t[1];
02774           while ( *r == SUBEXPRESSION && r[2] == replac && r[3] && r < m && r[4] == extractbuff ) {
02775               t = r; r += r[1];
02776           }
02777 foundit:;
02778           u = m;
02779           r = term;
```

```
02780             m = termout;
02781             do { *m++ = *r++; } while ( r < t );
02782             if ( t[1] > SUBEXPSIZE ) {
02783 /*
02784                 if this is a dollar expression there are no wildcards
02785 */
02786                 i = *--m;
02787                 if ( ( l2 = WildFill(BHEAD m,position,t) ) < 0 ) goto InsCall;
02788                 *m = i;
02789                 m += l2-1;
02790                 l2 = *m;
02791                 i = ( j = ABS(l2) ) - 1;
02792                 r = coef + i;
02793                 do { *--r = *--m; } while ( --i > 0 );
02794             }
02795             else {
02796                 v = t;
02797                 t = position;
02798                 r = t + *t;
02799                 l2 = r[-1];
02800                 r -= ( j = ABS(l2) );
02801                 t++;
02802                 if ( t < r ) do { *m++ = *t++; } while ( t < r );
02803                 t = v;
02804             }
02805             t += t[1];
02806             while ( t < u && *t == DOLLAREXPR2 ) t += t[1];
02807 ComAct:     if ( t < u ) do { *m++ = *t++; } while ( t < u );
02808             if ( *r == 1 && r[1] == 1 && j == 3 ) {
02809                 if ( l2 < 0 ) l1 = -l1;
02810                 i = ABS(l1)-1;
02811                 NCOPY(m,t,i);
02812                 *m++ = l1;
02813             }
02814             else {
02815                 if ( MulRat(BHEAD (UWORD *)u,REDLENG(l1),(UWORD *)r,REDLENG(l2),
02816                 (UWORD *)m,&l1) ) goto InsCall;
02817                 l2 = l1;
02818                 l2 *= 2;
02819                 if ( l2 < 0 ) {
02820                     m -= l2;
02821                     *m++ = l2-1;
02822                 }
02823                 else {
02824                     m += l2;
02825                     *m++ = l2+1;
02826                 }
02827             }
02828             *termout = WORDDIF(m,termout);
02829             if ( (*termout)*((LONG)sizeof(WORD)) > AM.MaxTer ) {
02830                 MLOCK(ErrorMessageLock);
02831                 MesPrint("Term too complex during substitution (%d words). MaxTermSize (%l words) is
    too small.", *termout, AM.MaxTer/(LONG)sizeof(WORD) );
02832                 goto InsCall2;
02833             }
02834             AT.WorkPointer = coef;
02835             return(0);
02836         }
02837         t += t[1];
02838     }
02839 /*
02840     The next action is for when there is no subexpression pointer.
02841     We append the extra term. Effectively the routine becomes now a
02842     merge routine for two terms.
02843 */
02844     v = t;
02845     u = m;
02846     r = term;
02847     m = termout;
02848     do { *m++ = *r++; } while ( r < t );
02849     t = position;
02850     r = t + *t;
02851     l2 = r[-1];
02852     r -= ( j = ABS(l2) );
02853     t++;
02854     if ( t < r ) do { *m++ = *t++; } while ( t < r );
02855     t = v;
02856     goto ComAct;
02857
02858 InsCall:
02859     MLOCK(ErrorMessageLock);
02860 InsCall2:
02861     MesCall("InsertTerm");
02862     MUNLOCK(ErrorMessageLock);
02863     SETERROR(-1)
02864 }
02865
```

```
02866 /*
02867         #] InsertTerm :
02868         #[ PasteFile :          WORD PasteFile(num,acc,pos,accf,renum,freeze,nexpr)
02869 */
02885 LONG PasteFile(PHEAD WORD number, WORD *accum, POSITION *position, WORD **accfill,
02886                 RENUMBER renumber, WORD *freeze, WORD nexpr)
02887 {
02888     GETBIDENTITY
02889     WORD *r, l, *m, i;
02890     WORD *stop, *s1, *s2;
02891 /*  POSITION AccPos; bug 12-apr-2008 JV */
02892     WORD InCompState;
02893     WORD *oldipointer;
02894     LONG retlength;
02895     stop = (WORD *)(((UBYTE *)(accum)) + 2*AM.MaxTer);
02896     *accum++ = number;
02897     while ( --number >= 0 ) accum += *accum;
02898     if ( freeze ) {
02899 /*      AccPos = *position; bug 12-apr-2008 JV */
02900         oldipointer = AR.CompressPointer;
02901         do {
02902             AR.CompressPointer = oldipointer;
02903 /*          if ( ( l = GetFromStore(accum,&AccPos,renumber,&InCompState,nexpr) ) < 0 ) bug 12-apr-2008
    JV */
02904             if ( ( l = GetFromStore(accum,position,renumber,&InCompState,nexpr) ) < 0 )
02905                 goto PasErr;
02906             if ( !l ) { *accum = 0; return(0); }
02907             r = accum;
02908             m = r + *r;
02909             m -= ABS(m[-1]);
02910             r++;
02911             while ( r < m && *r != HAAKJE ) r += r[1];
02912             if ( r >= m ) {
02913                 if ( *freeze != 4 ) l = -1;
02914             }
02915             else {
02916 /*
02917                 The algorithm for accepting terms with a given (freeze)
02918                 representation outside brackets is rather crude. A refinement
02919                 would be to store the part outside the bracket and skip the
02920                 term when this part doesn't alter (and is unacceptable).
02921                 Once accepting one can keep accepting till the bracket alters
02922                 and then one may stop the generation. It is necessary to
02923                 set up a struct to remember the bracket and the progress
02924                 status.
02925 */
02926                 m = AT.WorkPointer;
02927                 s2 = r;
02928                 r = accum;
02929                 *m++ = WORDDIF(s2,r) + 3;
02930                 r++;
02931                 while ( r < s2 ) *m++ = *r++;
02932                 *m++ = 1; *m++ = 1; *m++ = 3;
02933                 m = AT.WorkPointer;
02934                 if ( Normalize(BHEAD AT.WorkPointer) ) goto PasErr;
02935                 r = freeze;
02936                 i = *m;
02937                 while ( --i >= 0 && *m++ == *r++ ) {}
02938                 if ( i > 0 ) {
02939                     l = -1;
02940                 }
02941                 else {  /* Term to be accepted */
02942                     r = accum;
02943                     s1 = r + *r;
02944                     r++;
02945                     m = s2;
02946                     m += m[1];
02947                     do { *r++ = *m++; } while ( m < s1 );
02948                     *accum = l = WORDDIF(r,accum);
02949                 }
02950             }
02951         } while ( l < 0 );
02952         retlength = InCompState;
02953 /*      retlength = DIFBASE(AccPos,*position) / sizeof(WORD);  bug 12-apr-2008 JV */
02954     }
02955     else {
02956         if ( ( l = GetFromStore(accum,position,renumber,&InCompState,nexpr) ) < 0 ) {
02957             MLOCK(ErrorMessageLock);
02958             MesCall("PasteFile");
02959             MUNLOCK(ErrorMessageLock);
02960             SETERROR(-1)
02961         }
02962         if ( l == 0 ) { *accum = 0; return(0); }
02963         retlength = InCompState;
02964     }
02965     accum += l;
02966     if ( accum > stop ) {
```

```
02967 /* INTERNAL_ERROR_EXCL_START */
02968         MLOCK(ErrorMessageLock);
02969         MesPrint("!>Buffer too small in PasteFile");
02970         MUNLOCK(ErrorMessageLock);
02971         SETERROR(-1)
02972 /* INTERNAL_ERROR_EXCL_STOP */
02973     }
02974     *accum = 0;
02975     *accfill = accum;
02976     return(retlength);
02977 PasErr:
02978     MLOCK(ErrorMessageLock);
02979     MesCall("PasteFile");
02980     MUNLOCK(ErrorMessageLock);
02981     SETERROR(-1)
02982 }
02983
02984 /*
02985         #] PasteFile :
02986         #[ PasteTerm :          WORD PasteTerm(number,accum,position,times,divby)
02987 */
03009 WORD *PasteTerm(PHEAD WORD number, WORD *accum, WORD *position, WORD times, WORD divby)
03010 {
03011     GETBIDENTITY
03012     WORD *t, *r, x, y, z;
03013     WORD *m, *u, l1, a[2];
03014     m = (WORD *)(((UBYTE *)(accum)) + AM.MaxTer);
03015 /*    m = (WORD *)(((UBYTE *)(accum)) + 2*AM.MaxTer); */
03016     *accum++ = number;
03017     while ( --number >= 0 ) accum += *accum;
03018     if ( times == divby ) {
03019         t = position;
03020         r = t + *t;
03021         if ( t < r ) do { *accum++ = *t++; } while ( t < r );
03022     }
03023     else {
03024         u = accum;
03025         t = position;
03026         r = t + *t - 1;
03027         l1 = *r;
03028         r -= ABS(*r) - 1;
03029         if ( t < r ) do { *accum++ = *t++; } while ( t < r );
03030         if ( divby > times ) { x = divby; y = times; }
03031         else { x = times; y = divby; }
03032         z = x%y;
03033         while ( z ) { x = y; y = z; z = x%y; }
03034         if ( y != 1 ) { divby /= y; times /= y; }
03035         a[1] = divby;
03036         a[0] = times;
03037         if ( MulRat(BHEAD (UWORD *)t,REDLENG(l1),(UWORD *)a,1,(UWORD *)accum,&l1) ) {
03038             MLOCK(ErrorMessageLock);
03039             MesCall("PasteTerm");
03040             MUNLOCK(ErrorMessageLock);
03041             return(0);
03042         }
03043         x = l1;
03044         x *= 2;
03045         if ( x < 0 ) { accum -= x; *accum++ = x - 1; }
03046         else         { accum += x; *accum++ = x + 1; }
03047         *u = WORDDIF(accum,u);
03048     }
03049     if ( accum >= m ) {
03050 /* INTERNAL_ERROR_EXCL_START */
03051         MLOCK(ErrorMessageLock);
03052         MesPrint("!>Buffer too small in PasteTerm");
03053         MUNLOCK(ErrorMessageLock);
03054         return(0);
03055 /* INTERNAL_ERROR_EXCL_STOP */
03056     }
03057     *accum = 0;
03058     return(accum);
03059 }
03060
03061 /*
03062         #] PasteTerm :
03063         #[ FiniTerm :           WORD FiniTerm(term,accum,termout,number)
03064 */
03076 int FiniTerm(PHEAD WORD *term, WORD *accum, WORD *termout, WORD number, WORD tepos)
03077 {
03078     GETBIDENTITY
03079     WORD *m, *t, *r, i, numacc, l2, ipp;
03080     WORD *u, *v, l1, *coef = AT.WorkPointer, *oldaccum;
03081     if ( ( AT.WorkPointer = coef + 2*AM.MaxTal ) > AT.WorkTop ) {
03082         MLOCK(ErrorMessageLock);
03083         MesWork();
03084         MUNLOCK(ErrorMessageLock);
03085         return(-1);
```

```
03086     }
03087     oldaccum = accum;
03088     t = term;
03089     m = t + *t - 1;
03090     l1 = REDLENG(*m);
03091     i = ABS(*m) - 1;
03092     r = coef + i;
03093     do { *--r = *--m; } while ( --i > 0 ); /* Copies coefficient */
03094     if ( tepos > 0 ) {
03095         t = term + tepos;
03096         goto foundit;
03097     }
03098     t++;
03099     if ( t < m ) do {
03100         if ( ( ( *t == SUBEXPRESSION && ( *(r=t+t[1]) != SUBEXPRESSION
03101         || r >= m || !r[3] ) ) || *t == EXPRESSION ) && t[2] == number && t[3] ) {
03102 foundit:;
03103             u = m;
03104             r = term;
03105             m = termout;
03106             if ( r < t ) do { *m++ = *r++; } while ( r < t );
03107             numacc = *accum++;
03108             if ( numacc >= 0 ) do {
03109                 if ( *t == EXPRESSION ) {
03110                     v = t + t[1];
03111                     r = t + SUBEXPSIZE;
03112                     while ( r < v ) {
03113                         if ( *r == WILDCARDS ) {
03114                             r += 2;
03115                             i = *--m;
03116                             if ( ( l2 = WildFill(BHEAD m,accum,r) ) < 0 ) goto FiniCall;
03117                             goto AllWild;
03118                         }
03119                         r += r[1];
03120                     }
03121                     goto NoWild;
03122                 }
03123                 else if ( t[1] > SUBEXPSIZE && t[SUBEXPSIZE] != FROMBRAC ) {
03124                     i = *--m;
03125                     if ( ( l2 = WildFill(BHEAD m,accum,t) ) < 0 ) goto FiniCall;
03126 AllWild:            *m = i;
03127                     m += l2-1;
03128                     l2 = *m;
03129                     m -= ABS(l2) - 1;
03130                     r = m;
03131                 }
03132                 else {
03133 NoWild:             r = accum;
03134                     v = r + *r - 1;
03135                     l2 = *v;
03136                     v -= ABS(l2) - 1;
03137                     r++;
03138                     if ( r < v ) do { *m++ = *r++; } while ( r < v );
03139                 }
03140                 if ( *r == 1 && r[1] == 1 && ABS(l2) == 3 ) {
03141                     if ( l2 < 0 ) l1 = -l1;
03142                 }
03143                 else {
03144                     l2 = REDLENG(l2);
03145                     if ( l2 == 0 ) {
03146                         t = oldaccum;
03147                         numacc = *t++;
03148                         AO.OutSkip = 3;
03149                         FiniLine();
03150                         while ( --numacc >= 0 ) {
03151                             i = *t;
03152                             while ( --i >= 0 ) {
03153                                 TalToLine((UWORD)(*t++));
03154                                 TokenToLine((UBYTE *)"  ");
03155                             }
03156                         }
03157                         AO.OutSkip = 0;
03158                         FiniLine();
03159                         goto FiniCall;
03160                     }
03161                     if ( MulRat(BHEAD (UWORD *)coef,l1,(UWORD *)r,l2,(UWORD *)coef,&l1) ) goto
      FiniCall;
03162                     if ( AN.ncmod != 0 && TakeModulus((UWORD
      *)coef,&l1,AC.cmod,AN.ncmod,UNPACK|AC.modmode) ) goto FiniCall;
03163                 }
03164                 accum += *accum;
03165             } while ( --numacc >= 0 );
03166             if ( *t == SUBEXPRESSION ) {
03167                  while ( t+t[1] < u && t[t[1]] == DOLLAREXPR2 ) t += t[1];
03168             }
03169             t += t[1];
03170             if ( t < u ) do { *m++ = *t++; } while ( t < u );
```

```
03171             l2 = l1;
03172 /*
03173             Code to economize when taking x = (a+b)/2
03174 */
03175             r = termout+1;
03176             while ( r < m ) {
03177                 if ( *r == SUBEXPRESSION ) {
03178                     t = r + r[1];
03179                     l1 = (WORD)(cbuf[r[4]].CanCommu[r[2]]);
03180                     while ( t < m ) {
03181                         if ( *t == SUBEXPRESSION &&
03182                             t[1] == r[1] && t[2] == r[2] && t[4] == r[4] ) {
03183                                 i = t[1] - SUBEXPSIZE;
03184                                 u = r + SUBEXPSIZE; v = t + SUBEXPSIZE;
03185                                 while ( i > 0 ) {
03186                                     if ( *v++ != *u++ ) break;
03187                                     i--;
03188                                 }
03189                                 if ( i <= 0 ) {
03190                                     u = r;
03191                                     r[3] += t[3];
03192                                     r = t + t[1];
03193                                     while ( r < m ) *t++ = *r++;
03194                                     m = t;
03195                                     r = u;
03196                                     goto Nextr;
03197                                 }
03198                                 if ( l1 && cbuf[t[4]].CanCommu[t[2]] ) break;
03199                                 while ( t+t[1] < m && t[t[1]] == DOLLAREXPR2 ) t += t[1];
03200                         }
03201                         else if ( l1 ) {
03202                             if ( *t == SUBEXPRESSION && cbuf[t[4]].CanCommu[t[2]] )
03203                                 break;
03204                             if ( *t >= FUNCTION+WILDOFFSET )
03205                                 ipp = *t - WILDOFFSET;
03206                             else ipp = *t;
03207                             if ( *t >= FUNCTION
03208                              && functions[ipp-FUNCTION].commute && l1 ) break;
03209                             if ( *t == EXPRESSION ) break;
03210                         }
03211                         t += t[1];
03212                     }
03213                     r += r[1];
03214                 }
03215                 else r += r[1];
03216 Nextr:;
03217             }
03218
03219             i = ABS(l2);
03220             i *= 2;
03221             i++;
03222             l2 = ( l2 >= 0 ) ? i: -i;
03223             r = coef;
03224             while ( --i > 0 ) *m++ = *r++;
03225             *m++ = l2;
03226             *termout = WORDDIF(m,termout);
03227             AT.WorkPointer = coef;
03228             return(0);
03229         }
03230         t += t[1];
03231     } while ( t < m );
03232     AT.WorkPointer = coef;
03233     return(1);
03234
03235 FiniCall:
03236     MLOCK(ErrorMessageLock);
03237     MesCall("FiniTerm");
03238     MUNLOCK(ErrorMessageLock);
03239     SETERROR(-1)
03240 }
03241
03242 /*
03243         #] FiniTerm :
03244         #[ Generator :          WORD Generator(BHEAD term,level)
03245 */
03246
03247 static WORD zeroDollar[] = { 0, 0 };
03248 /*
03249 static LONG debugcounter = 0;
03250 */
03251
03275 int Generator(PHEAD WORD *term, WORD level)
03276 {
03277     GETBIDENTITY
03278     WORD replac, *accum, *termout, *t, i, j, tepos, applyflag = 0, *StartBuf;
03279     WORD *a, power, power1, DumNow = AR.CurDum, oldtoprhs, oldatoprhs, extractbuff;
03280     int ret;
```

```
03281     int *RepSto = AN.RepPoint, iscopy = 0;
03282     CBUF *C = cbuf+AM.rbufnum, *CC = cbuf + AT.ebufnum, *CCC = cbuf + AT.aebufnum;
03283     LONG posisub, oldcpointer, oldacpointer;
03284     DOLLARS d = 0;
03285     WORD numfac[5], idfunctionflag;
03286 #ifdef WITHPTHREADS
03287     int nummodopt, dtype = -1, id;
03288 #endif
03289     oldtoprhs = CC->numrhs;
03290     oldcpointer = CC->Pointer - CC->Buffer;
03291     oldatoprhs = CCC->numrhs;
03292     oldacpointer = CCC->Pointer - CCC->Buffer;
03293 ReStart:
03294     if ( ( replac = TestSub(BHEAD term,level) ) == 0 ) {
03295         if ( applyflag ) { TableReset(); applyflag = 0; }
03296 /*
03297         if ( AN.PolyNormFlag > 1 ) {
03298             if ( PolyFunMul(BHEAD term) < 0 ) goto GenCall;
03299             AN.PolyNormFlag = 0;
03300             if ( !*term ) goto Return0;
03301         }
03302 */
03303 Renormalize:
03304         AN.PolyNormFlag = 0;
03305         AN.idfunctionflag = 0;
03306         if ( ( ret = Normalize(BHEAD term) ) != 0 ) {
03307             if ( ret > 0 ) {
03308                 if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
03309                 goto ReStart;
03310             }
03311             goto GenCall;
03312         }
03313         idfunctionflag = AN.idfunctionflag;
03314         if ( !*term ) { AN.PolyNormFlag = 0; goto Return0; }
03315
03316         if ( AN.PolyNormFlag ) {
03317             if ( AN.PolyFunTodo == 0 ) {
03318                 if ( PolyFunMul(BHEAD term) < 0 ) goto GenCall;
03319                 if ( !*term ) { AN.PolyNormFlag = 0; goto Return0; }
03320             }
03321             else {
03322                 WORD oldPolyFunExp = AR.PolyFunExp;
03323                 AR.PolyFunExp = 0;
03324                 if ( PolyFunMul(BHEAD term) < 0 ) goto GenCall;
03325                 AT.WorkPointer = term+*term;
03326                 AR.PolyFunExp = oldPolyFunExp;
03327                 if ( !*term ) { AN.PolyNormFlag = 0; goto Return0; }
03328                 if ( Normalize(BHEAD term) < 0 ) goto GenCall;
03329                 if ( !*term ) { AN.PolyNormFlag = 0; goto Return0; }
03330                 AT.WorkPointer = term+*term;
03331                 if ( AN.PolyNormFlag ) {
03332                     if ( PolyFunMul(BHEAD term) < 0 ) goto GenCall;
03333                     if ( !*term ) { AN.PolyNormFlag = 0; goto Return0; }
03334                     AT.WorkPointer = term+*term;
03335                 }
03336                 AN.PolyFunTodo = 0;
03337             }
03338         }
03339         if ( idfunctionflag > 0 ) {
03340             if ( TakeIDfunction(BHEAD term) ) {
03341                 AT.WorkPointer = term + *term;
03342                 goto ReStart;
03343             }
03344         }
03345         if ( AT.WorkPointer < (WORD *)(((UBYTE *)(term)) + AM.MaxTer) )
03346              AT.WorkPointer = (WORD *)(((UBYTE *)(term)) + AM.MaxTer);
03347         do {
03348 SkipCount:  level++;
03349             if ( level > AR.Cnumlhs ) {
03350                 if ( AR.DeferFlag && AR.sLevel <= 0 ) {
03351 #ifdef WITHMPI
03352                     if ( PF.me != MASTER && AC.mparallelflag == PARALLELFLAG && PF.exprtodo < 0 ) {
03353                         if ( PF_Deferred(term,level) ) goto GenCall;
03354                     }
03355                     else
03356 #endif
03357                     {
03358                         if ( Deferred(BHEAD term,level) ) goto GenCall;
03359                     }
03360                     goto Return0;
03361                 }
03362                 if ( AN.ncmod != 0 ) {
03363                     if ( Modulus(term) ) goto GenCall;
03364                     if ( !*term ) goto Return0;
03365                 }
03366                 if ( AR.CurDum > AM.IndDum && AR.sLevel <= 0 ) {
03367                     WORD olddummies = AN.IndDum;
```

```
03368                     AN.IndDum = AM.IndDum;
03369                     ReNumber(BHEAD term);
03370                     Normalize(BHEAD term);
03371                     AN.IndDum = olddummies;
03372                     if ( !*term ) goto Return0;
03373                     olddummies = DetCurDum(BHEAD term);
03374                     if ( olddummies > AR.MaxDum ) AR.MaxDum = olddummies;
03375                 }
03376                 if ( AR.PolyFun > 0 && ( AR.sLevel <= 0 || AN.FunSorts[AR.sLevel]->PolyFlag > 0 ) ) {
03377                     if ( PrepPoly(BHEAD term,0) != 0 ) goto Return0;
03378                 }
03379                 else if ( AR.PolyFun > 0 ) {
03380                     if ( PrepPoly(BHEAD term,1) != 0 ) goto Return0;
03381                 }
03382                 if ( AR.sLevel <= 0 && AR.BracketOn ) {
03383                     if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
03384                     termout = AT.WorkPointer;
03385                     if ( AT.WorkPointer + *term + 3 > AT.WorkTop ) goto OverWork;
03386                     if ( PutBracket(BHEAD term) ) return(-1);
03387                     AN.RepPoint = RepSto;
03388                     *AT.WorkPointer = 0;
03389                     ret = StoreTerm(BHEAD termout);
03390                     AT.WorkPointer = termout;
03391                     CC->numrhs = oldtoprhs;
03392                     CC->Pointer = CC->Buffer + oldcpointer;
03393                     CCC->numrhs = oldatoprhs;
03394                     CCC->Pointer = CCC->Buffer + oldacpointer;
03395                     return(ret);
03396                 }
03397                 else {
03398                     if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
03399                     if ( AT.WorkPointer >= AT.WorkTop ) goto OverWork;
03400                     *AT.WorkPointer = 0;
03401                     AN.RepPoint = RepSto;
03402                     ret = StoreTerm(BHEAD term);
03403                     CC->numrhs = oldtoprhs;
03404                     CC->Pointer = CC->Buffer + oldcpointer;
03405                     CCC->numrhs = oldatoprhs;
03406                     CCC->Pointer = CCC->Buffer + oldacpointer;
03407                     return(ret);
03408                 }
03409             }
03410             i = C->lhs[level][0];
03411             if ( i >= TYPECOUNT ) {
03412 /*
03413             #[ Special action :
03414 */
03415                 switch ( i ) {
03416                   case TYPECOUNT:
03417                     if ( CountDo(term,C->lhs[level]) < C->lhs[level][2] ) {
03418                         AT.WorkPointer = term + *term;
03419                         goto Return0;
03420                     }
03421                     break;
03422                   case TYPEMULT:
03423                     if ( MultDo(BHEAD term,C->lhs[level]) ) goto GenCall;
03424                     goto ReStart;
03425                   case TYPEGOTO:
03426                     level = AC.Labels[C->lhs[level][2]];
03427                     break;
03428                   case TYPEDISCARD:
03429                     AT.WorkPointer = term + *term;
03430                     goto Return0;
03431                   case TYPEIF:
03432 #ifdef WITHPTHREADS
03433                     {
03434 /*
03435                         We may be writing in the space here when wildcards
03436                         are involved in a match(). Hence we have to make
03437                         a private copy here!!!!
03438 */
03439                         WORD ic, jc, *ifcode, *jfcode;
03440                         jfcode = C->lhs[level]; jc = jfcode[1];
03441                         ifcode = AT.WorkPointer; AT.WorkPointer += jc;
03442                         for ( ic = 0; ic < jc; ic++ ) ifcode[ic] = jfcode[ic];
03443                         while ( !DoIfStatement(BHEAD ifcode,term) ) {
03444                             level = C->lhs[level][2];
03445                             if ( C->lhs[level][0] != TYPEELIF ) break;
03446                         }
03447                         AT.WorkPointer = ifcode;
03448                     }
03449 #else
03450                     while ( !DoIfStatement(BHEAD C->lhs[level],term) ) {
03451                         level = C->lhs[level][2];
03452                         if ( C->lhs[level][0] != TYPEELIF ) break;
03453                     }
03454 #endif
```

```
03455                     break;
03456                   case TYPEELIF:
03457                     do {
03458                         level = C->lhs[level][2];
03459                     } while ( C->lhs[level][0] == TYPEELIF );
03460                     break;
03461                   case TYPEELSE:
03462                   case TYPEENDIF:
03463                     level = C->lhs[level][2];
03464                     break;
03465                   case TYPESUMFIX:
03466                     {
03467                         WORD *cp = AR.CompressPointer, *op = AR.CompressPointer;
03468                         WORD *tlhs = C->lhs[level] + 3, *m, jlhs;
03469                         WORD theindex = C->lhs[level][2];
03470                         if ( theindex < 0 ) {   /* $-variable */
03471 #ifdef WITHPTHREADS
03472                             int ddtype = -1;
03473                             theindex = -theindex;
03474                             d = Dollars + theindex;
03475                             if ( AS.MultiThreaded ) {
03476                                 for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
03477                                     if ( theindex == ModOptdollars[nummodopt].number ) break;
03478                                 }
03479                                 if ( nummodopt < NumModOptdollars ) {
03480                                     ddtype = ModOptdollars[nummodopt].type;
03481                                     if ( DollarLocalCopy(ddtype) ) {
03482                                         d = ModOptdollars[nummodopt].dstruct+AT.identity;
03483                                     }
03484                                     else {
03485                                         LOCK(d->pthreadslock);
03486                                     }
03487                                 }
03488                             }
03489 #else
03490                             theindex = -theindex;
03491                             d = Dollars + theindex;
03492 #endif
03493
03494                             if ( d->type != DOLINDEX
03495                             || d->index < AM.OffsetIndex
03496                             || d->index >= AM.OffsetIndex + WILDOFFSET ) {
03497                                 MLOCK(ErrorMessageLock);
03498                                 MesPrint("$%s should have been an index"
03499                                 ,AC.dollarnames->namebuffer+d->name);
03500                                 AN.currentTerm = term;
03501                                 MesPrint("Current term: %t");
03502                                 AN.listinprint = printscratch;
03503                                 printscratch[0] = DOLLAREXPRESSION;
03504                                 printscratch[1] = theindex;
03505                                 MesPrint("$%s = %$"
03506                                 ,AC.dollarnames->namebuffer+d->name);
03507                                 MUNLOCK(ErrorMessageLock);
03508 #ifdef WITHPTHREADS
03509                                 if ( ddtype > 0 && ! DollarLocalCopy(ddtype) ) { UNLOCK(d->pthreadslock);
    }
03510 #endif
03511                                 goto GenCall;
03512                             }
03513                             theindex = d->index;
03514 #ifdef WITHPTHREADS
03515                             if ( ddtype > 0 && ! DollarLocalCopy(ddtype) ) { UNLOCK(d->pthreadslock);
    }
03516 #endif
03517                         }
03518                         cp[1] = SUBEXPSIZE+4;
03519                         cp += SUBEXPSIZE;
03520                         *cp++ = INDTOIND;
03521                         *cp++ = 4;
03522                         *cp++ = theindex;
03523                         i = C->lhs[level][1] - 3;
03524                         cp++;
03525                         AR.CompressPointer = cp;
03526                         while ( --i >= 0 ) {
03527                             cp[-1] = *tlhs++;
03528                             termout = AT.WorkPointer;
03529                             if ( ( jlhs = WildFill(BHEAD termout,term,op)) < 0 )
03530                                 goto GenCall;
03531                             m = term;
03532                             jlhs = *m;
03533                             while ( --jlhs >= 0 ) {
03534                                 if ( *m++ != *termout++ ) break;
03535                             }
03536                             if ( jlhs >= 0 ) {
03537                                 termout = AT.WorkPointer;
03538                                 AT.WorkPointer = termout + *termout;
03539                                 if ( Generator(BHEAD termout,level) ) goto GenCall;
```

```
03540                                   AT.WorkPointer = termout;
03541                               }
03542                               else {
03543                                   AR.CompressPointer = op;
03544                                   goto SkipCount;
03545                               }
03546                           }
03547                           AR.CompressPointer = op;
03548                           goto CommonEnd;
03549                       }
03550                   case TYPESUM:
03551                       {
03552                           WORD *wp, *cp = AR.CompressPointer, *op = AR.CompressPointer;
03553                           WORD theindex;
03554                           WORD *ow;
03555 /*
03556                           At this point it is safest to determine CurDum
03557 */
03558                           AR.CurDum = DetCurDum(BHEAD term);
03559                           i = C->lhs[level][1]-2;
03560                           wp = C->lhs[level] + 2;
03561                           cp[1] = SUBEXPSIZE+4*i;
03562                           cp += SUBEXPSIZE;
03563                           while ( --i >= 0 ) {
03564                               theindex = *wp++;
03565                               if ( theindex < 0 ) {   /* $-variable */
03566 #ifdef WITHPTHREADS
03567                                   int ddtype = -1;
03568                                   theindex = -theindex;
03569                                   d = Dollars + theindex;
03570                                   if ( AS.MultiThreaded ) {
03571                                       for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
03572                                           if ( theindex == ModOptdollars[nummodopt].number ) break;
03573                                       }
03574                                       if ( nummodopt < NumModOptdollars ) {
03575                                           ddtype = ModOptdollars[nummodopt].type;
03576                                           if ( DollarLocalCopy(ddtype) ) {
03577                                               d = ModOptdollars[nummodopt].dstruct+AT.identity;
03578                                           }
03579                                           else {
03580                                               LOCK(d->pthreadslock);
03581                                           }
03582                                       }
03583                                   }
03584 #else
03585                                   theindex = -theindex;
03586                                   d = Dollars + theindex;
03587 #endif
03588                                   if ( d->type != DOLINDEX
03589                                   || d->index < AM.OffsetIndex
03590                                   || d->index >= AM.OffsetIndex + WILDOFFSET ) {
03591                                       MLOCK(ErrorMessageLock);
03592                                       MesPrint("$%s should have been an index"
03593                                       ,AC.dollarnames->namebuffer+d->name);
03594                                       AN.currentTerm = term;
03595                                       MesPrint("Current term: %t");
03596                                       AN.listinprint = printscratch;
03597                                       printscratch[0] = DOLLAREXPRESSION;
03598                                       printscratch[1] = theindex;
03599                                       MesPrint("$%s = %$"
03600                                       ,AC.dollarnames->namebuffer+d->name);
03601                                       MUNLOCK(ErrorMessageLock);
03602 #ifdef WITHPTHREADS
03603                                       if ( ddtype > 0 && ! DollarLocalCopy(ddtype) ) {
      UNLOCK(d->pthreadslock); }
03604 #endif
03605                                       goto GenCall;
03606                                   }
03607                                   theindex = d->index;
03608 #ifdef WITHPTHREADS
03609                                   if ( ddtype > 0 && ! DollarLocalCopy(ddtype) ) {
      UNLOCK(d->pthreadslock); }
03610 #endif
03611                               }
03612                               *cp++ = INDTOIND;
03613                               *cp++ = 4;
03614                               *cp++ = theindex;
03615                               *cp++ = ++AR.CurDum;
03616                           }
03617                           ow = AT.WorkPointer;
03618                           AR.CompressPointer = cp;
03619                           if ( WildFill(BHEAD ow,term,op) < 0 ) goto GenCall;
03620                           AR.CompressPointer = op;
03621                           i = ow[0];
03622                           WORD term_changed = 0;
03623                           for ( j = 0; j < i; j++ ) {
03624                               if ( term[j] != ow[j] ) term_changed = 1;
```

```
03625                           term[j] = ow[j];
03626                       }
03627                       // If the term was modified by WildFill, set RepCount.
03628                       if ( term_changed ) *AN.RepPoint = 1;
03629                       AT.WorkPointer = ow;
03630                       // Most other calls to ReNumber reset AN.IndDum to AM.IndDum first. Here it is
03631                       // not done, but doing it is one way to fix Issue #710. The fix that is
      actually
03632                       // implemented is to change a comparison in FunLevel and then a reset of
03633                       // AN.IndDum appears unnecessary here. But maybe one day this comment is
      useful.
03634                       ReNumber(BHEAD term);
03635                       goto Renormalize;
03636                     }
03637                   case TYPECHISHOLM:
03638                     if ( Chisholm(BHEAD term,level) ) goto GenCall;
03639 CommonEnd:
03640                     AT.WorkPointer = term + *term;
03641                     goto Return0;
03642                   case TYPEARG:
03643                     if ( ( i = execarg(BHEAD term,level) ) < 0 ) goto GenCall;
03644                     level = C->lhs[level][2];
03645                     if ( i > 0 ) goto ReStart;
03646                     break;
03647                   case TYPENORM:
03648                   case TYPENORM2:
03649                   case TYPENORM3:
03650                   case TYPENORM4:
03651                   case TYPESPLITARG:
03652                   case TYPESPLITARG2:
03653                   case TYPESPLITFIRSTARG:
03654                   case TYPESPLITLASTARG:
03655                   case TYPEARGTOEXTRASYMBOL:
03656                     if ( execarg(BHEAD term,level) < 0 ) goto GenCall;
03657                     level = C->lhs[level][2];
03658                     break;
03659                   case TYPEFACTARG:
03660                   case TYPEFACTARG2:
03661                     { WORD jjj;
03662                     if ( ( jjj = execarg(BHEAD term,level) ) < 0 ) goto GenCall;
03663                     if ( jjj > 0 ) goto ReStart;
03664                     level = C->lhs[level][2];
03665                     break; }
03666                   case TYPEEXIT:
03667                     if ( C->lhs[level][2] > 0 ) {
03668                         MLOCK(ErrorMessageLock);
03669                         MesPrint("%s",C->lhs[level]+3);
03670                         MUNLOCK(ErrorMessageLock);
03671                     }
03672                     Terminate(-1);
03673                     goto GenCall;
03674                   case TYPESETEXIT:
03675                     AM.exitflag = 1; /* no danger of race conditions */
03676                     break;
03677                   case TYPEPRINT:
03678                     AN.currentTerm = term;
03679                     AN.numlistinprint = (C->lhs[level][1] - C->lhs[level][4] - 5)/2;
03680                     AN.listinprint = C->lhs[level]+5+C->lhs[level][4];
03681                     MLOCK(ErrorMessageLock);
03682                     AO.ErrorBlock = 1;
03683                     MesPrint((char *)(C->lhs[level]+5));
03684                     AO.ErrorBlock = 0;
03685                     MUNLOCK(ErrorMessageLock);
03686                     break;
03687                   case TYPEFPRINT:
03688                     {
03689                     int oldFOflag;
03690                     WORD oldPrintType, oldLogHandle = AC.LogHandle;
03691                     AC.LogHandle = C->lhs[level][2];
03692                     MLOCK(ErrorMessageLock);
03693                     oldFOflag = AM.FileOnlyFlag;
03694                     oldPrintType = AO.PrintType;
03695                     if ( AC.LogHandle >= 0 ) {
03696                         AM.FileOnlyFlag = 1;
03697                         AO.PrintType |= PRINTLFILE;
03698                     }
03699                     AO.PrintType |= C->lhs[level][3];
03700                     AN.currentTerm = term;
03701                     AN.numlistinprint = (C->lhs[level][1] - C->lhs[level][4] - 5)/2;
03702                     AN.listinprint = C->lhs[level]+5+C->lhs[level][4];
03703                     MesPrint((char *)(C->lhs[level]+5));
03704                     AO.PrintType = oldPrintType;
03705                     AM.FileOnlyFlag = oldFOflag;
03706                     MUNLOCK(ErrorMessageLock);
03707                     AC.LogHandle = oldLogHandle;
03708                     }
03709                     break;
```

```
03710                   case TYPEREDEFPRE:
03711                     j = C->lhs[level][2];
03712 #ifdef WITHMPI
03713                     {
03714                         /*
03715                          * Regardless of parallel/nonparallel switch, we need to set
03716                          * AC.inputnumbers[ii], which indicates that the corresponding
03717                          * preprocessor variable is redefined and so we need to
03718                          * send/broadcast it.
03719                          */
03720                         int ii;
03721                         for ( ii = 0; ii < AC.numpfirstnum; ii++ ) {
03722                             if ( AC.pfirstnum[ii] == j ) break;
03723                         }
03724                         AC.inputnumbers[ii] = AN.ninterms;
03725                     }
03726 #endif
03727 #ifdef WITHPTHREADS
03728                     if ( AS.MultiThreaded ) {
03729                         int ii;
03730                         for ( ii = 0; ii < AC.numpfirstnum; ii++ ) {
03731                             if ( AC.pfirstnum[ii] == j ) break;
03732                         }
03733                         if ( AN.inputnumber < AC.inputnumbers[ii] ) break;
03734                         LOCK(AP.PreVarLock);
03735                         if ( AN.inputnumber >= AC.inputnumbers[ii] ) {
03736                             a = C->lhs[level]+4;
03737                             if ( a[a[-1]] == 0 )
03738                                 PutPreVar(PreVar[j].name,(UBYTE *)(a),0,1);
03739                             else
03740                                 PutPreVar(PreVar[j].name,(UBYTE *)(a)
03741                                     ,(UBYTE *)(a+a[-1]+1),1);
03742 /*
03743                             PutPreVar(PreVar[j].name,(UBYTE *)(C->lhs[level]+4),0,1);
03744 */
03745                             AC.inputnumbers[ii] = AN.inputnumber;
03746                         }
03747                         UNLOCK(AP.PreVarLock);
03748                     }
03749                     else
03750 #endif
03751                     {
03752                         a = C->lhs[level]+4;
03753                         LOCK(AP.PreVarLock);
03754                         if ( a[a[-1]] == 0 )
03755                             PutPreVar(PreVar[j].name,(UBYTE *)(a),0,1);
03756                         else
03757                             PutPreVar(PreVar[j].name,(UBYTE *)(a)
03758                                 ,(UBYTE *)(a+a[-1]+1),1);
03759                         UNLOCK(AP.PreVarLock);
03760                     }
03761                     break;
03762                   case TYPERENUMBER:
03763                     AT.WorkPointer = term + *term;
03764                     if ( FullRenumber(BHEAD term,C->lhs[level][2]) ) goto GenCall;
03765                     AT.WorkPointer = term + *term;
03766                     if ( *term == 0 ) goto Return0;
03767                     break;
03768                   case TYPETRY:
03769                     if ( TryDo(BHEAD term,C->lhs[level],level) ) goto GenCall;
03770                     AT.WorkPointer = term + *term;
03771                     goto Return0;
03772                   case TYPEASSIGN:
03773                     { WORD onc = AR.NoCompress, oldEside = AR.Eside;
03774                     WORD oldrepeat = *AN.RepPoint;
03775 /*
03776                     Here we have to assign an expression to a $ variable.
03777 */
03778                     AR.Eside = RHSIDE;
03779                     AR.NoCompress = 1;
03780                     AN.cTerm = AN.currentTerm = term;
03781                     AT.WorkPointer = term + *term;
03782                     *AT.WorkPointer++ = 0;
03783                     if ( AssignDollar(BHEAD term,level) ) goto GenCall;
03784                     AT.WorkPointer = term + *term;
03785                     AN.cTerm = 0;
03786                     *AN.RepPoint = oldrepeat;
03787                     AR.NoCompress = onc;
03788                     AR.Eside = oldEside;
03789                     break;
03790                     }
03791                   case TYPEFINDLOOP:
03792                     if ( Lus(term,C->lhs[level][3],C->lhs[level][4],
03793                     C->lhs[level][5],C->lhs[level][6],C->lhs[level][2]) ) {
03794                         AT.WorkPointer = term + *term;
03795                         goto Renormalize;
03796                     }
```

```
03797                     break;
03798                   case TYPEINSIDE:
03799                     if ( InsideDollar(BHEAD C->lhs[level],level) < 0 ) goto GenCall;
03800                     level = C->lhs[level][2];
03801                     break;
03802                   case TYPETERM:
03803                     ret = execterm(BHEAD term,level);
03804                     AN.RepPoint = RepSto;
03805                     AR.CurDum = DumNow;
03806                     CC->numrhs = oldtoprhs;
03807                     CC->Pointer = CC->Buffer + oldcpointer;
03808                     CCC->numrhs = oldatoprhs;
03809                     CCC->Pointer = CCC->Buffer + oldacpointer;
03810                     return(ret);
03811                   case TYPEDETCURDUM:
03812                     AT.WorkPointer = term + *term;
03813                     AR.CurDum = DetCurDum(BHEAD term);
03814                     break;
03815                   case TYPEINEXPRESSION:
03816                     {WORD *ll = C->lhs[level];
03817                     int numexprs = (int)(ll[1]-3);
03818                     ll += 3;
03819                     while ( numexprs-- >= 0 ) {
03820                         if ( *ll == AR.CurExpr ) break;
03821                         ll++;
03822                     }
03823                     if ( numexprs < 0 ) level = C->lhs[level][2];
03824                     }
03825                     break;
03826                   case TYPEMERGE:
03827                     AT.WorkPointer = term + *term;
03828                     if ( DoShuffle(term,level,C->lhs[level][2],C->lhs[level][3]) )
03829                         goto GenCall;
03830                     AT.WorkPointer = term + *term;
03831                     goto Return0;
03832                   case TYPESTUFFLE:
03833                     AT.WorkPointer = term + *term;
03834                     if ( DoStuffle(term,level,C->lhs[level][2],C->lhs[level][3]) )
03835                         goto GenCall;
03836                     AT.WorkPointer = term + *term;
03837                     goto Return0;
03838                   case TYPETESTUSE:
03839                     AT.WorkPointer = term + *term;
03840                     if ( TestUse(term,level) ) goto GenCall;
03841                     AT.WorkPointer = term + *term;
03842                     break;
03843                   case TYPEAPPLY:
03844                     AT.WorkPointer = term + *term;
03845                     if ( ApplyExec(term,C->lhs[level][2],level) < C->lhs[level][2] ) {
03846                         AT.WorkPointer = term + *term;
03847                         *AN.RepPoint = 1;
03848                         goto ReStart;
03849                     }
03850                     AT.WorkPointer = term + *term;
03851                     break;
03852 /*
03853                   case TYPEAPPLYRESET:
03854                     AT.WorkPointer = term + *term;
03855                     if ( ApplyReset(level) ) goto GenCall;
03856                     AT.WorkPointer = term + *term;
03857                     break;
03858 */
03859                   case TYPECHAININ:
03860                     { int lter = *term;
03861                     AT.WorkPointer = term + *term;
03862                     if ( ChainIn(BHEAD term,C->lhs[level][2]) ) goto GenCall;
03863                     AT.WorkPointer = term + *term;
03864                     /* Symmetry properties might mean the term has vanished */
03865                     if ( *term == 0 ) goto Return0;
03866                     if ( *term != lter ) *AN.RepPoint = 1;
03867                     }
03868                     break;
03869                   case TYPECHAINOUT:
03870                     { int lter = *term;
03871                     AT.WorkPointer = term + *term;
03872                     if ( ChainOut(BHEAD term,C->lhs[level][2]) ) goto GenCall;
03873                     AT.WorkPointer = term + *term;
03874                     if ( *term != lter ) *AN.RepPoint = 1;
03875                     }
03876                     break;
03877                   case TYPEFACTOR:
03878                     AT.WorkPointer = term + *term;
03879                     if ( DollarFactorize(BHEAD C->lhs[level][2]) ) goto GenCall;
03880                     AT.WorkPointer = term + *term;
03881                     break;
03882                   case TYPEARGIMPLODE:
03883                     AT.WorkPointer = term + *term;
```

```
03884                   if ( ArgumentImplode(BHEAD term,C->lhs[level]) ) goto GenCall;
03885                   AT.WorkPointer = term + *term;
03886                   break;
03887                 case TYPEARGEXPLODE:
03888                   AT.WorkPointer = term + *term;
03889                   if ( ArgumentExplode(BHEAD term,C->lhs[level]) ) goto GenCall;
03890                   AT.WorkPointer = term + *term;
03891                   break;
03892                 case TYPEDENOMINATORS:
03893                   if ( DenToFunction(term,C->lhs[level][2]) ) goto ReStart;
03894                   break;
03895                 case TYPEDROPCOEFFICIENT:
03896                   DropCoefficient(BHEAD term);
03897                   break;
03898                 case TYPETRANSFORM:
03899                   AT.WorkPointer = term + *term;
03900                   if ( RunTransform(BHEAD term,C->lhs[level]+2) ) goto GenCall;
03901                   AT.WorkPointer = term + *term;
03902                   if ( *term == 0 ) goto Return0;
03903                   goto ReStart;
03904                 case TYPETOPOLYNOMIAL:
03905                   AT.WorkPointer = term + *term;
03906                   termout = AT.WorkPointer;
03907                   if ( ConvertToPoly(BHEAD term,termout,C->lhs[level],0) < 0 ) goto GenCall;
03908                   if ( *termout == 0 ) goto Return0;
03909                   i = termout[0]; t = term; NCOPY(t,termout,i);
03910                   AT.WorkPointer = term + *term;
03911                   break;
03912                 case TYPEFROMPOLYNOMIAL:
03913                   AT.WorkPointer = term + *term;
03914                   termout = AT.WorkPointer;
03915                   if ( ConvertFromPoly(BHEAD term,termout,0,numxsymbol,0,0) < 0 ) goto GenCall;
03916                   if ( *term == 0 ) goto Return0;
03917                   i = termout[0]; t = term; NCOPY(t,termout,i);
03918                   AT.WorkPointer = term + *term;
03919                   goto ReStart;
03920                 case TYPEDOLOOP:
03921                   level = TestDoLoop(BHEAD C->lhs[level],level);
03922                   if ( level < 0 ) goto GenCall;
03923                   break;
03924                 case TYPEENDDOLOOP:
03925                   level = TestEndDoLoop(BHEAD C->lhs[C->lhs[level][2]],C->lhs[level][2]);
03926                   if ( level < 0 ) goto GenCall;
03927                   break;
03928                 case TYPEDROPSYMBOLS:
03929                   DropSymbols(BHEAD term);
03930                   break;
03931                 case TYPEPUTINSIDE:
03932                   AT.WorkPointer = term + *term;
03933                   if ( PutInside(BHEAD term,C->lhs[level]) < 0 ) goto GenCall;
03934                   AT.WorkPointer = term + *term;
03935                   /*
03936                    * We need to call Generator() to convert slow notation to
03937                    * fast notation, which fixes Issue #30.
03938                    */
03939                   if ( Generator(BHEAD term,level) < 0 ) goto GenCall;
03940                   goto Return0;
03941                 case TYPETOSPECTATOR:
03942                   if ( PutInSpectator(term,C->lhs[level][2]) < 0 ) goto GenCall;
03943                   goto Return0;
03944                 case TYPECANONICALIZE:
03945                   AT.WorkPointer = term + *term;
03946                   if ( DoCanonicalize(BHEAD term,C->lhs[level]) ) goto GenCall;
03947                   AT.WorkPointer = term + *term;
03948                   if ( *term == 0 ) goto Return0;
03949                   break;
03950                 case TYPESWITCH:
03951                   AT.WorkPointer = term + *term;
03952                   if ( DoSwitch(BHEAD term,C->lhs[level]) ) goto GenCall;
03953                   goto Return0;
03954                 case TYPEENDSWITCH:
03955                   AT.WorkPointer = term + *term;
03956                   if ( DoEndSwitch(BHEAD term,C->lhs[level]) ) goto GenCall;
03957                   goto Return0;
03958                 case TYPESETUSERFLAG:
03959                   Expressions[AR.CurExpr].uflags |= 1 << (C->lhs[level][2]);
03960                   break;
03961                 case TYPECLEARUSERFLAG:
03962                   Expressions[AR.CurExpr].uflags &= ~(1 << (C->lhs[level][2]));
03963                   break;
03964                 case TYPEALLLOOPS:
03965                   AT.WorkPointer = term + *term;
03966                   if ( AllLoops(BHEAD term,level) ) goto GenCall;
03967                   goto Return0;
03968                 case TYPEALLPATHS:
03969                   AT.WorkPointer = term + *term;
03970                   if ( AllPaths(BHEAD term,level) ) goto GenCall;
```

```
03971                     goto Return0;
03972 #ifdef WITHFLOAT
03973                   case TYPEEVALUATE:
03974                     AT.WorkPointer = term + *term;
03975                     if ( C->lhs[level][2] == MZV
03976                          || C->lhs[level][2] == EULER
03977                          || C->lhs[level][2] == MZVHALF
03978                          || C->lhs[level][2] == ALLMZVFUNCTIONS
03979                          ) {
03980                         if ( EvaluateEuler(BHEAD term,level,C->lhs[level][2]) ) goto GenCall;
03981                     }
03982                     else {
03983                         if ( EvaluateFun(BHEAD term,level,C->lhs[level]) ) goto GenCall;
03984                     }
03985 /*
03986                     else if ( C->lhs[level][2] == SQRTFUNCTION ) {
03987                         if ( EvaluateSqrt(BHEAD term,level,C->lhs[level][2]) ) goto GenCall;
03988                     }
03989                     else {
03990                         MLOCK(ErrorMessageLock);
03991                         MesPrint("Illegal function %d in evaluate statement.",C->lhs[level][2]);
03992                         MUNLOCK(ErrorMessageLock);
03993                         goto GenCall;
03994                     }
03995 */
03996                     goto Return0;
03997                   case TYPETOFLOAT:
03998                     AT.WorkPointer = term + *term;
03999                     if ( ToFloat(BHEAD term,level) ) goto GenCall;
04000                     goto Return0;
04001                   case TYPETORAT:
04002                     AT.WorkPointer = term + *term;
04003                     if ( ToRat(BHEAD term,level) ) goto GenCall;
04004                     goto Return0;
04005                   case TYPESTRICTROUNDING:
04006                     AT.WorkPointer = term + *term;
04007                     if ( StrictRounding(BHEAD term,level,C->lhs[level][2],C->lhs[level][3]) ) goto
      GenCall;
04008                     goto Return0;
04009                   case TYPECHOP:
04010                     AT.WorkPointer = term + *term;
04011                     if ( Chop(BHEAD term,level) ) goto GenCall;
04012                     goto Return0;
04013 #endif
04014                 }
04015                 goto SkipCount;
04016 /*
04017             #] Special action :
04018 */
04019             }
04020         } while ( ( i = TestMatch(BHEAD term,&level) ) == 0 );
04021         if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
04022         if ( i > 0 ) replac = TestSub(BHEAD term,level);
04023         else replac = i;
04024         if ( replac >= 0 || AT.TMout[1] != SYMMETRIZE ) {
04025             *AN.RepPoint = 1;
04026             AR.expchanged = 1;
04027         }
04028         if ( replac < 0 ) {     /* Terms come from automatic generation */
04029 AutoGen:    i = *AT.TMout;
04030             t = termout = AT.WorkPointer;
04031             if ( ( AT.WorkPointer += i ) > AT.WorkTop ) goto OverWork;
04032             accum = AT.TMout;
04033             while ( --i >= 0 ) *t++ = *accum++;
04034             if ( (*(FG.Operation[termout[1]]))(BHEAD term,termout,replac,level) ) goto GenCall;
04035             AT.WorkPointer = termout;
04036             goto Return0;
04037         }
04038     }
04039     if ( applyflag ) { TableReset(); applyflag = 0; }
04040
04041     if ( AN.TeInFun ) { /* Match in function argument */
04042         if ( AN.TeInFun < 0 && !AN.TeSuOut ) {
04043
04044             if ( AR.TePos >= 0 ) goto AutoGen;
04045             switch ( AN.TeInFun ) {
04046                 case -1:
04047                     if ( DoDistrib(BHEAD term,level) ) goto GenCall;
04048                     break;
04049                 case -2:
04050                     if ( DoDelta3(BHEAD term,level) ) goto GenCall;
04051                     break;
04052                 case -3:
04053                     if ( DoTableExpansion(term,level) ) goto GenCall;
04054                     break;
04055                 case -4:
04056                     if ( FactorIn(BHEAD term,level) ) goto GenCall;
```

```
04057                   break;
04058               case -5:
04059                   if ( FactorInExpr(BHEAD term,level) ) goto GenCall;
04060                   break;
04061               case -6:
04062                   if ( TermsInBracket(BHEAD term,level) < 0 ) goto GenCall;
04063                   break;
04064               case -7:
04065                   if ( ExtraSymFun(BHEAD term,level) < 0 ) goto GenCall;
04066                   break;
04067               case -8:
04068                   if ( GCDfunction(BHEAD term,level) < 0 ) goto GenCall;
04069                   break;
04070               case -9:
04071                   if ( DIVfunction(BHEAD term,level,0) < 0 ) goto GenCall;
04072                   break;
04073               case -10:
04074                   if ( DIVfunction(BHEAD term,level,1) < 0 ) goto GenCall;
04075                   break;
04076               case -11:
04077                   if ( DIVfunction(BHEAD term,level,2) < 0 ) goto GenCall;
04078                   break;
04079               case -12:
04080                   if ( DoPermutations(BHEAD term,level) ) goto GenCall;
04081                   break;
04082               case -13:
04083                   if ( DoPartitions(BHEAD term,level) ) goto GenCall;
04084                   break;
04085               case -14:
04086                   if ( DIVfunction(BHEAD term,level,3) < 0 ) goto GenCall;
04087                   break;
04088               case -15:
04089                   if ( GenDiagrams(BHEAD term,level) < 0 ) goto GenCall;
04090                   break;
04091           }
04092       }
04093       else {
04094           termout = AT.WorkPointer;
04095           AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
04096           if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
04097           if ( InFunction(BHEAD term,termout) ) goto GenCall;
04098           AT.WorkPointer = termout + *termout;
04099           *AN.RepPoint = 1;
04100           AR.expchanged = 1;
04101           if ( *termout && Generator(BHEAD termout,level) < 0 ) goto GenCall;
04102           AT.WorkPointer = termout;
04103       }
04104   }
04105   else if ( replac > 0 ) {
04106       power = AN.TeSuOut;
04107       tepos = AR.TePos;
04108       if ( power < 0 ) {  /* Table expansion */
04109           power = -power; tepos = 0;
04110       }
04111       extractbuff = AT.TMbuff;
04112       if ( extractbuff == AM.dbufnum ) {
04113           d = DolToTerms(BHEAD replac);
04114           if ( d && d->where != 0 ) {
04115               iscopy = 1;
04116               if ( AT.TMdolfac > 0 ) {    /* We need a factor */
04117                 if ( AT.TMdolfac == 1 ) {
04118                   if ( d->nfactors ) {
04119                       numfac[0] = 4;
04120                       numfac[1] = d->nfactors;
04121                       numfac[2] = 1;
04122                       numfac[3] = 3;
04123                       numfac[4] = 0;
04124                   }
04125                   else {
04126                       numfac[0] = 0;
04127                   }
04128                   StartBuf = numfac;
04129                 }
04130                 else {
04131                   if ( (AT.TMdolfac-1) > d->nfactors && d->nfactors > 0 ) {
04132                       MLOCK(ErrorMessageLock);
04133                       MesPrint("Attempt to use an nonexisting factor %d of a
    $-variable",(WORD)(AT.TMdolfac-1));
04134                       if ( d->nfactors == 1 )
04135                           MesPrint("There is only one factor");
04136                       else
04137                           MesPrint("There are only %d factors",(WORD)(d->nfactors));
04138                       MUNLOCK(ErrorMessageLock);
04139                       goto GenCall;
04140                   }
04141                   if ( d->nfactors > 1 ) {
04142                       DOLLARS dd;
```

```
04143                           LONG dsize;
04144                           WORD *td1, *td2;
04145                           dd = Dollars + replac;
04146 #ifdef WITHPTHREADS
04147                           {
04148                               int nummodopt, dtype = -1;
04149                               if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
04150                                   for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
04151                                       if ( replac == ModOptdollars[nummodopt].number ) break;
04152                                   }
04153                                   if ( nummodopt < NumModOptdollars ) {
04154                                       dtype = ModOptdollars[nummodopt].type;
04155                                       if ( DollarLocalCopy(dtype) ) {
04156                                           dd = ModOptdollars[nummodopt].dstruct+AT.identity;
04157                                       }
04158                                   }
04159                               }
04160                           }
04161 #endif
04162                           dsize = dd->factors[AT.TMdolfac-2].size;
04163 /*
04164                           We copy only the factor we need
04165 */
04166                           if ( dsize == 0 ) {
04167                               numfac[0] = 4;
04168                               numfac[1] = d->factors[AT.TMdolfac-2].value;
04169                               numfac[2] = 1;
04170                               numfac[3] = 3;
04171                               numfac[4] = 0;
04172                               StartBuf = numfac;
04173                               if ( numfac[1] < 0 ) {
04174                                   numfac[1] = -numfac[1];
04175                                   numfac[3] = -numfac[3];
04176                               }
04177                           }
04178                           else {
04179                           d->factors[AT.TMdolfac-2].where = td2 = (WORD *)Malloc1(
04180                               (dsize+1)*sizeof(WORD),"Copy of factor");
04181                           td1 = dd->factors[AT.TMdolfac-2].where;
04182                           StartBuf = td2;
04183                           d->size = dsize; d->type = DOLTERMS;
04184                           NCOPY(td2,td1,dsize);
04185                           *td2 = 0;
04186                           }
04187                       }
04188                       else if ( d->nfactors == 1 ) {
04189                           StartBuf = d->where;
04190                       }
04191                       else {
04192                           MLOCK(ErrorMessageLock);
04193                           if ( d->nfactors == 0 ) {
04194                               MesPrint("Attempt to use factor %d of an unfactored
    $-variable",(WORD)(AT.TMdolfac-1));
04195                           }
04196                           else {
04197 /* INTERNAL_ERROR_EXCL_START */
04198                               MesPrint("!>Internal error. Illegal number of factors for $-variable");
04199 /* INTERNAL_ERROR_EXCL_STOP */
04200                           }
04201                           MUNLOCK(ErrorMessageLock);
04202                           goto GenCall;
04203                       }
04204                   }
04205               }
04206               else StartBuf = d->where;
04207           }
04208           else {
04209               d = Dollars + replac;
04210               StartBuf = zeroDollar;
04211           }
04212           posisub = 0;
04213           i = DetCommu(d->where);
04214 #ifdef WITHPTHREADS
04215           if ( AS.MultiThreaded ) {
04216               for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
04217                   if ( replac == ModOptdollars[nummodopt].number ) break;
04218               }
04219               if ( nummodopt < NumModOptdollars ) {
04220                   dtype = ModOptdollars[nummodopt].type;
04221                   if ( dtype == MODMAX || dtype == MODMIN ) {
04222                       if ( StartBuf[0] && StartBuf[StartBuf[0]] ) {
04223                           MLOCK(ErrorMessageLock);
04224                           MesPrint("A dollar variable with modoption max or min can have only one
    term");
04225                           MUNLOCK(ErrorMessageLock);
04226                           goto GenCall;
04227                       }
```

```
04228                     }
04229                     if ( dtype > 0 && ! DollarLocalCopy(dtype) ) {
04230                         LOCK(d->pthreadslock);
04231                     }
04232                 }
04233             }
04234 #endif
04235         }
04236         else {
04237             StartBuf = cbuf[extractbuff].Buffer;
04238             posisub = cbuf[extractbuff].rhs[replac] - StartBuf;
04239             i = (WORD)cbuf[extractbuff].CanCommu[replac];
04240         }
04241         if ( power == 1 ) {     /* Just a single power */
04242             termout = AT.WorkPointer;
04243             AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
04244             if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
04245             while ( StartBuf[posisub] ) {
04246                 if ( extractbuff == AT.allbufnum ) WildDollars(BHEAD &(StartBuf[posisub]));
04247                 AT.WorkPointer = (WORD *)(((UBYTE *)(termout)) + AM.MaxTer);
04248                 if ( InsertTerm(BHEAD term,replac,extractbuff,
04249                     &(StartBuf[posisub]),termout,tepos) < 0 ) goto GenCall;
04250                 AT.WorkPointer = termout + *termout;
04251                 *AN.RepPoint = 1;
04252                 AR.expchanged = 1;
04253                 posisub += StartBuf[posisub];
04254 /*
04255                     For multiple table substitutions it may be better to
04256                     do modulus arithmetic right here
04257                     Turns out to be not very effective.
04258
04259                 if ( AN.ncmod != 0 ) {
04260                     if ( Modulus(termout) ) goto GenCall;
04261                     if ( !*termout ) goto Return0;
04262                 }
04263 */
04264 #ifdef WITHPTHREADS
04265                 if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
04266                 if ( ( AS.Balancing && CC->numrhs == 0 ) && StartBuf[posisub] ) {
04267 /* UNFINISHED_FEATURE_EXCL_START */
04268                     if ( ( id = ConditionalGetAvailableThread() ) >= 0 ) {
04269                         if ( BalanceRunThread(BHEAD id,termout,level) < 0 ) goto GenCall;
04270                     }
04271 /* UNFINISHED_FEATURE_EXCL_STOP */
04272                 }
04273                 else
04274 #endif
04275                 if ( Generator(BHEAD termout,level) < 0 ) goto GenCall;
04276 #ifdef WITHPTHREADS
04277                 if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { dtype = 0; break; }
04278 #endif
04279                 if ( iscopy == 0 && ( extractbuff != AM.dbufnum ) ) {
04280 /*
04281                     There are cases in which a bigger buffer is created
04282                     on the fly, like with wildcard buffers.
04283                     We play it safe here. Maybe we can be more selective
04284                     in some distant future?
04285 */
04286                     StartBuf = cbuf[extractbuff].Buffer;
04287                 }
04288             }
04289             if ( extractbuff == AT.allbufnum ) {
04290                 CBUF *Ce = cbuf + extractbuff;
04291                 Ce->Pointer = Ce->rhs[Ce->numrhs--];
04292             }
04293 #ifdef WITHPTHREADS
04294             if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); dtype = 0; }
04295 #endif
04296             if ( iscopy ) {
04297                 if ( d->nfactors > 1 ) {
04298                     int j;
04299                     for ( j = 0; j < d->nfactors; j++ ) {
04300                         if ( d->factors[j].where ) M_free(d->factors[j].where,"Copy of factor");
04301                     }
04302                     M_free(d->factors,"Dollar factors");
04303                 }
04304                 M_free(d,"Copy of dollar variable");
04305                 d = 0; iscopy = 0;
04306             }
04307             AT.WorkPointer = termout;
04308         }
04309         else if ( i <= 1 ) {        /* Use binomials */
04310             LONG posit, olw;
04311             WORD *same, *ow = AT.WorkPointer;
04312             LONG olpw = AT.posWorkPointer;
04313             power1 = power+1;
04314             WantAddLongs(power1);
```

```
04315             olw = posit = AT.lWorkPointer; AT.lWorkPointer += power1;
04316             same = ++AT.WorkPointer;
04317             a = accum = ( AT.WorkPointer += power1+1 );
04318             AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + 2*AM.MaxTer);
04319             if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
04320             AT.lWorkSpace[posit] = posisub;
04321             same[-1] = 0;
04322             *same = 1;
04323             *accum = 0;
04324             tepos = AR.TePos;
04325             i = 1;
04326             do {
04327                 if ( StartBuf[AT.lWorkSpace[posit]] ) {
04328                     if ( ( a = PasteTerm(BHEAD i-1,accum,
04329                         &(StartBuf[AT.lWorkSpace[posit]]),i,*same) ) == 0 )
04330                         goto GenCall;
04331                     AT.lWorkSpace[posit+1] = AT.lWorkSpace[posit];
04332                     same[1] = *same + 1;
04333                     if ( i > 1 && AT.lWorkSpace[posit] < AT.lWorkSpace[posit-1] ) *same = 1;
04334                     AT.lWorkSpace[posit] += StartBuf[AT.lWorkSpace[posit]];
04335                     i++;
04336                     posit++;
04337                     same++;
04338                 }
04339                 else {
04340                     i--; posit--; same--;
04341                 }
04342                 if ( i > power ) {
04343                     termout = AT.WorkPointer = a;
04344                     AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + 2*AM.MaxTer);
04345                     if ( AT.WorkPointer > AT.WorkTop )
04346                         goto OverWork;
04347                     if ( FiniTerm(BHEAD term,accum,termout,replac,tepos) ) goto GenCall;
04348                     AT.WorkPointer = termout + *termout;
04349                     *AN.RepPoint = 1;
04350                     AR.expchanged = 1;
04351 #ifdef WITHPTHREADS
04352                     if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
04353 /* UNFINISHED_FEATURE_EXCL_START */
04354                     if ( ( AS.Balancing && CC->numrhs == 0 ) && ( i > 0 )
04355                     && ( id = ConditionalGetAvailableThread() ) >= 0 ) {
04356                         if ( BalanceRunThread(BHEAD id,termout,level) < 0 ) goto GenCall;
04357                     }
04358 /* UNFINISHED_FEATURE_EXCL_STOP */
04359                     else
04360 #endif
04361                     if ( Generator(BHEAD termout,level) ) goto GenCall;
04362 #ifdef WITHPTHREADS
04363                     if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { dtype = 0; break; }
04364 #endif
04365                     if ( iscopy == 0 && ( extractbuff != AM.dbufnum ) )
04366                             StartBuf = cbuf[extractbuff].Buffer;
04367                     i--; posit--; same--;
04368                 }
04369             } while ( i > 0 );
04370 #ifdef WITHPTHREADS
04371             if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); dtype = 0; }
04372 #endif
04373             if ( iscopy ) {
04374                 if ( d->nfactors > 1 ) {
04375                     int j;
04376                     for ( j = 0; j < d->nfactors; j++ ) {
04377                         if ( d->factors[j].where ) M_free(d->factors[j].where,"Copy of factor");
04378                     }
04379                     M_free(d->factors,"Dollar factors");
04380                 }
04381                 M_free(d,"Copy of dollar variable");
04382                 d = 0; iscopy = 0;
04383             }
04384             AT.WorkPointer = ow; AT.lWorkPointer = olw; AT.posWorkPointer = olpw;
04385         }
04386         else {                          /* No binomials */
04387             LONG posit, olw, olpw = AT.posWorkPointer;
04388             WantAddLongs(power);
04389             posit = olw = AT.lWorkPointer; AT.lWorkPointer += power;
04390             a = accum = AT.WorkPointer;
04391             AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + 2*AM.MaxTer);
04392             if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
04393             for ( i = 0; i < power; i++ ) AT.lWorkSpace[posit++] = posisub;
04394             posit = olw;
04395             *accum = 0;
04396             tepos = AR.TePos;
04397             i = 0;
04398             while ( i >= 0 ) {
04399                 if ( StartBuf[AT.lWorkSpace[posit]] ) {
04400                     if ( ( a = PasteTerm(BHEAD i,accum,
04401                         &(StartBuf[AT.lWorkSpace[posit]]),1,1) ) == 0 ) goto GenCall;
```

```
04402                   AT.lWorkSpace[posit] += StartBuf[AT.lWorkSpace[posit]];
04403                   i++; posit++;
04404               }
04405               else {
04406                   AT.lWorkSpace[posit--] = posisub;
04407                   i--;
04408               }
04409               if ( i >= power ) {
04410                   termout = AT.WorkPointer = a;
04411                   AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + 2*AM.MaxTer);
04412                   if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
04413                   if ( FiniTerm(BHEAD term,accum,termout,replac,tepos) ) goto GenCall;
04414                   AT.WorkPointer = termout + *termout;
04415                   *AN.RepPoint = 1;
04416                   AR.expchanged = 1;
04417 #ifdef WITHPTHREADS
04418                   if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
04419                   if ( ( AS.Balancing && CC->numrhs == 0 ) && ( i > 0 ) && ( id =
      ConditionalGetAvailableThread() ) >= 0 ) {
04420 /* UNFINISHED_FEATURE_EXCL_START */
04421                       if ( BalanceRunThread(BHEAD id,termout,level) < 0 ) goto GenCall;
04422 /* UNFINISHED_FEATURE_EXCL_STOP */
04423                   }
04424                   else
04425 #endif
04426                   if ( Generator(BHEAD termout,level) ) goto GenCall;
04427 #ifdef WITHPTHREADS
04428                   if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { dtype = 0; break; }
04429 #endif
04430                   if ( iscopy == 0 && ( extractbuff != AM.dbufnum ) )
04431                           StartBuf = cbuf[extractbuff].Buffer;
04432                   i--; posit--;
04433               }
04434           }
04435 #ifdef WITHPTHREADS
04436           if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); dtype = 0; }
04437 #endif
04438           if ( iscopy ) {
04439               if ( d->nfactors > 1 ) {
04440                   int j;
04441                   for ( j = 0; j < d->nfactors; j++ ) {
04442                       if ( d->factors[j].where ) M_free(d->factors[j].where,"Copy of factor");
04443                   }
04444                   M_free(d->factors,"Dollar factors");
04445               }
04446               M_free(d,"Copy of dollar variable");
04447               d = 0; iscopy = 0;
04448           }
04449           AT.WorkPointer = accum;
04450           AT.lWorkPointer = olw;
04451           AT.posWorkPointer = olpw;
04452       }
04453   }
04454   else {                              /* Expression from disk */
04455       POSITION StartPos;
04456       LONG position, olpw, opw, comprev, extra;
04457       RENUMBER renumber;
04458       WORD *Freeze, *aa, *dummies;
04459       replac = -replac-1;
04460       power = AN.TeSuOut;
04461       Freeze = AN.Frozen;
04462       if ( Expressions[replac].status == STOREDEXPRESSION ) {
04463           POSITION firstpos;
04464           SETSTARTPOS(firstpos);
04465
04466 /*        Note that AT.TMaddr is needed for GetTable just once! */
04467 /*
04468           We need space for the previous term in the compression
04469           This is made available in AR.CompressBuffer, although we may get
04470           problems with this sooner or later. Hence we need to keep
04471           a set of pointers in AR.CompressBuffer
04472           Note that after the last call there has been no use made
04473           of AR.CompressPointer, so it points automatically at its original
04474           position!
04475 */
04476           WantAddPointers(power+1);
04477           comprev = opw = AT.pWorkPointer;
04478           AT.pWorkPointer += power+1;
04479           WantAddPositions(power+1);
04480           position = olpw = AT.posWorkPointer;
04481           AT.posWorkPointer += power + 1;
04482
04483           AT.pWorkSpace[comprev++] = AR.CompressPointer;
04484
04485           for ( i = 0; i < power; i++ ) {
04486               PUTZERO(AT.posWorkSpace[position]); position++;
04487           }
```

```
04488             position = olpw;
04489             if ( ( renumber = GetTable(replac,&(AT.posWorkSpace[position]),1) ) == 0 ) goto GenCall;
04490             dummies = AT.WorkPointer;
04491             *dummies++ = AR.CurDum;
04492             AT.WorkPointer += power+2;
04493             accum = AT.WorkPointer;
04494             AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + 2*AM.MaxTer);
04495             if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
04496             aa = AT.WorkPointer;
04497             *accum = 0;
04498             i = 0; StartPos = AT.posWorkSpace[position];
04499             dummies[i] = AR.CurDum;
04500             while ( i >= 0 ) {
04501 skippedfirst:
04502                 AR.CompressPointer = AT.pWorkSpace[comprev-1];
04503                 if ( ( extra = PasteFile(BHEAD i,accum,&(AT.posWorkSpace[position])
04504                         ,&a,renumber,Freeze,replac) ) < 0 ) goto GenCall;
04505                 if ( Expressions[replac].numdummies > 0 ) {
04506                     AR.CurDum = dummies[i] + Expressions[replac].numdummies;
04507                 }
04508                 if ( NOTSTARTPOS(firstpos) ) {
04509                     if ( ISMINPOS(firstpos) || ISEQUALPOS(firstpos,AT.posWorkSpace[position]) ) {
04510                         firstpos = AT.posWorkSpace[position];
04511 /*
04512                         ADDPOS(AT.posWorkSpace[position],extra * sizeof(WORD));
04513 */
04514                         goto skippedfirst;
04515                     }
04516                 }
04517                 if ( extra ) {
04518 /*
04519                     ADDPOS(AT.posWorkSpace[position],extra * sizeof(WORD));
04520 */
04521                     i++; AT.posWorkSpace[++position] = StartPos;
04522                     AT.pWorkSpace[comprev++] = AR.CompressPointer;
04523                     dummies[i] = AR.CurDum;
04524                 }
04525                 else {
04526                     PUTZERO(AT.posWorkSpace[position]); position--; i--;
04527                     AR.CurDum = dummies[i];
04528                     comprev--;
04529                 }
04530                 if ( i >= power ) {
04531                     termout = AT.WorkPointer = a;
04532                     AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + 2*AM.MaxTer);
04533                     if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
04534                     if ( FiniTerm(BHEAD term,accum,termout,replac,0) ) goto GenCall;
04535                     if ( *termout ) {
04536                         AT.WorkPointer = termout + *termout;
04537                         *AN.RepPoint = 1;
04538                         AR.expchanged = 1;
04539 #ifdef WITHPTHREADS
04540                         if ( ( AS.Balancing && CC->numrhs == 0 ) && ( i > 0 ) && ( id =
    ConditionalGetAvailableThread() ) >= 0 ) {
04541 /* UNFINISHED_FEATURE_EXCL_START */
04542                             if ( BalanceRunThread(BHEAD id,termout,level) < 0 ) goto GenCall;
04543 /* UNFINISHED_FEATURE_EXCL_STOP */
04544                         }
04545                         else
04546 #endif
04547                         if ( Generator(BHEAD termout,level) ) goto GenCall;
04548                     }
04549                     i--; position--;
04550                     AR.CurDum = dummies[i];
04551                     comprev--;
04552                 }
04553                 AT.WorkPointer = aa;
04554             }
04555             AT.WorkPointer = accum;
04556             AT.posWorkPointer = olpw;
04557             AT.pWorkPointer = opw;
04558 /*
04559         Bug fix. See also GetTable
04560 #ifdef WITHPTHREADS
04561             M_free(renumber->symb.lo,"VarSpace");
04562             M_free(renumber,"Renumber");
04563 #endif
04564 */
04565             if ( renumber->symb.lo != AN.dummyrenumlist )
04566                 M_free(renumber->symb.lo,"VarSpace");
04567             M_free(renumber,"Renumber");
04568 
04569         }
04570         else {          /* Active expression */
04571             aa = accum = AT.WorkPointer;
04572             if ( ( (WORD *)(((UBYTE *)(AT.WorkPointer)) + 2 * AM.MaxTer + sizeof(WORD)) ) > AT.WorkTop
    )
```

```
04573                   goto OverWork;
04574             *accum++ = -1; AT.WorkPointer++;
04575             if ( DoOnePow(BHEAD term,power,replac,accum,aa,level,Freeze) ) goto GenCall;
04576             AT.WorkPointer = aa;
04577         }
04578     }
04579 Return0:
04580     AR.CurDum = DumNow;
04581     AN.RepPoint = RepSto;
04582     CC->numrhs = oldtoprhs;
04583     CC->Pointer = CC->Buffer + oldcpointer;
04584     CCC->numrhs = oldatoprhs;
04585     CCC->Pointer = CCC->Buffer + oldacpointer;
04586     return(0);
04587
04588 GenCall:
04589     if ( AM.tracebackflag ) {
04590         termout = term;
04591         MLOCK(ErrorMessageLock);
04592         AO.OutFill = AO.OutputLine = (UBYTE *)AT.WorkPointer;
04593         AO.OutSkip = 3;
04594         FiniLine();
04595         i = *termout;
04596         while ( --i >= 0 ) {
04597             TalToLine((UWORD)(*termout++));
04598             TokenToLine((UBYTE *)"  ");
04599         }
04600         AO.OutSkip = 0;
04601         FiniLine();
04602         MesCall("Generator");
04603         MUNLOCK(ErrorMessageLock);
04604     }
04605     CC->numrhs = oldtoprhs;
04606     CC->Pointer = CC->Buffer + oldcpointer;
04607     CCC->numrhs = oldatoprhs;
04608     CCC->Pointer = CCC->Buffer + oldacpointer;
04609     return(-1);
04610 OverWork:
04611     CC->numrhs = oldtoprhs;
04612     CC->Pointer = CC->Buffer + oldcpointer;
04613     CCC->numrhs = oldatoprhs;
04614     CCC->Pointer = CCC->Buffer + oldacpointer;
04615     MLOCK(ErrorMessageLock);
04616     MesWork();
04617     MUNLOCK(ErrorMessageLock);
04618     return(-1);
04619 }
04620
04621 /*
04622         #] Generator :
04623         #[ DoOnePow :           WORD DoOnePow(term,power,nexp,accum,aa,level,freeze)
04624 */
04649 #ifdef WITHPTHREADS
04650 char freezestring[] = "freeze<-xxxx";
04651 #endif
04652
04653 int DoOnePow(PHEAD WORD *term, WORD power, WORD nexp, WORD * accum,
04654              WORD *aa, WORD level, WORD *freeze)
04655 {
04656     GETBIDENTITY
04657     POSITION oldposition, startposition;
04658     WORD *acc, *termout, fromfreeze = 0;
04659     WORD *oldipointer = AR.CompressPointer;
04660     FILEHANDLE *fi;
04661     WORD type, retval;
04662     WORD oldGetOneFile = AR.GetOneFile;
04663     WORD olddummies = AR.CurDum;
04664     WORD extradummies = Expressions[nexp].numdummies;
04665 /*
04666     The next code is for some tricky debugging. (5-jan-2010 JV)
04667     Normally it should be disabled.
04668 */
04669 /*
04670 #ifdef WITHPTHREADS
04671     if ( freeze ) {
04672         MLOCK(ErrorMessageLock);
04673         if ( AT.identity < 10 ) {
04674             freezestring[8] = '0'+AT.identity;
04675             freezestring[9] = '>';
04676             freezestring[10] = 0;
04677         }
04678         else if ( AT.identity < 100 ) {
04679             freezestring[8] = '0'+AT.identity/10;
04680             freezestring[9] = '0'+AT.identity%10;
04681             freezestring[10] = '>';
04682             freezestring[11] = 0;
04683         }
```

```
04684         else {
04685             freezestring[8] = 0;
04686         }
04687         PrintTerm(freeze,freezestring);
04688         MUNLOCK(ErrorMessageLock);
04689     }
04690 #else
04691     if ( freeze ) PrintTerm(freeze,"freeze");
04692 #endif
04693 */
04694     type = Expressions[nexp].status;
04695     if ( type == HIDDENLEXPRESSION || type == HIDDENGEXPRESSION
04696      || type == DROPHLEXPRESSION || type == DROPHGEXPRESSION
04697      || type == UNHIDELEXPRESSION || type == UNHIDEGEXPRESSION ) {
04698         AR.GetOneFile = 2; fi = AR.hidefile;
04699     }
04700     else {
04701         AR.GetOneFile = 0; fi = AR.infile;
04702     }
04703     if ( fi->handle >= 0 ) {
04704         PUTZERO(oldposition);
04705 #ifdef WITHSEEK
04706         LOCK(AS.inputslock);
04707         SeekFile(fi->handle,&oldposition,SEEK_CUR);
04708         UNLOCK(AS.inputslock);
04709 #endif
04710     }
04711     else {
04712         SETBASEPOSITION(oldposition,fi->POfill-fi->PObuffer);
04713     }
04714     if ( freeze && ( Expressions[nexp].bracketinfo != 0 ) ) {
04715         POSITION *brapos;
04716 /*
04717         There is a bracket index
04718         AR.CompressPointer = oldipointer;
04719 */
04720         (*aa)++;
04721         power--;
04722         if ( ( brapos = FindBracket(nexp,freeze) ) == 0 )
04723             goto EndExpr;
04724         startposition = *brapos;
04725         goto doterms;
04726     }
04727     startposition = AS.OldOnFile[nexp];
04728     retval = GetOneTerm(BHEAD accum,fi,&startposition,0);
04729     if ( retval > 0 ) {          /* Skip prototype */
04730         (*aa)++;
04731         power--;
04732 doterms:
04733         AR.CompressPointer = oldipointer;
04734         for (;;) {
04735             retval = GetOneTerm(BHEAD accum,fi,&startposition,0);
04736             if ( retval <= 0 ) break;
04737 /*
04738             Here should come the code to test for [].
04739 */
04740             if ( freeze ) {
04741                 WORD *t, *m, *r, *mstop;
04742                 WORD *tset;
04743                 t = accum;
04744                 m = freeze;
04745                 m += *m;
04746                 m -= ABS(m[-1]);
04747                 mstop = m;
04748                 m = freeze + 1;
04749                 r = t;
04750                 r += *t;
04751                 r -= ABS(r[-1]);
04752                 t++;
04753                 tset = t;
04754                 while ( t < r && *t != HAAKJE ) t += t[1];
04755                 if ( t >= r ) {
04756                     if ( m < mstop ) {
04757                         if ( fromfreeze ) goto EndExpr;
04758                         goto NextTerm;
04759                     }
04760                     t = tset;
04761                 }
04762                 else {
04763                     r = tset;
04764                     while ( r < t && m < mstop ) {
04765                         if ( *r == *m ) { m++; r++; }
04766                         else {
04767                             if ( fromfreeze ) goto EndExpr;
04768                             goto NextTerm;
04769                         }
04770                     }
```

```
04771                     if ( r < t || m < mstop ) {
04772                         if ( fromfreeze ) goto EndExpr;
04773                         goto NextTerm;
04774                     }
04775                 }
04776                 fromfreeze = 1;
04777                 r = tset;
04778                 m = accum;
04779                 m += *m;
04780                 while ( t < m ) *r++ = *t++;
04781                 *accum = WORDDIF(r,accum);
04782             }
04783             if ( extradummies > 0 ) {
04784                 if ( olddummies > AM.IndDum ) {
04785                     MoveDummies(BHEAD accum,olddummies-AM.IndDum);
04786                 }
04787                 AR.CurDum = olddummies+extradummies;
04788             }
04789             acc = accum;
04790             acc += *acc;
04791             if ( power <= 0 ) {
04792                 termout = acc;
04793                 AT.WorkPointer = (WORD *)(((UBYTE *)(acc)) + 2*AM.MaxTer);
04794                 if ( AT.WorkPointer > AT.WorkTop ) {
04795                     MLOCK(ErrorMessageLock);
04796                     MesWork();
04797                     MUNLOCK(ErrorMessageLock);
04798                     return(-1);
04799                 }
04800                 if ( FiniTerm(BHEAD term,aa,termout,nexp,0) ) goto PowCall;
04801                 if ( *termout ) {
04802                     MarkPolyRatFunDirty(termout)
04803 /*                  PolyFunDirty(BHEAD termout); */
04804                     AT.WorkPointer = termout + *termout;
04805                     *AN.RepPoint = 1;
04806                     AR.expchanged = 1;
04807                     if ( Generator(BHEAD termout,level) ) goto PowCall;
04808                 }
04809             }
04810             else {
04811                 if ( acc > AT.WorkTop ) {
04812                     MLOCK(ErrorMessageLock);
04813                     MesWork();
04814                     MUNLOCK(ErrorMessageLock);
04815                     return(-1);
04816                 }
04817                 if ( DoOnePow(BHEAD term,power,nexp,acc,aa,level,freeze) ) goto PowCall;
04818             }
04819 NextTerm:;
04820             AR.CompressPointer = oldipointer;
04821         }
04822 EndExpr:
04823         (*aa)--;
04824     }
04825     AR.CompressPointer = oldipointer;
04826     if ( fi->handle >= 0 ) {
04827 #ifdef WITHSEEK
04828         LOCK(AS.inputslock);
04829         SeekFile(fi->handle,&oldposition,SEEK_SET);
04830         UNLOCK(AS.inputslock);
04831         if ( ISNEGPOS(oldposition) ) {
04832 /* INTERNAL_ERROR_EXCL_START */
04833             MLOCK(ErrorMessageLock);
04834             MesPrint("!>File error");
04835             goto PowCall2;
04836 /* INTERNAL_ERROR_EXCL_STOP */
04837         }
04838 #endif
04839     }
04840     else {
04841         fi->POfill = fi->PObuffer + BASEPOSITION(oldposition);
04842     }
04843     AR.GetOneFile = oldGetOneFile;
04844     AR.CurDum = olddummies;
04845     return(0);
04846 PowCall:;
04847     MLOCK(ErrorMessageLock);
04848 #ifdef WITHSEEK
04849 PowCall2:;
04850 #endif
04851     MesCall("DoOnePow");
04852     MUNLOCK(ErrorMessageLock);
04853     SETERROR(-1)
04854 }
04855
04856 /*
04857         #] DoOnePow :
```

```
04858         #[ Deferred :           WORD Deferred(term,level)
04859 */
04876 int Deferred(PHEAD WORD *term, WORD level)
04877 {
04878     GETBIDENTITY
04879     POSITION startposition;
04880     WORD *t, *m, *mstop, *tstart, decr, oldb, *termout, i, *oldwork, retval;
04881     WORD *oldipointer = AR.CompressPointer, *oldPOfill = AR.infile->POfill;
04882     WORD oldGetOneFile = AR.GetOneFile;
04883     AR.GetOneFile = 1;
04884     oldwork = AT.WorkPointer;
04885     AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
04886     termout = AT.WorkPointer;
04887     AR.DeferFlag = 0;
04888     startposition = AR.DefPosition;
04889 /*
04890         Store old position
04891 */
04892     if ( AR.infile->handle >= 0 ) {
04893 /*
04894         PUTZERO(oldposition);
04895         SeekFile(AR.infile->handle,&oldposition,SEEK_CUR);
04896 */
04897     }
04898     else {
04899 /*
04900         SETBASEPOSITION(oldposition,AR.infile->POfill-AR.infile->PObuffer);
04901 */
04902         AR.infile->POfill = (WORD *)((UBYTE *)(AR.infile->PObuffer)
04903                     +BASEPOSITION(startposition));
04904     }
04905 /*
04906         Look in the CompressBuffer where the bracket contents start
04907 */
04908     t = m = AR.CompressBuffer;
04909     t += *t;
04910     mstop = t - ABS(t[-1]);
04911     m++;
04912     while ( *m != HAAKJE && m < mstop ) m += m[1];
04913     if ( m >= mstop ) { /* No deferred action! */
04914         AT.WorkPointer = term + *term;
04915         if ( Generator(BHEAD term,level) ) goto DefCall;
04916         AR.DeferFlag = 1;
04917         AT.WorkPointer = oldwork;
04918         AR.GetOneFile = oldGetOneFile;
04919         return(0);
04920     }
04921     mstop = m + m[1];
04922     decr = WORDDIF(mstop,AR.CompressBuffer)-1;
04923     tstart = AR.CompressPointer + decr;
04924
04925     m = AR.CompressBuffer;
04926     t = AR.CompressPointer;
04927     i = *m;
04928     NCOPY(t,m,i);
04929     oldb = *tstart;
04930     AR.TePos = 0;
04931     AN.TeSuOut = 0;
04932 /*
04933         Status:
04934         First bracket content starts at mstop.
04935         Next term starts at startposition.
04936         Decompression information is in AR.CompressPointer.
04937         The outside of the bracket runs from AR.CompressBuffer+1 to mstop.
04938 */
04939     for(;;) {
04940         *tstart = *(AR.CompressPointer)-decr;
04941         AR.CompressPointer = AR.CompressPointer+AR.CompressPointer[0];
04942         if ( InsertTerm(BHEAD term,0,AM.rbufnum,tstart,termout,0) < 0 ) {
04943             goto DefCall;
04944         }
04945         *tstart = oldb;
04946         AT.WorkPointer = termout + *termout;
04947         if ( Generator(BHEAD termout,level) ) goto DefCall;
04948         AR.CompressPointer = oldipointer;
04949         AT.WorkPointer = termout;
04950         retval = GetOneTerm(BHEAD AT.WorkPointer,AR.infile,&startposition,0);
04951         if ( retval >= 0 ) AR.CompressPointer = oldipointer;
04952         if ( retval <= 0 ) break;
04953         t = AR.CompressPointer;
04954         if ( *t < (1 + decr + ABS(*(t+*t-1))) ) break;
04955         t++;
04956         m = AR.CompressBuffer+1;
04957         while ( m < mstop ) {
04958             if ( *m != *t ) goto Thatsit;
04959             m++; t++;
04960         }
```

```
04961     }
04962 Thatsit:;
04963 /*
04964         Finished. Reposition the file, restore information and return.
04965 */
04966     if ( AR.infile->handle < 0 ) AR.infile->POfill = oldPOfill;
04967     AR.DeferFlag = 1;
04968     AR.GetOneFile = oldGetOneFile;
04969     AT.WorkPointer = oldwork;
04970     return(0);
04971 DefCall:;
04972     MLOCK(ErrorMessageLock);
04973     MesCall("Deferred");
04974     MUNLOCK(ErrorMessageLock);
04975     SETERROR(-1)
04976 }
04977
04978 /*
04979         #] Deferred :
04980         #[ PrepPoly :           WORD PrepPoly(term,par)
04981 */
05004 int PrepPoly(PHEAD WORD *term,WORD par)
05005 {
05006     GETBIDENTITY
05007     WORD count = 0, i, jcoef, ncoef;
05008     WORD *t, *m, *r, *tstop, *poly = 0, *v, *w, *vv, *ww;
05009     WORD *oldworkpointer = AT.WorkPointer;
05010 /*
05011     The problem here is that the function will be forced into 'long'
05012     notation. After this -SNUMBER,1 becomes 6,0,4,1,1,3 and the
05013     pattern matcher cannot match a short 1 with a long 1.
05014     But because this is an undocumented feature for very special
05015     purposes, we don't do anything about it. (30-aug-2011)
05016 */
05017     if ( AR.PolyFunType == 2 && AR.PolyFunExp != 2 ) {
05018         WORD oldtype = AR.SortType;
05019         AR.SortType = SORTHIGHFIRST;
05020         if ( poly_ratfun_normalize(BHEAD term) != 0 ) Terminate(-1);
05021 /*      if ( ReadPolyRatFun(BHEAD term) != 0 ) Terminate(-1); */
05022         oldworkpointer = AT.WorkPointer;
05023         AR.SortType = oldtype;
05024     }
05025     AT.PolyAct = 0;
05026     t = term;
05027     GETSTOP(t,tstop);
05028     t++;
05029     while ( t < tstop ) {
05030         if ( *t == AR.PolyFun ) {
05031             if ( count > 0 ) return(0);
05032             poly = t;
05033             count++;
05034         }
05035         t += t[1];
05036     }
05037     r = m = term + *term;
05038     i = ABS(m[-1]);
05039     if ( par > 0 ) {
05040         if ( count == 0 ) return(0);
05041         else if ( AR.PolyFunType == 1 || (AR.PolyFunType == 2 && AR.PolyFunExp == 2) )
05042             goto DoOne;
05043         else if ( AR.PolyFunType == 2 )
05044             goto DoTwo;
05045         else
05046             goto DoError;
05047     }
05048     else if ( count == 0 ) {
05049 /*
05050         #[ Create a PolyFun :
05051 */
05052         poly = t = tstop;
05053         if ( i == 3 && m[-2] == 1 && (m[-3]&MAXPOSITIVE) == m[-3] ) {
05054             *m++ = AR.PolyFun;
05055             if ( AR.PolyFunType == 1 || (AR.PolyFunType == 2 && AR.PolyFunExp == 2) ) {
05056                 *m++ = FUNHEAD+2;
05057                 FILLFUN(m)
05058                 *m++ = -SNUMBER;
05059                 *m = m[-2-FUNHEAD] < 0 ? -m[-4-FUNHEAD]: m[-4-FUNHEAD];
05060                 m++;
05061             }
05062             else if ( AR.PolyFunType == 2 ) {
05063                 *m++ = FUNHEAD+4;
05064                 FILLFUN(m)
05065                 *m++ = -SNUMBER;
05066                 *m = m[-2-FUNHEAD] < 0 ? -m[-4-FUNHEAD]: m[-4-FUNHEAD];
05067                 m++;
05068                 *m++ = -SNUMBER;
05069                 *m++ = 1;
```

```
05070             }
05071         }
05072         else {
05073             WORD *vm;
05074             r = tstop;
05075             if ( AR.PolyFunType == 1 || (AR.PolyFunType == 2 && AR.PolyFunExp == 2) ) {
05076                 *m++ = AR.PolyFun;
05077                 *m++ = FUNHEAD+ARGHEAD+i+1;
05078                 FILLFUN(m)
05079                 *m++ = ARGHEAD+i+1;
05080                 *m++ = 0;
05081                 FILLARG(m)
05082                 *m++ = i+1;
05083                 NCOPY(m,r,i);
05084             }
05085             else if ( AR.PolyFunType == 2 ) {
05086                 WORD *num, *den, size, sign, sizenum, sizeden;
05087                 if ( m[-1] < 0 ) { sign = -1; size = -m[-1]; }
05088                 else             { sign =  1; size =  m[-1]; }
05089                 num = m - size; size = (size-1)/2; den = num + size;
05090                 sizenum = size; while ( num[sizenum-1] == 0 ) sizenum--;
05091                 sizeden = size; while ( den[sizeden-1] == 0 ) sizeden--;
05092                 v = m;
05093                 AT.PolyAct = WORDDIF(v,term);
05094                 *v++ = AR.PolyFun;
05095                 v++;
05096                 FILLFUN(v);
05097                 vm = v;
05098                 *v++ = ARGHEAD+2*sizenum+2;
05099                 *v++ = 0;
05100                 FILLARG(v);
05101                 *v++ = 2*sizenum+2;
05102                 for ( i = 0; i < sizenum; i++ ) *v++ = num[i];
05103                 *v++ = 1;
05104                 for ( i = 1; i < sizenum; i++ ) *v++ = 0;
05105                 *v++ = sign*(2*sizenum+1);
05106                 if ( ToFast(vm,vm) ) v = vm+2;
05107                 vm = v;
05108                 *v++ = ARGHEAD+2*sizeden+2;
05109                 *v++ = 0;
05110                 FILLARG(v);
05111                 *v++ = 2*sizeden+2;
05112                 for ( i = 0; i < sizeden; i++ ) *v++ = den[i];
05113                 *v++ = 1;
05114                 for ( i = 1; i < sizeden; i++ ) *v++ = 0;
05115                 *v++ = 2*sizeden+1;
05116                 if ( ToFast(vm,vm) ) v = vm+2;
05117                 i = v-m;
05118                 m[1] = i;
05119                 w = num;
05120                 NCOPY(w,m,i);
05121                 *w++ = 1; *w++ = 1; *w++ = 3; *term = w - term;
05122                 return(0);
05123             }
05124         }
05125 /*
05126         #] Create a PolyFun :
05127 */
05128     }
05129     else if ( AR.PolyFunType == 1 || (AR.PolyFunType == 2 && AR.PolyFunExp == 2) ) {
05130         DoOne:;
05131 /*
05132         #[ One argument :
05133 */
05134         m = term + *term;
05135         r = poly + poly[1];
05136         if ( ( poly[1] == FUNHEAD+2 && poly[FUNHEAD+1] == 0
05137         && poly[FUNHEAD] == -SNUMBER ) || poly[1] == FUNHEAD ) return(1);
05138         t = poly + FUNHEAD;
05139         if ( t >= r ) return(0);
05140         if ( m[-1] == 3 && *tstop == 1 && tstop[1] == 1 ) {
05141             i = poly[1];
05142             t = poly;
05143             NCOPY(m,t,i);
05144         }
05145         else if ( *t <= -FUNCTION ) {
05146             if ( t+1 < r ) return(0);   /* More than one argument */
05147             r = tstop;
05148             *m++ = AR.PolyFun;
05149             *m++ = FUNHEAD*2+ARGHEAD+i+1;
05150             FILLFUN(m)
05151             *m++ = FUNHEAD+ARGHEAD+i+1;
05152             *m++ = 0;
05153             FILLARG(m)
05154             *m++ = FUNHEAD+i+1;
05155             *m++ = -*t++;
05156             *m++ = FUNHEAD;
```

```
05157             FILLFUN(m)
05158             NCOPY(m,r,i);
05159         }
05160         else if ( *t < 0 ) {
05161             if ( t+2 < r ) return(0);   /* More than one argument */
05162             r = tstop;
05163             if ( *t == -SNUMBER ) {
05164                 if ( t[1] == 0 ) return(1); /* Term should be zero now */
05165                 *m = AR.PolyFun;
05166                 w = m+1;
05167                 m += FUNHEAD+ARGHEAD;
05168                 v = m;
05169                 *m++ = 5+i;
05170                 *m++ = SNUMBER;
05171                 *m++ = 4;
05172                 *m++ = t[1];
05173                 *m++ = 1;
05174                 NCOPY(m,r,i);
05175                 if ( m >= AT.WorkSpace && m < AT.WorkTop )
05176                     AT.WorkPointer = m;
05177                 if ( Normalize(BHEAD v) ) Terminate(-1);
05178                 AT.WorkPointer = oldworkpointer;
05179                 m = w;
05180                 if ( *v == 4 && v[2] == 1 && (v[1]&MAXPOSITIVE) == v[1] ) {
05181                     *m++ = FUNHEAD+2;
05182                     FILLFUN(m)
05183                     *m++ = -SNUMBER;
05184                     *m++ = v[3] < 0 ? -v[1] : v[1];
05185                 }
05186                 else if ( *v == 0 ) return(1);
05187                 else {
05188                     *m++ = FUNHEAD+ARGHEAD+*v;
05189                     FILLFUN(m)
05190                     *m++ = ARGHEAD+*v;
05191                     *m++ = 0;
05192                     FILLARG(m)
05193                     m = v + *v;
05194                 }
05195             }
05196             else if ( *t == -SYMBOL ) {
05197                 *m++ = AR.PolyFun;
05198                 *m++ = FUNHEAD+ARGHEAD+5+i;
05199                 FILLFUN(m)
05200                 *m++ = ARGHEAD+5+i;
05201                 *m++ = 0;
05202                 FILLARG(m)
05203                 *m++ = 5+i;
05204                 *m++ = SYMBOL;
05205                 *m++ = 4;
05206                 *m++ = t[1];
05207                 *m++ = 1;
05208                 NCOPY(m,r,i);
05209             }
05210             else return(0);         /* Not symbol-like */
05211         }
05212         else {
05213             if ( t + *t < r ) return(0); /* More than one argument */
05214             i = m[-1];
05215             *m++ = AR.PolyFun;
05216             w = m;
05217             m += ARGHEAD+FUNHEAD-1;
05218             t += ARGHEAD;
05219             jcoef = i < 0 ? (i+1)>>1:(i-1)>>1;
05220             v = t;
05221 /*
05222             Test now the scalar nature of the argument.
05223             No indices allowed.
05224 */
05225             while ( t < r ) {
05226                 WORD *vstop;
05227                 vv = t + *t;
05228                 vstop = vv - ABS(vv[-1]);
05229                 t++;
05230                 while( t < vstop ) {
05231                     if ( *t == INDEX ) return(0);
05232                     t += t[1];
05233                 }
05234                 t = vv;
05235             }
05236 /*
05237             Now multiply each term by the coefficient.
05238 */
05239             t = v;
05240             while ( t < r ) {
05241                 ww = m;
05242                 v = t + *t;
05243                 ncoef = v[-1];
```

```
05244                 vv = v - ABS(ncoef);
05245                 if ( ncoef < 0 ) ncoef++;
05246                 else ncoef--;
05247                 ncoef >>= 1;
05248                 while ( t < vv ) *m++ = *t++;
05249                 if ( MulRat(BHEAD (UWORD *)vv,ncoef,(UWORD *)tstop,jcoef,
05250                     (UWORD *)m,&ncoef) ) Terminate(-1);
05251                 ncoef *= 2;
05252                 m += ABS(ncoef);
05253                 if ( ncoef < 0 ) ncoef--;
05254                 else ncoef++;
05255                 *m++ = ncoef;
05256                 *ww = WORDDIF(m,ww);
05257                 if ( AN.ncmod != 0 ) {
05258                     if ( Modulus(ww) ) Terminate(-1);
05259                     if ( *ww == 0 ) return(1);
05260                     m = ww + *ww;
05261                 }
05262                 t = v;
05263             }
05264             *w = (WORDDIF(m,w))+1;
05265             w[FUNHEAD-1] = w[0] - FUNHEAD;
05266             w[FUNHEAD] = 0;
05267             w[1] = 0; /* omission survived for years. 23-mar-2006 JV */
05268             w += FUNHEAD-1;
05269             if ( ToFast(w,w) ) {
05270                 if ( *w <= -FUNCTION ) { w[-FUNHEAD+1] = FUNHEAD+1; m = w+1; }
05271                 else { w[-FUNHEAD+1] = FUNHEAD+2; m = w+2; }
05272 
05273             }
05274         }
05275         t = poly + poly[1];
05276         while ( t < tstop ) *poly++ = *t++;
05277 /*
05278         #] One argument :
05279 */
05280     }
05281     else if ( AR.PolyFunType == 2 ) {
05282         DoTwo:;
05283 /*
05284         #[ Two arguments :
05285 */
05286         WORD *num, *den, size, sign, sizenum, sizeden;
05287 /*
05288         First make sure that the PolyFun is last
05289 */
05290         m = term + *term;
05291         if ( poly + poly[1] < tstop ) {
05292             for ( i = 0; i < poly[1]; i++ ) m[i] = poly[i];
05293             t = poly; v = poly + poly[1];
05294             while ( v < tstop ) *t++ = *v++;
05295             poly = t;
05296             for ( i = 0; i < m[1]; i++ ) t[i] = m[i];
05297             t += m[1];
05298         }
05299         AT.PolyAct = WORDDIF(poly,term);
05300 /*
05301         If needed we convert the coefficient into a PolyRatFun and then
05302         we call poly_ratfun_normalize
05303 */
05304         if ( m[-1] == 3 && m[-2] == 1 && m[-3] == 1 ) return(0);
05305         if ( AR.PolyFunExp != 1 ) {
05306         if ( m[-1] < 0 ) { sign = -1; size = -m[-1]; } else { sign = 1; size = m[-1]; }
05307         num = m - size; size = (size-1)/2; den = num + size;
05308         sizenum = size; while ( num[sizenum-1] == 0 ) sizenum--;
05309         sizeden = size; while ( den[sizeden-1] == 0 ) sizeden--;
05310         v = m;
05311         *v++ = AR.PolyFun;
05312         *v++ = FUNHEAD + 2*(ARGHEAD+sizenum+sizeden+2);
05313 /*      *v++ = MUSTCLEANPRF; */
05314         *v++ = 0;
05315         FILLFUN3(v);
05316         *v++ = ARGHEAD+2*sizenum+2;
05317         *v++ = 0;
05318         FILLARG(v);
05319         *v++ = 2*sizenum+2;
05320         for ( i = 0; i < sizenum; i++ ) *v++ = num[i];
05321         *v++ = 1;
05322         for ( i = 1; i < sizenum; i++ ) *v++ = 0;
05323         *v++ = sign*(2*sizenum+1);
05324         *v++ = ARGHEAD+2*sizeden+2;
05325         *v++ = 0;
05326         FILLARG(v);
05327         *v++ = 2*sizeden+2;
05328         for ( i = 0; i < sizeden; i++ ) *v++ = den[i];
05329         *v++ = 1;
05330         for ( i = 1; i < sizeden; i++ ) *v++ = 0;
```

```
05331         *v++ = 2*sizeden+1;
05332         w = num;
05333         i = v - m;
05334         NCOPY(w,m,i);
05335         }
05336         else {
05337             w = m-ABS(m[-1]);
05338         }
05339         *w++ = 1; *w++ = 1; *w++ = 3; *term = w - term;
05340         {
05341             WORD oldtype = AR.SortType;
05342             AR.SortType = SORTHIGHFIRST;
05343 /*
05344             if ( count > 0 )
05345                 poly_ratfun_normalize(BHEAD term);
05346             else
05347                 ReadPolyRatFun(BHEAD term);
05348 */
05349             poly_ratfun_normalize(BHEAD term);
05350 
05351 /*          oldworkpointer = AT.WorkPointer; */
05352             AR.SortType = oldtype;
05353         }
05354         goto endofit;
05355 /*
05356         #] Two arguments :
05357 */
05358     }
05359     else {
05360 /* INTERNAL_ERROR_EXCL_START */
05361         DoError:;
05362         MLOCK(ErrorMessageLock);
05363         MesPrint("!>Illegal value for PolyFunType in PrepPoly");
05364         MUNLOCK(ErrorMessageLock);
05365         Terminate(-1);
05366 /* INTERNAL_ERROR_EXCL_STOP */
05367     }
05368     r = term + *term;
05369     AT.PolyAct = WORDDIF(poly,term);
05370     while ( r < m ) *poly++ = *r++;
05371     *poly++ = 1;
05372     *poly++ = 1;
05373     *poly++ = 3;
05374     *term = WORDDIF(poly,term);
05375 endofit:;
05376     return(0);
05377 }
05378 
05379 /*
05380         #] PrepPoly :
05381         #[ PolyFunMul :         WORD PolyFunMul(term)
05382 */
05394 int PolyFunMul(PHEAD WORD *term)
05395 {
05396     GETBIDENTITY
05397     WORD *t, *fun1, *fun2, *t1, *t2, *m, *w, *ww, *tt1, *tt2, *tt4, *arg1, *arg2;
05398     WORD *tstop, i, dirty = 0, OldPolyFunPow = AR.PolyFunPow, minp1, minp2;
05399     WORD n1, n2, i1, i2, l1, l2, l3, l4, action = 0, noac = 0;
05400     int retval = 0;
05401     if ( AR.PolyFunType == 2 && AR.PolyFunExp == 1 ) {
05402         WORD pow = 0, pow1;
05403         t = term + 1; t1 = term + *term; t1 -= ABS(t1[-1]);
05404         w = t;
05405         while ( t < t1 ) {
05406             if ( *t != AR.PolyFun ) {
05407 SkipFun:
05408                 if ( t == w ) { t += t[1]; w = t; }
05409                 else { i = t[1]; NCOPY(w,t,i) }
05410                 continue;
05411             }
05412             pow1 = 0;
05413             t2 = t + t[1]; t += FUNHEAD;
05414             if ( *t < 0 ) {
05415                 if ( *t == -SYMBOL && t[1] == AR.PolyFunVar ) pow1++;
05416                 else if ( *t != -SNUMBER ) goto NoLegal;
05417                 t += 2;
05418             }
05419             else if ( t[0] == ARGHEAD+8 && t[ARGHEAD] == 8
05420              && t[ARGHEAD+1] == SYMBOL && t[ARGHEAD+3] == AR.PolyFunVar
05421              && t[ARGHEAD+5] == 1 && t[ARGHEAD+6] == 1 && t[ARGHEAD+7] == 3 ) {
05422                 pow1 += t[ARGHEAD+4];
05423                 t += *t;
05424             }
05425             else {
05426 NoLegal:
05427 /* INTERNAL_ERROR_EXCL_START */
05428                 MLOCK(ErrorMessageLock);
```

```
05429                 MesPrint("!>Illegal term with divergence in PolyRatFun");
05430                 MesCall("PolyFunMul");
05431                 MUNLOCK(ErrorMessageLock);
05432                 Terminate(-1);
05433 /* INTERNAL_ERROR_EXCL_STOP */
05434             }
05435             if ( *t < 0 ) {
05436                 if ( *t == -SYMBOL && t[1] == AR.PolyFunVar ) pow1--;
05437                 else if ( *t != -SNUMBER ) goto NoLegal;
05438                 t += 2;
05439             }
05440             else if ( t[0] == ARGHEAD+8 && t[ARGHEAD] == 8
05441              && t[ARGHEAD+1] == SYMBOL && t[ARGHEAD+3] == AR.PolyFunVar
05442              && t[ARGHEAD+5] == 1 && t[ARGHEAD+6] == 1 && t[ARGHEAD+7] == 3 ) {
05443                 pow1 -= t[ARGHEAD+4];
05444                 t += *t;
05445             }
05446             else goto NoLegal;
05447             if ( t == t2 ) pow += pow1;
05448             else goto SkipFun;
05449         }
05450         m = w;
05451         *w++ = AR.PolyFun; *w++ = 0; FILLFUN(w);
05452         if ( pow > 1 ) {
05453             *w++ = 8+ARGHEAD; *w++ = 0; FILLARG(w);
05454             *w++ = 8; *w++ = SYMBOL; *w++ = 4; *w++ = AR.PolyFunVar; *w++ = pow;
05455             *w++ = 1; *w++ = 1; *w++ = 3; *w++ = -SNUMBER; *w++ = 1;
05456         }
05457         else if ( pow == 1 ) {
05458             *w++ = -SYMBOL; *w++ = AR.PolyFunVar; *w++ = -SNUMBER; *w++ = 1;
05459         }
05460         else if ( pow < -1 ) {
05461             *w++ = -SNUMBER; *w++ = 1; *w++ = 8+ARGHEAD; *w++ = 0; FILLARG(w);
05462             *w++ = 8; *w++ = SYMBOL; *w++ = 4; *w++ = AR.PolyFunVar; *w++ = -pow;
05463             *w++ = 1; *w++ = 1; *w++ = 3;
05464         }
05465         else if ( pow == -1 ) {
05466             *w++ = -SNUMBER; *w++ = 1; *w++ = -SYMBOL; *w++ = AR.PolyFunVar;
05467         }
05468         else {
05469             *w++ = -SNUMBER; *w++ = 1; *w++ = -SNUMBER; *w++ = 1;
05470         }
05471         m[1] = w - m;
05472         *w++ = 1; *w++ = 1; *w++ = 3;
05473         *term = w - term;
05474         if ( w > AT.WorkSpace && w < AT.WorkTop ) AT.WorkPointer = w;
05475         return(0);
05476     }
05477 ReStart:
05478     if ( AR.PolyFunType == 2 && ( ( AR.PolyFunExp != 2 )
05479      || ( AR.PolyFunExp == 2 && AN.PolyNormFlag > 1 ) ) ) {
05480         WORD count1 = 0, count2 = 0, count3;
05481         WORD oldtype = AR.SortType;
05482         t = term + 1; t1 = term + *term; t1 -= ABS(t1[-1]);
05483         while ( t < t1 ) {
05484             if ( *t == AR.PolyFun ) {
05485               if ( t[2] && dirty == 0 ) { /* Any dirty flag on? */
05486                 dirty = 1;
05487 /*              ReadPolyRatFun(BHEAD term); */
05488 /*              ToPolyFunGeneral(BHEAD term); */
05489                 poly_ratfun_normalize(BHEAD term);
05490                 if ( term[0] == 0 ) return(0);
05491                 count1 = 0;
05492                 action++;
05493                 goto ReStart;
05494               }
05495               t2 = t + t[1]; tt2 = t+FUNHEAD; count3 = 0;
05496               while ( tt2 < t2 ) { count3++; NEXTARG(tt2); }
05497               if ( count3 == 2 ) {
05498                 count1++;
05499                 if ( ( t[2] & MUSTCLEANPRF ) != 0 ) {   /* Better civilize this guy */
05500                     action++;
05501                     w = AT.WorkPointer;
05502                     AR.SortType = SORTHIGHFIRST;
05503                     t2 = t + t[1]; tt2 = t+FUNHEAD;
05504                     while ( tt2 < t2 ) {
05505                         if ( *tt2 > 0 ) {
05506                             tt4 = tt2; tt1 = tt2 + ARGHEAD; tt2 += *tt2;
05507                             NewSort(BHEAD0);
05508                             while ( tt1 < tt2 ) {
05509                                 i = *tt1; ww = w; NCOPY(ww,tt1,i);
05510                                 AT.WorkPointer = ww;
05511                                 Normalize(BHEAD w);
05512                                 StoreTerm(BHEAD w);
05513                             }
05514                             EndSort(BHEAD w,1);
05515                             ww = w; while ( *ww ) ww += *ww;
```

```
05516                           if ( ww-w != *tt4-ARGHEAD ) { /* Little problem */
05517 /*
05518                               Solution: brute force copy
05519                               Maybe it will never come here????
05520 */
05521                               WORD *r1 = TermMalloc("PolyFunMul");
05522                               WORD ii = (ww-w)-(*tt4-ARGHEAD); /* increment */
05523                               WORD *r2 = tt4+ARGHEAD, *r3, *r4 = r1;
05524                               i = r2 - term; r3 = term; NCOPY(r4,r3,i);
05525                               i = ww-w; ww = w; NCOPY(r4,ww,i);
05526                               r3 = tt2; i = term+*term-tt2; NCOPY(r4,r3,i);
05527                               *r1 = i = r4-r1; r4 = term; r3 = r1;
05528                               NCOPY(r4,r3,i);
05529                               t[1] += ii; t1 += ii; *tt4 += ii;
05530                               tt2 = tt4 + *tt4;
05531                               TermFree(r1,"PolyFunMul");
05532                           }
05533                           else {
05534                               i = ww-w; ww = w; tt1 = tt4+ARGHEAD;
05535                               NCOPY(tt1,ww,i);
05536                               AT.WorkPointer = w;
05537                           }
05538                       }
05539                       else if ( *tt2 <= -FUNCTION ) tt2++;
05540                       else tt2 += 2;
05541                   }
05542                   AR.SortType = oldtype;
05543               }
05544             }
05545           }
05546           t += t[1];
05547       }
05548       if ( count1 <= 1 ) { goto checkaction; }
05549       if ( AR.PolyFunExp == 1 ) {
05550           t = term + *term; t -= ABS(t[-1]);
05551           *t++ = 1; *t++ = 1; *t++ = 3; *term = t - term;
05552       }
05553       {
05554           AR.SortType = SORTHIGHFIRST;
05555 /*        retval = ReadPolyRatFun(BHEAD term); */
05556 /*        ToPolyFunGeneral(BHEAD term); */
05557           retval = poly_ratfun_normalize(BHEAD term);
05558           if ( *term == 0 ) return(retval);
05559           AR.SortType = oldtype;
05560       }
05561
05562       t = term + 1; t1 = term + *term; t1 -= ABS(t1[-1]);
05563       while ( t < t1 ) {
05564           if ( *t == AR.PolyFun ) {
05565             t2 = t + t[1]; tt2 = t+FUNHEAD; count3 = 0;
05566             while ( tt2 < t2 ) { count3++; NEXTARG(tt2); }
05567             if ( count3 == 2 ) {
05568               count2++;
05569             }
05570           }
05571           t += t[1];
05572       }
05573       if ( count1 >= count2 ) {
05574           t = term + 1;
05575           while ( t < t1 ) {
05576               if ( *t == AR.PolyFun ) {
05577                   t2 = t;
05578                   t = t + t[1];
05579                   t2[2] |= (DIRTYFLAG|MUSTCLEANPRF);
05580                   t2 += FUNHEAD;
05581                   while ( t2 < t ) {
05582                       if ( *t2 > 0 ) t2[1] = DIRTYFLAG;
05583                       NEXTARG(t2);
05584                   }
05585               }
05586               else t += t[1];
05587           }
05588       }
05589
05590       w = term + *term;
05591       if ( w > AT.WorkSpace && w < AT.WorkTop ) AT.WorkPointer = w;
05592 checkaction:
05593       if ( action ) retval = action;
05594       return(retval);
05595   }
05596 retry:
05597   if ( term >= AT.WorkSpace && term+*term < AT.WorkTop )
05598       AT.WorkPointer = term + *term;
05599   GETSTOP(term,tstop);
05600   t = term+1;
05601   while ( *t != AR.PolyFun && t < tstop ) t += t[1];
05602   while ( t < tstop && *t == AR.PolyFun ) {
```

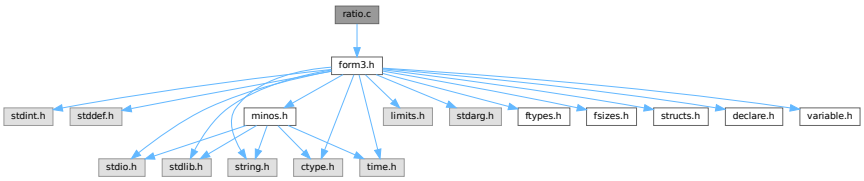
```
05603         if ( t[1] > FUNHEAD ) {
05604             if ( t[FUNHEAD] < 0 ) {
05605                 if ( t[FUNHEAD] <= -FUNCTION && t[1] == FUNHEAD+1 ) break;
05606                 if ( t[FUNHEAD]  > -FUNCTION && t[1] == FUNHEAD+2 ) {
05607                     if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] == 0 ) {
05608                         *term = 0;
05609                         return(0);
05610                     }
05611                     break;
05612                 }
05613             }
05614             else if ( t[FUNHEAD] == t[1] - FUNHEAD ) break;
05615         }
05616         noac = 1;
05617         t += t[1];
05618     }
05619     if ( *t != AR.PolyFun || t >= tstop ) goto done;
05620     fun1 = t;
05621     t += t[1];
05622     while ( t < tstop && *t == AR.PolyFun ) {
05623         if ( t[1] > FUNHEAD ) {
05624             if ( t[FUNHEAD] < 0 ) {
05625                 if ( t[FUNHEAD] <= -FUNCTION && t[1] == FUNHEAD+1 ) break;
05626                 if ( t[FUNHEAD]  > -FUNCTION && t[1] == FUNHEAD+2 ) {
05627                     if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] == 0 ) {
05628                         *term = 0;
05629                         return(0);
05630                     }
05631                     break;
05632                 }
05633             }
05634             else if ( t[FUNHEAD] == t[1] - FUNHEAD ) break;
05635         }
05636         noac = 1;
05637         t += t[1];
05638     }
05639     if ( *t != AR.PolyFun || t >= tstop ) goto done;
05640     fun2 = t;
05641 /*
05642     We have two functions of the proper type.
05643     Count terms (needed for the specials)
05644 */
05645     t = fun1 + FUNHEAD;
05646     if ( *t < 0 ) {
05647         n1 = 1; arg1 = AT.WorkPointer;
05648         ToGeneral(t,arg1,1);
05649         AT.WorkPointer = arg1 + *arg1;
05650     }
05651     else {
05652         t += ARGHEAD;
05653         n1 = 0; t1 = fun1 + fun1[1]; arg1 = t;
05654         while ( t < t1 ) { n1++; t += *t; }
05655     }
05656     t = fun2 + FUNHEAD;
05657     if ( *t < 0 ) {
05658         n2 = 1; arg2 = AT.WorkPointer;
05659         ToGeneral(t,arg2,1);
05660         AT.WorkPointer = arg2 + *arg2;
05661     }
05662     else {
05663         t += ARGHEAD;
05664         n2 = 0; t2 = fun2 + fun2[1]; arg2 = t;
05665         while ( t < t2 ) { n2++; t += *t; }
05666     }
05667 /*
05668     Now we can start the multiplications. We first multiply the terms
05669     without coefficients, then normalize, and finally put the coefficients
05670     in place. This is because one has often truncated series and the
05671     high powers may get killed, while their coefficients are the most
05672     expensive ones.
05673     Note: We may run into fun(-SNUMBER,value)
05674 */
05675     w = AT.WorkPointer;
05676     NewSort(BHEAD0);
05677     if ( AR.PolyFunType == 2 && AR.PolyFunExp == 2 ) {
05678         AT.TrimPower = 1;
05679 /*
05680         We have to find the lowest power in both polynomials.
05681         This will be needed to temporarily correct the AR.PolyFunPow
05682 */
05683         minp1 = MAXPOWER;
05684         for ( t1 = arg1, i1 = 0; i1 < n1; i1++, t1 += *t1 ) {
05685             if ( *t1 == 4 ) {
05686                 if ( minp1 > 0 ) minp1 = 0;
05687             }
05688             else if ( ABS(t1[*t1-1]) == (*t1-1) ) {
05689                 if ( minp1 > 0 ) minp1 = 0;
```

```
05690             }
05691             else {
05692                 if ( t1[1] == SYMBOL && t1[2] == 4 && t1[3] == AR.PolyFunVar ) {
05693                     if ( t1[4] < minp1 ) minp1 = t1[4];
05694                 }
05695                 else {
05696 /* INTERNAL_ERROR_EXCL_START */
05697                     MesPrint("!>Illegal term in expanded polyratfun.");
05698                     goto PolyCall;
05699 /* INTERNAL_ERROR_EXCL_STOP */
05700                 }
05701             }
05702         }
05703         minp2 = MAXPOWER;
05704         for ( t2 = arg2, i2 = 0; i2 < n2; i2++, t2 += *t2 ) {
05705             if ( *t2 == 4 ) {
05706                 if ( minp2 > 0 ) minp2 = 0;
05707             }
05708             else if ( ABS(t2[*t2-1]) == (*t2-1) ) {
05709                 if ( minp2 > 0 ) minp2 = 0;
05710             }
05711             else {
05712                 if ( t2[1] == SYMBOL && t2[2] == 4 && t2[3] == AR.PolyFunVar ) {
05713                     if ( t2[4] < minp2 ) minp2 = t2[4];
05714                 }
05715                 else {
05716 /* INTERNAL_ERROR_EXCL_START */
05717                     MesPrint("!>Illegal term in expanded polyratfun.");
05718                     goto PolyCall;
05719 /* INTERNAL_ERROR_EXCL_STOP */
05720                 }
05721             }
05722         }
05723         AR.PolyFunPow += minp1+minp2;
05724     }
05725     for ( t1 = arg1, i1 = 0; i1 < n1; i1++, t1 += *t1 ) {
05726     for ( t2 = arg2, i2 = 0; i2 < n2; i2++, t2 += *t2 ) {
05727         m = w;
05728         m++;
05729         GETSTOP(t1,tt1);
05730         t = t1 + 1;
05731         while ( t < tt1 ) *m++ = *t++;
05732         GETSTOP(t2,tt2);
05733         t = t2+1;
05734         while ( t < tt2 ) *m++ = *t++;
05735         *m++ = 1; *m++ = 1; *m++ = 3; *w = WORDDIF(m,w);
05736         AT.WorkPointer = m;
05737         if ( Normalize(BHEAD w) ) { LowerSortLevel(); goto PolyCall; }
05738         if ( *w ) {
05739             m = w + *w;
05740             if ( m[-1] != 3 || m[-2] != 1 || m[-3] != 1 ) {
05741                 l3 = REDLENG(m[-1]);
05742                 m -= ABS(m[-1]);
05743                 t = t1 + *t1 - 1;
05744                 l1 = REDLENG(*t);
05745                 if ( MulRat(BHEAD (UWORD *)m,l3,(UWORD *)tt1,l1,(UWORD *)m,&l4) ) {
05746                     LowerSortLevel(); goto PolyCall; }
05747                 if ( AN.ncmod != 0 && TakeModulus((UWORD *)m,&l4,AC.cmod,AN.ncmod,UNPACK|AC.modmode) )
    {
05748                     LowerSortLevel(); goto PolyCall; }
05749                 if ( l4 == 0 ) continue;
05750                 t = t2 + *t2 - 1;
05751                 l2 = REDLENG(*t);
05752                 if ( MulRat(BHEAD (UWORD *)m,l4,(UWORD *)tt2,l2,(UWORD *)m,&l3) ) {
05753                     LowerSortLevel(); goto PolyCall; }
05754                 if ( AN.ncmod != 0 && TakeModulus((UWORD *)m,&l3,AC.cmod,AN.ncmod,UNPACK|AC.modmode) )
    {
05755                     LowerSortLevel(); goto PolyCall; }
05756             }
05757             else {
05758                 m -= 3;
05759                 t = t1 + *t1 - 1;
05760                 l1 = REDLENG(*t);
05761                 t = t2 + *t2 - 1;
05762                 l2 = REDLENG(*t);
05763                 if ( MulRat(BHEAD (UWORD *)tt1,l1,(UWORD *)tt2,l2,(UWORD *)m,&l3) ) {
05764                     LowerSortLevel(); goto PolyCall; }
05765                 if ( AN.ncmod != 0 && TakeModulus((UWORD *)m,&l3,AC.cmod,AN.ncmod,UNPACK|AC.modmode) )
    {
05766                     LowerSortLevel(); goto PolyCall; }
05767             }
05768             if ( l3 == 0 ) continue;
05769             l3 = INCLENG(l3);
05770             m += ABS(l3);
05771             m[-1] = l3;
05772             *w = WORDDIF(m,w);
05773             AT.WorkPointer = m;
```

```
05774             if ( StoreTerm(BHEAD w) ) { LowerSortLevel(); goto PolyCall; }
05775         }
05776     }
05777     }
05778     if ( EndSort(BHEAD w,0) < 0 ) goto PolyCall;
05779     AR.PolyFunPow = OldPolyFunPow;
05780     AT.TrimPower = 0;
05781     if ( *w == 0 ) {
05782         *term = 0;
05783         return(0);
05784     }
05785     t = w;
05786     while ( *t ) t += *t;
05787     AT.WorkPointer = t;
05788     n1 = WORDDIF(t,w);
05789     t1 = term;
05790     while ( t1 < fun1 ) *t++ = *t1++;
05791     t2 = t;
05792     *t++ = AR.PolyFun;
05793     *t++ = FUNHEAD+ARGHEAD+n1;
05794     *t++ = 0;
05795     FILLFUN3(t)
05796     *t++ = ARGHEAD+n1;
05797     *t++ = 0;
05798     FILLARG(t)
05799     NCOPY(t,w,n1);
05800     if ( ToFast(t2+FUNHEAD,t2+FUNHEAD) ) {
05801         if ( t2[FUNHEAD] > -FUNCTION ) t2[1] = FUNHEAD+2;
05802         else t2[FUNHEAD] = FUNHEAD+1;
05803         t = t2 + t2[1];
05804     }
05805     t1 = fun1 + fun1[1];
05806     while ( t1 < fun2 ) *t++ = *t1++;
05807     t1 = fun2 + fun2[1];
05808     t2 = term + *term;
05809     while ( t1 < t2 ) *t++ = *t1++;
05810     *AT.WorkPointer = n1 = WORDDIF(t,AT.WorkPointer);
05811     if ( n1*((LONG)sizeof(WORD)) > AM.MaxTer ) {
05812         MLOCK(ErrorMessageLock);
05813         MesPrint("Term too complex (%d words). MaxTermSize (%l words) is too small.", n1,
      AM.MaxTer/(LONG)sizeof(WORD) );
05814         goto PolyCall2;
05815     }
05816     m = term; t = AT.WorkPointer;
05817     NCOPY(m,t,n1);
05818     action++;
05819     goto retry;
05820 done:
05821     AT.WorkPointer = term + *term;
05822     if ( action && noac ) {
05823         if ( Normalize(BHEAD term) ) goto PolyCall;
05824         AT.WorkPointer = term + *term;
05825     }
05826     return(0);
05827 PolyCall:;
05828     MLOCK(ErrorMessageLock);
05829 PolyCall2:;
05830     AR.PolyFunPow = OldPolyFunPow;
05831     MesCall("PolyFunMul");
05832     MUNLOCK(ErrorMessageLock);
05833     SETERROR(-1)
05834 }
05835
05836 /*
05837         #] PolyFunMul :
05838     #] Processor :
05839 */
```

4.119 ratio.c File Reference

`#include "form3.h"`
Include dependency graph for ratio.c:



Data Structures

- struct ARGBUFFER

Functions

- int RatioFind (PHEAD WORD *term, WORD *params)
- int RatioGen (PHEAD WORD *term, WORD *params, WORD num, WORD level)
- int BinomGen (PHEAD WORD *term, WORD level, WORD **tstops, WORD x1, WORD x2, WORD pow1, WORD pow2, WORD sign, UWORD *coef, WORD ncoef)
- int DoSumF1 (PHEAD WORD *term, WORD *params, WORD replac, WORD level)
- int Glue (PHEAD WORD *term1, WORD *term2, WORD *sub, WORD insert)
- int DoSumF2 (PHEAD WORD *term, WORD *params, WORD replac, WORD level)
- int GCDfunction (PHEAD WORD *term, WORD level)
- WORD * GCDfunction3 (PHEAD WORD *in1, WORD *in2)
- WORD * PutExtraSymbols (PHEAD WORD *in, WORD startebuf, int *actionflag)
- WORD * TakeExtraSymbols (PHEAD WORD *in, WORD startebuf)
- WORD * MultiplyWithTerm (PHEAD WORD *in, WORD *term, WORD par)
- WORD * TakeContent (PHEAD WORD *in, WORD *term)
- int MergeSymbolLists (PHEAD WORD *old, WORD *extra, int par)
- int MergeDotproductLists (PHEAD WORD *old, WORD *extra, int par)
- WORD * CreateExpression (PHEAD WORD nexp)
- int GCDterms (PHEAD WORD *term1, WORD *term2, WORD *termout)
- int ReadPolyRatFun (PHEAD WORD *term)
- int FromPolyRatFun (PHEAD WORD *fun, WORD **numout, WORD **denout)
- WORD * TakeSymbolContent (PHEAD WORD *in, WORD *term)
- void GCDclean (PHEAD WORD *num, WORD *den)
- WORD * PolyDiv (PHEAD WORD *a, WORD *b, char *text)
- int DIVfunction (PHEAD WORD *term, WORD level, int par)
- WORD * MULfunc (PHEAD WORD *p1, WORD *p2)
- WORD * ConvertArgument (PHEAD WORD *arg, int *type)
- int ExpandRat (PHEAD WORD *fun)
- int InvPoly (PHEAD WORD *inpoly, WORD maxpow, WORD sym)

Variables

- WORD divrem [4] = { DIVFUNCTION, REMFUNCTION, INVERSEFUNCTION, MULFUNCTION }
- char * TheErrorMessage [ ]

4.119.1 Detailed Description

A variety of routines: The ratio command for partial fractioning (rather old. Schoonschip inheritance) The sum routines.

Definition in file ratio.c.

4.119.2 Function Documentation

4.119.2.1 RatioFind()

```
int RatioFind (
            PHEAD WORD * term,
            WORD * params )
```

Definition at line 62 of file ratio.c.

4.119.2.2 RatioGen()

```
int RatioGen (
            PHEAD WORD * term,
            WORD * params,
            WORD num,
            WORD level )
```

Definition at line 170 of file ratio.c.

4.119.2.3 BinomGen()

```
int BinomGen (
            PHEAD WORD * term,
            WORD level,
            WORD ** tstops,
            WORD x1,
            WORD x2,
            WORD pow1,
            WORD pow2,
            WORD sign,
            UWORD * coef,
            WORD ncoef )
```

Definition at line 320 of file ratio.c.

4.119.2.4 DoSumF1()

```
int DoSumF1 (
            PHEAD WORD * term,
            WORD * params,
            WORD replac,
            WORD level )
```

Definition at line 395 of file ratio.c.

4.119.2.5 Glue()

```
int Glue (
            PHEAD WORD * term1,
            WORD * term2,
            WORD * sub,
            WORD insert )
```

Definition at line 461 of file ratio.c.

4.119.2.6 DoSumF2()

```
int DoSumF2 (
            PHEAD WORD * term,
            WORD * params,
            WORD replac,
            WORD level )
```

Definition at line 520 of file ratio.c.

4.119.2.7 GCDfunction()

```
int GCDfunction (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 609 of file ratio.c.

4.119.2.8 GCDfunction3()

```
WORD * GCDfunction3 (
            PHEAD WORD * in1,
            WORD * in2 )
```

Definition at line 1147 of file ratio.c.

4.119.2.9 PutExtraSymbols()

```
WORD * PutExtraSymbols (
            PHEAD WORD * in,
            WORD startebuf,
            int * actionflag )
```

Definition at line 1249 of file ratio.c.

4.119.2.10 TakeExtraSymbols()

```
WORD * TakeExtraSymbols (
            PHEAD WORD * in,
            WORD startebuf )
```

Definition at line 1278 of file ratio.c.

4.119.2.11 MultiplyWithTerm()

```
WORD * MultiplyWithTerm (
            PHEAD WORD * in,
            WORD * term,
            WORD par )
```

Definition at line 1317 of file ratio.c.

4.119.2.12 TakeContent()

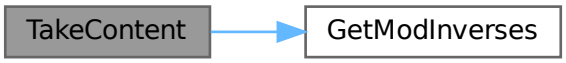
```
WORD * TakeContent (
            PHEAD WORD * in,
            WORD * term )
```

Implements part of the old ExecArg in which we take common factors from arguments with more than one term. Here the input is a sequence of terms in 'in' and the answer is a content-free sequence of terms. This sequence has been allocated by the Malloc1 routine in a call to EndSort, unless the expression was already content-free. In that case the input pointer is returned. The content is returned in term. This is supposed to be a separate allocation, made by TermMalloc in the calling routine.

Definition at line 1382 of file ratio.c.

References GetModInverses().

Here is the call graph for this function:



4.119.2.13 MergeSymbolLists()

```
int MergeSymbolLists (
            PHEAD WORD * old,
            WORD * extra,
            int par )
```

Definition at line 1830 of file ratio.c.

4.119.2.14 MergeDotproductLists()

```
int MergeDotproductLists (
            PHEAD WORD * old,
            WORD * extra,
            int par )
```

Definition at line 1950 of file ratio.c.

4.119.2.15 CreateExpression()

```
WORD * CreateExpression (
            PHEAD WORD nexp )
```

Definition at line 2077 of file ratio.c.

4.119.2.16 GCDterms()

```
int GCDterms (
            PHEAD WORD * term1,
            WORD * term2,
            WORD * termout )
```

Definition at line 2133 of file ratio.c.

4.119.2.17 ReadPolyRatFun()

```
int ReadPolyRatFun (
            PHEAD WORD * term )
```

Definition at line 2321 of file ratio.c.

4.119.2.18 FromPolyRatFun()

```
int FromPolyRatFun (
            PHEAD WORD * fun,
            WORD ** numout,
            WORD ** denout )
```

Definition at line 2440 of file ratio.c.

4.119.2.19 TakeSymbolContent()

```
WORD * TakeSymbolContent (
            PHEAD WORD * in,
            WORD * term )
```

Implements part of the old ExecArg in which we take common factors from arguments with more than one term. We allow only symbols as this code is used for the polyratfun only. We have a special routine, because the generic TakeContent does too much work and speed is at a premium here. Input: in is the input expression as a sequence of terms. Output: term: the content return value: the contentfree expression. it is in new allocation, made by TermMalloc. (should be in a TermMalloc space?)

Definition at line 2493 of file ratio.c.

References GetModInverses().

Here is the call graph for this function:

4.119.2.20 GCDclean()

```
void GCDclean (
             PHEAD WORD * num,
             WORD * den )
```

Definition at line 2703 of file ratio.c.

4.119.2.21 PolyDiv()

```
WORD * PolyDiv (
             PHEAD WORD * a,
             WORD * b,
             char * text )
```

Definition at line 2800 of file ratio.c.

4.119.2.22 DIVfunction()

```
int DIVfunction (
             PHEAD WORD * term,
             WORD level,
             int par )
```

Definition at line 2847 of file ratio.c.

4.119.2.23 MULfunc()

```
WORD * MULfunc (
             PHEAD WORD * p1,
             WORD * p2 )
```

Definition at line 3020 of file ratio.c.

4.119.2.24 ConvertArgument()

```
WORD * ConvertArgument (
             PHEAD WORD * arg,
             int * type )
```

Definition at line 3083 of file ratio.c.

4.119.2.25 ExpandRat()

```
int ExpandRat (
             PHEAD WORD * fun )
```

Definition at line 3179 of file ratio.c.

4.119.2.26 InvPoly()

```
int InvPoly (
              PHEAD WORD * inpoly,
              WORD maxpow,
              WORD sym )
```

Definition at line 3649 of file ratio.c.

4.119.3 Variable Documentation

4.119.3.1 divrem

```
WORD divrem[4] = { DIVFUNCTION, REMFUNCTION, INVERSEFUNCTION, MULFUNCTION }
```

Definition at line 2845 of file ratio.c.

4.119.3.2 TheErrorMessage

```
char* TheErrorMessage[]
```

Initial value:

```
= {
     "PolyRatFun not of a type that FORM will expand: incorrect variable inside."
    ,"Division by zero in PolyRatFun encountered in ExpandRat."
    ,"Irregular code in PolyRatFun encountered in ExpandRat."
    ,"Called from ExpandRat."
    ,"WorkSpace overflow. Change parameter WorkSpace in setup file?"
    ,"Illegal term in expanded polyratfun."
    }
```

Definition at line 3170 of file ratio.c.

4.120 ratio.c

Go to the documentation of this file.

```
00001
00008 /* #[ License : */
00009 /*
00010  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00011  *   When using this file you are requested to refer to the publication
00012  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013  *   This is considered a matter of courtesy as the development was paid
00014  *   for by FOM the Dutch physics granting agency and we would like to
00015  *   be able to track its scientific use to convince FOM of its value
00016  *   for the community.
00017  *
00018  *   This file is part of FORM.
00019  *
00020  *   FORM is free software: you can redistribute it and/or modify it under the
00021  *   terms of the GNU General Public License as published by the Free Software
00022  *   Foundation, either version 3 of the License, or (at your option) any later
00023  *   version.
00024  *
00025  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00027  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00028  *   details.
00029  *
00030  *   You should have received a copy of the GNU General Public License along
00031  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00032  */
00033 /* #] License : */
```

```
00034 /*
00035     #[ Includes : ratio.c
00036 */
00037
00038 #include "form3.h"
00039
00040 /*
00041     #] Includes :
00042     #[ Ratio :
00043
00044     These are the special operations regarding simple polynomials.
00045     The first and most needed is the partial fractioning expansion.
00046     Ratio,x1,x2,x3
00047
00048     The files belonging to the ratio command serve also as a good example
00049     of how to implement a new operation.
00050
00051         #[ RatioFind :
00052
00053         The routine that should locate the need for a ratio command.
00054         If located the corresponding symbols are removed and the
00055         operational parameters are loaded. A subexpression pointer
00056         is inserted and the code for success is returned.
00057
00058         params points at the compiler output block defined in RatioComp.
00059
00060 */
00061
00062 int RatioFind(PHEAD WORD *term, WORD *params)
00063 {
00064     GETBIDENTITY
00065     WORD *t, *m, *r;
00066     WORD x1, x2, i;
00067     WORD *y1, *y2, n1 = 0, n2 = 0;
00068     x1 = params[3];
00069     x2 = params[4];
00070     m = t = term;
00071     m += *m;
00072     m -= ABS(m[-1]);
00073     t++;
00074     if ( t < m ) do {
00075         if ( *t == SYMBOL ) {
00076             y1 = 0;
00077             y2 = 0;
00078             r = t + t[1];
00079             m = t + 2;
00080             do {
00081                 if ( *m == x1 ) { y1 = m; n1 = m[1]; }
00082                 else if ( *m == x2 ) { y2 = m; n2 = m[1]; }
00083                 m += 2;
00084             } while ( m < r );
00085             if ( !y1 || !y2 || ( n1 > 0 && n2 > 0 ) ) return(0);
00086             m -= 2;
00087             if ( y1 > y2 ) { r = y1; y1 = y2; y2 = r; }
00088             *y2 = *m; y2[1] = m[1];
00089             m -= 2;
00090             *y1 = *m; y1[1] = m[1];
00091             i = WORDDIF(m,t);
00092 #if SUBEXPSIZE > 6
00093 We have to revise the code for the second case.
00094 #endif
00095             if ( i > 2 ) {      /* Subexpression fits exactly */
00096                 t[1] = i;
00097                 y1 = term+*term;
00098                 y2 = y1+SUBEXPSIZE-4;
00099                 r = m+4;
00100                 while ( y1 > r ) *--y2 = *--y1;
00101                 *m++ = SUBEXPRESSION;
00102                 *m++ = SUBEXPSIZE;
00103                 *m++ = -1;
00104                 *m++ = 1;
00105                 *m++ = DUMMYBUFFER;
00106                 FILLSUB(m)
00107                 *term += SUBEXPSIZE-4;
00108             }
00109             else {              /* All symbols are gone. Rest has to be moved */
00110                 m -= 2;
00111                 *m++ = SUBEXPRESSION;
00112                 *m++ = SUBEXPSIZE;
00113                 *m++ = -1;
00114                 *m++ = 1;
00115                 *m++ = DUMMYBUFFER;
00116                 FILLSUB(m)
00117                 t = term;
00118                 t += *t;
00119                 *term += SUBEXPSIZE-6;
00120                 r = m + 6-SUBEXPSIZE;
```

```
00121                 do { *m++ = *r++; } while ( r < t );
00122             }
00123             t = AT.TMout;           /* Load up the TM out array for the generator */
00124             *t++ = 7;
00125             *t++ = RATIO;
00126             *t++ = x1;
00127             *t++ = x2;
00128             *t++ = params[5];
00129             *t++ = n1;
00130             *t++ = n2;
00131             return(1);
00132         }
00133         t += t[1];
00134     } while ( t < m );
00135     return(0);
00136 }
00137
00138 /*
00139         #] RatioFind :
00140         #[ RatioGen :
00141
00142         The algorithm:
00143         x1^-n1*x2^n2    ==>  x2 --> x1 + x3
00144         x1^n1*x2^-n2    ==>  x1 --> x2 - x3
00145         x1^-n1*x2^-n2   ==>
00146
00147            +sum(i=0,n1-1){(-1)^i*binom(n2-1+i,n2-1)
00148                    *x3^-(n2+i)*x1^-(n1-i)}
00149            +sum(i=0,n2-1){(-1)^(n1)*binom(n1-1+i,n1-1)
00150                    *x3^-(n1+i)*x2^-(n2-i)}
00151
00152         Actually there is an amount of arbitrariness in the first two
00153         formulae and the replacement x2 -> x1 + x3 could be made 'by hand'.
00154         It is better to use the nontrivial 'minimal change' formula:
00155
00156         x1^-n1*x2^n2:   if ( n1 >= n2 ) {
00157                            +sum(i=0,n2){x3^i*x1^-(n1-n2+i)*binom(n2,i)}
00158                         }
00159                         else {
00160                             sum(i=0,n2-n1){x2^(n2-n1-i)*x3^i*binom(n1-1+i,n1-1)}
00161                            +sum(i=0,n1-1){x3^(n2-i)*x1^-(n1-i)*binom(n2,i)}
00162                         }
00163         x1^n1*x2^-n2:   Same but x3 -> -x3.
00164
00165         The contents of the AT.TMout/params array are:
00166         length,type,x1,x2,x3,n1,n2
00167
00168 */
00169
00170 int RatioGen(PHEAD WORD *term, WORD *params, WORD num, WORD level)
00171 {
00172     GETBIDENTITY
00173     WORD *t, *m;
00174     WORD *tstops[3];
00175     WORD n1, n2, i, j;
00176     WORD x1,x2,x3;
00177     UWORD *coef;
00178     WORD ncoef, sign = 0;
00179     coef = (UWORD *)AT.WorkPointer;
00180     t = term;
00181     tstops[2] = m = t + *t;
00182     m -= ABS(m[-1]);
00183     t++;
00184     do {
00185         if ( *t == SUBEXPRESSION && t[2] == num ) break;
00186         t += t[1];
00187     } while ( t < m );
00188     tstops[0] = t;
00189     tstops[1] = t + t[1];
00190 /*
00191     Copying to termout will be from term to tstop1, then the induced part
00192     and finally from tstop2 to tstop3
00193
00194     Now separate the various cases:
00195
00196 */
00197     t = params + 2;
00198     x1 = *t++;
00199     x2 = *t++;
00200     x3 = *t++;
00201     n1 = *t++;
00202     n2 = *t++;
00203     if ( n1 > 0 ) {     /* Flip the variables and indicate -x3 */
00204         n2 = -n2;
00205         sign = 1;
00206         i = n1; n1 = n2; n2 = i;
00207         i = x1; x1 = x2; x2 = i;
```

```
00208         goto PosNeg;
00209     }
00210     else if ( n2 > 0 ) {
00211         n1 = -n1;
00212 PosNeg:
00213         if ( n2 <= n1 ) {   /* x1 -> x2 + x3 */
00214             *coef = 1;
00215             ncoef = 1;
00216             AT.WorkPointer = (WORD *)(coef + 1);
00217             j = n2;
00218             for ( i = 0; i <= n2; i++ ) {
00219                 if ( BinomGen(BHEAD term,level,tstops,x1,x3,n2-n1-i,i,sign&i
00220                 ,coef,ncoef) ) goto RatioCall;
00221                 if ( i < n2 ) {
00222                     if ( Product(coef,&ncoef,j) ) goto RatioCall;
00223                     if ( Quotient(coef,&ncoef,i+1) ) goto RatioCall;
00224                     j--;
00225                     AT.WorkPointer = (WORD *)(coef + ABS(ncoef));
00226                 }
00227             }
00228             AT.WorkPointer = (WORD *)(coef);
00229             return(0);
00230         }
00231         else {
00232 /*
00233             sum(i=0,n2-n1){x2^(n2-n1-i)*x3^i*binom(n1-1+i,n1-1)}
00234            +sum(i=0,n1-1){x3^(n2-i)*x1^-(n1-i)*binom(n2,i)}
00235 */
00236             *coef = 1;
00237             ncoef = 1;
00238             AT.WorkPointer = (WORD *)(coef + 1);
00239             j = n2 - n1;
00240             for ( i = 0; i <= j; i++ ) {
00241                 if ( BinomGen(BHEAD term,level,tstops,x2,x3,n2-n1-i,i,sign&i
00242                 ,coef,ncoef) ) goto RatioCall;
00243                 if ( i < j ) {
00244                     if ( Product(coef,&ncoef,n1+i) ) goto RatioCall;
00245                     if ( Quotient(coef,&ncoef,i+1) ) goto RatioCall;
00246                     AT.WorkPointer = (WORD *)(coef + ABS(ncoef));
00247                 }
00248             }
00249             *coef = 1;
00250             ncoef = 1;
00251             AT.WorkPointer = (WORD *)(coef + 1);
00252             j = n1-1;
00253             for ( i = 0; i <= j; i++ ) {
00254                 if ( BinomGen(BHEAD term,level,tstops,x1,x3,i-n1,n2-i,sign&(n2-i)
00255                 ,coef,ncoef) ) goto RatioCall;
00256                 if ( i < j ) {
00257                     if ( Product(coef,&ncoef,n2-i) ) goto RatioCall;
00258                     if ( Quotient(coef,&ncoef,i+1) ) goto RatioCall;
00259                     AT.WorkPointer = (WORD *)(coef + ABS(ncoef));
00260                 }
00261             }
00262             AT.WorkPointer = (WORD *)(coef);
00263             return(0);
00264         }
00265     }
00266     else {
00267         n2 = -n2;
00268         n1 = -n1;
00269 /*
00270            +sum(i=0,n1-1){(-1)^i*binom(n2-1+i,n2-1)
00271                     *x3^-(n2+i)*x1^-(n1-i)}
00272            +sum(i=0,n2-1){(-1)^(n1)*binom(n1-1+i,n1-1)
00273                     *x3^-(n1+i)*x2^-(n2-i)}
00274 */
00275         *coef = 1;
00276         ncoef = 1;
00277         AT.WorkPointer = (WORD *)(coef + 1);
00278         j = n1-1;
00279         for ( i = 0; i <= j; i++ ) {
00280             if ( BinomGen(BHEAD term,level,tstops,x1,x3,i-n1,-n2-i,i&1
00281             ,coef,ncoef) ) goto RatioCall;
00282             if ( i < j ) {
00283                 if ( Product(coef,&ncoef,n2+i) ) goto RatioCall;
00284                 if ( Quotient(coef,&ncoef,i+1) ) goto RatioCall;
00285                 AT.WorkPointer = (WORD *)(coef + ABS(ncoef));
00286             }
00287         }
00288         *coef = 1;
00289         ncoef = 1;
00290         AT.WorkPointer = (WORD *)(coef + 1);
00291         j = n2-1;
00292         for ( i = 0; i <= j; i++ ) {
00293             if ( BinomGen(BHEAD term,level,tstops,x2,x3,i-n2,-n1-i,n1&1
00294             ,coef,ncoef) ) goto RatioCall;
```

```
00295             if ( i < j ) {
00296                 if ( Product(coef,&ncoef,n1+i) ) goto RatioCall;
00297                 if ( Quotient(coef,&ncoef,i+1) ) goto RatioCall;
00298                 AT.WorkPointer = (WORD *)(coef + ABS(ncoef));
00299             }
00300         }
00301         AT.WorkPointer = (WORD *)(coef);
00302         return(0);
00303     }
00304
00305 RatioCall:
00306     MLOCK(ErrorMessageLock);
00307     MesCall("RatioGen");
00308     MUNLOCK(ErrorMessageLock);
00309     SETERROR(-1)
00310 }
00311
00312 /*
00313         #] RatioGen :
00314         #[ BinomGen :
00315
00316         Routine for the generation of terms in a binomialtype expansion.
00317
00318 */
00319
00320 int BinomGen(PHEAD WORD *term, WORD level, WORD **tstops, WORD x1, WORD x2,
00321              WORD pow1, WORD pow2, WORD sign, UWORD *coef, WORD ncoef)
00322 {
00323     GETBIDENTITY
00324     WORD *t, *r;
00325     WORD *termout;
00326     WORD k;
00327     termout = AT.WorkPointer;
00328     t = termout;
00329     r = term;
00330     do { *t++ = *r++; } while ( r < tstops[0] );
00331     *t++ = SYMBOL;
00332     if ( pow2 == 0 ) {
00333         if ( pow1 == 0 ) t--;
00334         else { *t++ = 4; *t++ = x1; *t++ = pow1; }
00335     }
00336     else if ( pow1 == 0 ) {
00337         *t++ = 4; *t++ = x2; *t++ = pow2;
00338     }
00339     else {
00340         *t++ = 6; *t++ = x1; *t++ = pow1; *t++ = x2; *t++ = pow2;
00341     }
00342     *t++ = LNUMBER;
00343     *t++ = ABS(ncoef) + 3;
00344     *t = ncoef;
00345     if ( sign ) *t = -*t;
00346     t++;
00347     ncoef = ABS(ncoef);
00348     for ( k = 0; k < ncoef; k++ ) *t++ = coef[k];
00349     r = tstops[1];
00350     do { *t++ = *r++; } while ( r < tstops[2] );
00351     *termout = WORDDIF(t,termout);
00352     AT.WorkPointer = t;
00353     if ( AT.WorkPointer > AT.WorkTop ) {
00354         MLOCK(ErrorMessageLock);
00355         MesWork();
00356         MUNLOCK(ErrorMessageLock);
00357         return(-1);
00358     }
00359     *AN.RepPoint = 1;
00360     AR.expchanged = 1;
00361     if ( Generator(BHEAD termout,level) ) {
00362         MLOCK(ErrorMessageLock);
00363         MesCall("BinomGen");
00364         MUNLOCK(ErrorMessageLock);
00365         SETERROR(-1)
00366     }
00367     AT.WorkPointer = termout;
00368     return(0);
00369 }
00370
00371 /*
00372         #] BinomGen :
00373     #] Ratio :
00374     #[ Sum :
00375         #[ DoSumF1 :
00376
00377         Routine expands a sum_ function.
00378         Its arguments are:
00379         The term in which the function occurs.
00380         The parameter list:
00381             length of parameter field
```

```
00382             function number (SUMNUM1)
00383             number of the symbol that is loop parameter
00384             min value
00385             max value
00386             increment
00387         the number of the subexpression to be removed
00388         the level in the generation tree.
00389 
00390         Note that the insertion of the loop parameter in the argument
00391         is done via the regular wildcard substitution mechanism.
00392 
00393 */
00394 
00395 int DoSumF1(PHEAD WORD *term, WORD *params, WORD replac, WORD level)
00396 {
00397     GETBIDENTITY
00398     WORD *termout, *t, extractbuff = AT.TMbuff;
00399     WORD isum, ival, iinc;
00400     LONG from;
00401     CBUF *C;
00402     ival = params[3];
00403     iinc = params[5];
00404     if ( ( iinc > 0 && params[4] >= ival )
00405       || ( iinc < 0 && params[4] <= ival ) ) {
00406         isum = (params[4] - ival)/iinc + 1;
00407     }
00408     else return(0);
00409     termout = AT.WorkPointer;
00410     AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
00411     if ( AT.WorkPointer > AT.WorkTop ) {
00412         MLOCK(ErrorMessageLock);
00413         MesWork();
00414         MUNLOCK(ErrorMessageLock);
00415         return(-1);
00416     }
00417     t = term + 1;
00418     while ( *t != SUBEXPRESSION || t[2] != replac || t[4] != extractbuff )
00419                     t += t[1];
00420     C = cbuf+t[4];
00421     t += SUBEXPSIZE;
00422     if ( params[2] < 0 ) {
00423         while ( *t != INDTOIND || t[2] != -params[2] ) t += t[1];
00424         *t = INDTOIND;
00425     }
00426     else {
00427         while ( *t > SYMTOSUB || t[2] != params[2] ) t += t[1];
00428         *t = SYMTONUM;
00429     }
00430     do {
00431         t[3] = ival;
00432         from = C->rhs[replac] - C->Buffer;
00433         while ( C->Buffer[from] ) {
00434             if ( InsertTerm(BHEAD term,replac,extractbuff,C->Buffer+from,termout,0) < 0 ) goto
      SumF1Call;
00435             AT.WorkPointer = termout + *termout;
00436             if ( Generator(BHEAD termout,level) < 0 ) goto SumF1Call;
00437             from += C->Buffer[from];
00438         }
00439         ival += iinc;
00440     } while ( --isum > 0 );
00441     AT.WorkPointer = termout;
00442     return(0);
00443 SumF1Call:
00444     MLOCK(ErrorMessageLock);
00445     MesCall("DoSumF1");
00446     MUNLOCK(ErrorMessageLock);
00447     SETERROR(-1)
00448 }
00449 
00450 /*
00451         #] DoSumF1 :
00452         #[ Glue :
00453 
00454         Routine multiplies two terms. The second term is subject
00455         to the wildcard substitutions in sub.
00456         Output in the first term. This routine is a variation on
00457         the routine InsertTerm.
00458 
00459 */
00460 
00461 int Glue(PHEAD WORD *term1, WORD *term2, WORD *sub, WORD insert)
00462 {
00463     GETBIDENTITY
00464     UWORD *coef;
00465     WORD ncoef, *t, *t1, *t2, i, nc2, nc3, old, newer;
00466     coef = (UWORD *)(TermMalloc("Glue"));
00467     t = term1;
```

```
00468     t += *t;
00469     i = t[-1];
00470     t -= ABS(i);
00471     old = WORDDIF(t,term1);
00472     ncoef = REDLENG(i);
00473     if ( i < 0 ) i = -i;
00474     i--;
00475     t1 = t;
00476     t2 = (WORD *)coef;
00477     while ( --i >= 0 ) *t2++ = *t1++;
00478     i = *--t;
00479     nc2 = WildFill(BHEAD t,term2,sub);
00480     *t = i;
00481     t += nc2;
00482     nc2 = t[-1];
00483     t -= ABS(nc2);
00484     newer = WORDDIF(t,term1);
00485     if ( MulRat(BHEAD (UWORD *)t,REDLENG(nc2),coef,ncoef,(UWORD *)t,&nc3) ) {
00486         MLOCK(ErrorMessageLock);
00487         MesCall("Glue");
00488         MUNLOCK(ErrorMessageLock);
00489         TermFree(coef,"Glue");
00490         SETERROR(-1)
00491     }
00492     i = (ABS(nc3))*2;
00493     t += i++;
00494     *t++ = (nc3 >= 0)?i:-i;
00495     *term1 = WORDDIF(t,term1);
00496 /*
00497     Switch the new piece with the old tail, so that noncommuting
00498     variables get into their proper spot.
00499 */
00500     i = old - insert;
00501     t1 = t;
00502     t2 = term1+insert;
00503     NCOPY(t1,t2,i);
00504     i = newer - old;
00505     t1 = term1+insert;
00506     t2 = term1+old;
00507     NCOPY(t1,t2,i);
00508     t2 = t;
00509     i = old - insert;
00510     NCOPY(t1,t2,i);
00511     TermFree(coef,"Glue");
00512     return(0);
00513 }
00514
00515 /*
00516         #] Glue :
00517         #[ DoSumF2 :
00518 */
00519
00520 int DoSumF2(PHEAD WORD *term, WORD *params, WORD replac, WORD level)
00521 {
00522     GETBIDENTITY
00523     WORD *termout, *t, *from, *sub, *to, extractbuff = AT.TMbuff;
00524     WORD isum, ival, iinc, insert, i;
00525     CBUF *C;
00526     ival = params[3];
00527     iinc = params[5];
00528     if ( ( iinc > 0 && params[4] >= ival )
00529       || ( iinc < 0 && params[4] <= ival ) ) {
00530         isum = (params[4] - ival)/iinc + 1;
00531     }
00532     else return(0);
00533     termout = AT.WorkPointer;
00534     AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
00535     if ( AT.WorkPointer > AT.WorkTop ) {
00536         MLOCK(ErrorMessageLock);
00537         MesWork();
00538         MUNLOCK(ErrorMessageLock);
00539         return(-1);
00540     }
00541     t = term + 1;
00542     while ( *t != SUBEXPRESSION || t[2] != replac || t[4] != extractbuff ) t += t[1];
00543     insert = WORDDIF(t,term);
00544
00545     from = term;
00546     to = termout;
00547     while ( from < t ) *to++ = *from++;
00548     from += t[1];
00549     sub = term + *term;
00550     while ( from < sub ) *to++ = *from++;
00551     *termout -= t[1];
00552
00553     sub = t;
00554     C = cbuf+t[4];
```

```
00555     t += SUBEXPSIZE;
00556     if ( params[2] < 0 ) {
00557         while ( *t != INDTOIND || t[2] != -params[2] ) t += t[1];
00558         *t = INDTOIND;
00559     }
00560     else {
00561         while ( *t > SYMTOSUB || t[2] != params[2] ) t += t[1];
00562         *t = SYMTONUM;
00563     }
00564     t[3] = ival;
00565     for(;;) {
00566         AT.WorkPointer = termout + *termout;
00567         to = AT.WorkPointer;
00568         if ( ( to + *termout ) > AT.WorkTop ) {
00569             MLOCK(ErrorMessageLock);
00570             MesWork();
00571             MUNLOCK(ErrorMessageLock);
00572             return(-1);
00573         }
00574         from = termout;
00575         i = *termout;
00576         NCOPY(to,from,i);
00577         from = AT.WorkPointer;
00578         AT.WorkPointer = to;
00579         if ( Generator(BHEAD from,level) < 0 ) goto SumF2Call;
00580         if ( --isum <= 0 ) break;
00581         ival += iinc;
00582         t[3] = ival;
00583         if ( Glue(BHEAD termout,C->rhs[replac],sub,insert) < 0 ) goto SumF2Call;
00584     }
00585     AT.WorkPointer = termout;
00586     return(0);
00587 SumF2Call:
00588     MLOCK(ErrorMessageLock);
00589     MesCall("DoSumF2");
00590     MUNLOCK(ErrorMessageLock);
00591     SETERROR(-1)
00592 }
00593
00594 /*
00595         #] DoSumF2 :
00596     #] Sum :
00597     #[ GCDfunction :
00598         #[ GCDfunction :
00599 */
00600
00601 typedef struct {
00602     WORD *buffer;
00603     DOLLARS dollar;
00604     LONG size;
00605     int type;
00606     int dummy;
00607 } ARGBUFFER;
00608
00609 int GCDfunction(PHEAD WORD *term,WORD level)
00610 {
00611     GETBIDENTITY
00612     WORD *t, *tstop, *tf, *termout, *tin, *tout, *m, *mnext, *mstop, *mm;
00613     int todo, i, ii, j, istart, sign = 1, action = 0;
00614     WORD firstshort = 0, firstvalue = 0, gcdisone = 0, mlength, tlength, newlength;
00615     WORD totargs = 0, numargs, argsdone = 0, *mh, oldval1, *g, *gcdout = 0;
00616     WORD *arg1, *arg2;
00617     UWORD x1,x2,x3;
00618     LONG args;
00619 #if ( FUNHEAD > 4 )
00620     WORD sh[FUNHEAD+5];
00621 #else
00622     WORD sh[9];
00623 #endif
00624     DOLLARS d;
00625     ARGBUFFER *abuf = 0, ab;
00626 /*
00627     #[ Find Function. Count arguments :
00628
00629     First find the proper function
00630 */
00631     t = term + *term; tlength = t[-1];
00632     tstop = t - ABS(tlength);
00633     t = term + 1;
00634     while ( t < tstop ) {
00635         if ( *t != GCDFUNCTION ) { t += t[1]; continue; }
00636         todo = 1; totargs = 0;
00637         tf = t + FUNHEAD;
00638         while ( tf < t + t[1] ) {
00639             totargs++;
00640             if ( *tf > 0 && tf[1] != 0 ) todo = 0;
00641             NEXTARG(tf);
```

```
00642         }
00643         if ( todo ) break;
00644         t += t[1];
00645     }
00646     if ( t >= tstop ) {
00647 /* INTERNAL_ERROR_EXCL_START */
00648         MLOCK(ErrorMessageLock);
00649         MesPrint("!>Internal error. Indicated gcd_ function not encountered.");
00650         MUNLOCK(ErrorMessageLock);
00651         Terminate(-1);
00652 /* INTERNAL_ERROR_EXCL_STOP */
00653     }
00654     WantAddPointers(totargs);
00655     args = AT.pWorkPointer; AT.pWorkPointer += totargs;
00656 /*
00657     #] Find Function. Count arguments :
00658     #[ Do short arguments :
00659
00660     The function we need, in agreement with TestSub, is now in t
00661     Make first a compilation of the short arguments (except $-s and expressions)
00662     to see whether we need to do much work.
00663     This means that after this scan we can ignore all short arguments with
00664     the exception of unevaluated $-s and expressions.
00665 */
00666     numargs = 0;
00667     firstshort = 0;
00668     tf = t + FUNHEAD;
00669     while ( tf < t + t[1] ) {
00670         if ( *tf == -SNUMBER && tf[1] == 0 ) { NEXTARG(tf); continue; }
00671         if ( *tf > 0 || *tf == -DOLLAREXPRESSION || *tf == -EXPRESSION ) {
00672             AT.pWorkSpace[args+numargs++] = tf;
00673             NEXTARG(tf); continue;
00674         }
00675         if ( firstshort == 0 ) {
00676             firstshort = *tf;
00677             if ( *tf <= -FUNCTION ) { firstvalue = -(*tf); }
00678             else                    { firstvalue = tf[1]; }
00679             NEXTARG(tf);
00680             argsdone++;
00681             continue;
00682         }
00683         else if ( *tf != firstshort ) {
00684             if ( *tf != -INDEX && *tf != -VECTOR && *tf != -MINVECTOR ) {
00685                 argsdone++; gcdisone = 1; break;
00686             }
00687             if ( firstshort != -INDEX && firstshort != -VECTOR && firstshort != -MINVECTOR ) {
00688                 argsdone++; gcdisone = 1; break;
00689             }
00690             if ( tf[1] != firstvalue ) {
00691                 argsdone++; gcdisone = 1; break;
00692             }
00693             if ( *t == -MINVECTOR ) { firstshort = -VECTOR; }
00694             if ( firstshort == -MINVECTOR ) { firstshort = -VECTOR; }
00695         }
00696         else if ( *tf > -FUNCTION && *tf != -SNUMBER && tf[1] != firstvalue ) {
00697             argsdone++; gcdisone = 1; break;
00698         }
00699         if ( *tf == -SNUMBER && firstvalue != tf[1] ) {
00700 /*
00701             make a new firstvalue which is gcd_(firstvalue,tf[1])
00702 */
00703             if ( firstvalue == 1 || tf[1] == 1 ) { gcdisone = 1; break; }
00704             if ( firstvalue < 0 && tf[1] < 0 ) {
00705                 x1 = -firstvalue; x2 = -tf[1]; sign = -1;
00706             }
00707             else {
00708                 x1 = ABS(firstvalue); x2 = ABS(tf[1]); sign = 1;
00709             }
00710             while ( ( x3 = x1%x2 ) != 0 ) { x1 = x2; x2 = x3; }
00711             firstvalue = ((WORD)x2)*sign;
00712             argsdone++;
00713             if ( firstvalue == 1 ) { gcdisone = 1; break; }
00714         }
00715         NEXTARG(tf);
00716     }
00717     termout = AT.WorkPointer;
00718     AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
00719     if ( AT.WorkPointer > AT.WorkTop ) {
00720         MLOCK(ErrorMessageLock);
00721         MesWork();
00722         MUNLOCK(ErrorMessageLock);
00723         return(-1);
00724     }
00725 /*
00726     #] Do short arguments :
00727     #[ Do trivial GCD :
00728
```

```
00729     Copy head
00730 */
00731     i = t - term; tin = term; tout = termout;
00732     NCOPY(tout,tin,i);
00733     if ( gcdisone || ( firstshort == -SNUMBER && firstvalue == 1 ) ) {
00734         sign = 1;
00735 gcdone:
00736         tin += t[1]; tstop = term + *term;
00737         while ( tin < tstop ) *tout++ = *tin++;
00738         *termout = tout - termout;
00739         if ( sign < 0 ) tout[-1] = -tout[-1];
00740         AT.WorkPointer = tout;
00741         if ( argsdone && Generator(BHEAD termout,level) < 0 ) goto CalledFrom;
00742         AT.WorkPointer = termout;
00743         AT.pWorkPointer = args;
00744         return(0);
00745     }
00746 /*
00747     #] Do trivial GCD :
00748     #[ Do short argument GCD :
00749 */
00750     if ( numargs == 0 ) {   /* basically we are done */
00751 doshort:
00752         sign = 1;
00753         if ( firstshort == 0 ) goto gcdone;
00754         if ( firstshort == -SNUMBER ) {
00755             *tout++ = SNUMBER; *tout++ = 4; *tout++ = firstvalue; *tout++ = 1;
00756             goto gcdone;
00757         }
00758         else if ( firstshort == -SYMBOL ) {
00759             *tout++ = SYMBOL; *tout++ = 4; *tout++ = firstvalue; *tout++ = 1;
00760             goto gcdone;
00761         }
00762         else if ( firstshort == -VECTOR || firstshort == -INDEX ) {
00763             *tout++ = INDEX; *tout++ = 3; *tout++ = firstvalue; goto gcdone;
00764         }
00765         else if ( firstshort == -MINVECTOR ) {
00766             sign = -1;
00767             *tout++ = INDEX; *tout++ = 3; *tout++ = firstvalue; goto gcdone;
00768         }
00769         else if ( firstshort <= -FUNCTION ) {
00770             *tout++ = firstvalue; *tout++ = FUNHEAD; FILLFUN(tout);
00771             goto gcdone;
00772         }
00773         else {
00774 /* INTERNAL_ERROR_EXCL_START */
00775             MLOCK(ErrorMessageLock);
00776             MesPrint("!>Internal error. Illegal short argument in GCDfunction.");
00777             MUNLOCK(ErrorMessageLock);
00778             Terminate(-1);
00779 /* INTERNAL_ERROR_EXCL_STOP */
00780         }
00781     }
00782 /*
00783     #] Do short argument GCD :
00784     #[ Convert short argument :
00785
00786     Now we allocate space for the arguments in general notation.
00787     First the special one if there were short arguments
00788 */
00789     if ( firstshort ) {
00790       switch ( firstshort ) {
00791         case -SNUMBER:
00792             sh[0] = 4; sh[1] = ABS(firstvalue); sh[2] = 1;
00793             if ( firstvalue < 0 ) sh[3] = -3;
00794             else sh[3] = 3;
00795             sh[4] = 0;
00796             break;
00797         case -MINVECTOR:
00798         case -VECTOR:
00799         case -INDEX:
00800             sh[0] = 8; sh[1] = INDEX; sh[2] = 3; sh[3] = firstvalue;
00801             sh[4] = 1; sh[5] = 1;
00802             if ( firstshort == -MINVECTOR ) sh[6] = -3;
00803             else sh[6] = 3;
00804             sh[7] = 0;
00805             break;
00806         case -SYMBOL:
00807             sh[0] = 8; sh[1] = SYMBOL; sh[2] = 4; sh[3] = firstvalue; sh[4] = 1;
00808             sh[5] = 1; sh[6] = 1; sh[7] = 3; sh[8] = 0;
00809             break;
00810         default:
00811             sh[0] = FUNHEAD+4; sh[1] = firstshort; sh[2] = FUNHEAD;
00812             for ( i = 2; i < FUNHEAD; i++ ) sh[i+1] = 0;
00813             sh[FUNHEAD+1] = 1; sh[FUNHEAD+2] = 1; sh[FUNHEAD+3] = 3; sh[FUNHEAD+4] = 0;
00814         break;
00815       }
```

```
00816     }
00817 /*
00818     #] Convert short argument :
00819     #[ Sort arguments :
00820
00821     Now we should sort the arguments in a way that the dollars and the
00822     expressions come last. That way we may never need them.
00823 */
00824     for ( i = 1; i < numargs; i++ ) {
00825         for ( ii = i; ii > 0; ii-- ) {
00826             arg1 = AT.pWorkSpace[args+ii];
00827             arg2 = AT.pWorkSpace[args+ii-1];
00828             if ( *arg1 < 0 ) {
00829                 if ( *arg2 < 0 ) {
00830                     if ( *arg1 == -EXPRESSION ) break;
00831                     if ( *arg2 == -DOLLAREXPRESSION ) break;
00832                     AT.pWorkSpace[args+ii] = arg2;
00833                     AT.pWorkSpace[args+ii-1] = arg1;
00834                 }
00835                 else break;
00836             }
00837             else if ( *arg2 < 0 ) {
00838                 AT.pWorkSpace[args+ii] = arg2;
00839                 AT.pWorkSpace[args+ii-1] = arg1;
00840             }
00841             else {
00842                 if ( *arg1 > *arg2 ) {
00843                     AT.pWorkSpace[args+ii] = arg2;
00844                     AT.pWorkSpace[args+ii-1] = arg1;
00845                 }
00846                 else break;
00847             }
00848         }
00849     }
00850 /*
00851     #] Sort arguments :
00852     #[ There is a single term argument :
00853 */
00854     if ( firstshort ) {
00855         mh = sh; istart = 0;
00856 oneterm:;
00857         for ( i = istart; i < numargs; i++ ) {
00858             arg1 = AT.pWorkSpace[args+i];
00859             if ( *arg1 > 0 ) {
00860                 oldval1 = arg1[*arg1]; arg1[*arg1] = 0;
00861                 m = arg1+ARGHEAD;
00862                 while ( *m ) {
00863                     GCDterms(BHEAD mh,m,mh); m += *m;
00864                     if ( mh[0] == 4 && mh[1] == 1 && mh[2] == 1 && mh[3] == 3 ) {
00865                         gcdisone = 1; sign = 1; arg1[*arg1] = oldval1; goto gcdone;
00866                     }
00867                 }
00868                 arg1[*arg1] = oldval1;
00869             }
00870             else if ( *arg1 == -DOLLAREXPRESSION ) {
00871                 if ( ( d = DolToTerms(BHEAD arg1[1]) ) != 0 ) {
00872                     m = d->where;
00873                     while ( *m ) {
00874                         GCDterms(BHEAD mh,m,mh); m += *m;
00875                         argsdone++;
00876                         if ( mh[0] == 4 && mh[1] == 1 && mh[2] == 1 && mh[3] == 3 ) {
00877                             gcdisone = 1; sign = 1;
00878                             if ( d->factors ) M_free(d->factors,"Dollar factors");
00879                             M_free(d,"Copy of dollar variable"); goto gcdone;
00880                         }
00881                     }
00882                     if ( d->factors ) M_free(d->factors,"Dollar factors");
00883                     M_free(d,"Copy of dollar variable");
00884                 }
00885             }
00886             else {
00887                 mm = CreateExpression(BHEAD arg1[1]);
00888                 m = mm;
00889                 while ( *m ) {
00890                     GCDterms(BHEAD mh,m,mh); m += *m;
00891                     argsdone++;
00892                     if ( mh[0] == 4 && mh[1] == 1 && mh[2] == 1 && mh[3] == 3 ) {
00893                         gcdisone = 1; sign = 1; M_free(mm,"CreateExpression"); goto gcdone;
00894                     }
00895                 }
00896                 M_free(mm,"CreateExpression");
00897             }
00898         }
00899         if ( firstshort ) {
00900             if ( mh[0] == 4 ) {
00901                 firstshort = -SNUMBER; firstvalue = mh[1] * (mh[3]/3);
00902             }
```

```
00903             else if ( mh[1] == SYMBOL ) {
00904                 firstshort = -SYMBOL; firstvalue = mh[3];
00905             }
00906             else if ( mh[1] == INDEX ) {
00907                 firstshort = -INDEX; firstvalue = mh[3];
00908                 if ( mh[6] == -3 ) firstshort = -MINVECTOR;
00909             }
00910             else if ( mh[1] >= FUNCTION ) {
00911                 firstshort = -mh[1]; firstvalue = mh[1];
00912             }
00913             goto doshort;
00914         }
00915         else {
00916 /*
00917             We have a GCD that is only a single term.
00918             Paste it in and combine the coefficients.
00919 */
00920             mh[mh[0]] = 0;
00921             mm = mh;
00922             ii = 0;
00923             goto multiterms;
00924         }
00925     }
00926 /*
00927     Now we have only regular arguments.
00928     But some have not yet been expanded.
00929     Check whether there are proper long arguments and if so if there is
00930     one with just a single term
00931 */
00932     for ( i = 0; i < numargs; i++ ) {
00933         arg1 = AT.pWorkSpace[args+i];
00934         if ( *arg1 > 0 && arg1[ARGHEAD]+ARGHEAD == *arg1 ) {
00935 /*
00936             We have an argument with a single term
00937 */
00938             if ( i != 0 ) {
00939                 arg2 = AT.pWorkSpace[args];
00940                 AT.pWorkSpace[args] = arg1;
00941                 AT.pWorkSpace[args+1] = arg2;
00942             }
00943             m = mh = AT.WorkPointer;
00944             mm = arg1+ARGHEAD; i = *mm;
00945             NCOPY(m,mm,i);
00946             AT.WorkPointer = m;
00947             istart = 1;
00948             argsdone++;
00949             goto oneterm;
00950         }
00951     }
00952 /*
00953     #] There is a single term argument :
00954     #[ Expand $ and expr :
00955
00956     We have: 1: regular multiterm arguments
00957              2: dollars
00958              3: expressions.
00959     The sum of them is numargs. Their addresses are in args. The problem is
00960     that expansion will lead to allocations that we have to return and all
00961     these allocations are different in nature.
00962 */
00963     action = 1;
00964     abuf = (ARGBUFFER *)Malloc1(numargs*sizeof(ARGBUFFER),"argbuffer");
00965     for ( i = 0; i < numargs; i++ ) {
00966         arg1 = AT.pWorkSpace[args+i];
00967         if ( *arg1 > 0 ) {
00968             m = (WORD *)Malloc1(*arg1*sizeof(WORD),"argbuffer type 0");
00969             abuf[i].buffer = m;
00970             abuf[i].type = 0;
00971             mm = arg1+ARGHEAD;
00972             j = *arg1-ARGHEAD;
00973             abuf[i].size = j;
00974             if ( j ) argsdone++;
00975             NCOPY(m,mm,j);
00976             *m = 0;
00977         }
00978         else if ( *arg1 == -DOLLAREXPRESSION ) {
00979             d = DolToTerms(BHEAD arg1[1]);
00980             abuf[i].buffer = d->where;
00981             abuf[i].type = 1;
00982             abuf[i].dollar = d;
00983             m = abuf[i].buffer;
00984             if ( *m ) argsdone++;
00985             while ( *m ) m+= *m;
00986             abuf[i].size = m-abuf[i].buffer;
00987         }
00988         else if ( *arg1 == -EXPRESSION ) {
00989             abuf[i].buffer = CreateExpression(BHEAD arg1[1]);
```

```
00990             abuf[i].type = 2;
00991             m = abuf[i].buffer;
00992             if ( *m ) argsdone++;
00993             while ( *m ) m+= *m;
00994             abuf[i].size = m-abuf[i].buffer;
00995         }
00996         else {
00997 /* INTERNAL_ERROR_EXCL_START */
00998             MLOCK(ErrorMessageLock);
00999             MesPrint("!>What argument is this?");
01000             MUNLOCK(ErrorMessageLock);
01001             goto CalledFrom;
01002 /* INTERNAL_ERROR_EXCL_STOP */
01003         }
01004     }
01005     for ( i = 0; i < numargs; i++ ) {
01006         arg1 = abuf[i].buffer;
01007         if ( *arg1 == 0 ) {}
01008         else if ( arg1[*arg1] == 0 ) {
01009 /*
01010             After expansion there is an argument with a single term
01011 */
01012             ab = abuf[i]; abuf[i] = abuf[0]; abuf[0] = ab;
01013             mh = abuf[0].buffer;
01014             for ( j = 1; j < numargs; j++ ) {
01015                 m = abuf[j].buffer;
01016                 while ( *m ) {
01017                     GCDterms(BHEAD mh,m,mh); m += *m;
01018                     argsdone++;
01019                     if ( mh[0] == 4 && mh[1] == 1 && mh[2] == 1 && mh[3] == 3 ) {
01020                         gcdisone = 1; sign = 1; break;
01021                     }
01022                 }
01023                 if ( *m ) break;
01024             }
01025             mm = mh + *mh; if ( mm[-1] < 0 ) { sign = -1; mm[-1] = -mm[-1]; }
01026             mstop = mm - mm[-1]; m = mh+1; mlength = mm[-1];
01027             while ( tin < t ) *tout++ = *tin++;
01028             while ( m < mstop ) *tout++ = *m++;
01029             tin += tin[1];
01030             while ( tin < tstop ) *tout++ = *tin++;
01031             tlength = REDLENG(tlength);
01032             mlength = REDLENG(mlength);
01033             if ( MulRat(BHEAD (UWORD *)tstop,tlength,(UWORD *)mstop,mlength,
01034                     (UWORD *)tout,&newlength) < 0 ) goto CalledFrom;
01035             mlength = INCLENG(newlength);
01036             tout += ABS(mlength);
01037             tout[-1] = mlength*sign;
01038             *termout = tout - termout;
01039             AT.WorkPointer = tout;
01040             if ( argsdone && Generator(BHEAD termout,level) < 0 ) goto CalledFrom;
01041             goto cleanup;
01042         }
01043     }
01044 /*
01045     There are only arguments with more than one term.
01046     We order them by size to make the computations as easy as possible.
01047 */
01048     for ( i = 1; i < numargs; i++ ) {
01049         for ( ii = i; ii > 0; ii-- ) {
01050             if ( abuf[ii-1].size <= abuf[ii].size ) break;
01051             ab = abuf[ii-1]; abuf[ii-1] = abuf[ii]; abuf[ii] = ab;
01052         }
01053     }
01054 /*
01055     #] Expand $ and expr :
01056     #[ Multiterm subexpressions :
01057 */
01058     ii = 0;
01059     gcdout = abuf[ii].buffer;
01060     for ( i = 0; i < numargs; i++ ) {
01061         if ( abuf[i].buffer[0] ) { gcdout = abuf[i].buffer; ii = i; i++; argsdone++; break; }
01062     }
01063     for ( ; i < numargs; i++ ) {
01064       if ( abuf[i].buffer[0] ) {
01065         g = GCDfunction3(BHEAD gcdout,abuf[i].buffer);
01066         argsdone++;
01067         if ( gcdout != abuf[ii].buffer ) M_free(gcdout,"gcdout");
01068         gcdout = g;
01069         if ( gcdout[*gcdout] == 0 && gcdout[0] == 4 && gcdout[1] == 1
01070         && gcdout[2] == 1 && gcdout[3] == 3 ) break;
01071       }
01072     }
01073     mm = gcdout;
01074 multiterms:;
01075     tlength = REDLENG(tlength);
01076     while ( *mm ) {
```

```
01077         tin = term; tout = termout; while ( tin < t ) *tout++ = *tin++;
01078         tin += t[1];
01079         mnext = mm + *mm; mlength = mnext[-1]; mstop = mnext - ABS(mlength);
01080         mm++;
01081         while ( mm < mstop ) *tout++ = *mm++;
01082         while ( tin < tstop ) *tout++ = *tin++;
01083         mlength = REDLENG(mlength);
01084         if ( MulRat(BHEAD (UWORD *)tstop,tlength,(UWORD *)mm,mlength,
01085                 (UWORD *)tout,&newlength) < 0 ) goto CalledFrom;
01086         mlength = INCLENG(newlength);
01087         tout += ABS(mlength);
01088         tout[-1] = mlength;
01089         *termout = tout - termout;
01090         AT.WorkPointer = tout;
01091         if ( argsdone && Generator(BHEAD termout,level) < 0 ) goto CalledFrom;
01092         mm = mnext; /* next term */
01093     }
01094     if ( action && ( gcdout != abuf[ii].buffer ) ) M_free(gcdout,"gcdout");
01095 /*
01096     #] Multiterm subexpressions :
01097     #[ Cleanup :
01098 */
01099 cleanup:;
01100     if ( action ) {
01101         for ( i = 0; i < numargs; i++ ) {
01102             if ( abuf[i].type == 0 ) { M_free(abuf[i].buffer,"argbuffer type 0"); }
01103             else if ( abuf[i].type == 1 ) {
01104                 d = abuf[i].dollar;
01105                 if ( d->factors ) M_free(d->factors,"Dollar factors");
01106                 M_free(d,"Copy of dollar variable");
01107             }
01108             else if ( abuf[i].type == 2 ) { M_free(abuf[i].buffer,"CreateExpression"); }
01109         }
01110         M_free(abuf,"argbuffer");
01111     }
01112 /*
01113     #] Cleanup :
01114 */
01115     AT.pWorkPointer = args;
01116     AT.WorkPointer = termout;
01117     return(0);
01118
01119 CalledFrom:
01120     MLOCK(ErrorMessageLock);
01121     MesCall("GCDfunction");
01122     MUNLOCK(ErrorMessageLock);
01123     SETERROR(-1)
01124     return(-1);
01125 }
01126
01127 /*
01128         #] GCDfunction :
01129         #[ GCDfunction3 :
01130
01131     Finds the GCD of the two arguments which are buffers with terms.
01132     In principle the first buffer can have only one term.
01133
01134     If both buffers have more than one term, we need to replace all
01135     non-symbolic objects by generated symbols and substitute that back
01136     afterwards. The rest we leave to the powerful routines.
01137     Philosophical problem: What do we do with GCD_(x/z+y,x+y*z) ?
01138
01139     Method:
01140     If we have only negative powers of z and no positive powers we let
01141     the EXTRASYMBOLS do their job. When mixed, multiply the arguments with
01142     the negative powers with enough powers of z to eliminate the negative powers.
01143     The DENOMINATOR function is always eliminated with the mechanism as we
01144     cannot tell whether there are positive powers of its contents.
01145 */
01146
01147 WORD *GCDfunction3(PHEAD WORD *in1, WORD *in2)
01148 {
01149     GETBIDENTITY
01150     WORD oldsorttype = AR.SortType, *ow = AT.WorkPointer;;
01151     WORD *t, *tt, *gcdout, *term1, *term2, *confree1, *confree2, *gcdout1, *proper1, *proper2;
01152     int i, actionflag1, actionflag2;
01153     WORD startebuf = cbuf[AT.ebufnum].numrhs;
01154     WORD tryterm1, tryterm2;
01155     if ( in2[*in2] == 0 ) { t = in1; in1 = in2; in2 = t; }
01156     if ( in1[*in1] == 0 ) { /* First input with only one term */
01157         gcdout = (WORD *)Malloc1((*in1+1)*sizeof(WORD),"gcdout");
01158         i = *in1; t = gcdout; tt = in1; NCOPY(t,tt,i); *t = 0;
01159         t = in2;
01160         while ( *t ) {
01161             GCDterms(BHEAD gcdout,t,gcdout);
01162             if ( gcdout[0] == 4 && gcdout[1] == 1
01163              && gcdout[2] == 1 && gcdout[3] == 3 ) break;
```

```
01164             t += *t;
01165         }
01166         AT.WorkPointer = ow;
01167         return(gcdout);
01168     }
01169 /*
01170     We need to take out the content from the two expressions
01171     and determine their GCD. This plays with the negative powers!
01172 */
01173     AR.SortType = SORTHIGHFIRST;
01174     term1 = TermMalloc("GCDfunction3-a");
01175     term2 = TermMalloc("GCDfunction3-b");
01176     confree1 = TakeContent(BHEAD in1,term1);
01177     tryterm1 = AN.tryterm; AN.tryterm = 0;
01178     confree2 = TakeContent(BHEAD in2,term2);
01179     tryterm2 = AN.tryterm; AN.tryterm = 0;
01180 /*
01181     confree1 = TakeSymbolContent(BHEAD in1,term1);
01182     confree2 = TakeSymbolContent(BHEAD in2,term2);
01183 */
01184     GCDterms(BHEAD term1,term2,term1);
01185     TermFree(term2,"GCDfunction3-b");
01186 /*
01187     Now we have to replace all non-symbols and symbols to a negative power
01188     by extra symbols.
01189 */
01190     if ( ( proper1 = PutExtraSymbols(BHEAD confree1,startebuf,&actionflag1) ) == 0 ) goto CalledFrom;
01191     if ( confree1 != in1 ) {
01192         if ( tryterm1 ) { TermFree(confree1,"TakeContent"); }
01193         else { M_free(confree1,"TakeContent"); }
01194     }
01195 /*
01196     TermFree(confree1,"TakeSymbolContent");
01197 */
01198     if ( ( proper2 = PutExtraSymbols(BHEAD confree2,startebuf,&actionflag2) ) == 0 ) goto CalledFrom;
01199     if ( confree2 != in2 ) {
01200         if ( tryterm2 ) { TermFree(confree2,"TakeContent"); }
01201         else { M_free(confree2,"TakeContent"); }
01202     }
01203 /*
01204     TermFree(confree2,"TakeSymbolContent");
01205 */
01206 /*
01207     And now the real work:
01208 */
01209     gcdout1 = poly_gcd(BHEAD proper1,proper2,0);
01210     M_free(proper1,"PutExtraSymbols");
01211     M_free(proper2,"PutExtraSymbols");
01212
01213     AR.SortType = oldsorttype;
01214     if ( actionflag1 || actionflag2 ) {
01215         if ( ( gcdout = TakeExtraSymbols(BHEAD gcdout1,startebuf) ) == 0 ) goto CalledFrom;
01216         M_free(gcdout1,"gcdout");
01217     }
01218     else {
01219         gcdout = gcdout1;
01220     }
01221
01222     cbuf[AT.ebufnum].numrhs = startebuf;
01223 /*
01224     Now multiply gcdout by term1
01225 */
01226     if ( term1[0] != 4 || term1[3] != 3 || term1[1] != 1 || term1[2] != 1 ) {
01227         AN.tryterm = -1;
01228         if ( ( gcdout1 = MultiplyWithTerm(BHEAD gcdout,term1,2) ) == 0 ) goto CalledFrom;
01229         AN.tryterm = 0;
01230         M_free(gcdout,"gcdout");
01231         gcdout = gcdout1;
01232     }
01233     TermFree(term1,"GCDfunction3-a");
01234     AT.WorkPointer = ow;
01235     return(gcdout);
01236 CalledFrom:
01237     AN.tryterm = 0;
01238     MLOCK(ErrorMessageLock);
01239     MesCall("GCDfunction3");
01240     MUNLOCK(ErrorMessageLock);
01241     return(0);
01242 }
01243
01244 /*
01245         #] GCDfunction3 :
01246         #[ PutExtraSymbols :
01247 */
01248
01249 WORD *PutExtraSymbols(PHEAD WORD *in,WORD startebuf,int *actionflag)
01250 {
```

```
01251     WORD *termout = AT.WorkPointer;
01252     int action;
01253     *actionflag = 0;
01254     NewSort(BHEAD0);
01255     while ( *in ) {
01256         if ( ( action = LocalConvertToPoly(BHEAD in,termout,startebuf,0) ) < 0 ) {
01257             LowerSortLevel();
01258             goto CalledFrom;
01259         }
01260         if ( action > 0 ) *actionflag = 1;
01261         StoreTerm(BHEAD termout);
01262         in += *in;
01263     }
01264     if ( EndSort(BHEAD (WORD *)((void *)(&termout)),2) < 0 ) goto CalledFrom;
01265     return(termout);
01266 CalledFrom:
01267     MLOCK(ErrorMessageLock);
01268     MesCall("PutExtraSymbols");
01269     MUNLOCK(ErrorMessageLock);
01270     return(0);
01271 }
01272
01273 /*
01274         #] PutExtraSymbols :
01275         #[ TakeExtraSymbols :
01276 */
01277
01278 WORD *TakeExtraSymbols(PHEAD WORD *in,WORD startebuf)
01279 {
01280     CBUF *C = cbuf+AC.cbufnum;
01281     CBUF *CC = cbuf+AT.ebufnum;
01282     WORD *oldworkpointer = AT.WorkPointer, *termout;
01283
01284     termout = AT.WorkPointer;
01285     NewSort(BHEAD0);
01286     while ( *in ) {
01287         if ( ConvertFromPoly(BHEAD
      in,termout,numxsymbol,CC->numrhs-startebuf+numxsymbol,startebuf-numxsymbol,1) <= 0 ) {
01288             LowerSortLevel();
01289             goto CalledFrom;
01290         }
01291         in += *in;
01292         AT.WorkPointer = termout + *termout;
01293 /*
01294         ConvertFromPoly leaves terms with subexpressions. Hence:
01295 */
01296         if ( Generator(BHEAD termout,C->numlhs) ) {
01297             LowerSortLevel();
01298             goto CalledFrom;
01299         }
01300     }
01301     AT.WorkPointer = oldworkpointer;
01302     if ( EndSort(BHEAD (WORD *)((void *)(&termout)),2) < 0 ) goto CalledFrom;
01303     return(termout);
01304
01305 CalledFrom:
01306     MLOCK(ErrorMessageLock);
01307     MesCall("TakeExtraSymbols");
01308     MUNLOCK(ErrorMessageLock);
01309     return(0);
01310 }
01311
01312 /*
01313         #] TakeExtraSymbols :
01314         #[ MultiplyWithTerm :
01315 */
01316
01317 WORD *MultiplyWithTerm(PHEAD WORD *in, WORD *term, WORD par)
01318 {
01319     WORD *termout, *t, *tt, *tstop, *ttstop;
01320     WORD length, length1, length2;
01321     WORD oldsorttype = AR.SortType;
01322     COMPARE oldcompareroutine = (COMPARE)(AR.CompareRoutine);
01323     AR.CompareRoutine = (COMPAREDUMMY)(&CompareSymbols);
01324
01325     if ( par == 0 || par == 2 ) AR.SortType = SORTHIGHFIRST;
01326     else                        AR.SortType = SORTLOWFIRST;
01327     termout = AT.WorkPointer;
01328     NewSort(BHEAD0);
01329     while ( *in ) {
01330         tt = termout + 1;
01331         tstop = in + *in; tstop -= ABS(tstop[-1]); t = in + 1;
01332         while ( t < tstop ) *tt++ = *t++;
01333         ttstop = term + *term; ttstop -= ABS(ttstop[-1]); t = term + 1;
01334         while ( t < ttstop ) *tt++ = *t++;
01335         length1 = REDLENG(in[*in-1]); length2 = REDLENG(term[*term-1]);
01336         if ( MulRat(BHEAD (UWORD *)tstop,length1,
```

```
01337                 (UWORD *)ttstop,length2,(UWORD *)tt,&length) ) goto CalledFrom;
01338         length = INCLENG(length);
01339         tt += ABS(length); tt[-1] = length;
01340         *termout = tt - termout;
01341         // in or term can contain non-symbols: we call this in TakeContent
01342         Normalize(BHEAD termout);
01343         StoreTerm(BHEAD termout);
01344         in += *in;
01345     }
01346     if ( par == 2 ) {
01347 /*      if ( AN.tryterm == 0 ) AN.tryterm = 1; */
01348         AN.tryterm = 0; /* For now */
01349         if ( EndSort(BHEAD (WORD *)((void *)(&termout)),2) < 0 ) goto CalledFrom;
01350     }
01351     else {
01352         if ( EndSort(BHEAD termout,1) < 0 ) goto CalledFrom;
01353     }
01354
01355     AR.CompareRoutine = (COMPAREDUMMY)oldcompareroutine;
01356
01357     AR.SortType = oldsorttype;
01358     return(termout);
01359
01360 CalledFrom:
01361     MLOCK(ErrorMessageLock);
01362     MesCall("MultiplyWithTerm");
01363     MUNLOCK(ErrorMessageLock);
01364     return(0);
01365 }
01366
01367 /*
01368         #] MultiplyWithTerm :
01369         #[ TakeContent :
01370 */
01382 WORD *TakeContent(PHEAD WORD *in, WORD *term)
01383 {
01384     GETBIDENTITY
01385     WORD *t, *tstop, *tcom, *tout, *tstore, *r, *rstop, *m, *mm, *w, *ww, *wterm;
01386     WORD *tnext, *tt, *tterm, code[2];
01387     WORD *inp, a, *den;
01388     int i, j, k, action = 0, sign;
01389     UWORD *GCDbuffer, *GCDbuffer2, *LCMbuffer, *LCMbuffer2, *ap;
01390     WORD GCDlen, GCDlen2, LCMlen, LCMlen2, length, redlength, len1, len2;
01391     tout = tstore = term+1;
01392 /*
01393     #[ INDEX :
01394 */
01395     t = in;
01396     tnext = t + *t;
01397     tstop = tnext-ABS(tnext[-1]);
01398     t++;
01399     while ( t < tstop ) {
01400         if ( *t == INDEX ) {
01401             i = t[1]; NCOPY(tout,t,i); break;
01402         }
01403         else t += t[1];
01404     }
01405     if ( tout > tstore ) { /* There are indices in the first term */
01406         t = tnext;
01407         while ( *t ) {
01408             tnext = t + *t;
01409             rstop = tnext - ABS(tnext[-1]);
01410             r = t+1;
01411             if ( r == rstop ) goto noindices;
01412             while ( r < rstop ) {
01413                 if ( *r != INDEX ) { r += r[1]; continue; }
01414                 m = tstore+2;
01415                 while ( m < tout ) {
01416                     for ( i = 2; i < r[1]; i++ ) {
01417                         if ( *m == r[i] ) break;
01418                     }
01419                     if ( i == r[1] ) { // index at m was not found, scratch from list
01420                         mm = m+1;
01421                         while ( mm < tout ) { mm[-1] = mm[0]; mm++; }
01422                         tout--; tstore[1]--; m--;
01423                     }
01424                     m++;
01425                 }
01426                 r += r[1];
01427             }
01428             if ( tout <= tstore+2 ) {
01429                 tout = tstore; break;
01430             }
01431             t = tnext;
01432         }
01433         if ( tout > tstore+2 ) { /* Now we have to take out what is in tstore */
01434             t = in; w = in;
```

```
01435             while ( *t ) {
01436                 wterm = w;
01437                 tnext = t + *t; t++; w++;
01438                 while ( *t != INDEX ) { i = t[1]; NCOPY(w,t,i); }
01439                 tt = t + t[1]; t += 2; r = tstore+2; ww = w; *w++ = INDEX; w++;
01440                 while ( r < tout && t < tt ) {
01441                     if ( *r > *t ) { *w++ = *t++; }
01442                     else if ( *r == *t ) { r++; t++; }
01443                     else goto CalledFrom;
01444                 }
01445                 if ( r < tout ) goto CalledFrom;
01446                 while (  t < tt ) *w++ = *t++;
01447                 ww[1] = w - ww;
01448                 if ( ww[1] == 2 ) w = ww;
01449                 while ( t < tnext ) *w++ = *t++;
01450                 *wterm = w - wterm;
01451             }
01452             *w = 0;
01453         }
01454 noindices:
01455         tstore = tout;
01456     }
01457 /*
01458     #] INDEX :
01459     #[ VECTOR/DELTA :
01460 */
01461     code[0] = VECTOR; code[1] = DELTA;
01462     for ( k = 0; k < 2; k++ ) {
01463         t = in;
01464         tnext = t + *t;
01465         tstop = tnext-ABS(tnext[-1]);
01466         t++;
01467         while ( t < tstop ) {
01468             if ( *t == code[k] ) {
01469                 i = t[1]; NCOPY(tout,t,i); break;
01470             }
01471             else t += t[1];
01472         }
01473         if ( tout > tstore ) { /* There are vectors in the first term */
01474             t = tnext;
01475             while ( *t ) {
01476                 tnext = t + *t;
01477                 rstop = tnext - ABS(tnext[-1]);
01478                 r = t+1;
01479                 if ( r == rstop ) { tstore = tout; goto novectors; }
01480                 while ( r < rstop ) {
01481                     if ( *r != code[k] ) { r += r[1]; continue; }
01482                     m = tstore+2;
01483                     while ( m < tout ) {
01484                         for ( i = 2; i < r[1]; i += 2 ) {
01485                             if ( *m == r[i] && m[1] == r[i+1] ) break;
01486                         }
01487                         if ( i == r[1] ) { // object was not found, scratch from list
01488                             mm = m+2;
01489                             while ( mm < tout ) { mm[-2] = mm[0]; mm[-1] = mm[1]; mm += 2; }
01490                             tout -= 2; tstore[1] -= 2; m -= 2;
01491                         }
01492                         m += 2;
01493                     }
01494                     r += r[1];
01495                 }
01496                 if ( tout <= tstore+2 ) {
01497                     tout = tstore; break;
01498                 }
01499                 t = tnext;
01500             }
01501             if ( tout > tstore+2 ) { /* Now we have to take out what is in tstore */
01502                 t = in; w = in;
01503                 while ( *t ) {
01504                     wterm = w;
01505                     tnext = t + *t; t++; w++;
01506                     while ( *t != code[k] ) { i = t[1]; NCOPY(w,t,i); }
01507                     tt = t + t[1]; t += 2; r = tstore+2; ww = w; *w++ = code[k]; w++;
01508                     while ( r < tout && t < tt ) {
01509                         if ( ( *r > *t ) || ( *r == *t && r[1] > t[1] ) )
01510                             { *w++ = *t++; *w++ = *t++; }
01511                         else if ( *r == *t && r[1] == t[1] ) { r += 2; t += 2; }
01512                         else goto CalledFrom;
01513                     }
01514                     if ( r < tout ) goto CalledFrom;
01515                     while (  t < tt ) *w++ = *t++;
01516                     ww[1] = w - ww;
01517                     if ( ww[1] == 2 ) w = ww;
01518                     while ( t < tnext ) *w++ = *t++;
01519                     *wterm = w - wterm;
01520                 }
01521                 *w = 0;
```

```
01522             }
01523             tstore = tout;
01524         }
01525     }
01526 novectors:;
01527 /*
01528     #] VECTOR/DELTA :
01529     #[ FUNCTIONS :
01530 */
01531     t = in;
01532     tnext = t + *t;
01533     tstop = tnext-ABS(tnext[-1]);
01534     t++;
01535     tcom = 0;
01536     while ( t < tstop ) {
01537         if ( *t >= FUNCTION ) {
01538             if ( functions[*t-FUNCTION].commute ) {
01539                 if ( tcom == 0 ) { tcom = tout; }
01540                 else {
01541                     for ( i = 0; i < t[1]; i++ ) {
01542                         if ( t[i] != tcom[i] ) {
01543                             MLOCK(ErrorMessageLock);
01544                             MesPrint("GCD or factorization of more than one noncommuting object not
    allowed");
01545                             MUNLOCK(ErrorMessageLock);
01546                             goto CalledFrom;
01547                         }
01548                     }
01549                 }
01550             }
01551             i = t[1]; NCOPY(tout,t,i);
01552         }
01553         else t += t[1];
01554     }
01555     if ( tout > tstore ) { /* There are functions in the first term */
01556         t = tnext;
01557         while ( *t ) {
01558             tnext = t + *t; tstop = tnext - ABS(tnext[-1]); t++;
01559             if ( t == tstop ) goto nofunctions;
01560             r = tstore;
01561             while ( r < tout ) {
01562                 tt = t;
01563                 while ( tt < tstop ) {
01564                     for ( i = 0; i < r[1]; i++ ) {
01565                         if ( r[i] != tt[i] ) break;
01566                     }
01567                     if ( i == r[1] ) { r += r[1]; goto nextr1; }
01568                     tt += tt[1];
01569                 }
01570 /*
01571                 Not encountered in this term. Scratch from list
01572 */
01573                 m = r; mm = r + r[1];
01574                 while ( mm < tout ) *m++ = *mm++;
01575                 tout = m;
01576 nextr1:;
01577             }
01578             if ( tout <= tstore ) break;
01579             t = tnext;
01580         }
01581     }
01582     if ( tout > tstore ) {
01583 /*
01584         Now we have one or more functions left that are common in all terms.
01585         Take them out. We do this one by one.
01586 */
01587         r = tstore;
01588         while ( r < tout ) {
01589             t = in; ww = w = in;
01590             while ( *t ) {
01591                 ww = w; // store the location of the term size
01592                 w += 1; // term data copied here, following the size
01593                 tnext = t + *t;
01594                 t++;
01595                 for(;;) {
01596                     for ( i = 0; i < r[1]; i++ ) { // search for the current tstore object
01597                         if ( t[i] != r[i] ) {
01598                             j = t[1]; NCOPY(w,t,j);
01599                             break;
01600                         }
01601                     }
01602                     if ( i == r[1] ) { // we found the current tstore object at t
01603                         t += t[1]; // skip over it
01604                         while ( t < tnext ) *w++ = *t++; // copy the rest of the term
01605                         *ww = w - ww; // update the size
01606                         break;
01607                     }
```

```
01608                 }
01609             }
01610             r += r[1];
01611             *w = 0;
01612         }
01613 nofunctions:
01614         tstore = tout;
01615     }
01616 /*
01617     #] FUNCTIONS :
01618     #[ SYMBOL :
01619
01620     We make a list of symbols and their minimal powers.
01621     This includes negative powers. In the end we have to multiply by the
01622     inverse of this list. That takes out all negative powers and leaves
01623     things ready for further processing.
01624 */
01625     tterm = AT.WorkPointer; tt = tterm+1;
01626     tout[0] = SYMBOL; tout[1] = 2;
01627     t = in;
01628     tnext = t + *t; tstop = tnext - ABS(tnext[-1]); t++;
01629     while ( t < tstop ) {
01630         if ( *t == SYMBOL ) {
01631             for ( i = 0; i < t[1]; i++ ) tout[i] = t[i];
01632             break;
01633         }
01634         t += t[1];
01635     }
01636     t = tnext;
01637     while ( *t ) {
01638         tnext = t + *t; tstop = tnext - ABS(tnext[-1]); t++;
01639         if ( t == tstop ) {
01640             tout[1] = 2;
01641             break;
01642         }
01643         else {
01644             while ( t < tstop ) {
01645                 if ( *t == SYMBOL ) {
01646                     MergeSymbolLists(BHEAD tout,t,-1);
01647                     break;
01648                 }
01649                 t += t[1];
01650             }
01651             t = tnext;
01652         }
01653     }
01654     if ( tout[1] > 2 ) {
01655         t = tout;
01656         tt[0] = t[0]; tt[1] = t[1];
01657         for ( i = 2; i < t[1]; i += 2 ) {
01658             tt[i] = t[i]; tt[i+1] = -t[i+1];
01659         }
01660         tt += tt[1];
01661         tout += tout[1];
01662         action++;
01663     }
01664 /*
01665     #] SYMBOL :
01666     #[ DOTPRODUCT :
01667
01668     We make a list of dotproducts and their minimal powers.
01669     This includes negative powers. In the end we have to multiply by the
01670     inverse of this list. That takes out all negative powers and leaves
01671     things ready for further processing.
01672 */
01673     tout[0] = DOTPRODUCT; tout[1] = 2;
01674     t = in;
01675     tnext = t + *t; tstop = tnext - ABS(tnext[-1]); t++;
01676     while ( t < tstop ) {
01677         if ( *t == DOTPRODUCT ) {
01678             for ( i = 0; i < t[1]; i++ ) tout[i] = t[i];
01679             break;
01680         }
01681         t += t[1];
01682     }
01683     t = tnext;
01684     while ( *t ) {
01685         tnext = t + *t; tstop = tnext - ABS(tnext[-1]); t++;
01686         if ( t == tstop ) {
01687             tout[1] = 2;
01688             break;
01689         }
01690         while ( t < tstop ) {
01691             if ( *t == DOTPRODUCT ) {
01692                 MergeDotproductLists(BHEAD tout,t,-1);
01693                 break;
01694             }
```

```
01695             t += t[1];
01696         }
01697         t = tnext;
01698     }
01699     if ( tout[1] > 2 ) {
01700         t = tout;
01701         tt[0] = t[0]; tt[1] = t[1];
01702         for ( i = 2; i < t[1]; i += 3 ) {
01703             tt[i] = t[i]; tt[i+1] = t[i+1]; tt[i+2] = -t[i+2];
01704         }
01705         tt += tt[1];
01706         tout += tout[1];
01707         action++;
01708     }
01709 /*
01710     #] DOTPRODUCT :
01711     #[ Coefficient :
01712
01713     Now we have to collect the GCD of the numerators
01714     and the LCM of the denominators.
01715 */
01716     AT.WorkPointer = tt;
01717     if ( AN.cmod != 0 ) {
01718       WORD x, ix, ip;
01719       t = in; tnext = t + *t; tstop = tnext - ABS(tnext[-1]);
01720       x = tstop[0];
01721       if ( tnext[-1] < 0 ) x += AC.cmod[0];
01722       if ( GetModInverses(x,(WORD)(AN.cmod[0]),&ix,&ip) ) goto CalledFrom;
01723       *tout++ = x; *tout++ = 1; *tout++ = 3;
01724       *tt++ = ix; *tt++ = 1; *tt++ = 3;
01725     }
01726     else {
01727       GCDbuffer  = NumberMalloc("MakeInteger");
01728       GCDbuffer2 = NumberMalloc("MakeInteger");
01729       LCMbuffer  = NumberMalloc("MakeInteger");
01730       LCMbuffer2 = NumberMalloc("MakeInteger");
01731       t = in;
01732       tnext = t + *t; length = tnext[-1];
01733       if ( length < 0 ) { sign = -1; length = -length; }
01734       else              { sign = 1; }
01735       tstop = tnext - length;
01736       redlength = (length-1)/2;
01737       for ( i = 0; i < redlength; i++ ) {
01738         GCDbuffer[i] = (UWORD)(tstop[i]);
01739         LCMbuffer[i] = (UWORD)(tstop[redlength+i]);
01740       }
01741       GCDlen = LCMlen = redlength;
01742       while ( GCDbuffer[GCDlen-1] == 0 ) GCDlen--;
01743       while ( LCMbuffer[LCMlen-1] == 0 ) LCMlen--;
01744       t = tnext;
01745       while ( *t ) {
01746         tnext = t + *t; length = ABS(tnext[-1]);
01747         tstop = tnext - length; redlength = (length-1)/2;
01748         len1 = len2 = redlength;
01749         den = tstop + redlength;
01750         while ( tstop[len1-1] == 0 ) len1--;
01751         while ( den[len2-1] == 0 ) len2--;
01752         if ( GCDlen == 1 && GCDbuffer[0] == 1 ) {}
01753         else {
01754             GcdLong(BHEAD (UWORD *)tstop,len1,GCDbuffer,GCDlen,GCDbuffer2,&GCDlen2);
01755             ap = GCDbuffer; GCDbuffer = GCDbuffer2; GCDbuffer2 = ap;
01756             a = GCDlen; GCDlen = GCDlen2; GCDlen2 = a;
01757         }
01758         if ( len2 == 1 && den[0] == 1 ) {}
01759         else {
01760             GcdLong(BHEAD LCMbuffer,LCMlen,(UWORD *)den,len2,LCMbuffer2,&LCMlen2);
01761             DivLong((UWORD *)den,len2,LCMbuffer2,LCMlen2,
01762                 GCDbuffer2,&GCDlen2,(UWORD *)AT.WorkPointer,&a);
01763             MulLong(LCMbuffer,LCMlen,GCDbuffer2,GCDlen2,LCMbuffer2,&LCMlen2);
01764             ap = LCMbuffer; LCMbuffer = LCMbuffer2; LCMbuffer2 = ap;
01765             a = LCMlen; LCMlen = LCMlen2; LCMlen2 = a;
01766         }
01767         t = tnext;
01768       }
01769       if ( GCDlen != 1 || GCDbuffer[0] != 1 || LCMlen != 1 || LCMbuffer[0] != 1 ) {
01770         redlength = GCDlen; if ( LCMlen > GCDlen ) redlength = LCMlen;
01771         for ( i = 0; i < GCDlen; i++ ) *tout++ = (WORD)(GCDbuffer[i]);
01772         for ( ; i < redlength; i++ ) *tout++ = 0;
01773         for ( i = 0; i < LCMlen; i++ ) *tout++ = (WORD)(LCMbuffer[i]);
01774         for ( ; i < redlength; i++ ) *tout++ = 0;
01775         *tout++ = (2*redlength+1)*sign;
01776         for ( i = 0; i < LCMlen; i++ ) *tt++ = (WORD)(LCMbuffer[i]);
01777         for ( ; i < redlength; i++ ) *tt++ = 0;
01778         for ( i = 0; i < GCDlen; i++ ) *tt++ = (WORD)(GCDbuffer[i]);
01779         for ( ; i < redlength; i++ ) *tt++ = 0;
01780         *tt++ = (2*redlength+1)*sign;
01781         action++;
```

```
01782       }
01783       else {
01784         *tout++ = 1; *tout++ = 1; *tout++ = 3*sign;
01785         *tt++ = 1; *tt++ = 1; *tt++ = 3*sign;
01786         if ( sign != 1 ) action++;
01787       }
01788       *tout = 0;
01789       NumberFree(LCMbuffer2,"MakeInteger");
01790       NumberFree(LCMbuffer ,"MakeInteger");
01791       NumberFree(GCDbuffer2,"MakeInteger");
01792       NumberFree(GCDbuffer ,"MakeInteger");
01793     }
01794 /*
01795     #] Coefficient :
01796     #[ Multiply by the inverse content :
01797 */
01798     if ( action ) {
01799         *tterm = tt - tterm;
01800         AT.WorkPointer = tt;
01801         inp = MultiplyWithTerm(BHEAD in,tterm,2);
01802         AT.WorkPointer = tterm;
01803         in = inp;
01804     }
01805 /*
01806     #] Multiply by the inverse content :
01807 */
01808     *term = tout - term;
01809     AT.WorkPointer = tterm;
01810     return(in);
01811 CalledFrom:
01812     MLOCK(ErrorMessageLock);
01813     MesCall("TakeContent");
01814     MUNLOCK(ErrorMessageLock);
01815     return(0);
01816 }
01817
01818 /*
01819         #] TakeContent :
01820         #[ MergeSymbolLists :
01821
01822         Merges the extra list into the old.
01823         If par == -1 we take minimum powers
01824         If par ==  1 we take maximum powers
01825         If par ==  0 we take minimum of the absolute value of the powers
01826                      if one is positive and the other negative we get zero.
01827         We assume that the symbols are in order in both lists
01828 */
01829
01830 int MergeSymbolLists(PHEAD WORD *old, WORD *extra, int par)
01831 {
01832     GETBIDENTITY
01833     WORD *new = TermMalloc("MergeSymbolLists");
01834     WORD *t1, *t2, *fill;
01835     int i1,i2;
01836     fill = new + 2;
01837     i1 = old[1] - 2; i2 = extra[1] - 2;
01838     t1 = old + 2; t2 = extra + 2;
01839     switch ( par ) {
01840         case -1:
01841             while ( i1 > 0 && i2 > 0 ) {
01842                 if ( *t1 > *t2 ) {
01843                     if ( t2[1] < 0 ) { *fill++ = *t2++; *fill++ = *t2++; }
01844                     else t2 += 2;
01845                     i2 -= 2;
01846                 }
01847                 else if ( *t1 < *t2 ) {
01848                     if ( t1[1] < 0 ) { *fill++ = *t1++; *fill++ = *t1++; }
01849                     else t1 += 2;
01850                     i1 -= 2;
01851                 }
01852                 else if ( t1[1] < t2[1] ) {
01853                     *fill++ = *t1++; *fill++ = *t1++; t2 += 2;
01854                     i1 -= 2; i2 -=2;
01855                 }
01856                 else {
01857                     *fill++ = *t2++; *fill++ = *t2++; t1 += 2;
01858                     i1 -= 2; i2 -=2;
01859                 }
01860             }
01861             for ( ; i1 > 0; i1 -= 2 ) {
01862                 if ( t1[1] < 0 ) { *fill++ = *t1++; *fill++ = *t1++; }
01863                 else t1 += 2;
01864             }
01865             for ( ; i2 > 0; i2 -= 2 ) {
01866                 if ( t2[1] < 0 ) { *fill++ = *t2++; *fill++ = *t2++; }
01867                 else t2 += 2;
01868             }
```

```
01869             break;
01870         case 1:
01871             while ( i1 > 0 && i2 > 0 ) {
01872                 if ( *t1 > *t2 ) {
01873                     if ( t2[1] > 0 ) { *fill++ = *t2++; *fill++ = *t2++; }
01874                     else t2 += 2;
01875                     i2 -=2;
01876                 }
01877                 else if ( *t1 < *t2 ) {
01878                     if ( t1[1] > 0 ) { *fill++ = *t1++; *fill++ = *t1++; }
01879                     else t1 += 2;
01880                     i1 -= 2;
01881                 }
01882                 else if ( t1[1] > t2[1] ) {
01883                     *fill++ = *t1++; *fill++ = *t1++; t2 += 2;
01884                     i1 -= 2; i2 -=2;
01885                 }
01886                 else {
01887                     *fill++ = *t2++; *fill++ = *t2++; t1 += 2;
01888                     i1 -= 2; i2 -=2;
01889                 }
01890             }
01891             for ( ; i1 > 0; i1 -= 2 ) {
01892                 if ( t1[1] > 0 ) { *fill++ = *t1++; *fill++ = *t1++; }
01893                 else t1 += 2;
01894             }
01895             for ( ; i2 > 0; i2 -= 2 ) {
01896                 if ( t2[1] > 0 ) { *fill++ = *t2++; *fill++ = *t2++; }
01897                 else t2 += 2;
01898             }
01899             break;
01900         case 0:
01901             while ( i1 > 0 && i2 > 0 ) {
01902                 if ( *t1 > *t2 ) {
01903                     t2 += 2; i2 -= 2;
01904                 }
01905                 else if ( *t1 < *t2 ) {
01906                     t1 += 2; i1 -= 2;
01907                 }
01908                 else if ( ( t1[1] > 0 ) && ( t2[1] < 0 ) ) { t1 += 2; t2 += 2; i1 -= 2; i2 -= 2; }
01909                 else if ( ( t1[1] < 0 ) && ( t2[1] > 0 ) ) { t1 += 2; t2 += 2; i1 -= 2; i2 -= 2; }
01910                 else if ( t1[1] > 0 ) {
01911                     if ( t1[1] < t2[1] ) {
01912                         *fill++ = *t1++; *fill++ = *t1++; t2 += 2; i2 -= 2;
01913                     }
01914                     else {
01915                         *fill++ = *t2++; *fill++ = *t2++; t1 += 2; i1 -= 2;
01916                     }
01917                 }
01918                 else {
01919                     if ( t2[1] < t1[1] ) {
01920                         *fill++ = *t2++; *fill++ = *t2++; t1 += 2; i1 -= 2; i2 -= 2;
01921                     }
01922                     else {
01923                         *fill++ = *t1++; *fill++ = *t1++; t2 += 2; i1 -= 2; i2 -= 2;
01924                     }
01925                 }
01926             }
01927             for ( ; i1 > 0; i1-- ) *fill++ = *t1++;
01928             for ( ; i2 > 0; i2-- ) *fill++ = *t2++;
01929             break;
01930     }
01931     new[0] = SYMBOL;
01932     i1 = new[1] = fill - new;
01933     t2 = new; t1 = old; NCOPY(t1,t2,i1);
01934     TermFree(new,"MergeSymbolLists");
01935     return(0);
01936 }
01937
01938 /*
01939         #] MergeSymbolLists :
01940         #[ MergeDotproductLists :
01941
01942         Merges the extra list into the old.
01943         If par == -1 we take minimum powers
01944         If par ==  1 we take maximum powers
01945         If par ==  0 we take minimum of the absolute value of the powers
01946                      if one is positive and the other negative we get zero.
01947         We assume that the dotproducts are in order in both lists
01948 */
01949
01950 int MergeDotproductLists(PHEAD WORD *old, WORD *extra, int par)
01951 {
01952     GETBIDENTITY
01953     WORD *new = TermMalloc("MergeDotproductLists");
01954     WORD *t1, *t2, *fill;
01955     int i1,i2;
```

```
01956     fill = new + 2;
01957     i1 = old[1] - 2; i2 = extra[1] - 2;
01958     t1 = old + 2; t2 = extra + 2;
01959     switch ( par ) {
01960         case -1:
01961             while ( i1 > 0 && i2 > 0 ) {
01962                 if ( ( *t1 > *t2 ) || ( *t1 == *t2 && t1[1] > t2[1] ) ) {
01963                     if ( t2[2] < 0 ) { *fill++ = *t2++; *fill++ = *t2++; *fill++ = *t2++; }
01964                     else t2 += 3;
01965                     i2 -= 3;
01966                 }
01967                 else if ( ( *t1 < *t2 ) || ( *t1 == *t2 && t1[1] < t2[1] ) ) {
01968                     if ( t1[2] < 0 ) { *fill++ = *t1++; *fill++ = *t1++; *fill++ = *t1++; }
01969                     else t1 += 3;
01970                     i1 -= 3;
01971                 }
01972                 else if ( t1[2] < t2[2] ) {
01973                     *fill++ = *t1++; *fill++ = *t1++; *fill++ = *t1++; t2 += 3;
01974                     i1 -= 3; i2 -= 3;
01975                 }
01976                 else {
01977                     *fill++ = *t2++; *fill++ = *t2++; *fill++ = *t2++; t1 += 3;
01978                     i2 -= 3; i1 -= 3;
01979                 }
01980             }
01981             for ( ; i1 > 0; i1 -= 3 ) {
01982                 if ( t1[2] < 0 ) { *fill++ = *t1++; *fill++ = *t1++; *fill++ = *t1++; }
01983                 else t1 += 3;
01984             }
01985             for ( ; i2 > 0; i2 -= 3 ) {
01986                 if ( t2[2] < 0 ) { *fill++ = *t2++; *fill++ = *t2++; *fill++ = *t2++; }
01987                 else t2 += 3;
01988             }
01989             break;
01990         case 1:
01991             while ( i1 > 0 && i2 > 0 ) {
01992                 if ( ( *t1 > *t2 ) || ( *t1 == *t2 && t1[1] > t2[1] ) ) {
01993                     if ( t2[2] > 0 ) { *fill++ = *t2++; *fill++ = *t2++; *fill++ = *t2++; }
01994                     else t2 += 3;
01995                     i2 -= 3;
01996                 }
01997                 else if ( ( *t1 < *t2 ) || ( *t1 == *t2 && t1[1] < t2[1] ) ) {
01998                     if ( t1[2] > 0 ) { *fill++ = *t1++; *fill++ = *t1++; *fill++ = *t1++; }
01999                     else t1 += 3;
02000                     i1 -= 3;
02001                 }
02002                 else if ( t1[2] > t2[2] ) {
02003                     *fill++ = *t1++; *fill++ = *t1++; *fill++ = *t1++; t2 += 3;
02004                     i1 -= 3; i2 -= 3;
02005                 }
02006                 else {
02007                     *fill++ = *t2++; *fill++ = *t2++; *fill++ = *t2++; t1 += 3;
02008                     i2 -= 3; i1 -= 3;
02009                 }
02010             }
02011             for ( ; i1 > 0; i1 -= 3 ) {
02012                 if ( t1[2] > 0 ) { *fill++ = *t1++; *fill++ = *t1++; *fill++ = *t1++; }
02013                 else t1 += 3;
02014             }
02015             for ( ; i2 > 0; i2 -= 3 ) {
02016                 if ( t2[2] > 0 ) { *fill++ = *t2++; *fill++ = *t2++; *fill++ = *t2++; }
02017                 else t2 += 3;
02018             }
02019             break;
02020         case 0:
02021             while ( i1 > 0 && i2 > 0 ) {
02022                 if ( ( *t1 > *t2 ) || ( *t1 == *t2 && t1[1] > t2[1] ) ) {
02023                     t2 += 3;
02024                     i2 -= 3;
02025                 }
02026                 else if ( ( *t1 < *t2 ) || ( *t1 == *t2 && t1[1] < t2[1] ) ) {
02027                     t1 += 3;
02028                     i1 -= 3;
02029                 }
02030                 else if ( ( t1[2] > 0 ) && ( t2[2] < 0 ) ) { t1 += 3; t2 += 3; i1 -= 3; i2 -= 3; }
02031                 else if ( ( t1[2] < 0 ) && ( t2[2] > 0 ) ) { t1 += 3; t2 += 3; i1 -= 3; i2 -= 3; }
02032                 else if ( t1[2] > 0 ) {
02033                     if ( t1[2] < t2[2] ) {
02034                         *fill++ = *t1++; *fill++ = *t1++; *fill++ = *t1++; t2 += 3;
02035                         i1 -= 3; i2 -= 3;
02036                     }
02037                     else {
02038                         *fill++ = *t2++; *fill++ = *t2++; *fill++ = *t2++; t1 += 3;
02039                         i2 -= 3; i1 -= 3;
02040
02041                     }
02042                 }
```

```
02043                 else {
02044                     if ( t2[2] < t1[2] ) {
02045                         *fill++ = *t2++; *fill++ = *t2++; *fill++ = *t2++; t1 += 3;
02046                         i2 -= 3; i1 -= 3;
02047                     }
02048                     else {
02049                         *fill++ = *t1++; *fill++ = *t1++; *fill++ = *t1++; t2 += 3;
02050                         i1 -= 3; i2 -= 3;
02051                     }
02052                 }
02053             }
02054             for ( ; i1 > 0; i1-- ) *fill++ = *t1++;
02055             for ( ; i2 > 0; i2-- ) *fill++ = *t2++;
02056             break;
02057     }
02058     new[0] = DOTPRODUCT;
02059     i1 = new[1] = fill - new;
02060     t2 = new; t1 = old; NCOPY(t1,t2,i1);
02061     TermFree(new,"MergeDotproductLists");
02062     return(0);
02063 }
02064
02065 /*
02066         #] MergeDotproductLists :
02067         #[ CreateExpression :
02068
02069         Looks for the expression in the argument, reads it and puts it
02070         in a buffer. Returns the address of the buffer.
02071         We send the expression through the Generator system, because there
02072         may be unsubstituted (sub)expressions as in
02073         Local F = (a+b);
02074         Local G = gcd_(F,...);
02075 */
02076
02077 WORD *CreateExpression(PHEAD WORD nexp)
02078 {
02079     GETBIDENTITY
02080     CBUF *C = cbuf+AC.cbufnum;
02081     POSITION startposition, oldposition;
02082     FILEHANDLE *fi;
02083     WORD *term, *oldipointer = AR.CompressPointer;
02084 ;
02085     switch ( Expressions[nexp].status ) {
02086         case HIDDENLEXPRESSION:
02087         case HIDDENGEXPRESSION:
02088         case DROPHLEXPRESSION:
02089         case DROPHGEXPRESSION:
02090         case UNHIDELEXPRESSION:
02091         case UNHIDEGEXPRESSION:
02092             AR.GetOneFile = 2; fi = AR.hidefile;
02093             break;
02094         default:
02095             AR.GetOneFile = 0; fi = AR.infile;
02096             break;
02097     }
02098     SeekScratch(fi,&oldposition);
02099     startposition = AS.OldOnFile[nexp];
02100     term = AT.WorkPointer;
02101     if ( GetOneTerm(BHEAD term,fi,&startposition,0) <= 0 ) goto CalledFrom;
02102     NewSort(BHEAD0);
02103     AR.CompressPointer = oldipointer;
02104     while ( GetOneTerm(BHEAD term,fi,&startposition,0) > 0 ) {
02105         AT.WorkPointer = term + *term;
02106         if ( Generator(BHEAD term,C->numlhs) ) {
02107             LowerSortLevel();
02108             goto CalledFrom;
02109         }
02110         AR.CompressPointer = oldipointer;
02111     }
02112     AT.WorkPointer = term;
02113     if ( EndSort(BHEAD (WORD *)((void *)(&term)),2) < 0 ) goto CalledFrom;
02114     SetScratch(fi,&oldposition);
02115     return(term);
02116 CalledFrom:
02117     MLOCK(ErrorMessageLock);
02118     MesCall("CreateExpression");
02119     MUNLOCK(ErrorMessageLock);
02120     Terminate(-1);
02121     return(0);
02122 }
02123
02124 /*
02125         #] CreateExpression :
02126         #[ GCDterms :       GCD of two terms
02127
02128     Computes the GCD of two terms.
02129     Output in termout.
```

```
02130     termout may overlap with term1.
02131 */
02132
02133 int GCDterms(PHEAD WORD *term1, WORD *term2, WORD *termout)
02134 {
02135     GETBIDENTITY
02136     WORD *t1, *t1stop, *t1next, *t2, *t2stop, *t2next, *tout, *tt1, *tt2;
02137     int count1, count2, i, ii, x1, sign;
02138     WORD length1, length2;
02139     t1 = term1 + *term1; t1stop = t1 - ABS(t1[-1]); t1 = term1+1;
02140     t2 = term2 + *term2; t2stop = t2 - ABS(t2[-1]); t2 = term2+1;
02141     tout = termout+1;
02142     while ( t1 < t1stop ) {
02143         t1next = t1 + t1[1];
02144         t2 = term2+1;
02145         if ( *t1 == SYMBOL ) {
02146             while ( t2 < t2stop && *t2 != SYMBOL ) t2 += t2[1];
02147             if ( t2 < t2stop && *t2 == SYMBOL ) {
02148                 t2next = t2+t2[1];
02149                 tt1 = t1+2; tt2 = t2+2; count1 = 0;
02150                 while ( tt1 < t1next && tt2 < t2next ) {
02151                     if ( *tt1 < *tt2 ) tt1 += 2;
02152                     else if ( *tt1 > *tt2 ) tt2 += 2;
02153                     else if ( ( tt1[1] > 0 && tt2[1] < 0 ) ||
02154                               ( tt2[1] > 0 && tt1[1] < 0 ) ) {
02155                         tt1 += 2; tt2 += 2;
02156                     }
02157                     else {
02158                         x1 = tt1[1];
02159                         if ( tt1[1] < 0 ) { if ( tt2[1] > x1 ) x1 = tt2[1]; }
02160                         else              { if ( tt2[1] < x1 ) x1 = tt2[1]; }
02161                         tout[count1+2] = *tt1;
02162                         tout[count1+3] = x1;
02163                         tt1 += 2; tt2 += 2;
02164                         count1 += 2;
02165                     }
02166                 }
02167                 if ( count1 > 0 ) {
02168                     *tout = SYMBOL; tout[1] = count1+2; tout += tout[1];
02169                 }
02170             }
02171         }
02172         else if ( *t1 == DOTPRODUCT ) {
02173             while ( t2 < t2stop && *t2 != DOTPRODUCT ) t2 += t2[1];
02174             if ( t2 < t2stop && *t2 == DOTPRODUCT ) {
02175                 t2next = t2+t2[1];
02176                 tt1 = t1+2; tt2 = t2+2; count1 = 0;
02177                 while ( tt1 < t1next && tt2 < t2next ) {
02178                     if ( *tt1 < *tt2 || ( *tt1 == *tt2 && tt1[1] < tt2[1] ) ) tt1 += 3;
02179                     else if ( *tt1 > *tt2 || ( *tt1 == *tt2 && tt1[1] > tt2[1] ) ) tt2 += 3;
02180                     else if ( ( tt1[2] > 0 && tt2[2] < 0 ) ||
02181                               ( tt2[2] > 0 && tt1[2] < 0 ) ) {
02182                         tt1 += 3; tt2 += 3;
02183                     }
02184                     else {
02185                         x1 = tt1[2];
02186                         if ( tt1[2] < 0 ) { if ( tt2[2] > x1 ) x1 = tt2[2]; }
02187                         else              { if ( tt2[2] < x1 ) x1 = tt2[2]; }
02188                         tout[count1+2] = *tt1;
02189                         tout[count1+3] = tt1[1];
02190                         tout[count1+4] = x1;
02191                         tt1 += 3; tt2 += 3;
02192                         count1 += 3;
02193                     }
02194                 }
02195                 if ( count1 > 0 ) {
02196                     *tout = DOTPRODUCT; tout[1] = count1+2; tout += tout[1];
02197                 }
02198             }
02199         }
02200         else if ( *t1 == VECTOR ) {
02201             while ( t2 < t2stop && *t2 != VECTOR ) t2 += t2[1];
02202             if ( t2 < t2stop && *t2 == VECTOR ) {
02203                 t2next = t2+t2[1];
02204                 tt1 = t1+2; tt2 = t2+2; count1 = 0;
02205                 while ( tt1 < t1next && tt2 < t2next ) {
02206                     if ( *tt1 < *tt2 || ( *tt1 == *tt2 && tt1[1] < tt2[1] ) ) tt1 += 2;
02207                     else if ( *tt1 > *tt2 || ( *tt1 == *tt2 && tt1[1] > tt2[1] ) ) tt2 += 2;
02208                     else {
02209                         tout[count1+2] = *tt1;
02210                         tout[count1+3] = tt1[1];
02211                         tt1 += 2; tt2 += 2;
02212                         count1 += 2;
02213                     }
02214                 }
02215                 if ( count1 > 0 ) {
02216                     *tout = VECTOR; tout[1] = count1+2; tout += tout[1];
```

```
02217                 }
02218             }
02219         }
02220         else if ( *t1 == INDEX ) {
02221             while ( t2 < t2stop && *t2 != INDEX ) t2 += t2[1];
02222             if ( t2 < t2stop && *t2 == INDEX ) {
02223                 t2next = t2+t2[1];
02224                 tt1 = t1+2; tt2 = t2+2; count1 = 0;
02225                 while ( tt1 < t1next && tt2 < t2next ) {
02226                     if ( *tt1 < *tt2 ) tt1 += 1;
02227                     else if ( *tt1 > *tt2 ) tt2 += 1;
02228                     else {
02229                         tout[count1+2] = *tt1;
02230                         tt1 += 1; tt2 += 1;
02231                         count1 += 1;
02232                     }
02233                 }
02234                 if ( count1 > 0 ) {
02235                     *tout = INDEX; tout[1] = count1+2; tout += tout[1];
02236                 }
02237             }
02238         }
02239         else if ( *t1 == DELTA ) {
02240             while ( t2 < t2stop && *t2 != DELTA ) t2 += t2[1];
02241             if ( t2 < t2stop && *t2 == DELTA ) {
02242                 t2next = t2+t2[1];
02243                 tt1 = t1+2; tt2 = t2+2; count1 = 0;
02244                 while ( tt1 < t1next && tt2 < t2next ) {
02245                     if ( *tt1 < *tt2 || ( *tt1 == *tt2 && tt1[1] < tt2[1] ) ) tt1 += 2;
02246                     else if ( *tt1 > *tt2 || ( *tt1 == *tt2 && tt1[1] > tt2[1] ) ) tt2 += 2;
02247                     else {
02248                         tout[count1+2] = *tt1;
02249                         tout[count1+3] = tt1[1];
02250                         tt1 += 2; tt2 += 2;
02251                         count1 += 2;
02252                     }
02253                 }
02254                 if ( count1 > 0 ) {
02255                     *tout = DELTA; tout[1] = count1+2; tout += tout[1];
02256                 }
02257             }
02258         }
02259         else if ( *t1 >= FUNCTION ) { /* noncommuting functions? Forbidden! */
02260 /*
02261             Count how many times this function occurs.
02262             Then count how many times it is in term2.
02263 */
02264             count1 = 1;
02265             while ( t1next < t1stop && *t1 == *t1next && t1[1] == t1next[1] ) {
02266                 for ( i = 2; i < t1[1]; i++ ) {
02267                     if ( t1[i] != t1next[i] ) break;
02268                 }
02269                 if ( i < t1[1] ) break;
02270                 count1++;
02271                 t1next += t1next[1];
02272             }
02273             count2 = 0;
02274             while ( t2 < t2stop ) {
02275                 if ( *t2 == *t1 && t2[1] == t1[1] ) {
02276                     for ( i = 2; i < t1[1]; i++ ) {
02277                         if ( t2[i] != t1[i] ) break;
02278                     }
02279                     if ( i >= t1[1] ) count2++;
02280                 }
02281                 t2 += t2[1];
02282             }
02283             if ( count1 < count2 ) count2 = count1; /* number of common occurrences */
02284             if ( count2 > 0 ) {
02285                 if ( tout == t1 ) {
02286                     while ( count2 > 0 ) { tout += tout[1]; count2--; }
02287                 }
02288                 else {
02289                     i = t1[1]*count2;
02290                     NCOPY(tout,t1,i);
02291                 }
02292             }
02293         }
02294         t1 = t1next;
02295     }
02296 /*
02297         Now the coefficients. They are in t1stop and t2stop. Should go to tout.
02298 */
02299     sign = 1;
02300     length1 = term1[*term1-1]; ii = i = ABS(length1); t1 = t1stop;
02301     if ( t1 != tout ) { NCOPY(tout,t1,i); tout -= ii; }
02302     length2 = term2[*term2-1];
02303     if ( length1 < 0 && length2 < 0 ) sign = -1;
```

```
02304     if ( AccumGCD(BHEAD (UWORD *)tout,&length1,(UWORD *)t2stop,length2) ) {
02305         MLOCK(ErrorMessageLock);
02306         MesCall("GCDterms");
02307         MUNLOCK(ErrorMessageLock);
02308         SETERROR(-1)
02309     }
02310     if ( sign < 0 && length1 > 0 ) length1 = -length1;
02311     tout += ABS(length1); tout[-1] = length1;
02312     *termout = tout - termout; *tout = 0;
02313     return(0);
02314 }
02315
02316 /*
02317         #] GCDterms :
02318         #[ ReadPolyRatFun :
02319 */
02320
02321 int ReadPolyRatFun(PHEAD WORD *term)
02322 {
02323     WORD *oldworkpointer = AT.WorkPointer;
02324     int flag, i;
02325     WORD *t, *fun, *nextt, *num, *den, *t1, *t2, size, numsize, densize;
02326     WORD *term1, *term2, *confree1, *confree2, *gcd, *num1, *den1, move, *newnum, *newden;
02327     WORD *tstop, *m1, *m2;
02328     WORD oldsorttype = AR.SortType;
02329     COMPARE oldcompareroutine = (COMPARE)(AR.CompareRoutine);
02330     AR.SortType = SORTHIGHFIRST;
02331     AR.CompareRoutine = (COMPAREDUMMY)(&CompareSymbols);
02332
02333     tstop = term + *term; tstop -= ABS(tstop[-1]);
02334     if ( term + *term == AT.WorkPointer ) flag = 1;
02335     else flag = 0;
02336     t = term+1;
02337     while ( t < tstop ) {
02338         if ( *t != AR.PolyFun ) { t += t[1]; continue; }
02339         if ( ( t[2] & MUSTCLEANPRF ) == 0 ) { t += t[1]; continue; }
02340         fun = t;
02341         nextt = t + t[1];
02342         if ( fun[1] > FUNHEAD && fun[FUNHEAD] == -SNUMBER && fun[FUNHEAD+1] == 0 )
02343             { *term = 0; break; }
02344         if ( FromPolyRatFun(BHEAD fun, &num, &den) > 0 ) { t = nextt; continue; }
02345         if ( *num == ARGHEAD ) { *term = 0; break; }
02346 /*
02347         Now we have num and den. Both are in general argument notation,
02348         but can also be used as expressions as in num+ARGHEAD, den+ARGHEAD.
02349         We need the gcd. For this we have to take out the contents
02350         because PreGCD does not like contents.
02351 */
02352         term1 = TermMalloc("ReadPolyRatFun");
02353         term2 = TermMalloc("ReadPolyRatFun");
02354         confree1 = TakeSymbolContent(BHEAD num+ARGHEAD,term1);
02355         confree2 = TakeSymbolContent(BHEAD den+ARGHEAD,term2);
02356         GCDclean(BHEAD term1,term2);
02357 /*      gcd = PreGCD(BHEAD confree1,confree2,1); */
02358         gcd = poly_gcd(BHEAD confree1,confree2,1);
02359         newnum = PolyDiv(BHEAD confree1,gcd,"ReadPolyRatFun");
02360         newden = PolyDiv(BHEAD confree2,gcd,"ReadPolyRatFun");
02361         TermFree(confree2,"ReadPolyRatFun");
02362         TermFree(confree1,"ReadPolyRatFun");
02363         num1 = MULfunc(BHEAD term1,newnum);
02364         den1 = MULfunc(BHEAD term2,newden);
02365         TermFree(newnum,"ReadPolyRatFun");
02366         TermFree(newden,"ReadPolyRatFun");
02367 /*      M_free(gcd,"poly_gcd"); */
02368         TermFree(gcd,"poly_gcd");
02369         TermFree(term1,"ReadPolyRatFun");
02370         TermFree(term2,"ReadPolyRatFun");
02371 /*
02372         Now we can put the function back together.
02373         Notice that we cannot use ToFast, because there is no reservation
02374         for the header of the argument. Fortunately there are only two
02375         types of fast arguments.
02376 */
02377         if ( num1[0] == 4 && num1[4] == 0 && num1[2] == 1 && num1[1] > 0 ) {
02378             numsize = 2; num1[0] = -SNUMBER;
02379             if ( num1[3] < 0 ) num1[1] = -num1[1];
02380         }
02381         else if ( num1[0] == 8 && num1[8] == 0 && num1[7] == 3 && num1[6] == 1
02382             && num1[5] == 1 && num1[1] == SYMBOL && num1[4] == 1 ) {
02383             numsize = 2; num1[0] = -SYMBOL; num1[1] = num1[3];
02384         }
02385         else { m1 = num1; while ( *m1 ) m1 += *m1; numsize = (m1-num1)+ARGHEAD; }
02386         if ( den1[0] == 4 && den1[4] == 0 && den1[2] == 1 && den1[1] > 0 ) {
02387             densize = 2; den1[0] = -SNUMBER;
02388             if ( den1[3] < 0 ) den1[1] = -den1[1];
02389         }
02390         else if ( den1[0] == 8 && den1[8] == 0 && den1[7] == 3 && den1[6] == 1
```

```
02391             && den1[5] == 1 && den1[1] == SYMBOL && den1[4] == 1 ) {
02392             densize = 2; den1[0] = -SYMBOL; den1[1] = den1[3];
02393         }
02394         else { m2 = den1; while ( *m2 ) m2 += *m2; densize = (m2-den1)+ARGHEAD; }
02395         size = FUNHEAD+numsize+densize;
02396
02397         if ( size > fun[1] ) {
02398             move = size - fun[1];
02399             t1 = term+*term; t2 = t1+move;
02400             while ( t1 > nextt ) *--t2 = *--t1;
02401             tstop += move; nextt += move;
02402             *term += move;
02403         }
02404         else if ( size < fun[1] ) {
02405             move = fun[1]-size;
02406             t2 = fun+size; t1 = nextt;
02407             tstop -= move; nextt -= move;
02408             t = term+*term;
02409             while ( t1 < t ) *t2++ = *t1++;
02410             *term -= move;
02411         }
02412         else { /* no need to move anything */ }
02413         fun[1] = size; fun[2] = 0;
02414         t2 = fun+FUNHEAD; t1 = num1;
02415         if ( *num1 < 0 ) { *t2++ = num1[0]; *t2++ = num1[1]; }
02416         else { *t2++ = numsize; *t2++ = 0; FILLARG(t2);
02417             i = numsize-ARGHEAD; NCOPY(t2,t1,i) }
02418         t1 = den1;
02419         if ( *den1 < 0 ) { *t2++ = den1[0]; *t2++ = den1[1]; }
02420         else { *t2++ = densize; *t2++ = 0; FILLARG(t2);
02421             i = densize-ARGHEAD; NCOPY(t2,t1,i) }
02422
02423         TermFree(num1,"MULfunc");
02424         TermFree(den1,"MULfunc");
02425         t = nextt;
02426     }
02427
02428     if ( flag ) AT.WorkPointer = term +*term;
02429     else AT.WorkPointer = oldworkpointer;
02430     AR.CompareRoutine = (COMPAREDUMMY)oldcompareroutine;
02431     AR.SortType = oldsorttype;
02432     return(0);
02433 }
02434
02435 /*
02436         #] ReadPolyRatFun :
02437         #[ FromPolyRatFun :
02438 */
02439
02440 int FromPolyRatFun(PHEAD WORD *fun, WORD **numout, WORD **denout)
02441 {
02442     WORD *nextfun, *tt, *num, *den;
02443     int i;
02444     nextfun = fun + fun[1];
02445     fun += FUNHEAD;
02446     num = AT.WorkPointer;
02447     if ( *fun < 0 ) {
02448         if ( *fun != -SNUMBER && *fun != -SYMBOL ) goto Improper;
02449         ToGeneral(fun,num,0);
02450         tt = num + *num; *tt++ = 0;
02451         fun += 2;
02452     }
02453     else { i = *fun; tt = num; NCOPY(tt,fun,i); *tt++ = 0; }
02454     den = tt;
02455     if ( *fun < 0 ) {
02456         if ( *fun != -SNUMBER && *fun != -SYMBOL ) goto Improper;
02457         ToGeneral(fun,den,0);
02458         tt = den + *den; *tt++ = 0;
02459         fun += 2;
02460     }
02461     else { i = *fun; tt = den; NCOPY(tt,fun,i); *tt++ = 0; }
02462     *numout = num; *denout = den;
02463     if ( fun != nextfun ) { return(1); }
02464     AT.WorkPointer = tt;
02465     return(0);
02466 Improper:
02467 /* INTERNAL_ERROR_EXCL_START */
02468     MLOCK(ErrorMessageLock);
02469     MesPrint("!>Improper use of PolyRatFun");
02470     MesCall("FromPolyRatFun");
02471     MUNLOCK(ErrorMessageLock);
02472     SETERROR(-1);
02473 /* INTERNAL_ERROR_EXCL_STOP */
02474 }
02475
02476 /*
02477         #] FromPolyRatFun :
```

```
02478         #[ TakeSymbolContent :
02479 */
02493 WORD *TakeSymbolContent(PHEAD WORD *in, WORD *term)
02494 {
02495     GETBIDENTITY
02496     WORD *t, *tstop, *tout, *tstore;
02497     WORD *tnext, *tt, *tterm;
02498     WORD *inp, a, *den, *oldworkpointer = AT.WorkPointer;
02499     int i, action = 0, sign, first;
02500     UWORD *GCDbuffer, *GCDbuffer2, *LCMbuffer, *LCMbuffer2, *ap;
02501     WORD GCDlen, GCDlen2, LCMlen, LCMlen2, length, redlength, len1, len2;
02502     LONG j;
02503     tout = tstore = term+1;
02504 /*
02505     #[ SYMBOL :
02506
02507     We make a list of symbols and their minimal powers.
02508     This includes negative powers. In the end we have to multiply by the
02509     inverse of this list. That takes out all negative powers and leaves
02510     things ready for further processing.
02511 */
02512     tterm = AT.WorkPointer; tt = tterm+1;
02513     tout[0] = SYMBOL; tout[1] = 2;
02514     t = in; first = 1;
02515     while ( *t ) {
02516         tnext = t + *t; tstop = tnext - ABS(tnext[-1]); t++;
02517         while ( t < tstop ) {
02518             if ( first ) {
02519                 if ( *t == SYMBOL ) {
02520                     for ( i = 0; i < t[1]; i++ ) tout[i] = t[i];
02521                     goto didwork;
02522                 }
02523                 else {
02524                     MLOCK(ErrorMessageLock);
02525                     MesPrint ((char*)"ERROR: polynomials and polyratfuns must contain symbols only");
02526                     MUNLOCK(ErrorMessageLock);
02527                     Terminate(1);
02528                 }
02529             }
02530             else if ( *t == SYMBOL ) {
02531                 MergeSymbolLists(BHEAD tout,t,-1);
02532                 goto didwork;
02533             }
02534             else {
02535                 t += t[1];
02536             }
02537         }
02538 /*
02539         Here we come when there were no symbols. Only keep the negative ones.
02540 */
02541         if ( first == 0 ) {
02542             int j = 2;
02543             for ( i = 2; i < tout[1]; i += 2 ) {
02544                 if ( tout[i+1] < 0 ) {
02545                     if ( i == j ) { j += 2; }
02546                     else { tout[j] = tout[i]; tout[j+1] = tout[i+1]; j += 2; }
02547                 }
02548             }
02549             tout[1] = j;
02550         }
02551 didwork:;
02552         first = 0;
02553         t = tnext;
02554     }
02555     if ( tout[1] > 2 ) {
02556         t = tout;
02557         tt[0] = t[0]; tt[1] = t[1];
02558         for ( i = 2; i < t[1]; i += 2 ) {
02559             tt[i] = t[i]; tt[i+1] = -t[i+1];
02560         }
02561         tt += tt[1];
02562         tout += tout[1];
02563         action++;
02564     }
02565 /*
02566     #] SYMBOL :
02567     #[ Coefficient :
02568
02569     Now we have to collect the GCD of the numerators
02570     and the LCM of the denominators.
02571 */
02572     AT.WorkPointer = tt;
02573     if ( AN.cmod != 0 ) {
02574       WORD x, ix, ip;
02575       t = in; tnext = t + *t; tstop = tnext - ABS(tnext[-1]);
02576       x = tstop[0];
02577       if ( tnext[-1] < 0 ) x += AC.cmod[0];
```

```
02578       if ( GetModInverses(x,(WORD)(AN.cmod[0]),&ix,&ip) ) goto CalledFrom;
02579       *tout++ = x; *tout++ = 1; *tout++ = 3;
02580       *tt++ = ix; *tt++ = 1; *tt++ = 3;
02581     }
02582     else {
02583       GCDbuffer  = NumberMalloc("MakeInteger");
02584       GCDbuffer2 = NumberMalloc("MakeInteger");
02585       LCMbuffer  = NumberMalloc("MakeInteger");
02586       LCMbuffer2 = NumberMalloc("MakeInteger");
02587       t = in;
02588       tnext = t + *t; length = tnext[-1];
02589       if ( length < 0 ) { sign = -1; length = -length; }
02590       else              { sign = 1; }
02591       tstop = tnext - length;
02592       redlength = (length-1)/2;
02593       for ( i = 0; i < redlength; i++ ) {
02594         GCDbuffer[i] = (UWORD)(tstop[i]);
02595         LCMbuffer[i] = (UWORD)(tstop[redlength+i]);
02596       }
02597       GCDlen = LCMlen = redlength;
02598       while ( GCDbuffer[GCDlen-1] == 0 ) GCDlen--;
02599       while ( LCMbuffer[LCMlen-1] == 0 ) LCMlen--;
02600       t = tnext;
02601       while ( *t ) {
02602         tnext = t + *t; length = ABS(tnext[-1]);
02603         tstop = tnext - length; redlength = (length-1)/2;
02604         len1 = len2 = redlength;
02605         den = tstop + redlength;
02606         while ( tstop[len1-1] == 0 ) len1--;
02607         while ( den[len2-1] == 0 ) len2--;
02608         if ( GCDlen == 1 && GCDbuffer[0] == 1 ) {}
02609         else {
02610             GcdLong(BHEAD (UWORD *)tstop,len1,GCDbuffer,GCDlen,GCDbuffer2,&GCDlen2);
02611             ap = GCDbuffer; GCDbuffer = GCDbuffer2; GCDbuffer2 = ap;
02612             a = GCDlen; GCDlen = GCDlen2; GCDlen2 = a;
02613         }
02614         if ( len2 == 1 && den[0] == 1 ) {}
02615         else {
02616             GcdLong(BHEAD LCMbuffer,LCMlen,(UWORD *)den,len2,LCMbuffer2,&LCMlen2);
02617             DivLong((UWORD *)den,len2,LCMbuffer2,LCMlen2,
02618                 GCDbuffer2,&GCDlen2,(UWORD *)AT.WorkPointer,&a);
02619             MulLong(LCMbuffer,LCMlen,GCDbuffer2,GCDlen2,LCMbuffer2,&LCMlen2);
02620             ap = LCMbuffer; LCMbuffer = LCMbuffer2; LCMbuffer2 = ap;
02621             a = LCMlen; LCMlen = LCMlen2; LCMlen2 = a;
02622         }
02623         t = tnext;
02624       }
02625       if ( GCDlen != 1 || GCDbuffer[0] != 1 || LCMlen != 1 || LCMbuffer[0] != 1 ) {
02626         redlength = GCDlen; if ( LCMlen > GCDlen ) redlength = LCMlen;
02627         for ( i = 0; i < GCDlen; i++ ) *tout++ = (WORD)(GCDbuffer[i]);
02628         for ( ; i < redlength; i++ ) *tout++ = 0;
02629         for ( i = 0; i < LCMlen; i++ ) *tout++ = (WORD)(LCMbuffer[i]);
02630         for ( ; i < redlength; i++ ) *tout++ = 0;
02631         *tout++ = (2*redlength+1)*sign;
02632         for ( i = 0; i < LCMlen; i++ ) *tt++ = (WORD)(LCMbuffer[i]);
02633         for ( ; i < redlength; i++ ) *tt++ = 0;
02634         for ( i = 0; i < GCDlen; i++ ) *tt++ = (WORD)(GCDbuffer[i]);
02635         for ( ; i < redlength; i++ ) *tt++ = 0;
02636         *tt++ = (2*redlength+1)*sign;
02637         action++;
02638       }
02639       else {
02640         *tout++ = 1; *tout++ = 1; *tout++ = 3*sign;
02641         *tt++ = 1; *tt++ = 1; *tt++ = 3*sign;
02642         if ( sign != 1 ) action++;
02643       }
02644       NumberFree(LCMbuffer2,"MakeInteger");
02645       NumberFree(LCMbuffer ,"MakeInteger");
02646       NumberFree(GCDbuffer2,"MakeInteger");
02647       NumberFree(GCDbuffer ,"MakeInteger");
02648     }
02649 /*
02650     #] Coefficient :
02651     #[ Multiply by the inverse content :
02652 */
02653     if ( action ) {
02654         *term = tout - term; *tout = 0;
02655         *tterm = tt - tterm; *tt = 0;
02656         AT.WorkPointer = tt;
02657         inp = MultiplyWithTerm(BHEAD in,tterm,2);
02658         AT.WorkPointer = tterm;
02659         t = inp; while ( *t ) t += *t;
02660         j = (t-inp); t = inp;
02661         if ( j*sizeof(WORD) > (size_t)(AM.MaxTer) ) goto OverWork;
02662         in = tout = TermMalloc("TakeSymbolContent");
02663         NCOPY(tout,t,j); *tout = 0;
02664         if ( AN.tryterm > 0 ) { TermFree(inp,"MultiplyWithTerm"); AN.tryterm = 0; }
```

```
02665         else { M_free(inp,"MultiplyWithTerm"); }
02666     }
02667     else {
02668         t = in; while ( *t ) t += *t;
02669         j = (t-in); t = in;
02670         if ( j*sizeof(WORD) > (size_t)(AM.MaxTer) ) goto OverWork;
02671         in = tout = TermMalloc("TakeSymbolContent");
02672         NCOPY(tout,t,j); *tout = 0;
02673         term[0] = 4; term[1] = 1; term[2] = 1; term[3] = 3; term[4] = 0;
02674     }
02675 /*
02676     #] Multiply by the inverse content :
02677     AT.WorkPointer = tterm + *tterm;
02678 */
02679     AT.WorkPointer = oldworkpointer;
02680
02681     return(in);
02682 OverWork:
02683     MLOCK(ErrorMessageLock);
02684     MesPrint("Term too complex. Maybe increasing MaxTermSize can help");
02685     MUNLOCK(ErrorMessageLock);
02686 CalledFrom:
02687     MLOCK(ErrorMessageLock);
02688     MesCall("TakeSymbolContent");
02689     MUNLOCK(ErrorMessageLock);
02690     Terminate(-1);
02691     return(0);
02692 }
02693
02694 /*
02695         #] TakeSymbolContent :
02696         #[ GCDclean :
02697
02698         Takes a numerator and a denominator that each consist of a
02699         single term with only a coefficient and symbols and makes them
02700         into a proper fraction. Output overwrites input.
02701 */
02702
02703 void GCDclean(PHEAD WORD *num, WORD *den)
02704 {
02705     WORD *out1 = TermMalloc("GCDclean");
02706     WORD *out2 = TermMalloc("GCDclean");
02707     WORD *t1, *t2, *r1, *r2, *t1stop, *t2stop, csize1, csize2, csize3, pow, sign;
02708     int i;
02709     t1stop = num+*num; sign = ( t1stop[-1] < 0 ) ? -1 : 1;
02710     csize1 = ABS(t1stop[-1]); t1stop -= csize1;
02711     t2stop = den+*den; if ( t2stop[-1] < 0 ) sign = -sign;
02712     csize2 = ABS(t2stop[-1]); t2stop -= csize2;
02713     t1 = num+1; t2 = den+1;
02714     r1 = out1+3; r2 = out2+3;
02715     if ( t1 == t1stop ) {
02716         if ( t2 < t2stop ) {
02717             for ( i = 2; i < t2[1]; i += 2 ) {
02718                 if ( t2[i+1] < 0 ) { *r1++ = t2[i]; *r1++ = -t2[i+1]; }
02719                 else { *r2++ = t2[i]; *r2++ = t2[i+1]; }
02720             }
02721         }
02722     }
02723     else if ( t2 == t2stop ) {
02724         for ( i = 2; i < t1[1]; i += 2 ) {
02725             if ( t1[i+1] < 0 ) { *r2++ = t1[i]; *r2++ = -t1[i+1]; }
02726             else { *r1++ = t1[i]; *r1++ = t1[i+1]; }
02727         }
02728     }
02729     else {
02730         t1 += 2; t2 += 2;
02731         while ( t1 < t1stop && t2 < t2stop ) {
02732             if ( *t1 < *t2 ) {
02733                 if ( t1[1] > 0 ) { *r1++ = *t1; *r1++ = t1[1]; t1 += 2; }
02734                 else if ( t1[1] < 0 ) { *r2++ = *t1; *r2++ = -t1[1]; t1 += 2; }
02735             }
02736             else if ( *t1 > *t2 ) {
02737                 if ( t2[1] > 0 ) { *r2++ = *t2; *r2++ = t2[1]; t2 += 2; }
02738                 else if ( t2[1] < 0 ) { *r1++ = *t2; *r1++ = -t2[1]; t2 += 2; }
02739             }
02740             else {
02741                 pow = t1[1]-t2[1];
02742                 if ( pow > 0 ) { *r1++ = *t1; *r1++ = pow; }
02743                 else if ( pow < 0 ) { *r2++ = *t1; *r2++ = -pow; }
02744                 t1 += 2; t2 += 2;
02745             }
02746         }
02747         while ( t1 < t1stop ) {
02748             if ( t1[1] < 0 ) { *r2++ = *t1; *r2++ = -t1[1]; }
02749             else { *r1++ = *t1; *r1++ = t1[1]; }
02750             t1 += 2;
02751         }
```

```
02752         while ( t2 < t2stop ) {
02753             if ( t2[1] < 0 ) { *r1++ = *t2; *r1++ = -t2[1]; }
02754             else { *r2++ = *t2; *r2++ = t2[1]; }
02755             t2 += 2;
02756         }
02757     }
02758     if ( r1 > out1+3 ) { out1[1] = SYMBOL; out1[2] = r1 - out1 - 1; }
02759     else r1 = out1+1;
02760     if ( r2 > out2+3 ) { out2[1] = SYMBOL; out2[2] = r2 - out2 - 1; }
02761     else r2 = out2+1;
02762 /*
02763     Now the coefficients.
02764 */
02765     csize1 = REDLENG(csize1);
02766     csize2 = REDLENG(csize2);
02767     if ( DivRat(BHEAD (UWORD *)t1stop,csize1,(UWORD *)t2stop,csize2,(UWORD *)r1,&csize3) ) {
02768         MLOCK(ErrorMessageLock);
02769         MesCall("GCDclean");
02770         MUNLOCK(ErrorMessageLock);
02771         Terminate(-1);
02772     }
02773     UnPack((UWORD *)r1,csize3,&csize2,&csize1);
02774     t2 = r1+ABS(csize3);
02775     for ( i = 0; i < csize2; i++ ) r2[i] = t2[i];
02776     r2 += csize2; *r2++ = 1;
02777     for ( i = 1; i < csize2; i++ ) *r2++ = 0;
02778     csize2 = INCLENG(csize2); *r2++ = csize2; *out2 = r2-out2;
02779     r1 += ABS(csize1); *r1++ = 1;
02780     for ( i = 1; i < ABS(csize1); i++ ) *r1++ = 0;
02781     csize1 = INCLENG(csize1); *r1++ = csize1; *out1 = r1-out1;
02782
02783     t1 = num; t2 = out1; i = *out1; NCOPY(t1,t2,i); *t1 = 0;
02784     if ( sign < 0 ) t1[-1] = -t1[-1];
02785     t1 = den; t2 = out2; i = *out2; NCOPY(t1,t2,i); *t1 = 0;
02786
02787     TermFree(out2,"GCDclean");
02788     TermFree(out1,"GCDclean");
02789 }
02790
02791 /*
02792         #] GCDclean :
02793         #[ PolyDiv :
02794
02795         Special stub function for polynomials that should fit inside a term.
02796         We make sure that the space is allocated by TermMalloc.
02797         This makes things much easier on the calling routines.
02798 */
02799
02800 WORD *PolyDiv(PHEAD WORD *a,WORD *b,char *text)
02801 {
02802 /*
02803     Probably the following would work now
02804 */
02805     DUMMYUSE(text);
02806     return(poly_div(BHEAD a,b,1));
02807 /*
02808     WORD *quo, *qq;
02809     WORD *x, *xx;
02810     LONG i;
02811     quo = poly_div(BHEAD a,b,1);
02812     x = TermMalloc(text);
02813     qq = quo; while ( *qq ) qq += *qq;
02814     i = (qq-quo+1);
02815     if ( i*sizeof(WORD) > (size_t)(AM.MaxTer) ) {
02816         DUMMYUSE(text);
02817         MLOCK(ErrorMessageLock);
02818         MesPrint("PolyDiv: Term too complex. Maybe increasing MaxTermSize can help");
02819         MUNLOCK(ErrorMessageLock);
02820         Terminate(-1);
02821     }
02822     xx = x; qq = quo;
02823     NCOPY(xx,qq,i)
02824     TermFree(quo,"poly_div");
02825     return(x);
02826 */
02827 }
02828
02829 /*
02830         #] PolyDiv :
02831     #] GCDfunction :
02832     #[ DIVfunction :
02833
02834     Input: a div_ function that has two arguments inside a term.
02835     Action: Calculates [arg1/arg2] using polynomial techniques if needed.
02836     Output: The output result is combined with the remainder of the term
02837     and sent to Generator for further processing.
02838     Note that the output can be just a number or many terms.
```

```
02839     In case par == 0 the output is [arg1/arg2]
02840     In case par == 1 the output is [arg1%arg2]
02841     In case par == 2 the output is [inverse of arg1 modulus arg2]
02842     In case par == 3 the output is [arg1*arg2]
02843 */
02844
02845 WORD divrem[4] = { DIVFUNCTION, REMFUNCTION, INVERSEFUNCTION, MULFUNCTION };
02846
02847 int DIVfunction(PHEAD WORD *term,WORD level,int par)
02848 {
02849     GETBIDENTITY
02850     WORD *t, *tt, *r, *arg1 = 0, *arg2 = 0, *arg3 = 0, *termout;
02851     WORD *tstop, *tend, *r3, *rr, *rstop, tlength, rlength, newlength;
02852     WORD *proper1, *proper2, *proper3 = 0, numdol = -1;
02853     int numargs = 0, type1, type2, actionflag1, actionflag2;
02854     WORD startebuf = cbuf[AT.ebufnum].numrhs;
02855     int division = ( par <= 2 );  /* false for mul_ */
02856     if ( par < 0 || par > 3 ) {
02857 /* INTERNAL_ERROR_EXCL_START */
02858         MLOCK(ErrorMessageLock);
02859         MesPrint("!>Internal error. Illegal parameter %d in DIVfunction.",par);
02860         MUNLOCK(ErrorMessageLock);
02861         Terminate(-1);
02862 /* INTERNAL_ERROR_EXCL_STOP */
02863     }
02864 /*
02865     Find the function
02866 */
02867     tend = term + *term; tstop = tend - ABS(tend[-1]);
02868     t = term+1;
02869     while ( t < tstop ) {
02870         if ( *t != divrem[par] ) { t += t[1]; continue; }
02871         r = t + FUNHEAD;
02872         tt = t + t[1]; numargs = 0;
02873         while ( r < tt ) {
02874             if ( numargs == 0 ) { arg1 = r; }
02875             if ( numargs == 1 ) { arg2 = r; }
02876             if ( numargs == 2 && *r == -DOLLAREXPRESSION ) { numdol = r[1]; }
02877             numargs++;
02878             NEXTARG(r);
02879         }
02880         if ( numargs == 2 ) break;
02881         if ( division && numargs == 3 ) break;
02882         t = tt;
02883     }
02884     if ( t >= tstop ) {
02885 /* INTERNAL_ERROR_EXCL_START */
02886         MLOCK(ErrorMessageLock);
02887         MesPrint("!>Internal error. Indicated div_ or rem_ function not encountered.");
02888         MUNLOCK(ErrorMessageLock);
02889         Terminate(-1);
02890 /* INTERNAL_ERROR_EXCL_STOP */
02891     }
02892 /*
02893     We have two arguments in arg1 and arg2.
02894 */
02895     if ( division && *arg1 == -SNUMBER && arg1[1] == 0 ) {
02896         if ( *arg2 == -SNUMBER && arg2[1] == 0 ) {
02897 zerozero:;
02898             MLOCK(ErrorMessageLock);
02899             MesPrint("0/0 in either div_ or rem_ function.");
02900             MUNLOCK(ErrorMessageLock);
02901             Terminate(-1);
02902         }
02903         if ( numdol >= 0 ) PutTermInDollar(0,numdol);
02904         return(0);
02905     }
02906     if ( division && *arg2 == -SNUMBER && arg2[1] == 0 ) {
02907 divzero:;
02908         MLOCK(ErrorMessageLock);
02909         MesPrint("Division by zero in either div_ or rem_ function.");
02910         MUNLOCK(ErrorMessageLock);
02911         Terminate(-1);
02912     }
02913     if ( !division ) {
02914         if ( (*arg1 == -SNUMBER && arg1[1] == 0) ||
02915              (*arg2 == -SNUMBER && arg2[1] == 0) ) {
02916             return(0);
02917         }
02918     }
02919     if ( ( arg1 = ConvertArgument(BHEAD arg1, &type1) ) == 0 ) goto CalledFrom;
02920     if ( ( arg2 = ConvertArgument(BHEAD arg2, &type2) ) == 0 ) goto CalledFrom;
02921     if ( division && *arg1 == 0 ) {
02922         if ( *arg2 == 0 ) {
02923             M_free(arg2,"DIVfunction");
02924             M_free(arg1,"DIVfunction");
02925             goto zerozero;
```

```
02926         }
02927         M_free(arg2,"DIVfunction");
02928         M_free(arg1,"DIVfunction");
02929         if ( numdol >= 0 ) PutTermInDollar(0,numdol);
02930         return(0);
02931     }
02932     if ( division && *arg2 == 0 ) {
02933         M_free(arg2,"DIVfunction");
02934         M_free(arg1,"DIVfunction");
02935         goto divzero;
02936     }
02937     if ( !division && (*arg1 == 0 || *arg2 == 0) ) {
02938         M_free(arg2,"DIVfunction");
02939         M_free(arg1,"DIVfunction");
02940         return(0);
02941     }
02942     if ( ( proper1 = PutExtraSymbols(BHEAD arg1,startebuf,&actionflag1) ) == 0 ) goto CalledFrom;
02943     if ( ( proper2 = PutExtraSymbols(BHEAD arg2,startebuf,&actionflag2) ) == 0 ) goto CalledFrom;
02944 /*
02945     if ( type2 == 0 ) M_free(arg2,"DIVfunction");
02946     else {
02947         DOLLARS d = ((DOLLARS)arg2)-1;
02948         if ( d->factors ) M_free(d->factors,"Dollar factors");
02949         M_free(d,"Copy of dollar variable");
02950     }
02951 */
02952     M_free(arg2,"DIVfunction");
02953 /*
02954     if ( type1 == 0 ) M_free(arg1,"DIVfunction");
02955     else {
02956         DOLLARS d = ((DOLLARS)arg1)-1;
02957         if ( d->factors ) M_free(d->factors,"Dollar factors");
02958         M_free(d,"Copy of dollar variable");
02959     }
02960 */
02961     M_free(arg1,"DIVfunction");
02962     if ( par == 0 )      proper3 = poly_div(BHEAD proper1, proper2,0);
02963     else if ( par == 1 ) proper3 = poly_rem(BHEAD proper1, proper2,0);
02964     else if ( par == 2 ) proper3 = poly_inverse(BHEAD proper1, proper2);
02965     else if ( par == 3 ) proper3 = poly_mul(BHEAD proper1, proper2);
02966     if ( proper3 == 0 ) goto CalledFrom;
02967     if ( actionflag1 || actionflag2 ) {
02968         if ( ( arg3 = TakeExtraSymbols(BHEAD proper3,startebuf) ) == 0 ) goto CalledFrom;
02969         M_free(proper3,"DIVfunction");
02970     }
02971     else {
02972         arg3 = proper3;
02973     }
02974     M_free(proper2,"DIVfunction");
02975     M_free(proper1,"DIVfunction");
02976     cbuf[AT.ebufnum].numrhs = startebuf;
02977     if ( *arg3 ) {
02978         termout = AT.WorkPointer;
02979         tlength = tend[-1];
02980         tlength = REDLENG(tlength);
02981         r3 = arg3;
02982         while ( *r3 ) {
02983             tt = term + 1; rr = termout + 1;
02984             while ( tt < t ) *rr++ = *tt++;
02985             r = r3 + 1;
02986             r3 = r3 + *r3;
02987             rstop = r3 - ABS(r3[-1]);
02988             while ( r < rstop ) *rr++ = *r++;
02989             tt += t[1];
02990             while ( tt < tstop ) *rr++ = *tt++;
02991             rlength = r3[-1];
02992             rlength = REDLENG(rlength);
02993             if ( MulRat(BHEAD (UWORD *)tstop,tlength,(UWORD *)rstop,rlength,
02994                     (UWORD *)rr,&newlength) < 0 ) goto CalledFrom;
02995             rlength = INCLENG(newlength);
02996             rr += ABS(rlength);
02997             rr[-1] = rlength;
02998             *termout = rr - termout;
02999             AT.WorkPointer = rr;
03000             if ( Generator(BHEAD termout,level) ) goto CalledFrom;
03001         }
03002         AT.WorkPointer = termout;
03003     }
03004     M_free(arg3,"DIVfunction");
03005     return(0);
03006 CalledFrom:
03007     MLOCK(ErrorMessageLock);
03008     MesCall("DIVfunction");
03009     MUNLOCK(ErrorMessageLock);
03010     SETERROR(-1)
03011 }
03012
```

```
03013 /*
03014     #] DIVfunction :
03015     #[ MULfunc :
03016
03017     Multiplies two polynomials and puts the results in TermMalloc space.
03018 */
03019
03020 WORD *MULfunc(PHEAD WORD *p1, WORD *p2)
03021 {
03022     WORD *prod,size1,size2,size3,*t,*tfill,*ps1,*ps2,sign1,sign2, error, *p3;
03023     UWORD *num1, *num2, *num3;
03024     int i;
03025     WORD oldsorttype = AR.SortType;
03026     COMPARE oldcompareroutine = (COMPARE)(AR.CompareRoutine);
03027     AR.SortType = SORTHIGHFIRST;
03028     AR.CompareRoutine = (COMPAREDUMMY)(&CompareSymbols);
03029     num3 = NumberMalloc("MULfunc");
03030     prod = TermMalloc("MULfunc");
03031     NewSort(BHEAD0);
03032     while ( *p1 ) {
03033         ps1 = p1+*p1; num1 = (UWORD *)(ps1 - ABS(ps1[-1])); size1 = ps1[-1];
03034         if ( size1 < 0 ) { sign1 = -1; size1 = -size1; }
03035         else               sign1 = 1;
03036         size1 = (size1-1)/2;
03037         p3 = p2;
03038         while ( *p3 ) {
03039             ps2 = p3+*p3; num2 = (UWORD *)(ps2 - ABS(ps2[-1])); size2 = ps2[-1];
03040             if ( size2 < 0 ) { sign2 = -1; size2 = -size2; }
03041             else               sign2 = 1;
03042             size2 = (size2-1)/2;
03043             if ( MulLong(num1,size1,num2,size2,num3,&size3) ) {
03044 /* INTERNAL_ERROR_EXCL_START */
03045                 error = 1;
03046 CalledFrom:
03047                 MLOCK(ErrorMessageLock);
03048                 MesPrint("!>Error %d",error);
03049                 MesCall("MulFunc");
03050                 MUNLOCK(ErrorMessageLock);
03051                 Terminate(-1);
03052 /* INTERNAL_ERROR_EXCL_STOP */
03053             }
03054             tfill = prod+1;
03055             t = p1+1; while ( t < (WORD *)num1 ) *tfill++ = *t++;
03056             t = p3+1; while ( t < (WORD *)num2 ) *tfill++ = *t++;
03057             t = (WORD *)num3;
03058             for ( i = 0; i < size3; i++ ) *tfill++ = *t++;
03059             *tfill++ = 1;
03060             for ( i = 1; i < size3; i++ ) *tfill++ = 0;
03061             *tfill++ = (2*size3+1)*sign1*sign2;
03062             prod[0] = tfill - prod;
03063             if ( SymbolNormalize(prod) ) { error = 2; goto CalledFrom; }
03064             if ( StoreTerm(BHEAD prod) ) { error = 3; goto CalledFrom; }
03065             p3 += *p3;
03066         }
03067         p1 += *p1;
03068     }
03069     NumberFree(num3,"MULfunc");
03070     EndSort(BHEAD prod,1);
03071     AR.CompareRoutine = (COMPAREDUMMY)oldcompareroutine;
03072     AR.SortType = oldsorttype;
03073     return(prod);
03074 }
03075
03076 /*
03077     #] MULfunc :
03078     #[ ConvertArgument :
03079
03080     Converts an argument to a general notation in allocated space.
03081 */
03082
03083 WORD *ConvertArgument(PHEAD WORD *arg, int *type)
03084 {
03085     WORD *output, *t, *r;
03086     int i, size;
03087     if ( *arg > 0 ) {
03088         output = (WORD *)Malloc1((*arg)*sizeof(WORD),"ConvertArgument");
03089         i = *arg - ARGHEAD; t = arg + ARGHEAD; r = output;
03090         NCOPY(r,t,i);
03091         *r = 0; *type = 0;
03092         return(output);
03093     }
03094     if ( *arg == -EXPRESSION ) {
03095         *type = 0;
03096         return(CreateExpression(BHEAD arg[1]));
03097     }
03098     if ( *arg == -DOLLAREXPRESSION ) {
03099         DOLLARS d;
```

```
03100         *type = 1;
03101         d = DolToTerms(BHEAD arg[1]);
03102 /*
03103         The problem is that DolToTerms creates a copy of the dollar variable.
03104         If we just return d->where we create a memory leak. Hence we have to
03105         copy the contents of d->where to a new buffer
03106 */
03107         output = (WORD *)Malloc1((d->size+1)*sizeof(WORD),"Copy of dollar content");
03108         WCOPY(output,d->where,d->size+1);
03109         if ( d->factors ) { M_free(d->factors,"Dollar factors"); d->factors = 0; }
03110         M_free(d,"Copy of dollar variable");
03111         return(output);
03112     }
03113 #if ( FUNHEAD > 4 )
03114     size = FUNHEAD+5;
03115 #else
03116     size = 9;
03117 #endif
03118     output = (WORD *)Malloc1(size*sizeof(WORD),"ConvertArgument");
03119     switch(*arg) {
03120         case -SYMBOL:
03121             output[0] = 8; output[1] = SYMBOL; output[2] = 4; output[3] = arg[1];
03122             output[4] = 1; output[5] = 1; output[6] = 1; output[7] = 3;
03123             output[8] = 0;
03124             break;
03125         case -INDEX:
03126         case -VECTOR:
03127         case -MINVECTOR:
03128             output[0] = 7; output[1] = INDEX; output[2] = 3; output[3] = arg[1];
03129             output[4] = 1; output[5] = 1;
03130             if ( *arg == -MINVECTOR ) output[6] = -3;
03131             else output[6] = 3;
03132             output[7] = 0;
03133             break;
03134         case -SNUMBER:
03135             output[0] = 4;
03136             if ( arg[1] < 0 ) {
03137                 output[1] = -arg[1]; output[2] = 1; output[3] = -3;
03138             }
03139             else {
03140                 output[1] =  arg[1]; output[2] = 1; output[3] =  3;
03141             }
03142             output[4] = 0;
03143             break;
03144         default:
03145             output[0] = FUNHEAD+4;
03146             output[1] = -*arg;
03147             output[2] = FUNHEAD;
03148             for ( i = 3; i <= FUNHEAD; i++ ) output[i] = 0;
03149             output[FUNHEAD+1] = 1;
03150             output[FUNHEAD+2] = 1;
03151             output[FUNHEAD+3] = 3;
03152             output[FUNHEAD+4] = 0;
03153             break;
03154     }
03155     *type = 0;
03156     return(output);
03157 }
03158
03159 /*
03160     #] ConvertArgument :
03161     #[ ExpandRat :
03162
03163     Expands the denominator of a PolyRatFun in the variable PolyFunVar.
03164     The output is a polyratfun with a single argument.
03165     In the case that there is a polyratfun with more than one argument
03166     or the dirtyflag is on, the argument(s) is/are normalized.
03167     The output overwrites the input.
03168 */
03169
03170 char *TheErrorMessage[] = {
03171      "PolyRatFun not of a type that FORM will expand: incorrect variable inside."
03172     ,"Division by zero in PolyRatFun encountered in ExpandRat."
03173     ,"Irregular code in PolyRatFun encountered in ExpandRat."
03174     ,"Called from ExpandRat."
03175     ,"WorkSpace overflow. Change parameter WorkSpace in setup file?"
03176     ,"Illegal term in expanded polyratfun."
03177     };
03178
03179 int ExpandRat(PHEAD WORD *fun)
03180 {
03181     WORD *r, *rr, *rrr, *tt, *tnext, *arg1, *arg2, *rmin = NULL, *rmininv;
03182     WORD *rcoef, rsize, rcopy, *ow = AT.WorkPointer;
03183     WORD *numerator, *denominator, *rnext;
03184     WORD *thecopy, *rc, ncoef, newcoef, *m, *mm, nco, *outarg = NULL;
03185     UWORD co[2], co1[2], co2[2];
03186     WORD OldPolyFunPow = AR.PolyFunPow;
```

```
03187     int i, j, minpow = 0, eppow, first, error = 0, ipoly;
03188     if ( fun[1] == FUNHEAD ) { return(0); }
03189     tnext = fun + fun[1];
03190     if ( fun[1] == fun[FUNHEAD]+FUNHEAD ) { /* Single argument */
03191         if ( fun[2] == 0 ) { goto done; }
03192 /*
03193         We have to normalize the argument. This could make it shorter.
03194 */
03195 NormArg:;
03196         if ( outarg == NULL ) outarg = TermMalloc("ExpandRat")+ARGHEAD;
03197         AT.TrimPower = 1;
03198         NewSort(BHEAD0);
03199         r = fun+FUNHEAD+ARGHEAD;
03200         if ( AR.PolyFunExp ==  2 ) {    /* Find minimum power */
03201             WORD minpow2 = MAXPOWER, *rrm;
03202             rrm = r;
03203             while ( rrm < tnext ) {
03204                 if ( *rrm == 4 ) {
03205                     if ( minpow2 > 0 ) minpow2 = 0;
03206                 }
03207                 else if ( ABS(rrm[*rrm-1]) == (*rrm-1) ) {
03208                     if ( minpow2 > 0 ) minpow2 = 0;
03209                 }
03210                 else {
03211                     if ( rrm[1] == SYMBOL && rrm[2] == 4 && rrm[3] == AR.PolyFunVar ) {
03212                         if ( rrm[4] < minpow2 ) minpow2 = rrm[4];
03213                     }
03214                     else { error = 5; goto onerror; }
03215                 }
03216                 rrm += *rrm;
03217             }
03218             AR.PolyFunPow += minpow2;
03219         }
03220         while ( r < tnext ) {
03221             rr = r + *r;
03222             i = *r; rrr = outarg; NCOPY(rrr,r,i);
03223             Normalize(BHEAD outarg);
03224             if ( *outarg > 0 ) StoreTerm(BHEAD outarg);
03225         }
03226         r = fun+FUNHEAD+ARGHEAD;
03227         EndSort(BHEAD r,1);
03228         AT.TrimPower = 0;
03229         if ( *r == 0 ) {
03230             fun[FUNHEAD] = -SNUMBER; fun[FUNHEAD+1] = 0;
03231             fun[1] = FUNHEAD+2;
03232         }
03233         else {
03234             rr = fun+FUNHEAD;
03235             if ( ToFast(rr,rr) ) {
03236                 NEXTARG(rr); fun[1] = rr - fun;
03237             }
03238             else {
03239                 while ( *r ) r += *r;
03240                 *rr = r-rr; rr[1] = CLEANFLAG;
03241                 fun[1] = r - fun;
03242             }
03243         }
03244         fun[2] = CLEANFLAG;
03245         goto done;
03246     }
03247 /*
03248     First test whether we have only AR.PolyFunVar in the denominator
03249 */
03250     tt = fun + FUNHEAD;
03251     arg1 = arg2 = NULL;
03252     if ( tt < tnext ) {
03253         arg1 = tt; NEXTARG(tt);
03254         if ( tt < tnext ) {
03255             arg2 = tt; NEXTARG(tt);
03256             if ( tt != tnext ) { arg1 = arg2 = NULL; } /* more than two arguments */
03257         }
03258     } else { error = 5; goto onerror; }
03259     if ( arg2 == NULL ) {
03260         if ( *arg1 < 0 ) { fun[2] = CLEANFLAG; goto done; }
03261         if ( fun[2] == CLEANFLAG ) goto done;
03262         goto NormArg;   /* Note: should not come here */
03263     }
03264 /*
03265     Produce the output argument in outarg
03266 */
03267     if ( outarg == NULL ) outarg = TermMalloc("ExpandRat")+ARGHEAD;
03268
03269     if ( *arg2 <= 0 ) {
03270 /*
03271         These cases are trivial.
03272         We try as much as possible to write the output directly into the
03273         function. We just have to be extremely careful not to overwrite
```

```
03274         relevant information before we are finished with it.
03275 */
03276         if ( *arg2 == -SYMBOL && arg2[1] == AR.PolyFunVar ) {
03277             rr = r = fun+FUNHEAD+ARGHEAD;
03278             if ( *arg1 < 0 ) {
03279                 if ( *arg1 == -SYMBOL ) {
03280                     if ( arg1[1] == AR.PolyFunVar ) {
03281                         *r++ = 4; *r++ = 1; *r++ = 1; *r++ = 3; *r = 0;
03282                     }
03283                     else {
03284                         *r++ = 10; *r++ = SYMBOL; *r++ = 6;
03285                         *r++ = arg1[1]; *r++ = 1;
03286                         *r++ = AR.PolyFunVar; *r++ = -1;
03287                         *r++ = 1; *r++ = 1; *r++ = 3; *r = 0;
03288                         Normalize(BHEAD rr);
03289                     }
03290                 }
03291                 else if ( *arg1 == -SNUMBER ) {
03292                     nco = arg1[1];
03293                     if ( nco == 0 ) { *r++ = 0; }
03294                     else {
03295                         *r++ = 8; *r++ = SYMBOL; *r++ = 4;
03296                         *r++ = AR.PolyFunVar; *r++ = -1;
03297                         *r++ = ABS(nco); *r++ = 1;
03298                         if ( nco < 0 ) *r++ = -3;
03299                         else *r++ = 3;
03300                         *r = 0;
03301                     }
03302                 }
03303                 else { error = 2; goto onerror; }  /* should not happen! */
03304             }
03305             else {  /* Multi-term numerator. */
03306                 m = arg1+ARGHEAD;
03307                 NewSort(BHEAD0); /* Technically maybe not needed */
03308                 if ( AR.PolyFunExp ==  2 ) {    /* Find minimum power */
03309                     WORD minpow2 = MAXPOWER, *rrm;
03310                     rrm = m;
03311                     while ( rrm < arg2 ) {
03312                         if ( *rrm == 4 ) {
03313                             if ( minpow2 > 0 ) minpow2 = 0;
03314                         }
03315                         else if ( ABS(rrm[*rrm-1]) == (*rrm-1) ) {
03316                             if ( minpow2 > 0 ) minpow2 = 0;
03317                         }
03318                         else {
03319                             if ( rrm[1] == SYMBOL && rrm[2] == 4 && rrm[3] == AR.PolyFunVar ) {
03320                                 if ( rrm[4] < minpow2 ) minpow2 = rrm[4];
03321                             }
03322                             else { error = 5; goto onerror; }
03323                         }
03324                         rrm += *rrm;
03325                     }
03326                     AR.PolyFunPow += minpow2-1;
03327                 }
03328                 while ( m < arg2 ) {
03329                     r = outarg;
03330                     rrr = r++; mm = m + *m;
03331                     *r++ = SYMBOL; *r++ = 4; *r++ = AR.PolyFunVar; *r++ = -1;
03332                     m++; while ( m < mm ) *r++ = *m++;
03333                     *rrr = r-rrr;
03334                     Normalize(BHEAD rrr);
03335                     StoreTerm(BHEAD rrr);
03336                 }
03337                 EndSort(BHEAD rr,1);
03338                 r = rr; while ( *r ) r += *r;
03339             }
03340             if ( *rr == 0 ) {
03341                 fun[FUNHEAD] = -SNUMBER; fun[FUNHEAD+1] = CLEANFLAG;
03342                 fun[1] = FUNHEAD+2;
03343             }
03344             else {
03345                 rr = fun+FUNHEAD;
03346                 *rr = r-rr;
03347                 rr[1] = CLEANFLAG;
03348                 if ( ToFast(rr,rr) ) {
03349                     NEXTARG(rr);
03350                     fun[1] = rr - fun;
03351                 }
03352                 else { fun[1] = r - fun; }
03353             }
03354             fun[2] = CLEANFLAG;
03355             goto done;
03356         }
03357         else if ( *arg2 == -SNUMBER ) {
03358             rr = r = outarg;
03359             if ( arg2[1] == 0 ) { error = 1; goto onerror; }
03360             if ( *arg1  == -SNUMBER ) { /* Things may not be normalized */
```

```
03361                 if ( arg1[1] == 0 ) { *r++ = 0; }
03362                 else {
03363                     co1[0] = ABS(arg1[1]); co1[1] = 1;
03364                     co2[0] = 1; co2[1] = ABS(arg2[1]);
03365                     MulRat(BHEAD co1,1,co2,1,co,&nco);
03366                     *r++ = 4; *r++ = (WORD)(co[0]); *r++ = (WORD)(co[1]);
03367                     if ( ( arg1[1] < 0 && arg2[1] > 0 ) ||
03368                          ( arg1[1] > 0 && arg2[1] < 0 ) ) *r++ = -3;
03369                     else *r++ = 3;
03370                     *r = 0;
03371                 }
03372             }
03373             else if ( *arg1 == -SYMBOL ) {
03374                 *r++ = 8; *r++ = SYMBOL; *r++ = 4;
03375                 *r++ = arg1[1]; *r++ = 1;
03376                 *r++ = 1; *r++ = ABS(arg2[1]);
03377                 if ( arg2[1] < 0 ) *r++ = -3;
03378                 else *r++ = 3;
03379                 *r = 0;
03380             }
03381             else if ( *arg1 < 0 ) { error = 2; goto onerror; }
03382             else {  /* Multi-term numerator. */
03383                 m = arg1+ARGHEAD;
03384                 NewSort(BHEAD0); /* Technically maybe not needed */
03385                 if ( AR.PolyFunExp ==  2 ) {    /* Find minimum power */
03386                     WORD minpow2 = MAXPOWER, *rrm;
03387                     rrm = m;
03388                     while ( rrm < arg2 ) {
03389                         if ( *rrm == 4 ) {
03390                             if ( minpow2 > 0 ) minpow2 = 0;
03391                         }
03392                         else if ( ABS(rrm[*rrm-1]) == (*rrm-1) ) {
03393                             if ( minpow2 > 0 ) minpow2 = 0;
03394                         }
03395                         else {
03396                             if ( rrm[1] == SYMBOL && rrm[2] == 4 && rrm[3] == AR.PolyFunVar ) {
03397                                 if ( rrm[4] < minpow2 ) minpow2 = rrm[4];
03398                             }
03399                             else { error = 5; goto onerror; }
03400                         }
03401                         rrm += *rrm;
03402                     }
03403                     AR.PolyFunPow += minpow2;
03404                 }
03405                 while ( m < arg2 ) {
03406                     r = rr;
03407                     rrr = r++; mm = m + *m;
03408                     *r++ = DENOMINATOR; *r++ = FUNHEAD + 2; *r++ = DIRTYFLAG;
03409                     FILLFUN3(r);
03410                     *r++ = arg2[0]; *r++ = arg2[1];
03411                     m++; while ( m < mm ) *r++ = *m++;
03412                     *rrr = r-rrr;
03413                     if ( r < AT.WorkTop && r >= AT.WorkSpace )
03414                                 AT.WorkPointer = r;
03415                     Normalize(BHEAD rrr);
03416                     if ( ABS(rrr[*rrr-1]) == *rrr-1 ) {
03417                         if ( AR.PolyFunPow >= 0 ) {
03418                             StoreTerm(BHEAD rrr);
03419                         }
03420                     }
03421                     else if ( rrr[1] == SYMBOL && rrr[2] == 4 &&
03422                     rrr[3] == AR.PolyFunVar && rrr[4] <= AR.PolyFunPow ) {
03423                         StoreTerm(BHEAD rrr);
03424                     }
03425                 }
03426                 EndSort(BHEAD rr,1);
03427             }
03428             r = rr; while ( *r ) r += *r;
03429             i = r-rr;
03430             r = fun + FUNHEAD + ARGHEAD;
03431             NCOPY(r,rr,i);
03432             rr = fun + FUNHEAD;
03433             *rr = r - rr; rr[1] = CLEANFLAG;
03434             if ( ToFast(rr,rr) ) {
03435                 NEXTARG(rr);
03436                 fun[1] = rr - fun;
03437             }
03438             else { fun[1] = r - fun; }
03439             fun[2] = CLEANFLAG;
03440             goto done;
03441         }
03442         else { error = 0; goto onerror; }
03443     }
03444     else {
03445         r = arg2+ARGHEAD; /* The argument ends at tnext */
03446         first = 1;
03447         while ( r < tnext ) {
```

```
03448             rr = r + *r; rr -= ABS(rr[-1]);
03449             if ( r+1 == rr ) {
03450                 if ( first ) { minpow = 0; first = 0; rmin = r; }
03451                 else if ( minpow > 0 ) { minpow = 0; rmin = r; }
03452             }
03453             else if ( r[1] != SYMBOL || r[2] != 4 || r[3] != AR.PolyFunVar
03454                 || r[4] > MAXPOWER ) { error = 0; goto onerror; }
03455             else if ( first ) { minpow = r[4]; first = 0; rmin = r; }
03456             else if ( r[4] < minpow ) { minpow = r[4]; rmin = r; }
03457             r += *r;
03458         }
03459 /*
03460         We have now:
03461         1: a numerator in arg1 which can contain several variables.
03462         2: a denominator in arg2 with at most only AR.PolyFunVar (ep).
03463         3: the minimum power in the denominator is minpow and the
03464            term with that minimum power is in rmin.
03465         Divide numerator and denominator by this minimum power.
03466         Determine the power range in the numerator.
03467         Call InvPoly.
03468         Multiply by the inverse in such a way that we never take more
03469         powers of ep than necessary.
03470 */
03471 /*
03472         One: put 1/rmin in AT.WorkPointer -> rmininv
03473 */
03474         AT.WorkPointer += AM.MaxTer/sizeof(WORD);
03475         if ( AT.WorkPointer + (AM.MaxTer/sizeof(WORD)) >= AT.WorkTop ) {
03476             error = 4; goto onerror;
03477         }
03478         rmininv = r = AT.WorkPointer;
03479         rr = rmin; i = *rmin; NCOPY(r,rr,i)
03480         if ( minpow != 0 ) { rmininv[4] = -rmininv[4]; }
03481         rsize = ABS(r[-1]);
03482         rcoef = r - rsize;
03483         rsize = (rsize-1)/2; rr = rcoef + rsize;
03484         for ( i = 0; i < rsize; i++ ) {
03485             rcopy = rcoef[i]; rcoef[i] = rr[i]; rr[i] = rcopy;
03486         }
03487         AT.WorkPointer = r;
03488         if ( *arg1 < 0 ) {
03489             ToGeneral(arg1,r,0);
03490             arg1 = r; r += *r; *r++ = 0; rcopy = 0;
03491             AT.WorkPointer = r;
03492         }
03493         else {
03494             r = arg1 + *arg1;
03495             rcopy = *r; *r++ = 0;
03496         }
03497 /*
03498         We can use MultiplyWithTerm.
03499 */
03500         AT.LeaveNegative = 1;
03501         numerator = MultiplyWithTerm(BHEAD arg1+ARGHEAD,rmininv,0);
03502         AT.LeaveNegative = 0;
03503         r[-1] = rcopy;
03504         r = numerator; while ( *r ) r += *r;
03505         AT.WorkPointer = r+1;
03506         rcopy = arg2[*arg2]; arg2[*arg2] = 0;
03507         denominator = MultiplyWithTerm(BHEAD arg2+ARGHEAD,rmininv,1);
03508         arg2[*arg2] = rcopy;
03509         r = denominator; while ( *r ) r += *r;
03510         AT.WorkPointer = r+1;
03511 /*
03512         Now find the minimum power of ep in the numerator.
03513 */
03514         r = numerator;
03515         first = 1;
03516         while ( *r ) {
03517             rr = r + *r; rr -= ABS(rr[-1]);
03518             if ( r+1 == rr ) {
03519                 if ( first ) { minpow = 0; first = 0; }
03520                 else if ( minpow > 0 ) { minpow = 0; }
03521             }
03522             else if ( r[1] != SYMBOL ) { error = 0; goto onerror; }
03523             else {
03524                 for ( i = 3; i < r[2]; i += 2 ) {
03525                     if ( r[i] == AR.PolyFunVar ) {
03526                         if ( first ) { minpow = r[i+1]; first = 0; }
03527                         else if ( r[i+1] < minpow ) minpow = r[i+1];
03528                         break;
03529                     }
03530                 }
03531                 if ( i >= r[2] ) {
03532                     if ( first ) { minpow = 0; first = 0; }
03533                     else if ( minpow > 0 ) minpow = 0;
03534                 }
```

```
03535             }
03536             r += *r;
03537         }
03538 /*
03539         We can invert the denominator.
03540         Note that the return value is an offset in AT.pWorkSpace.
03541         Hence there is no need to free memory afterwards.
03542 */
03543         if ( AR.PolyFunExp == 3 ) {
03544             ipoly = InvPoly(BHEAD denominator,AR.PolyFunPow-minpow,AR.PolyFunVar);
03545         }
03546         else {
03547             ipoly = InvPoly(BHEAD denominator,AR.PolyFunPow,AR.PolyFunVar);
03548         }
03549 /*
03550         Now we start the multiplying
03551 */
03552         NewSort(BHEAD0);
03553         r = numerator;
03554         while ( *r ) {
03555 /*
03556             1: Find power of ep.
03557 */
03558             rnext = r + *r;
03559             rrr = rnext - ABS(rnext[-1]);
03560             rr = r+1;
03561             eppow = 0;
03562             if ( rr < rrr ) {
03563                 j = rr[1] - 2; rr += 2;
03564                 while ( j > 0 ) {
03565                     if ( *rr == AR.PolyFunVar ) { eppow = rr[1]; break; }
03566                     j -= 2; rr += 2;
03567                 }
03568             }
03569 /*
03570             2: Multiply by the proper terms in ipoly
03571 */
03572             for ( i = 0; i <= AR.PolyFunPow-eppow+minpow; i++ ) {
03573                 if ( AT.pWorkSpace[ipoly+i] == 0 ) continue;
03574 /*
03575                 Copy the term, add i to the power of ep and multiply coef.
03576 */
03577                 rc = r;
03578                 rr = thecopy = AT.WorkPointer;
03579                 while ( rc < rrr ) *rr++ = *rc++;
03580                 if ( i != 0 ) {
03581                     *rr++ = SYMBOL; *rr++ = 4; *rr++ = AR.PolyFunVar; *rr++ = i;
03582                 }
03583                 ncoef = REDLENG(rnext[-1]);
03584                 MulRat(BHEAD (UWORD *)rrr,ncoef,
03585                     (UWORD *)(AT.pWorkSpace[ipoly+i])+1,AT.pWorkSpace[ipoly+i][0]
03586                     ,(UWORD *)rr,&newcoef);
03587                 ncoef = ABS(newcoef); rr += 2*ncoef;
03588                 newcoef = INCLENG(newcoef);
03589                 *rr++ = newcoef;
03590                 *thecopy = rr - thecopy;
03591                 AT.WorkPointer = rr;
03592                 Normalize(BHEAD thecopy);
03593                 if ( *thecopy > 0 ) StoreTerm(BHEAD thecopy);
03594                 AT.WorkPointer = thecopy;
03595             }
03596             r = rnext;
03597         }
03598 /*
03599         Now we have all.
03600 */
03601         rr = fun + FUNHEAD; r = rr + ARGHEAD;
03602         EndSort(BHEAD r,1);
03603         if ( *r == 0 ) {
03604             fun[1] = FUNHEAD+2; fun[2] = CLEANFLAG;
03605             fun[FUNHEAD] = -SNUMBER; fun[FUNHEAD+1] = 0;
03606         }
03607         else {
03608             while ( *r ) r += *r;
03609             rr[0] = r-rr; rr[1] = CLEANFLAG;
03610             if ( ToFast(rr,rr) ) { NEXTARG(rr); fun[1] = rr-fun; }
03611             else { fun[1] = r-fun; }
03612             fun[2] = CLEANFLAG;
03613         }
03614     }
03615 done:
03616     if ( outarg ) TermFree(outarg-ARGHEAD,"ExpandRat");
03617     AR.PolyFunPow = OldPolyFunPow;
03618     AT.WorkPointer = ow;
03619     AN.PolyNormFlag = 1;
03620     return(0);
03621 onerror:
```

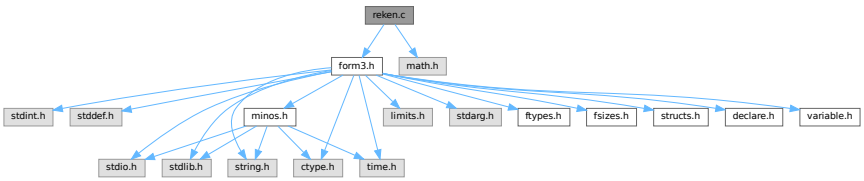
```
03622     if ( outarg ) TermFree(outarg-ARGHEAD,"ExpandRat");
03623     AR.PolyFunPow = OldPolyFunPow;
03624     AT.WorkPointer = ow;
03625     MLOCK(ErrorMessageLock);
03626     MesPrint(TheErrorMessage[error]);
03627     MUNLOCK(ErrorMessageLock);
03628     Terminate(-1);
03629     return(-1);
03630 }
03631
03632 /*
03633     #] ExpandRat :
03634     #[ InvPoly :
03635
03636     The input polynomial is represented as a sequence of terms in ascending
03637     power. The first coefficient is 1. If we call this 1-a and
03638     a = sum_(j,1,n,x^j*a(j)), and b = 1/(1-a) we can find the coefficients
03639     of b with the recursion
03640         b(0) = 1, b(n) = sum_(j,1,n,a(j)*b(n-j))
03641     The variable is the symbol sym and we need maxpow powers in the answer.
03642     The answer is an array of pointers to the coefficients of the various
03643     powers as rational numbers in the notation signedsize,numerator,denominator
03644     We put these powers in the workspace and the answer is in AT.pWorkSpace.
03645     Hence the return value is an offset in the pWorkSpace.
03646     A zero pointer indicates that this coefficient is zero.
03647 */
03648
03649 int InvPoly(PHEAD WORD *inpoly, WORD maxpow, WORD sym)
03650 {
03651     int needed, inpointers, outpointers, maxinput = 0, i, j;
03652     WORD *t, *tt, *ttt, *w, *c, *cc, *ccc, lenc, lenc1, lenc2, rc, *c1, *c2;
03653 /*
03654     Step 0: allocate the space
03655 */
03656     needed = (maxpow+1)*2;
03657     WantAddPointers(needed);
03658     inpointers = AT.pWorkPointer;
03659     outpointers = AT.pWorkPointer+maxpow+1;
03660     for ( i = 0; i < needed; i++ ) AT.pWorkSpace[inpointers+i] = 0;
03661 /*
03662     Step 1: determine the coefficients in inpoly
03663             often there is a maximum power that is much smaller than maxpow.
03664             keeping track of this can speed up things.
03665 */
03666     t = inpoly;
03667     w = AT.WorkPointer;
03668     while ( *t ) {
03669         if ( *t == 4 ) {
03670             if ( t[1] != 1 || t[2] != 1 || t[3] != 3 ) goto onerror;
03671             AT.pWorkSpace[inpointers] = 0;
03672         }
03673         else if ( t[1] != SYMBOL || t[2] != 4 || t[3] != sym || t[4] < 0 ) goto onerror;
03674         else if ( t[4] > maxpow ) {} /* power outside useful range */
03675         else {
03676             if ( t[4] > maxinput ) maxinput = t[4];
03677             AT.pWorkSpace[inpointers+t[4]] = w;
03678             tt = t + *t; rc = -*--tt; /* we need - the coefficient! */
03679             rc = REDLENG(rc); *w++ = rc;
03680             ttt = t+5;
03681             while ( ttt < tt ) *w++ = *ttt++;
03682         }
03683         t += *t;
03684     }
03685 /*
03686     Step 2: compute the output. b(0) = 1.
03687     then the recursion starts.
03688 */
03689     AT.pWorkSpace[outpointers] = w;
03690     *w++ = 1; *w++ = 1; *w++ = 1;
03691     c  = TermMalloc("InvPoly");
03692     c1 = TermMalloc("InvPoly");
03693     c2 = TermMalloc("InvPoly");
03694     for ( j = 1; j <= maxpow; j++ ) {
03695 /*
03696         Start at c = a(j)*b(0) = a(j)
03697 */
03698         if ( ( cc = AT.pWorkSpace[inpointers+j] ) != 0 ) {
03699             lenc = *cc++; /* reduced length */
03700             i = 2*ABS(lenc); ccc = c;
03701             NCOPY(ccc,cc,i);
03702         }
03703         else { lenc = 0; }
03704         for ( i = MiN(j-1,maxinput); i > 0; i-- ) {
03705 /*
03706             c -> c + a(i)*b(j-i)
03707 */
03708             if ( AT.pWorkSpace[inpointers+i] == 0
```

```
03709                   || AT.pWorkSpace[outpointers+j-i] == 0 ) {
03710               }
03711               else {
03712                   if ( MulRat(BHEAD (UWORD
    *)(AT.pWorkSpace[inpointers+i]+1),AT.pWorkSpace[inpointers+i][0],
03713                    (UWORD *)(AT.pWorkSpace[outpointers+j-i]+1),AT.pWorkSpace[outpointers+j-i][0],
03714                    (UWORD *)c1,&lenc1) ) goto calcerror;
03715                   if ( lenc == 0 ) {
03716                       cc = c; c = c1; c1 = cc;
03717                       lenc = lenc1;
03718                   }
03719                   else {
03720                       if ( AddRat(BHEAD (UWORD *)c,lenc,(UWORD *)c1,lenc1,(UWORD *)c2,&lenc2) )
03721                           goto calcerror;
03722                       cc = c; c = c2; c2 = cc;
03723                       lenc = lenc2;
03724                   }
03725               }
03726           }
03727 /*
03728           Copy c to the proper location
03729 */
03730           if ( lenc == 0 ) AT.pWorkSpace[outpointers+j] = 0;
03731           else {
03732               AT.pWorkSpace[outpointers+j] = w;
03733               *w++ = lenc;
03734               i = 2*ABS(lenc); ccc = c;
03735               NCOPY(w,ccc,i);
03736           }
03737       }
03738       AT.WorkPointer = w;
03739       TermFree(c2,"InvPoly");
03740       TermFree(c1,"InvPoly");
03741       TermFree(c ,"InvPoly");
03742
03743       return(outpointers);
03744 onerror:
03745       MLOCK(ErrorMessageLock);
03746       MesPrint("Incorrect symbol field in InvPoly.");
03747       MUNLOCK(ErrorMessageLock);
03748       Terminate(-1);
03749       return(-1);
03750 calcerror:
03751       MLOCK(ErrorMessageLock);
03752       MesPrint("Called from InvPoly.");
03753       MUNLOCK(ErrorMessageLock);
03754       Terminate(-1);
03755       return(-1);
03756 }
03757
03758 /*
03759       #] InvPoly :
03760 */
```

4.121 reken.c File Reference

```
#include "form3.h"
#include <math.h>
```

Include dependency graph for reken.c:



Macros

- #define GCDMAX 3

- #define NEWTRICK 1
- #define COPYLONG(x1, nx1, x2, nx2) { int i; for(i=0;i<ABS(nx2);i++)x1[i]=x2[i];nx1=nx2; }
- #define WARMUP 6

Functions

- void Pack (UWORD *a, WORD *na, UWORD *b, WORD nb)
- void UnPack (UWORD *a, WORD na, WORD *denom, WORD *numer)
- int Mully (PHEAD UWORD *a, WORD *na, UWORD *b, WORD nb)
- int Divvy (PHEAD UWORD *a, WORD *na, UWORD *b, WORD nb)
- int AddRat (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- int MulRat (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- int DivRat (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- int Simplify (PHEAD UWORD *a, WORD *na, UWORD *b, WORD *nb)
- int AccumGCD (PHEAD UWORD *a, WORD *na, UWORD *b, WORD nb)
- int TakeRatRoot (UWORD *a, WORD *n, WORD power)
- int AddLong (UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- int AddPLon (UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- void SubPLon (UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- int MulLong (UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- int BigLong (UWORD *a, WORD na, UWORD *b, WORD nb)
- int DivLong (UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc, UWORD *d, WORD *nd)
- int RaisPow (PHEAD UWORD *a, WORD *na, UWORD b)
- void RaisPowCached (PHEAD WORD x, WORD n, UWORD **c, WORD *nc)
- WORD RaisPowMod (WORD x, WORD n, WORD m)
- int NormalModulus (UWORD *a, WORD *na)
- int MakeInverses (void)
- int GetModInverses (WORD m1, WORD m2, WORD *im1, WORD *im2)
- int GetLongModInverses (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *ia, WORD *nia, UWORD *ib, WORD *nib)
- int Product (UWORD *a, WORD *na, WORD b)
- UWORD Quotient (UWORD *a, WORD *na, WORD b)
- WORD Remain10 (UWORD *a, WORD *na)
- WORD Remain4 (UWORD *a, WORD *na)
- void PrtLong (UWORD *a, WORD na, UBYTE *s)
- int GetLong (UBYTE *s, UWORD *a, WORD *na)
- int GcdLong (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- int GetBinom (UWORD *a, WORD *na, WORD i1, WORD i2)
- int LcmLong (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- int TakeLongRoot (UWORD *a, WORD *n, WORD power)
- int MakeRational (WORD a, WORD m, WORD *b, WORD *c)
- int MakeLongRational (PHEAD UWORD *a, WORD na, UWORD *m, WORD nm, UWORD *b, WORD *nb)
- WORD CompCoef (WORD *term1, WORD *term2)
- int Modulus (WORD *term)
- int TakeModulus (UWORD *a, WORD *na, UWORD *cmodvec, WORD ncmod, WORD par)
- int TakeNormalModulus (UWORD *a, WORD *na, UWORD *c, WORD nc, WORD par)
- int MakeModTable (void)
- int Factorial (PHEAD WORD n, UWORD *a, WORD *na)
- int Bernoulli (WORD n, UWORD *a, WORD *na)
- WORD NextPrime (PHEAD WORD num)
- WORD Moebius (PHEAD WORD nn)
- void iniwranf (PHEAD0)
- UWORD wranf (PHEAD0)
- UWORD iranf (PHEAD UWORD imax)
- UBYTE * PreRandom (UBYTE *s)

4.121.1 Detailed Description

This file contains the numerical routines. The arithmetic in FORM is normally over the rational numbers. Hence there are routines for dealing with integers and with rational of 'arbitrary precision' (within limits) There are also routines for that calculus modulus an integer. In addition there are the routines for factorials and Bernoulli numbers. The random number function is currently only for internal purposes.

Definition in file reken.c.

4.121.2 Macro Definition Documentation

4.121.2.1 GCDMAX

```
#define GCDMAX 3
```

Definition at line 49 of file reken.c.

4.121.2.2 NEWTRICK

```
#define NEWTRICK 1
```

Definition at line 51 of file reken.c.

4.121.2.3 COPYLONG

```
#define COPYLONG(
              x1,
              nx1,
              x2,
              nx2 ) { int i; for(i=0;i<ABS(nx2);i++)x1[i]=x2[i];nx1=nx2; }
```

Definition at line 2920 of file reken.c.

4.121.2.4 WARMUP

```
#define WARMUP 6
```

Definition at line 3824 of file reken.c.

4.121.3 Function Documentation

4.121.3.1 Pack()

```
void Pack (
             UWORD * a,
             WORD * na,
             UWORD * b,
             WORD nb )
```

Definition at line 62 of file reken.c.

4.121.3.2 UnPack()

```
void UnPack (
             UWORD * a,
             WORD na,
             WORD * denom,
             WORD * numer )
```

Definition at line 104 of file reken.c.

4.121.3.3 Mully()

```
int Mully (
             PHEAD UWORD * a,
             WORD * na,
             UWORD * b,
             WORD nb )
```

Definition at line 130 of file reken.c.

4.121.3.4 Divvy()

```
int Divvy (
             PHEAD UWORD * a,
             WORD * na,
             UWORD * b,
             WORD nb )
```

Definition at line 176 of file reken.c.

4.121.3.5 AddRat()

```
int AddRat (
             PHEAD UWORD * a,
             WORD na,
             UWORD * b,
             WORD nb,
             UWORD * c,
             WORD * nc )
```

Definition at line 216 of file reken.c.

4.121.3.6 MulRat()

```
int MulRat (
             PHEAD UWORD * a,
             WORD na,
             UWORD * b,
             WORD nb,
             UWORD * c,
             WORD * nc )
```

Definition at line 384 of file reken.c.

4.121.3.7 DivRat()

```
int DivRat (
              PHEAD UWORD * a,
              WORD na,
              UWORD * b,
              WORD nb,
              UWORD * c,
              WORD * nc )
```

Definition at line 506 of file reken.c.

4.121.3.8 Simplify()

```
int Simplify (
              PHEAD UWORD * a,
              WORD * na,
              UWORD * b,
              WORD * nb )
```

Definition at line 539 of file reken.c.

4.121.3.9 AccumGCD()

```
int AccumGCD (
              PHEAD UWORD * a,
              WORD * na,
              UWORD * b,
              WORD nb )
```

Definition at line 656 of file reken.c.

4.121.3.10 TakeRatRoot()

```
int TakeRatRoot (
              UWORD * a,
              WORD * n,
              WORD power )
```

Definition at line 689 of file reken.c.

4.121.3.11 AddLong()

```
int AddLong (
              UWORD * a,
              WORD na,
              UWORD * b,
              WORD nb,
              UWORD * c,
              WORD * nc )
```

Definition at line 714 of file reken.c.

4.121.3.12 AddPLon()

```
int AddPLon (
              UWORD * a,
              WORD na,
              UWORD * b,
              WORD nb,
              UWORD * c,
              WORD * nc )
```

Definition at line 759 of file reken.c.

4.121.3.13 SubPLon()

```
void SubPLon (
              UWORD * a,
              WORD na,
              UWORD * b,
              WORD nb,
              UWORD * c,
              WORD * nc )
```

Definition at line 812 of file reken.c.

4.121.3.14 MulLong()

```
int MulLong (
              UWORD * a,
              WORD na,
              UWORD * b,
              WORD nb,
              UWORD * c,
              WORD * nc )
```

Definition at line 850 of file reken.c.

4.121.3.15 BigLong()

```
int BigLong (
              UWORD * a,
              WORD na,
              UWORD * b,
              WORD nb )
```

Definition at line 953 of file reken.c.

4.121.3.16 DivLong()

```
int DivLong (
              UWORD * a,
              WORD na,
              UWORD * b,
              WORD nb,
              UWORD * c,
              WORD * nc,
              UWORD * d,
              WORD * nd )
```

Definition at line 981 of file reken.c.

4.121.3.17 RaisPow()

```
int RaisPow (
              PHEAD UWORD * a,
              WORD * na,
              UWORD b )
```

Definition at line 1241 of file reken.c.

4.121.3.18 RaisPowCached()

```
void RaisPowCached (
              PHEAD WORD x,
              WORD n,
              UWORD ** c,
              WORD * nc )
```

Computes power x^n and caches the value

4.121.4 Description

Calculates the power x^n and stores the results for caching purposes. The pointer c (i.e., the pointer, and not what it points to) is overwritten. What it points to should not be overwritten in the calling function.

4.121.5 Notes

- Caching is done in AT.small_power[]. This array is extended if necessary.

Definition at line 1309 of file reken.c.

Referenced by poly::divmod_heap(), poly::divmod_univar(), and poly::mul_heap().

4.121.5.1 RaisPowMod()

```
WORD RaisPowMod (
              WORD x,
              WORD n,
              WORD m )
```

Definition at line 1396 of file reken.c.

4.121.5.2 NormalModulus()

```
int NormalModulus (
              UWORD * a,
              WORD * na )
```

Brings a modular representation in the range -p/2 to +p/2 The return value tells whether anything was done. Routine made in the general modulus revamp of July 2008 (JV).

Definition at line 1416 of file reken.c.

Referenced by AddCoef(), and MergePatches().

4.121.5.3 MakeInverses()

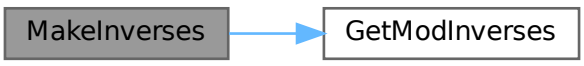
```
int MakeInverses (
              void  )
```

Makes a table of inverses in modular calculus The modulus is in AC.cmod and AC.ncmod One should notice that the table of inverses can only be made if the modulus fits inside a single FORM word. Otherwise the table lookup becomes too difficult and the table too long.

Definition at line 1453 of file reken.c.

References GetModInverses().

Here is the call graph for this function:



4.121.5.4 GetModInverses()

```
int GetModInverses (
              WORD m1,
              WORD m2,
              WORD * im1,
              WORD * im2 )
```

Input m1 and m2, which are relative prime. determines a∗m1+b∗m2 = 1 (and 1 is the gcd of m1 and m2) then a∗m1 = 1 mod m2 and hence im1 = a. and b∗m2 = 1 mod m1 and hence im2 = b. Set m1 = 0∗m1+1∗m2 = a1∗m1+b1∗m2 m2 = 1∗m1+0∗m2 = a2∗m1+b2∗m2 If everything is OK, the return value is zero

Definition at line 1489 of file reken.c.

Referenced by poly::divmod_heap(), poly::divmod_univar(), MakeDollarMod(), MakeInverses(), MakeMod(), TakeContent(), and TakeSymbolContent().

4.121.5.5 GetLongModInverses()

```
int GetLongModInverses (
              PHEAD UWORD * a,
              WORD na,
              UWORD * b,
              WORD nb,
              UWORD * ia,
              WORD * nia,
              UWORD * ib,
              WORD * nib )
```

Definition at line 1523 of file reken.c.

4.121.5.6 Product()

```
int Product (
              UWORD * a,
              WORD * na,
              WORD b )
```

Definition at line 1600 of file reken.c.

4.121.5.7 Quotient()

```
UWORD Quotient (
              UWORD * a,
              WORD * na,
              WORD b )
```

Definition at line 1635 of file reken.c.

4.121.5.8 Remain10()

```
WORD Remain10 (
             UWORD * a,
             WORD * na )
```

Definition at line 1674 of file reken.c.

4.121.5.9 Remain4()

```
WORD Remain4 (
             UWORD * a,
             WORD * na )
```

Definition at line 1701 of file reken.c.

4.121.5.10 PrtLong()

```
void PrtLong (
             UWORD * a,
             WORD na,
             UBYTE * s )
```

Definition at line 1727 of file reken.c.

4.121.5.11 GetLong()

```
int GetLong (
             UBYTE * s,
             UWORD * a,
             WORD * na )
```

Definition at line 1789 of file reken.c.

4.121.5.12 GcdLong()

```
int GcdLong (
             PHEAD UWORD * a,
             WORD na,
             UWORD * b,
             WORD nb,
             UWORD * c,
             WORD * nc )
```

Definition at line 2293 of file reken.c.

4.121.5.13 GetBinom()

```
int GetBinom (
              UWORD * a,
              WORD * na,
              WORD i1,
              WORD i2 )
```

Definition at line 2686 of file reken.c.

4.121.5.14 LcmLong()

```
int LcmLong (
              PHEAD UWORD * a,
              WORD na,
              UWORD * b,
              WORD nb,
              UWORD * c,
              WORD * nc )
```

Definition at line 2720 of file reken.c.

4.121.5.15 TakeLongRoot()

```
int TakeLongRoot (
              UWORD * a,
              WORD * n,
              WORD power )
```

Definition at line 2754 of file reken.c.

4.121.5.16 MakeRational()

```
int MakeRational (
              WORD a,
              WORD m,
              WORD * b,
              WORD * c )
```

Definition at line 2874 of file reken.c.

4.121.5.17 MakeLongRational()

```
int MakeLongRational (
              PHEAD UWORD * a,
              WORD na,
              UWORD * m,
              WORD nm,
              UWORD * b,
              WORD * nb )
```

Definition at line 2922 of file reken.c.

4.121.5.18 CompCoef()

```
WORD CompCoef (
             WORD * term1,
             WORD * term2 )
```

```
Routine takes a1 mod m1 and a2 mod m2 and returns a mod m1*m2 with
a mod m1 = a1 and a mod m2 = a2
```

Chinese remainder: a%(m1∗m2) = q1∗m1+a1 a%(m1∗m2) = q2∗m2+a2 Compute n1 such that (n1∗m1)m2 is one Compute n2 such that (n2∗m2)m1 is one Then (a1∗n2∗m2+a2∗n1∗m1)%(m1∗m2) is a%(m1∗m2)

Definition at line 3066 of file reken.c.

Referenced by Compare1().

4.121.5.19 Modulus()

```
int Modulus (
             WORD * term )
```

Definition at line 3121 of file reken.c.

4.121.5.20 TakeModulus()

```
int TakeModulus (
             UWORD * a,
             WORD * na,
             UWORD * cmodvec,
             WORD ncmod,
             WORD par )
```

Definition at line 3168 of file reken.c.

4.121.5.21 TakeNormalModulus()

```
int TakeNormalModulus (
             UWORD * a,
             WORD * na,
             UWORD * c,
             WORD nc,
             WORD par )
```

Definition at line 3340 of file reken.c.

4.121.5.22 MakeModTable()

```
int MakeModTable (
             void  )
```

Definition at line 3382 of file reken.c.

4.121.5.23 Factorial()

```
int Factorial (
              PHEAD WORD n,
              UWORD * a,
              WORD * na )
```

Definition at line 3470 of file reken.c.

4.121.5.24 Bernoulli()

```
int Bernoulli (
              WORD n,
              UWORD * a,
              WORD * na )
```

Definition at line 3550 of file reken.c.

4.121.5.25 NextPrime()

```
WORD NextPrime (
              PHEAD WORD num )
```

Gives the next prime number in the list of prime numbers.

If the list isn't long enough we expand it. For ease in ParForm and because these lists shouldn't be very big we let each worker keep its own list.

The list is cut off at MAXPOWER, because we don't want to get into trouble that the power of a variable gets larger than the prime number.

Definition at line 3685 of file reken.c.

4.121.5.26 Moebius()

```
WORD Moebius (
              PHEAD WORD nn )
```

Definition at line 3751 of file reken.c.

4.121.5.27 iniwranf()

```
void iniwranf (
              PHEAD0  )
```

Definition at line 3842 of file reken.c.

4.121.5.28 wranf()

```
UWORD wranf (
            PHEAD0  )
```

Definition at line 3891 of file reken.c.

4.121.5.29 iranf()

```
UWORD iranf (
            PHEAD UWORD imax )
```

Definition at line 3910 of file reken.c.

4.121.5.30 PreRandom()

```
UBYTE * PreRandom (
            UBYTE * s )
```

Definition at line 3935 of file reken.c.

4.122 reken.c

Go to the documentation of this file.

```
00001
00011 /* #[ License : */
00012 /*
00013  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00014  *   When using this file you are requested to refer to the publication
00015  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00016  *   This is considered a matter of courtesy as the development was paid
00017  *   for by FOM the Dutch physics granting agency and we would like to
00018  *   be able to track its scientific use to convince FOM of its value
00019  *   for the community.
00020  *
00021  *   This file is part of FORM.
00022  *
00023  *   FORM is free software: you can redistribute it and/or modify it under the
00024  *   terms of the GNU General Public License as published by the Free Software
00025  *   Foundation, either version 3 of the License, or (at your option) any later
00026  *   version.
00027  *
00028  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00029  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00030  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00031  *   details.
00032  *
00033  *   You should have received a copy of the GNU General Public License along
00034  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00035  */
00036 /* #] License : */
00037 /*
00038     #[ Includes : reken.c
00039 */
00040
00041 #include "form3.h"
00042 #include <math.h>
00043
00044 #ifdef WITHGMP
00045 #include <gmp.h>
00046 #define GMPSPREAD (GMP_LIMB_BITS/BITSINWORD)
00047 #endif
00048
00049 #define GCDMAX 3
00050
00051 #define NEWTRICK 1
```

```
00052 /*
00053     #] Includes :
00054     #[ RekenRational :
00055         #[ Pack :           void Pack(a,na,b,nb)
00056
00057     Packs the contents of the numerator a and the denominator b into
00058     one normalized fraction a.
00059
00060 */
00061
00062 void Pack(UWORD *a, WORD *na, UWORD *b, WORD nb)
00063 {
00064     WORD c, sgn = 1, i;
00065     UWORD *to,*from;
00066     if ( (c = *na) == 0 ) {
00067 /* INTERNAL_ERROR_EXCL_START */
00068         MLOCK(ErrorMessageLock);
00069         MesPrint("!>Caught a zero in Pack");
00070         MUNLOCK(ErrorMessageLock);
00071         return;
00072 /* INTERNAL_ERROR_EXCL_STOP */
00073     }
00074     if ( nb == 0 ) {
00075 /* INTERNAL_ERROR_EXCL_START */
00076         MLOCK(ErrorMessageLock);
00077         MesPrint("!>Division by zero in Pack");
00078         MUNLOCK(ErrorMessageLock);
00079         return;
00080 /* INTERNAL_ERROR_EXCL_STOP */
00081     }
00082     if ( *na < 0 ) { sgn = -sgn; c = -c; }
00083     if ( nb < 0 ) { sgn = -sgn; nb = -nb; }
00084     *na = MaX(c,nb);
00085     to = a + c;
00086     i = *na - c;
00087     while ( --i >= 0 ) *to++ = 0;
00088     i = *na - nb;
00089     from = b;
00090     NCOPY(to,from,nb);
00091     while ( --i >= 0 ) *to++ = 0;
00092     if ( sgn < 0 ) *na = -*na;
00093 }
00094
00095 /*
00096         #] Pack :
00097         #[ UnPack :         void UnPack(a,na,denom,numer)
00098
00099     Determines the sizes of the numerator and the denominator in the
00100     normalized fraction a with length na.
00101
00102 */
00103
00104 void UnPack(UWORD *a, WORD na, WORD *denom, WORD *numer)
00105 {
00106     UWORD *pos;
00107     WORD i, sgn = na;
00108     if ( na < 0 ) { na = -na; }
00109     i = na;
00110     if ( i > 1 ) {      /* Find the respective leading words */
00111         a += i;
00112         a--;
00113         pos = a + i;
00114         while ( !(*a) ) { i--; a--; }
00115         while ( !(*pos) ) { na--; pos--; }
00116     }
00117     *denom = na;
00118     if ( sgn < 0 ) i = -i;
00119     *numer = i;
00120 }
00121
00122 /*
00123         #] UnPack :
00124         #[ Mully :          WORD Mully(a,na,b,nb)
00125
00126     Multiplies the rational a by the Long b.
00127
00128 */
00129
00130 int Mully(PHEAD UWORD *a, WORD *na, UWORD *b, WORD nb)
00131 {
00132     GETBIDENTITY
00133     UWORD *d, *e;
00134     WORD i, sgn = 1;
00135     WORD nd, ne, adenom, anumer;
00136     if ( !nb ) { *na = 0; return(0); }
00137     else if ( *b == 1 ) {
00138         if ( nb == 1 ) return(0);
```

```
00139         else if ( nb == -1 ) { *na = -*na; return(0); }
00140     }
00141     if ( *na < 0 ) { sgn = -sgn; *na = -*na; }
00142     if ( nb < 0 ) { sgn = -sgn; nb = -nb; }
00143     UnPack(a,*na,&adenom,&anumer);
00144     d = NumberMalloc("Mully"); e = NumberMalloc("Mully");
00145     for ( i = 0; i < nb; i++ ) { e[i] = *b++; }
00146     ne = nb;
00147     if ( Simplify(BHEAD a+*na,&adenom,e,&ne) ) goto MullyEr;
00148     if ( MulLong(a,anumer,e,ne,d,&nd) ) goto MullyEr;
00149     b = a+*na;
00150     for ( i = 0; i < *na; i++ ) { e[i] = *b++; }
00151     ne = adenom;
00152     *na = nd;
00153     b = d;
00154     *na = nd;
00155     for ( i = 0; i < *na; i++ ) { a[i] = *b++; }
00156     Pack(a,na,e,ne);
00157     if ( sgn < 0 ) *na = -*na;
00158     NumberFree(d,"Mully"); NumberFree(e,"Mully");
00159     return(0);
00160 MullyEr:
00161     MLOCK(ErrorMessageLock);
00162     MesCall("Mully");
00163     MUNLOCK(ErrorMessageLock);
00164     NumberFree(d,"Mully"); NumberFree(e,"Mully");
00165     SETERROR(-1)
00166 }
00167
00168 /*
00169         #] Mully :
00170         #[ Divvy :          WORD Divvy(a,na,b,nb)
00171
00172     Divides the rational a by the Long b.
00173
00174 */
00175
00176 int Divvy(PHEAD UWORD *a, WORD *na, UWORD *b, WORD nb)
00177 {
00178     GETBIDENTITY
00179     UWORD *d,*e;
00180     WORD i, sgn = 1;
00181     WORD nd, ne, adenom, anumer;
00182     if ( !nb ) {
00183 /* INTERNAL_ERROR_EXCL_START */
00184         MLOCK(ErrorMessageLock);
00185         MesPrint("!>Division by zero in Divvy");
00186         MUNLOCK(ErrorMessageLock);
00187         return(-1);
00188 /* INTERNAL_ERROR_EXCL_STOP */
00189     }
00190     d = NumberMalloc("Divvy"); e = NumberMalloc("Divvy");
00191     if ( nb < 0 ) { sgn = -sgn; nb = -nb; }
00192     if ( *na < 0 ) { sgn = -sgn; *na = -*na; }
00193     UnPack(a,*na,&adenom,&anumer);
00194     for ( i = 0; i < nb; i++ ) { e[i] = *b++; }
00195     ne = nb;
00196     if ( Simplify(BHEAD a,&anumer,e,&ne) ) goto DivvyEr;
00197     if ( MulLong(a+*na,adenom,e,ne,d,&nd) ) goto DivvyEr;
00198     *na = anumer;
00199     Pack(a,na,d,nd);
00200     if ( sgn < 0 ) *na = -*na;
00201     NumberFree(d,"Divvy"); NumberFree(e,"Divvy");
00202     return(0);
00203 DivvyEr:
00204     MLOCK(ErrorMessageLock);
00205     MesCall("Divvy");
00206     MUNLOCK(ErrorMessageLock);
00207     NumberFree(d,"Divvy"); NumberFree(e,"Divvy");
00208     SETERROR(-1)
00209 }
00210
00211 /*
00212         #] Divvy :
00213         #[ AddRat :          WORD AddRat(a,na,b,nb,c,nc)
00214 */
00215
00216 int AddRat(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
00217 {
00218     GETBIDENTITY
00219     UWORD *d, *e, *f, *g;
00220     WORD nd, ne, nf, ng, adenom, anumer, bdenom, bnumer;
00221     if ( !na ) {
00222         WORD i;
00223         *nc = nb;
00224         if ( nb < 0 ) nb = -nb;
00225         nb *= 2;
```

```
00226         for ( i = 0; i < nb; i++ ) *c++ = *b++;
00227         return(0);
00228     }
00229     else if ( !nb ) {
00230         WORD i;
00231         *nc = na;
00232         if ( na < 0 ) na = -na;
00233         na *= 2;
00234         for ( i = 0; i < na; i++ ) *c++ = *a++;
00235         return(0);
00236     }
00237     else if ( b[1] == 1 && a[1] == 1 ) {
00238         if ( na == 1 ) {
00239             if ( nb == 1 ) {
00240                 *c = *a + *b;
00241                 c[1] = 1;
00242                 if ( *c < *a ) { c[2] = 1; c[3] = 0; *nc = 2; }
00243                 else { *nc = 1; }
00244                 return(0);
00245             }
00246             else if ( nb == -1 ) {
00247                 if ( *b > *a ) {
00248                     *c = *b - *a; *nc = -1;
00249                 }
00250                 else if ( *b < *a ) {
00251                     *c = *a - *b; *nc = 1;
00252                 }
00253                 else *nc = 0;
00254                 c[1] = 1;
00255                 return(0);
00256             }
00257         }
00258         else if ( na == -1 ){
00259             if ( nb == -1 ) {
00260                 c[1] = 1;
00261                 *c = *a + *b;
00262                 if ( *c < *a ) { c[2] = 1; c[3] = 0; *nc = -2; }
00263                 else { *nc = -1; }
00264                 return(0);
00265             }
00266             else if ( nb == 1 ) {
00267                 if ( *b > *a ) {
00268                     *c = *b - *a; *nc = 1;
00269                 }
00270                 else if ( *b < *a ) {
00271                     *c = *a - *b; *nc = -1;
00272                 }
00273                 else *nc = 0;
00274                 c[1] = 1;
00275                 return(0);
00276             }
00277         }
00278     }
00279     UnPack(a,na,&adenom,&anumer);
00280     UnPack(b,nb,&bdenom,&bnumer);
00281     if ( na < 0 ) na = -na;
00282     if ( nb < 0 ) nb = -nb;
00283     if ( na == 1 && nb == 1 ) {
00284         RLONG t1, t2, t3;
00285         t3 = ((RLONG)a[1])*((RLONG)b[1]);
00286         t1 = ((RLONG)a[0])*((RLONG)b[1]);
00287         t2 = ((RLONG)a[1])*((RLONG)b[0]);
00288         if ( ( anumer > 0 && bnumer > 0 ) || ( anumer < 0 && bnumer < 0 ) ) {
00289             if ( ( t1 = t1 + t2 ) < t2 ) {
00290                 c[2] = 1;
00291                 c[0] = (UWORD)t1;
00292                 c[1] = (UWORD)(t1 » BITSINWORD);
00293                 *nc = 3;
00294             }
00295             else {
00296                 c[0] = (UWORD)t1;
00297                 if ( ( c[1] = (UWORD)(t1 » BITSINWORD) ) != 0 ) *nc = 2;
00298                 else *nc = 1;
00299             }
00300         }
00301         else {
00302             if ( t1 == t2 ) { *nc = 0; return(0); }
00303             if ( t1 > t2 ) {
00304                 t1 -= t2;
00305             }
00306             else {
00307                 t1 = t2 - t1;
00308                 anumer = -anumer;
00309             }
00310             c[0] = (UWORD)t1;
00311             if ( ( c[1] = (UWORD)(t1 » BITSINWORD) ) != 0 ) *nc = 2;
00312             else *nc = 1;
```

```
00313         }
00314         if ( anumer < 0 ) *nc = -*nc;
00315         d = NumberMalloc("AddRat");
00316         d[0] = (UWORD)t3;
00317         if ( ( d[1] = (UWORD)(t3 » BITSINWORD) ) != 0 ) nd = 2;
00318         else nd = 1;
00319         if ( Simplify(BHEAD c,nc,d,&nd) ) goto AddRer1;
00320     }
00321 /*
00322     else if ( a[na] == 1 && b[nb] == 1 && adenom == 1 && bdenom == 1 ) {
00323         if ( AddLong(a,na,b,nb,c,&nc) ) goto AddRer2;
00324         i = ABS(nc); d = c + i; *d++ = 1;
00325         while ( --i > 0 ) *d++ = 0 ;
00326         return(0);
00327     }
00328 */
00329     else {
00330         d = NumberMalloc("AddRat"); e = NumberMalloc("AddRat");
00331         f = NumberMalloc("AddRat"); g = NumberMalloc("AddRat");
00332         if ( GcdLong(BHEAD a+na,adenom,b+nb,bdenom,d,&nd) ) goto AddRer;
00333         if ( *d == 1 && nd == 1 ) nd = 0;
00334         if ( nd ) {
00335             if ( DivLong(a+na,adenom,d,nd,e,&ne,c,nc) ) goto AddRer;
00336             if ( DivLong(b+nb,bdenom,d,nd,f,&nf,c,nc) ) goto AddRer;
00337             if ( MulLong(a,anumer,f,nf,c,nc) ) goto AddRer;
00338             if ( MulLong(b,bnumer,e,ne,g,&ng) ) goto AddRer;
00339         }
00340         else {
00341             if ( MulLong(a+na,adenom,b,bnumer,c,nc) ) goto AddRer;
00342             if ( MulLong(b+nb,bdenom,a,anumer,g,&ng) ) goto AddRer;
00343         }
00344         if ( AddLong(c,*nc,g,ng,c,nc) ) goto AddRer;
00345         if ( !*nc ) {
00346             NumberFree(g,"AddRat"); NumberFree(f,"AddRat");
00347             NumberFree(e,"AddRat"); NumberFree(d,"AddRat");
00348             return(0);
00349         }
00350         if ( nd ) {
00351             if ( Simplify(BHEAD c,nc,d,&nd) ) goto AddRer;
00352             if ( MulLong(e,ne,d,nd,g,&ng) ) goto AddRer;
00353             if ( MulLong(g,ng,f,nf,d,&nd) ) goto AddRer;
00354         }
00355         else {
00356             if ( MulLong(a+na,adenom,b+nb,bdenom,d,&nd) ) goto AddRer;
00357         }
00358         NumberFree(g,"AddRat"); NumberFree(f,"AddRat"); NumberFree(e,"AddRat");
00359     }
00360     Pack(c,nc,d,nd);
00361     NumberFree(d,"AddRat");
00362     return(0);
00363 AddRer:
00364     NumberFree(g,"AddRat"); NumberFree(f,"AddRat"); NumberFree(e,"AddRat");
00365 AddRer1:
00366     NumberFree(d,"AddRat");
00367 /* AddRer2: */
00368     MLOCK(ErrorMessageLock);
00369     MesCall("AddRat");
00370     MUNLOCK(ErrorMessageLock);
00371     SETERROR(-1)
00372 }
00373
00374 /*
00375         #] AddRat :
00376         #[ MulRat :         WORD MulRat(a,na,b,nb,c,nc)
00377
00378     Multiplies the rationals a and b. The Gcd of the individual
00379     pieces is divided out first to minimize the chances of spurious
00380     overflows.
00381
00382 */
00383
00384 int MulRat(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
00385 {
00386     WORD i;
00387     WORD sgn = 1;
00388     if ( *b == 1 && b[1] == 1 ) {
00389         if ( nb == 1 ) {
00390             *nc = na;
00391             i = ABS(na); i *= 2;
00392             while ( --i >= 0 ) *c++ = *a++;
00393             return(0);
00394         }
00395         else if ( nb == -1 ) {
00396             *nc = - na;
00397             i = ABS(na); i *= 2;
00398             while ( --i >= 0 ) *c++ = *a++;
00399             return(0);
```

```
00400         }
00401     }
00402     if ( *a == 1 && a[1] == 1 ) {
00403         if ( na == 1 ) {
00404             *nc = nb;
00405             i = ABS(nb); i *= 2;
00406             while ( --i >= 0 ) *c++ = *b++;
00407             return(0);
00408         }
00409         else if ( na == -1 ) {
00410             *nc = - nb;
00411             i = ABS(nb); i *= 2;
00412             while ( --i >= 0 ) *c++ = *b++;
00413             return(0);
00414         }
00415     }
00416     if ( na < 0 ) { na = -na; sgn = -sgn; }
00417     if ( nb < 0 ) { nb = -nb; sgn = -sgn; }
00418     if ( !na || !nb ) { *nc = 0; return(0); }
00419     if ( na != 1 || nb != 1 ) {
00420         GETBIDENTITY
00421         UWORD *xd,*xe, *xf,*xg;
00422         WORD dden, dnumr, eden, enumr;
00423         UnPack(a,na,&dden,&dnumr);
00424         UnPack(b,nb,&eden,&enumr);
00425         xd = NumberMalloc("MulRat"); xf = NumberMalloc("MulRat");
00426         for ( i = 0; i < dnumr; i++ ) xd[i] = a[i];
00427         a += na;
00428         for ( i = 0; i < dden; i++ ) xf[i] = a[i];
00429         xe = NumberMalloc("MulRat"); xg = NumberMalloc("MulRat");
00430         for ( i = 0; i < enumr; i++ ) xe[i] = b[i];
00431         b += nb;
00432         for ( i = 0; i < eden; i++ ) xg[i] = b[i];
00433         if ( Simplify(BHEAD xd,&dnumr,xg,&eden) ||
00434              Simplify(BHEAD xe,&enumr,xf,&dden) ||
00435              MulLong(xd,dnumr,xe,enumr,c,nc) ||
00436              MulLong(xf,dden,xg,eden,xd,&dnumr) ) {
00437             MLOCK(ErrorMessageLock);
00438             MesCall("MulRat");
00439             MUNLOCK(ErrorMessageLock);
00440             NumberFree(xd,"MulRat"); NumberFree(xe,"MulRat"); NumberFree(xf,"MulRat");
      NumberFree(xg,"MulRat");
00441             SETERROR(-1)
00442         }
00443         Pack(c,nc,xd,dnumr);
00444         NumberFree(xd,"MulRat"); NumberFree(xe,"MulRat"); NumberFree(xf,"MulRat");
      NumberFree(xg,"MulRat");
00445     }
00446     else {
00447         UWORD y;
00448         UWORD a0,a1,b0,b1;
00449         RLONG xx;
00450         y = a[0]; b1=b[1];
00451         do { a0 = y % b1; y = b1; } while ( ( b1 = a0 ) != 0 );
00452         if ( y != 1 ) {
00453             a0 = a[0] / y;
00454             b1 = b[1] / y;
00455         }
00456         else {
00457             a0 = a[0];
00458             b1 = b[1];
00459         }
00460         y=b[0]; a1=a[1];
00461         do { b0 = y % a1; y = a1; } while ( ( a1 = b0 ) != 0 );
00462         if ( y != 1 ) {
00463             a1 = a[1] / y;
00464             b0 = b[0] / y;
00465         }
00466         else {
00467             a1 = a[1];
00468             b0 = b[0];
00469         }
00470         xx = ((RLONG)a0)*b0;
00471         if ( xx & AWORDMASK ) {
00472             *nc = 2;
00473             c[0] = (UWORD)xx;
00474             c[1] = (UWORD)(xx » BITSINWORD);
00475             xx = ((RLONG)a1)*b1;
00476             c[2] = (UWORD)xx;
00477             c[3] = (UWORD)(xx » BITSINWORD);
00478         }
00479         else {
00480             c[0] = (UWORD)xx;
00481             xx = ((RLONG)a1)*b1;
00482             if ( xx & AWORDMASK ) {
00483                 c[1] = 0;
00484                 c[2] = (UWORD)xx;
```

```
00485                 c[3] = (UWORD)(xx » BITSINWORD);
00486                 *nc = 2;
00487             }
00488             else {
00489                 c[1] = (UWORD)xx;
00490                 *nc = 1;
00491             }
00492         }
00493     }
00494     if ( sgn < 0 ) *nc = -*nc;
00495     return(0);
00496 }
00497
00498 /*
00499         #] MulRat :
00500         #[ DivRat :         WORD DivRat(a,na,b,nb,c,nc)
00501
00502     Divides the rational a by the rational b.
00503
00504 */
00505
00506 int DivRat(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
00507 {
00508     GETBIDENTITY
00509     WORD i, j;
00510     int ret;
00511     UWORD *xd,*xe,xx;
00512     if ( !nb ) {
00513 /* INTERNAL_ERROR_EXCL_START */
00514         MLOCK(ErrorMessageLock);
00515         MesPrint("!>Rational division by zero");
00516         MUNLOCK(ErrorMessageLock);
00517         return(-1);
00518 /* INTERNAL_ERROR_EXCL_STOP */
00519     }
00520     j = i = (nb >= 0)? nb: -nb;
00521     xd = b; xe = b + i;
00522     do { xx = *xd; *xd++ = *xe; *xe++ = xx; } while ( --j > 0 );
00523     ret = MulRat(BHEAD a,na,b,nb,c,nc);
00524     xd = b; xe = b + i;
00525     do { xx = *xd; *xd++ = *xe; *xe++ = xx; } while ( --i > 0 );
00526     return(ret);
00527 }
00528
00529 /*
00530         #] DivRat :
00531         #[ Simplify :       WORD Simplify(a,na,b,nb)
00532
00533     Determines the greatest common denominator of a and b and
00534     divides both by it. A possible sign is put in a. This is
00535     the simplification of the fraction a/b.
00536
00537 */
00538
00539 int Simplify(PHEAD UWORD *a, WORD *na, UWORD *b, WORD *nb)
00540 {
00541     GETBIDENTITY
00542     UWORD *x1,*x2,*x3;
00543     UWORD *x4;
00544     WORD n1,n2,n3,n4,sgn = 1;
00545     WORD i;
00546     UWORD *Siscrat5, *Siscrat6, *Siscrat7, *Siscrat8;
00547     if ( *na < 0 ) { *na = -*na; sgn = -sgn; }
00548     if ( *nb < 0 ) { *nb = -*nb; sgn = -sgn; }
00549     Siscrat5 = NumberMalloc("Simplify"); Siscrat6 = NumberMalloc("Simplify");
00550     Siscrat7 = NumberMalloc("Simplify"); Siscrat8 = NumberMalloc("Simplify");
00551     x1 = Siscrat8; x2 = Siscrat7;
00552     if ( *nb == 1 ) {
00553         x3 = Siscrat6;
00554         if ( DivLong(a,*na,b,*nb,x1,&n1,x2,&n2) ) goto SimpErr;
00555         if ( !n2 ) {
00556             for ( i = 0; i < n1; i++ ) *a++ = *x1++;
00557             *na = n1;
00558             *b = 1;
00559         }
00560         else {
00561             UWORD y1, y2, y3;
00562             y2 = *b;
00563             y3 = *x2;
00564             do { y1 = y2 % y3; y2 = y3; } while ( ( y3 = y1 ) != 0 );
00565             if ( ( *x2 = y2 ) != 1 ) {
00566                 *b /= y2;
00567                 if ( DivLong(a,*na,x2,(WORD)1,x1,&n1,x3,&n3) ) goto SimpErr;
00568                 for ( i = 0; i < n1; i++ ) *a++ = *x1++;
00569                 *na = n1;
00570             }
00571         }
```

```
00572     }
00573 #ifdef NEWTRICK
00574     else if ( *na >= GCDMAX && *nb >= GCDMAX ) {
00575         n1 = i = *na; x3 = a;
00576         NCOPY(x1,x3,i);
00577         x3 = b; n2 = i = *nb;
00578         NCOPY(x2,x3,i);
00579         x4 = Siscrat5;
00580         x2 = Siscrat6;
00581         x3 = Siscrat7;
00582         if ( GcdLong(BHEAD Siscrat8,n1,Siscrat7,n2,x2,&n3) ) goto SimpErr;
00583         n2 = n3;
00584         if ( *x2 != 1 || n2 != 1 ) {
00585             DivLong(a,*na,x2,n2,x1,&n1,x4,&n4);
00586             *na = i = n1;
00587             NCOPY(a,x1,i);
00588             DivLong(b,*nb,x2,n2,x3,&n3,x4,&n4);
00589             *nb = i = n3;
00590             NCOPY(b,x3,i);
00591         }
00592     }
00593 #endif
00594     else {
00595         x4 = Siscrat5;
00596         n1 = i = *na; x3 = a;
00597         NCOPY(x1,x3,i);
00598         x3 = b; n2 = i = *nb;
00599         NCOPY(x2,x3,i);
00600         x1 = Siscrat8; x2 = Siscrat7; x3 = Siscrat6;
00601         for(;;){
00602             if ( DivLong(x1,n1,x2,n2,x4,&n4,x3,&n3) ) goto SimpErr;
00603             if ( !n3 ) break;
00604             if ( n2 == 1 ) {
00605                 while ( ( *x1 = (*x2) % (*x3) ) != 0 ) { *x2 = *x3; *x3 = *x1; }
00606                 *x2 = *x3;
00607                 break;
00608             }
00609             if ( DivLong(x2,n2,x3,n3,x4,&n4,x1,&n1) ) goto SimpErr;
00610             if ( !n1 ) { x2 = x3; n2 = n3; x3 = Siscrat7; break; }
00611             if ( n3 == 1 ) {
00612                 while ( ( *x2 = (*x3) % (*x1) ) != 0 ) { *x3 = *x1; *x1 = *x2; }
00613                 *x2 = *x1;
00614                 n2 = 1;
00615                 break;
00616             }
00617             if ( DivLong(x3,n3,x1,n1,x4,&n4,x2,&n2) ) goto SimpErr;
00618             if ( !n2 ) { x2 = x1; n2 = n1; x1 = Siscrat7; break; }
00619             if ( n1 == 1 ) {
00620                 while ( ( *x3 = (*x1) % (*x2) ) != 0 ) { *x1 = *x2; *x2 = *x3; }
00621                 break;
00622             }
00623         }
00624         if ( *x2 != 1 || n2 != 1 ) {
00625             DivLong(a,*na,x2,n2,x1,&n1,x4,&n4);
00626             *na = i = n1;
00627             NCOPY(a,x1,i);
00628             DivLong(b,*nb,x2,n2,x3,&n3,x4,&n4);
00629             *nb = i = n3;
00630             NCOPY(b,x3,i);
00631         }
00632     }
00633     if ( sgn < 0 ) *na = -*na;
00634     NumberFree(Siscrat5,"Simplify"); NumberFree(Siscrat6,"Simplify");
00635     NumberFree(Siscrat7,"Simplify"); NumberFree(Siscrat8,"Simplify");
00636     return(0);
00637 SimpErr:
00638     MLOCK(ErrorMessageLock);
00639     MesCall("Simplify");
00640     MUNLOCK(ErrorMessageLock);
00641     NumberFree(Siscrat5,"Simplify"); NumberFree(Siscrat6,"Simplify");
00642     NumberFree(Siscrat7,"Simplify"); NumberFree(Siscrat8,"Simplify");
00643     SETERROR(-1)
00644 }
00645
00646 /*
00647         #] Simplify :
00648         #[ AccumGCD :       WORD AccumGCD(PHEAD a,na,b,nb)
00649
00650         Routine takes the rational GCD of the fractions in a and b and
00651         replaces a by the GCD of the two.
00652         The rational GCD is defined as the rational that consists of
00653         the GCD of the numerators divided by the GCD of the denominators
00654 */
00655
00656 int AccumGCD(PHEAD UWORD *a, WORD *na, UWORD *b, WORD nb)
00657 {
00658     GETBIDENTITY
```

```
00659     WORD nna,nnb,numa,numb,dena,denb,numc,denc;
00660     UWORD *GCDbuffer = NumberMalloc("AccumGCD");
00661     int i;
00662     nna = *na; if ( nna < 0 ) nna = -nna; nna = (nna-1)/2;
00663     nnb = nb;  if ( nnb < 0 ) nnb = -nnb; nnb = (nnb-1)/2;
00664     UnPack(a,nna,&dena,&numa);
00665     UnPack(b,nnb,&denb,&numb);
00666     if ( GcdLong(BHEAD a,numa,b,numb,GCDbuffer,&numc) ) goto AccErr;
00667     numa = numc;
00668     for ( i = 0; i < numa; i++ ) a[i] = GCDbuffer[i];
00669     if ( GcdLong(BHEAD a+nna,dena,b+nnb,denb,GCDbuffer,&denc) ) goto AccErr;
00670     dena = denc;
00671     for ( i = 0; i < dena; i++ ) a[i+nna] = GCDbuffer[i];
00672     Pack(a,&numa,a+nna,dena);
00673     *na = INCLENG(numa);
00674     NumberFree(GCDbuffer,"AccumGCD");
00675     return(0);
00676 AccErr:
00677     MLOCK(ErrorMessageLock);
00678     MesCall("AccumGCD");
00679     MUNLOCK(ErrorMessageLock);
00680     NumberFree(GCDbuffer,"AccumGCD");
00681     SETERROR(-1)
00682 }
00683
00684 /*
00685         #] AccumGCD :
00686         #[ TakeRatRoot:
00687 */
00688
00689 int TakeRatRoot(UWORD *a, WORD *n, WORD power)
00690 {
00691     WORD numer,denom, nn;
00692     if ( ( power & 1 ) == 0 && *n < 0 ) return(1);
00693     if ( ABS(*n) == 1 && a[0] == 1 && a[1] == 1 ) return(0);
00694     nn = ABS(*n);
00695     UnPack(a,nn,&denom,&numer);
00696     if ( TakeLongRoot(a+nn,&denom,power) ) return(1);
00697     if ( TakeLongRoot(a,&numer,power) ) return(1);
00698     Pack(a,&numer,a+nn,denom);
00699     if ( *n < 0 ) *n = -numer;
00700     else          *n = numer;
00701     return(0);
00702 }
00703
00704 /*
00705         #] TakeRatRoot:
00706     #] RekenRational :
00707     #[ RekenLong :
00708         #[ AddLong :        WORD AddLong(a,na,b,nb,c,nc)
00709
00710     Long addition. Uses addition and subtraction of positive numbers.
00711
00712 */
00713
00714 int AddLong(UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
00715 {
00716     WORD sgn, res;
00717     if ( na < 0 ) {
00718         if ( nb < 0 ) {
00719             if ( AddPLon(a,-na,b,-nb,c,nc) ) return(-1);
00720             *nc = -*nc;
00721             return(0);
00722         }
00723         else {
00724             na = -na;
00725             sgn = -1;
00726         }
00727     }
00728     else {
00729         if ( nb < 0 ) {
00730             nb = -nb;
00731             sgn = 1;
00732         }
00733         else { return( AddPLon(a,na,b,nb,c,nc) ); }
00734     }
00735     if ( ( res = BigLong(a,na,b,nb) ) > 0 ) {
00736         SubPLon(a,na,b,nb,c,nc);
00737         if ( sgn < 0 ) *nc = -*nc;
00738     }
00739     else if ( res < 0 ) {
00740         SubPLon(b,nb,a,na,c,nc);
00741         if ( sgn > 0 ) *nc = -*nc;
00742     }
00743     else {
00744         *nc = 0;
00745         *c = 0;
```

```
00746     }
00747     return(0);
00748 }
00749
00750 /*
00751         #] AddLong :
00752         #[ AddPLon :        WORD AddPLon(a,na,b,nb,c,nc)
00753
00754     Adds two long integers a and b and puts the result in c.
00755     The length of a and b are na and nb. The length of c is returned in nc.
00756     c can be a or b.
00757 */
00758
00759 int AddPLon(UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
00760 {
00761     UWORD carry = 0, e, nd = 0;
00762     while ( na && nb ) {
00763         e = *a;
00764         *c = e + *b + carry;
00765         if ( carry ) {
00766             if ( e < *c ) carry = 0;
00767         }
00768         else {
00769             if ( e > *c ) carry = 1;
00770         }
00771         a++; b++; c++; nd++; na--; nb--;
00772     }
00773     while ( na ) {
00774         if ( carry ) {
00775             *c = *a++ + carry;
00776             if ( *c++ ) carry = 0;
00777         }
00778         else *c++ = *a++;
00779         nd++; na--;
00780     }
00781     while ( nb ) {
00782         if ( carry ) {
00783             *c = *b++ + carry;
00784             if ( *c++ ) carry = 0;
00785         }
00786         else *c++ = *b++;
00787         nd++; nb--;
00788     }
00789     if ( carry ) {
00790         nd++;
00791         if ( nd > (UWORD)AM.MaxTal ) {
00792             MLOCK(ErrorMessageLock);
00793             MesPrint("Overflow in addition");
00794             MUNLOCK(ErrorMessageLock);
00795             return(-1);
00796         }
00797         *c++ = carry;
00798     }
00799     *nc = nd;
00800     return(0);
00801 }
00802
00803 /*
00804         #] AddPLon :
00805         #[ SubPLon :        void SubPLon(a,na,b,nb,c,nc)
00806
00807     Subtracts b from a. Assumes that a > b. Result in c.
00808     c can be a or b.
00809
00810 */
00811
00812 void SubPLon(UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
00813 {
00814     UWORD borrow = 0, e, nd = 0;
00815     while ( nb ) {
00816         e = *a;
00817         if ( borrow ) {
00818             *c = e - *b - borrow;
00819             if ( *c < e ) borrow = 0;
00820         }
00821         else {
00822             *c = e - *b;
00823             if ( *c > e ) borrow = 1;
00824         }
00825         a++; b++; c++; na--; nb--; nd++;
00826     }
00827     while ( na ) {
00828         if ( borrow ) {
00829             if ( *a ) { *c++ = *a++ - 1; borrow = 0; }
00830             else { *c++ = (UWORD)(-1); a++; }
00831         }
00832         else *c++ = *a++;
```

```
00833         na--; nd++;
00834     }
00835     while ( nd && !*--c ) { nd--; }
00836     *nc = (WORD)nd;
00837 }
00838
00839 /*
00840         #] SubPLon :
00841         #[ MulLong :        WORD MulLong(a,na,b,nb,c,nc)
00842
00843     Does a Long multiplication. Assumes that WORD is half the size
00844     of a LONG to work out the scheme! The number of operations is
00845     the canonical na*nm multiplications.
00846     c should not overlap with a or b.
00847
00848 */
00849
00850 int MulLong(UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
00851 {
00852     WORD sgn = 1;
00853     UWORD i, *ic, *ia;
00854     RLONG t, bb;
00855     if ( !na || !nb ) { *nc = 0; return(0); }
00856     if ( na < 0 ) { na = -na; sgn = -sgn; }
00857     if ( nb < 0 ) { nb = -nb; sgn = -sgn; }
00858     *nc = i = na + nb;
00859     if ( i > (UWORD)(AM.MaxTal+1) ) goto MulLov;
00860     ic = c;
00861 /*
00862     #[ GMP stuff :
00863 */
00864 #ifdef WITHGMP
00865     if (na > 3 && nb > 3) {
00866 /*      mp_limb_t res;  */
00867         UWORD *to, *from;
00868         int j;
00869         GETIDENTITY
00870         UWORD *DLscrat9 = NumberMalloc("MulLong"), *DLscratA = NumberMalloc("MulLong"), *DLscratB =
      NumberMalloc("MulLong");
00871 #if ( GMPSPREAD != 1 )
00872         if ( na & 1 ) {
00873             from = a; a = to = DLscrat9; j = na; NCOPY(to, from, j);
00874             a[na++] = 0;
00875             ++*nc;
00876         } else
00877 #endif
00878         if ( (LONG)a & (sizeof(mp_limb_t)-1) ) {
00879             from = a; a = to = DLscrat9; j = na; NCOPY(to, from, j);
00880         }
00881
00882 #if ( GMPSPREAD != 1 )
00883         if ( nb & 1 ) {
00884             from = b; b = to = DLscratA; j = nb; NCOPY(to, from, j);
00885             b[nb++] = 0;
00886             ++*nc;
00887         } else
00888 #endif
00889         if ( (LONG)b & (sizeof(mp_limb_t)-1) ) {
00890             from = b; b = to = DLscratA; j = nb; NCOPY(to, from, j);
00891         }
00892
00893         if ( ( *nc > (WORD)i ) || ( (LONG)c & (LONG)(sizeof(mp_limb_t)-1) ) ) {
00894             ic = DLscratB;
00895         }
00896         if ( na < nb ) {
00897             /* res = */
00898             mpn_mul((mp_ptr)ic, (mp_srcptr)b, nb/GMPSPREAD, (mp_srcptr)a, na/GMPSPREAD);
00899         } else {
00900             /* res = */
00901             mpn_mul((mp_ptr)ic, (mp_srcptr)a, na/GMPSPREAD, (mp_srcptr)b, nb/GMPSPREAD);
00902         }
00903         while ( ic[i-1] == 0 ) i--;
00904         *nc = i;
00905 /*
00906         if ( res == 0 ) *nc -= GMPSPREAD;
00907         else if ( res <= WORDMASK ) --*nc;
00908 */
00909         if ( ic != c ) {
00910             j = *nc; NCOPY(c, ic, j);
00911         }
00912         if ( sgn < 0 ) *nc = -(*nc);
00913         NumberFree(DLscrat9,"MulLong"); NumberFree(DLscratA,"MulLong");
      NumberFree(DLscratB,"MulLong");
00914         return(0);
00915     }
00916 #endif
00917 /*
```

```
00918     #] GMP stuff :
00919 */
00920     do { *ic++ = 0; } while ( --i > 0 );
00921     do {
00922         ia = a;
00923         ic = c++;
00924         t = 0;
00925         i = na;
00926         bb = (RLONG)(*b++);
00927         do {
00928             t = (*ia++) * bb + t + *ic;
00929             *ic++ = (WORD)t;
00930             t »= BITSINWORD;       /* should actually be a swap */
00931         } while ( --i > 0 );
00932         if ( t ) *ic = (UWORD)t;
00933     } while ( --nb > 0 );
00934     if ( !*ic ) (*nc)--;
00935     if ( *nc > AM.MaxTal ) goto MulLov;
00936     if ( sgn < 0 ) *nc = -(*nc);
00937     return(0);
00938 MulLov:
00939     MLOCK(ErrorMessageLock);
00940     MesPrint("Overflow in Multiplication");
00941     MUNLOCK(ErrorMessageLock);
00942     return(-1);
00943 }
00944
00945 /*
00946         #] MulLong :
00947         #[ BigLong :        WORD BigLong(a,na,b,nb)
00948
00949     Returns > 0 if a > b, < 0 if b > a and 0 if a == b
00950
00951 */
00952
00953 int BigLong(UWORD *a, WORD na, UWORD *b, WORD nb)
00954 {
00955     a += na;
00956     b += nb;
00957     while ( na && !*--a ) na--;
00958     while ( nb && !*--b ) nb--;
00959     if ( nb < na ) return(1);
00960     if ( nb > na ) return(-1);
00961     while ( --na >= 0 ) {
00962         if ( *a > *b ) return(1);
00963         else if ( *b > *a ) return(-1);
00964         a--; b--;
00965     }
00966     return(0);
00967 }
00968
00969 /*
00970         #] BigLong :
00971         #[ DivLong :        WORD DivLong(a,na,b,nb,c,nc,d,nd)
00972
00973     This is the long division which knows a couple of exceptions.
00974     It uses therefore a recursive call for the renormalization.
00975     The quotient comes in c and the remainder in d.
00976     d may be overlapping with b. It may also be identical to a.
00977     c should not overlap with a, but it can overlap with b.
00978
00979 */
00980
00981 int DivLong(UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c,
00982             WORD *nc, UWORD *d, WORD *nd)
00983 {
00984     WORD sgna = 1, sgnb = 1, ne, nf, ng, nh;
00985     WORD i, ni;
00986     UWORD *w1, *w2;
00987     RLONG t, v;
00988     UWORD *e, *f, *ff, *g, norm, estim;
00989 #ifdef WITHGMP
00990     UWORD *DLscrat9, *DLscratA, *DLscratB, *DLscratC;
00991 #endif
00992     RLONG esthelp;
00993     if ( !nb ) {
00994         MLOCK(ErrorMessageLock);
00995         MesPrint("Division by zero");
00996         MUNLOCK(ErrorMessageLock);
00997         return(-1);
00998     }
00999     if ( !na ) { *nc = *nd = 0; return(0); }
01000     if ( na < 0 ) { sgna = -sgna; na = -na; }
01001     if ( nb < 0 ) { sgnb = -sgnb; nb = -nb; }
01002     if ( na < nb ) {
01003         for ( i = 0; i < na; i++ ) *d++ = *a++;
01004         *nd = na;
```

```
01005         *nc = 0;
01006     }
01007     else if ( nb == na && ( i = BigLong(b,nb,a,na) ) >= 0 ) {
01008         if ( i > 0 ) {
01009             for ( i = 0; i < na; i++ ) *d++ = *a++;
01010             *nd = na;
01011             *nc = 0;
01012         }
01013         else {
01014             *c = 1;
01015             *nc = 1;
01016             *nd = 0;
01017         }
01018     }
01019     else if ( nb == 1 ) {
01020         if ( *b == 1 ) {
01021             for ( i = 0; i < na; i++ ) *c++ = *a++;
01022             *nc = na;
01023             *nd = 0;
01024         }
01025         else {
01026             w1 = a+na;
01027             *nc = ni = na;
01028             *nd = 1;
01029             w2 = c+ni;
01030             v = (RLONG)(*b);
01031             t = (RLONG)(*--w1);
01032             while ( --ni >= 0 ) {
01033                 *--w2 = t / v;
01034                 t -= v * (*w2);
01035                 if ( ni ) {
01036                     t <<= BITSINWORD;
01037                     t += *--w1;
01038                 }
01039             }
01040             if ( ( *d = (UWORD)t ) == 0 ) *nd = 0;
01041             if ( !*(c+na-1) ) (*nc)--;
01042         }
01043     }
01044     else {
01045         GETIDENTITY
01046 /*
01047         #[ GMP stuff :
01048
01049         We start with copying a and b.
01050         Then we make space for c and d.
01051         Next we call mpn_tdiv_qr
01052         We adjust sizes and copy to c and d if needed.
01053         Finally the signs are settled.
01054 */
01055 #ifdef WITHGMP
01056         if ( na > 4 && nb > 3 ) {
01057           UWORD *ic, *id, *to, *from;
01058           int j = na - nb;
01059           DLscrat9 = NumberMalloc("DivLong"); DLscratA = NumberMalloc("DivLong");
01060           DLscratB = NumberMalloc("DivLong"); DLscratC = NumberMalloc("DivLong");
01061
01062 #if ( GMPSPREAD != 1 )
01063           if ( na & 1 ) {
01064             from = a; a = to = DLscrat9; i = na; NCOPY(to, from, i);
01065             a[na++] = 0;
01066           } else
01067 #endif
01068           if ( (LONG)a & (sizeof(mp_limb_t)-1) ) {
01069             from = a; a = to = DLscrat9; i = na; NCOPY(to, from, i);
01070           }
01071
01072 #if ( GMPSPREAD != 1 )
01073           if ( nb & 1 ) {
01074             from = b; b = to = DLscratA; i = nb; NCOPY(to, from, i);
01075             b[nb++] = 0;
01076           } else
01077 #endif
01078           if ( ( (LONG)b & (sizeof(mp_limb_t)-1) ) != 0 ) {
01079             from = b; b = to = DLscratA; i = nb; NCOPY(to, from, i);
01080           }
01081 #if ( GMPSPREAD != 1 )
01082 /*
01083           Not recognizing this case was a nasty and extremely rare bug.
01084           In the case that c == a and na is odd and nc == na and b
01085           still needs to be used, the least significant UWORD of b got
01086           overwritten by zero. (22-mar-2023)
01087 */
01088           ic = DLscratB; id = DLscratC;
01089 #else
01090           if ( ( (LONG)c & (sizeof(mp_limb_t)-1) ) != 0 ) ic = DLscratB;
01091           else                                            ic = c;
```

```
01092
01093           if ( ( (LONG)d & (sizeof(mp_limb_t)-1) ) != 0 ) id = DLscratC;
01094           else                                            id = d;
01095 #endif
01096           mpn_tdiv_qr((mp_limb_t *)ic,(mp_limb_t *)id,(mp_size_t)0,
01097             (const mp_limb_t *)a,(mp_size_t)(na/GMPSPREAD),
01098             (const mp_limb_t *)b,(mp_size_t)(nb/GMPSPREAD));
01099           while ( j >= 0 && ic[j] == 0 ) j--;
01100           j++; *nc = j;
01101           if ( c != ic ) { NCOPY(c,ic,j); }
01102           j = nb-1;
01103           while ( j >= 0 && id[j] == 0 ) j--;
01104           j++; *nd = j;
01105           if ( d != id ) { NCOPY(d,id,j); }
01106           if ( sgna < 0 ) { *nc = -(*nc); *nd = -(*nd); }
01107           if ( sgnb < 0 ) { *nc = -(*nc); }
01108           NumberFree(DLscrat9,"DivLong"); NumberFree(DLscratA,"DivLong");
01109           NumberFree(DLscratB,"DivLong"); NumberFree(DLscratC,"DivLong");
01110           return(0);
01111         }
01112 #endif
01113 /*
01114         #] GMP stuff :
01115 */
01116         /* Start with normalization operation */
01117
01118         e = NumberMalloc("DivLong"); f = NumberMalloc("DivLong"); g = NumberMalloc("DivLong");
01119         if ( b[nb-1] == (FULLMAX-1) ) norm = 1;
01120         else {
01121             norm = (UWORD)(((ULONG)FULLMAX) / (ULONG)((b[nb-1]+1L)));
01122         }
01123         f[na] = 0;
01124         if ( MulLong(b,nb,&norm,1,e,&ne) ||
01125              MulLong(a,na,&norm,1,f,&nf) ) {
01126             NumberFree(e,"DivLong"); NumberFree(f,"DivLong"); NumberFree(g,"DivLong");
01127             return(-1);
01128         }
01129         if ( BigLong(f+nf-ne,ne,e,ne) >= 0 ) {
01130             SubPLon(f+nf-ne,ne,e,ne,f+nf-ne,&nh);
01131             w1 = c + (nf-ne);
01132             *nc = nf-ne+1;
01133         }
01134         else {
01135             nh = ne;
01136             *nc = nf-ne;
01137             w1 = 0;
01138         }
01139         w2 = c; i = *nc; do { *w2++ = 0; } while ( --i > 0 );
01140         nf = na;
01141         ni = nf-ne;
01142         esthelp = (RLONG)(e[ne-1]) + 1L;
01143         while ( nf >= ne ) {
01144             if ( (WORD)esthelp == 0 ) {
01145                 estim = (WORD)(((((RLONG)(f[nf]))«BITSINWORD)+f[nf-1])»BITSINWORD);
01146             }
01147             else {
01148                 estim = (WORD)(((((RLONG)(f[nf]))«BITSINWORD)+f[nf-1])/esthelp);
01149             }
01150             /* This estimate may be up to two too small */
01151             if ( estim ) {
01152                 MulLong(e,ne,&estim,1,g,&ng);
01153                 nh = ne + 1; if ( !f[ni+ne] ) nh--;
01154                 SubPLon(f+ni,nh,g,ng,f+ni,&nh);
01155             }
01156             else {
01157                 w2 = f+ni+ne; nh = ne+1;
01158                 while ( ( nh > 0 ) && !*w2 ) { nh--; w2--; }
01159             }
01160             if ( BigLong(f+ni,nh,e,ne) >= 0 ) {
01161                 estim++;
01162                 SubPLon(f+ni,nh,e,ne,f+ni,&nh);
01163                 if ( BigLong(f+ni,nh,e,ne) >= 0 ) {
01164                     estim++;
01165                     SubPLon(f+ni,nh,e,ne,f+ni,&nh);
01166                     if ( BigLong(f+ni,nh,e,ne) >= 0 ) {
01167 /* INTERNAL_ERROR_EXCL_START */
01168                         MLOCK(ErrorMessageLock);
01169                         MesPrint("!>Problems in DivLong");
01170                         AO.OutSkip = 3;
01171                         FiniLine();
01172                         i = na;
01173                         while ( --i >= 0 ) { TalToLine((UWORD)(*a++)); TokenToLine((UBYTE *)"  "); }
01174                         FiniLine();
01175                         i = nb;
01176                         while ( --i >= 0 ) { TalToLine((UWORD)(*b++)); TokenToLine((UBYTE *)"  "); }
01177                         AO.OutSkip = 0;
01178                         FiniLine();
```

```
01179                       MUNLOCK(ErrorMessageLock);
01180                       NumberFree(e,"DivLong"); NumberFree(f,"DivLong"); NumberFree(g,"DivLong");
01181                       return(-1);
01182 /* INTERNAL_ERROR_EXCL_STOP */
01183                   }
01184               }
01185           }
01186           c[ni] = estim;
01187           nf--;
01188           ni--;
01189       }
01190       if ( w1 ) *w1 = 1;
01191
01192       /* Finish with the renormalization operation */
01193
01194       if ( nh > 0 ) {
01195           if ( norm == 1 ) {
01196               *nd = i = nh; ff = f;
01197               NCOPY(d,ff,i);
01198           }
01199           else {
01200               w1 = f+nh;
01201               *nd = ni = nh;
01202               w2 = d+ni;
01203               v = norm;
01204               t = (RLONG)(*--w1);
01205               while ( --ni >= 0 ) {
01206                   *--w2 = t / v;
01207                   t -= v * (*w2);
01208                   if ( ni ) {
01209                       t «= BITSINWORD;
01210                       t += *--w1;
01211                   }
01212               }
01213               if ( t ) {
01214 /* INTERNAL_ERROR_EXCL_START */
01215                   MLOCK(ErrorMessageLock);
01216                   MesPrint("!>Error in DivLong");
01217                   MUNLOCK(ErrorMessageLock);
01218                   NumberFree(e,"DivLong"); NumberFree(f,"DivLong"); NumberFree(g,"DivLong");
01219                   return(-1);
01220 /* INTERNAL_ERROR_EXCL_STOP */
01221               }
01222               if ( !*(d+nh-1) ) (*nd)--;
01223           }
01224       }
01225       else { *nd = 0; }
01226       NumberFree(e,"DivLong"); NumberFree(f,"DivLong"); NumberFree(g,"DivLong");
01227   }
01228   if ( sgna < 0 ) { *nc = -(*nc); *nd = -(*nd); }
01229   if ( sgnb < 0 ) { *nc = -(*nc); }
01230   return(0);
01231 }
01232
01233 /*
01234       #] DivLong :
01235       #[ RaisPow :        WORD RaisPow(a,na,b)
01236
01237   Raises a to the power b. a is a Long integer and b >= 0.
01238   The method that is used works with a bitdecomposition of b.
01239 */
01240
01241 int RaisPow(PHEAD UWORD *a, WORD *na, UWORD b)
01242 {
01243   GETBIDENTITY
01244   WORD i, nu;
01245   UWORD *it, *iu, c;
01246   UWORD *is, *iss;
01247   WORD ns, nt, nmod;
01248   nmod = ABS(AN.ncmod);
01249   if ( !*na || ( ( *na == 1 ) && ( *a == 1 ) ) ) return(0);
01250   if ( !b ) { *na=1; *a=1; return(0); }
01251   is = NumberMalloc("RaisPow");
01252   it = NumberMalloc("RaisPow");
01253   for ( i = 0; i < ABS(*na); i++ ) is[i] = a[i];
01254   ns = *na;
01255   c = b;
01256   for ( i = 0; i < BITSINWORD; i++ ) {
01257       if ( !c ) break;
01258       c /= 2;
01259   }
01260   i--;
01261   c = 1u « i;
01262   while ( --i >= 0 ) {
01263       c /= 2;
01264       if(MulLong(is,ns,is,ns,it,&nt)) goto RaisOvl;
01265       if ( b & c ) {
```

```
01266             if ( MulLong(it,nt,a,*na,is,&ns) ) goto RaisOvl;
01267         }
01268         else {
01269             iu = is; is = it; it = iu;
01270             nu = ns; ns = nt; nt = nu;
01271         }
01272         if ( nmod != 0 ) {
01273             if ( DivLong(is,ns,(UWORD *)AC.cmod,nmod,it,&nt,is,&ns) ) goto RaisOvl;
01274         }
01275     }
01276     if ( ( nmod != 0 ) && ( ( AC.modmode & POSNEG ) != 0 ) ) {
01277         NormalModulus(is,&ns);
01278     }
01279     if ( ( *na = i = ns ) != 0 ) { iss = is; i=ABS(i); NCOPY(a,iss,i); }
01280     NumberFree(is,"RaisPow"); NumberFree(it,"RaisPow");
01281     return(0);
01282 RaisOvl:
01283     MLOCK(ErrorMessageLock);
01284     MesCall("RaisPow");
01285     MUNLOCK(ErrorMessageLock);
01286     NumberFree(is,"RaisPow"); NumberFree(it,"RaisPow");
01287     SETERROR(-1)
01288 }
01289
01290 /*
01291         #] RaisPow :
01292         #[ RaisPowCached :
01293 */
01294
01309 void RaisPowCached (PHEAD WORD x, WORD n, UWORD **c, WORD *nc) {
01310
01311     int i,j;
01312     WORD new_small_power_maxx, new_small_power_maxn, ID;
01313     WORD *new_small_power_n;
01314     UWORD **new_small_power;
01315
01316     /* check whether to extend the array */
01317     if (x>=AT.small_power_maxx || n>=AT.small_power_maxn) {
01318
01319         new_small_power_maxx = AT.small_power_maxx;
01320         if (x>=AT.small_power_maxx)
01321             new_small_power_maxx = MaX(2*AT.small_power_maxx, x+1);
01322
01323         new_small_power_maxn = AT.small_power_maxn;
01324         if (n>=AT.small_power_maxn)
01325             new_small_power_maxn = MaX(2*AT.small_power_maxn, n+1);
01326
01327         new_small_power_n = (WORD*)
      Malloc1(new_small_power_maxx*new_small_power_maxn*sizeof(WORD),"RaisPowCached");
01328         new_small_power = (UWORD **) Malloc1(new_small_power_maxx*new_small_power_maxn*sizeof(UWORD
      *),"RaisPowCached");
01329
01330         for (i=0; i<new_small_power_maxx * new_small_power_maxn; i++) {
01331             new_small_power_n[i] = 0;
01332             new_small_power  [i] = NULL;
01333         }
01334
01335         for (i=0; i<AT.small_power_maxx; i++)
01336             for (j=0; j<AT.small_power_maxn; j++) {
01337                 new_small_power_n[i*new_small_power_maxn+j] =
      AT.small_power_n[i*AT.small_power_maxn+j];
01338                 new_small_power  [i*new_small_power_maxn+j] = AT.small_power
      [i*AT.small_power_maxn+j];
01339             }
01340
01341         if (AT.small_power_n != NULL) {
01342             M_free(AT.small_power_n,"RaisPowCached");
01343             M_free(AT.small_power,"RaisPowCached");
01344         }
01345
01346         AT.small_power_maxx = new_small_power_maxx;
01347         AT.small_power_maxn = new_small_power_maxn;
01348         AT.small_power_n    = new_small_power_n;
01349         AT.small_power      = new_small_power;
01350     }
01351
01352     /* check whether the results is already calculated */
01353     ID = x * AT.small_power_maxn + n;
01354
01355     if (AT.small_power[ID] == NULL) {
01356 #ifdef OLDRAISPOWCACHED
01357         AT.small_power[ID] = NumberMalloc("RaisPowCached");
01358         AT.small_power_n[ID] = 1;
01359         AT.small_power[ID][0] = x;
01360         RaisPow(BHEAD AT.small_power[ID],&AT.small_power_n[ID],n);
01361 #else
01362         UWORD *c = NumberMalloc("RaisPowCached");
```

```
01363         WORD i, k = 1;
01364         c[0] = x;
01365         RaisPow(BHEAD c,&k,n);
01366 /*
01367         And now get the proper amount.
01368 */
01369         if ( AT.InNumMem < k ) {    /* We should start a new buffer */
01370             AT.InNumMem = 5*AM.MaxTal;
01371             AT.NumMem = (UWORD *)Malloc1(AT.InNumMem*sizeof(UWORD),"RaisPowCached");
01372 /*
01373             MesPrint("  Got an extra %l UWORDS in RaisPowCached",AT.InNumMem);
01374 */
01375         }
01376         for ( i = 0; i < k; i++ ) AT.NumMem[i] = c[i];
01377         AT.small_power[ID] = AT.NumMem;
01378         AT.small_power_n[ID] = k;
01379         AT.NumMem += k;
01380         AT.InNumMem -= k;
01381         NumberFree(c,"RaisPowCached");
01382 #endif
01383     }
01384
01385     /* return the result */
01386     *c  = AT.small_power[ID];
01387     *nc = AT.small_power_n[ID];
01388 }
01389
01390 /*
01391         #] RaisPowCached :
01392         #[ RaisPowMod :
01393
01394    Computes the power x^n mod m
01395  */
01396 WORD RaisPowMod (WORD x, WORD n, WORD m) {
01397     LONG y=1, z=x;
01398     while (n) {
01399         if (n&1) { y*=z; y%=m; }
01400         z*=z; z%=m;
01401         n /= 2;
01402     }
01403     return (WORD)y;
01404 }
01405
01406 /*
01407     #] RaisPowMod :
01408         #[ NormalModulus :  int NormalModulus(UWORD *a,WORD *na)
01409 */
01416 int NormalModulus(UWORD *a,WORD *na)
01417 {
01418     WORD n;
01419     if ( AC.halfmod == 0 ) {
01420       LOCK(AC.halfmodlock);
01421       if ( AC.halfmod == 0 ) {
01422         UWORD two[1],remain[1];
01423         WORD dummy;
01424         two[0] = 2;
01425         AC.halfmod = (UWORD *)Malloc1((ABS(AC.ncmod))*sizeof(UWORD),"halfmod");
01426         DivLong((UWORD *)AC.cmod,(ABS(AC.ncmod)),two,1
01427                 ,(UWORD *)AC.halfmod,&(AC.nhalfmod),remain,&dummy);
01428       }
01429       UNLOCK(AC.halfmodlock);
01430     }
01431     n = ABS(*na);
01432     if ( BigLong(a,n,AC.halfmod,AC.nhalfmod) > 0 ) {
01433         SubPLon((UWORD *)AC.cmod,(ABS(AC.ncmod)),a,n,a,&n);
01434         if ( *na > 0 ) { *na = -n; }
01435         else { *na = n; }
01436         return(1);
01437     }
01438     return(0);
01439 }
01440
01441 /*
01442         #] NormalModulus :
01443         #[ MakeInverses :
01444 */
01453 int MakeInverses(void)
01454 {
01455     WORD n = AC.cmod[0], i, inv2;
01456     if ( AC.ncmod != 1 ) return(1);
01457     if ( AC.modinverses == 0 ) {
01458       LOCK(AC.halfmodlock);
01459       if ( AC.modinverses == 0 ) {
01460         AC.modinverses = (UWORD *)Malloc1(n*sizeof(UWORD),"modinverses");
01461         AC.modinverses[0] = 0;
01462         AC.modinverses[1] = 1;
01463         for ( i = 2; i < n; i++ ) {
```

```
01464             if ( GetModInverses(i,n,
01465                         (WORD *)(&(AC.modinverses[i])),&inv2) ) {
01466                 SETERROR(-1)
01467             }
01468         }
01469       }
01470       UNLOCK(AC.halfmodlock);
01471     }
01472     return(0);
01473 }
01474
01475 /*
01476         #] MakeInverses :
01477         #[ GetModInverses :
01478 */
01489 int GetModInverses(WORD m1, WORD m2, WORD *im1, WORD *im2)
01490 {
01491     WORD a1, a2, a3;
01492     WORD b1, b2, b3;
01493     WORD x = m1, y, c, d = m2;
01494     if ( x < 1 || d <= 1 ) goto somethingwrong;
01495     a1 = 0; a2 = 1;
01496     b1 = 1; b2 = 0;
01497     for(;;) {
01498         c = d/x; y = d%x; /* a good compiler makes this faster than y=d-c*x */
01499         if ( y == 0 ) break;
01500         a3 = a1-c*a2; a1 = a2; a2 = a3;
01501         b3 = b1-c*b2; b1 = b2; b2 = b3;
01502         d = x; x = y;
01503     }
01504     if ( x != 1 ) goto somethingwrong;
01505     if ( a2 < 0 ) a2 += m2;
01506     if ( b2 < 0 ) b2 += m1;
01507     if (im1!=NULL) *im1 = a2;
01508     if (im2!=NULL) *im2 = b2;
01509     return(0);
01510 somethingwrong:
01511 /* INTERNAL_ERROR_EXCL_START */
01512     MLOCK(ErrorMessageLock);
01513     MesPrint("!>Error trying to determine inverses in GetModInverses");
01514     MUNLOCK(ErrorMessageLock);
01515     return(-1);
01516 /* INTERNAL_ERROR_EXCL_STOP */
01517 }
01518 /*
01519         #] GetModInverses :
01520         #[ GetLongModInverses :
01521 */
01522
01523 int GetLongModInverses(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *ia, WORD *nia, UWORD *ib,
    WORD *nib) {
01524
01525     UWORD *s, *t, *sa, *sb, *ta, *tb, *x, *y, *swap1;
01526     WORD ns, nt, nsa, nsb, nta, ntb, nx, ny, swap2;
01527
01528     s = NumberMalloc("GetLongModInverses");
01529     ns = na;
01530     WCOPY(s, a, ABS(ns));
01531
01532     t = NumberMalloc("GetLongModInverses");
01533     nt = nb;
01534     WCOPY(t, b, ABS(nt));
01535
01536     sa = NumberMalloc("GetLongModInverses");
01537     nsa = 1;
01538     sa[0] = 1;
01539
01540     sb = NumberMalloc("GetLongModInverses");
01541     nsb = 0;
01542
01543     ta = NumberMalloc("GetLongModInverses");
01544     nta = 0;
01545
01546     tb = NumberMalloc("GetLongModInverses");
01547     ntb = 1;
01548     tb[0] = 1;
01549
01550     x = NumberMalloc("GetLongModInverses");
01551     y = NumberMalloc("GetLongModInverses");
01552
01553     while (nt != 0) {
01554         DivLong(s,ns,t,nt,x,&nx,y,&ny);
01555         swap1=s; s=y; y=swap1;
01556         ns=ny;
01557         MulLong(x,nx,ta,nta,y,&ny);
01558         AddLong(sa,nsa,y,-ny,sa,&nsa);
01559         MulLong(x,nx,tb,ntb,y,&ny);
```

```
01560         AddLong(sb,nsb,y,-ny,sb,&nsb);
01561
01562         swap1=s; s=t; t=swap1;
01563         swap2=ns; ns=nt; nt=swap2;
01564         swap1=sa; sa=ta; ta=swap1;
01565         swap2=nsa; nsa=nta; nta=swap2;
01566         swap1=sb; sb=tb; tb=swap1;
01567         swap2=nsb; nsb=ntb; ntb=swap2;
01568     }
01569
01570     if (ia!=NULL) {
01571         *nia = nsa*ns;
01572         WCOPY(ia,sa,ABS(*nia));
01573     }
01574
01575     if (ib!=NULL) {
01576         *nib = nsb*ns;
01577         WCOPY(ib,sb,ABS(*nib));
01578     }
01579
01580     NumberFree(s,"GetLongModInverses");
01581     NumberFree(t,"GetLongModInverses");
01582     NumberFree(sa,"GetLongModInverses");
01583     NumberFree(sb,"GetLongModInverses");
01584     NumberFree(ta,"GetLongModInverses");
01585     NumberFree(tb,"GetLongModInverses");
01586     NumberFree(x,"GetLongModInverses");
01587     NumberFree(y,"GetLongModInverses");
01588
01589     return 0;
01590 }
01591
01592 /*
01593         #] GetLongModInverses :
01594         #[ Product :        WORD Product(a,na,b)
01595
01596     Multiplies the Long number in a with the WORD b.
01597
01598 */
01599
01600 int Product(UWORD *a, WORD *na, WORD b)
01601 {
01602     WORD i, sgn = 1;
01603     RLONG t, u;
01604     if ( *na < 0 ) { *na = -(*na); sgn = -sgn; }
01605     if ( b < 0 ) { b = -b; sgn = -sgn; }
01606     t = 0;
01607     u = (RLONG)b;
01608     for ( i = 0; i < *na; i++ ) {
01609         t += *a * u;
01610         *a++ = (UWORD)t;
01611         t »= BITSINWORD;
01612     }
01613     if ( t > 0 ) {
01614         if ( ++(*na) > AM.MaxTal ) {
01615             MLOCK(ErrorMessageLock);
01616             MesPrint("Overflow in Product");
01617             MUNLOCK(ErrorMessageLock);
01618             return(-1);
01619         }
01620         *a = (UWORD)t;
01621     }
01622     if ( sgn < 0 ) *na = -(*na);
01623     return(0);
01624 }
01625
01626 /*
01627         #] Product :
01628         #[ Quotient :       UWORD Quotient(a,na,b)
01629
01630         Routine divides the long number a by b with the assumption that
01631         there is no remainder (like while computing binomials).
01632
01633 */
01634
01635 UWORD Quotient(UWORD *a, WORD *na, WORD b)
01636 {
01637     RLONG v, t;
01638     WORD i, j, sgn = 1;
01639     if ( ( i = *na ) < 0 ) { sgn = -1; i = -i; }
01640     if ( b < 0 ) { b = -b; sgn = -sgn; }
01641     if ( i == 1 ) {
01642         if ( ( *a /= (UWORD)b ) == 0 ) *na = 0;
01643         if ( sgn < 0 ) *na = -*na;
01644         return(0);
01645     }
01646     a += i;
```

```
01647     j = i;
01648     v = (RLONG)b;
01649     t = (RLONG)(*--a);
01650     while ( --i >= 0 ) {
01651         *a = t / v;
01652         t -= v * (*a);
01653         if ( i ) {
01654             t «= BITSINWORD;
01655             t += *--a;
01656         }
01657     }
01658     a += j - 1;
01659     if ( !*a ) j--;
01660     if ( sgn < 0 ) j = -j;
01661     *na = j;
01662     return(0);
01663 }
01664
01665 /*
01666         #] Quotient :
01667         #[ Remain10 :       WORD Remain10(a,na)
01668
01669     Routine divides a by 10 and gives the remainder as return value.
01670     The value of a will be the quotient! a must be positive.
01671
01672 */
01673
01674 WORD Remain10(UWORD *a, WORD *na)
01675 {
01676     WORD i;
01677     RLONG t, u;
01678     UWORD *b;
01679     i = *na;
01680     t = 0;
01681     b = a + i - 1;
01682     while ( --i >= 0 ) {
01683         t += *b;
01684         *b-- = u = t / 10;
01685         t -= u * 10;
01686         if ( i > 0 ) t «= BITSINWORD;
01687     }
01688     if ( ( *na > 0 ) && !a[*na-1] ) (*na)--;
01689     return((WORD)t);
01690 }
01691
01692 /*
01693         #] Remain10 :
01694         #[ Remain4 :        WORD Remain4(a,na)
01695
01696     Routine divides a by 10000 and gives the remainder as return value.
01697     The value of a will be the quotient! a must be positive.
01698
01699 */
01700
01701 WORD Remain4(UWORD *a, WORD *na)
01702 {
01703     WORD i;
01704     RLONG t, u;
01705     UWORD *b;
01706     i = *na;
01707     t = 0;
01708     b = a + i - 1;
01709     while ( --i >= 0 ) {
01710         t += *b;
01711         *b-- = u = t / 10000;
01712         t -= u * 10000;
01713         if ( i > 0 ) t «= BITSINWORD;
01714     }
01715     if ( ( *na > 0 ) && !a[*na-1] ) (*na)--;
01716     return((WORD)t);
01717 }
01718
01719 /*
01720         #] Remain4 :
01721         #[ PrtLong :        void PrtLong(a,na,s)
01722
01723     Puts the long number a in string s.
01724
01725 */
01726
01727 void PrtLong(UWORD *a, WORD na, UBYTE *s)
01728 {
01729     GETIDENTITY
01730     WORD q, i;
01731     UBYTE *sa, *sb;
01732     UBYTE c;
01733     UWORD *bb, *b;
```

```
01734
01735     if ( na < 0 ) {
01736         *s++ = '-';
01737         na = -na;
01738     }
01739
01740     b = NumberMalloc("PrtLong");
01741     bb = b;
01742     i = na; while ( --i >= 0 ) *bb++ = *a++;
01743     a = b;
01744     if ( na > 2 ) {
01745         sa = s;
01746         do {
01747             q = Remain4(a,&na);
01748             *sa++ = (UBYTE)('0' + (q%10));
01749             q /= 10;
01750             *sa++ = (UBYTE)('0' + (q%10));
01751             q /= 10;
01752             *sa++ = (UBYTE)('0' + (q%10));
01753             q /= 10;
01754             *sa++ = (UBYTE)('0' + (q%10));
01755         } while ( na );
01756         while ( sa[-1] == '0' ) sa--;
01757         sb = s;
01758         s = sa;
01759         sa--;
01760         while ( sa > sb ) { c = *sa; *sa = *sb; *sb = c; sa--; sb++; }
01761     }
01762     else if ( na ) {
01763         sa = s;
01764         do {
01765             q = Remain10(a,&na);
01766             *sa++ = (UBYTE)('0' + q);
01767         } while ( na );
01768         sb = s;
01769         s = sa;
01770         sa--;
01771         while ( sa > sb ) { c = *sa; *sa = *sb; *sb = c; sa--; sb++; }
01772     }
01773     else *s++ = '0';
01774     *s = '\0';
01775     NumberFree(b,"PrtLong");
01776 }
01777
01778 /*
01779         #] PrtLong :
01780         #[ GetLong :        WORD GetLong(s,a,na)
01781
01782     Reads a long number from a string.
01783     The string is zero terminated and contains only digits!
01784
01785     New algorithm: try to read 4 digits together before the result
01786     is accumulated.
01787 */
01788
01789 int GetLong(UBYTE *s, UWORD *a, WORD *na)
01790 {
01791 /*
01792     UWORD digit;
01793     *a = 0;
01794     *na = 0;
01795     while ( FG.cTable[*s] == 1 ) {
01796         digit = *s++ - '0';
01797         if ( *na && Product(a,na,(WORD)10) ) return(-1);
01798         if ( digit && AddLong(a,*na,&digit,(WORD)1,a,na) ) return(-1);
01799     }
01800     return(0);
01801 */
01802     UWORD digit, x = 0, y = 0;
01803     *a = 0;
01804     *na = 0;
01805     while ( FG.cTable[*s] == 1 ) {
01806         x = *s++ - '0';
01807         if ( FG.cTable[*s] != 1 ) { y = 10; break; }
01808         x = 10*x + *s++ - '0';
01809         if ( FG.cTable[*s] != 1 ) { y = 100; break; }
01810         x = 10*x + *s++ - '0';
01811         if ( FG.cTable[*s] != 1 ) { y = 1000; break; }
01812         x = 10*x + *s++ - '0';
01813         if ( *na && Product(a,na,(WORD)10000) ) return(-1);
01814         if ( ( digit = x ) != 0 && AddLong(a,*na,&digit,(WORD)1,a,na) )
01815             return(-1);
01816         y = 0;
01817     }
01818     if ( y ) {
01819         if ( *na && Product(a,na,(WORD)y) ) return(-1);
01820         if ( ( digit = x ) != 0 && AddLong(a,*na,&digit,(WORD)1,a,na) )
```

```
01821             return(-1);
01822     }
01823     return(0);
01824 }
01825
01826 /*
01827         #] GetLong :
01828         #[ GCD :            WORD GCD(a,na,b,nb,c,nc)
01829
01830     Algorithm to compute the GCD of two long numbers.
01831     See Knuth, sec 4.5.2 algorithm L.
01832
01833     We assume that both numbers are positive
01834
01835     NOTE!!!!!. NumberMalloc gets called and it may not be freed
01836 */
01837
01838 #ifdef EXTRAGCD
01839
01840 #define Convert(ia,aa,naa) \
01841     if ( (LONG)ia < 0 ) {         \
01842         ia = (ULONG)(-(LONG)ia); \
01843         aa[0] = ia;               \
01844         if ( ( aa[1] = ia » BITSINWORD ) != 0 ) naa = -2; \
01845         else naa = -1;            \
01846     }                             \
01847     else if ( ia == 0 ) { aa[0] = 0; naa = 0; } \
01848     else {                        \
01849         aa[0] = ia;               \
01850         if ( ( aa[1] = ia » BITSINWORD ) != 0 ) naa = 2; \
01851         else naa = 1;             \
01852     }
01853
01854 void GCD(UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
01855 {
01856     int ja = 0, jb = 0, j;
01857     UWORD *r,*t;
01858     UWORD *x1, *x2, *x3;
01859     WORD nd,naa,nbb;
01860     ULONG ia,ib,ic,id,u,v,w,q,T;
01861     UWORD aa[2], bb[2];
01862 /*
01863     First eliminate easy powers of 2^...
01864 */
01865     while ( a[0] == 0 ) { na--; ja++; a++; }
01866     while ( b[0] == 0 ) { nb--; jb++; b++; }
01867     if ( ja > jb ) ja = jb;
01868     if ( ja > 0 ) {
01869         j = ja;
01870         do { *c++ = 0; } while ( --j > 0 );
01871     }
01872 /*
01873     Now arrange things such that a >= b
01874 */
01875     if ( na < nb ) {
01876         jb = na; na = nb; nb = jb;
01877 exch:
01878         r = a; a = b; b = r;
01879     }
01880     else if ( na == nb ) {
01881         r = a+na;
01882         t = b+nb;
01883         j = na;
01884         while ( --j >= 0 ) {
01885             if ( *--r > *--t ) break;
01886             if ( *r < *t ) goto exch;
01887         }
01888         if ( j < 0 ) {
01889 out:
01890             j = nb;
01891             NCOPY(c,b,j);
01892             *nc = nb+ja;
01893             return;
01894         }
01895     }
01896 /*
01897     {
01898         MLOCK(ErrorMessageLock);
01899         MesPrint("Ordered input, ja = %d",(WORD)ja);
01900         AO.OutSkip = 3;
01901         FiniLine();
01902         j = na; r = a;
01903         while ( --j >= 0 ) { TalToLine((UWORD)(*r++)); TokenToLine((UBYTE *)"  "); }
01904         FiniLine();
01905         j = nb; r = b;
01906         while ( --j >= 0 ) { TalToLine((UWORD)(*r++)); TokenToLine((UBYTE *)"  "); }
01907         AO.OutSkip = 0;
```

```
01908         FiniLine();
01909         MUNLOCK(ErrorMessageLock);
01910     }
01911 */
01912 /*
01913     We have now that A > B
01914     The loop recognizes the case that na-nb >= 1
01915     In that case we just have to divide!
01916 */
01917     r = x1 = NumberMalloc("GCD"); t = x2 = NumberMalloc("GCD"); x3 = NumberMalloc("GCD");
01918     j = na;
01919     NCOPY(r,a,j);
01920     j = nb;
01921     NCOPY(t,b,j);
01922
01923     for(;;) {
01924
01925         while ( na > nb ) {
01926 toobad:
01927             DivLong(x1,na,x2,nb,c,nc,x3,&nd);
01928             if ( nd == 0 ) { b = x2; goto out; }
01929             t = x1; x1 = x2; x2 = x3; x3 = t; na = nb; nb = nd;
01930             if ( na == 2 ) break;
01931         }
01932 /*
01933     Here we can use the shortcut.
01934 */
01935     if ( na == 2 ) {
01936         v = x1[0] + ( ((ULONG)x1[1]) « BITSINWORD );
01937         w = x2[0];
01938         if ( nb == 2 ) w += ((ULONG)x2[1]) « BITSINWORD;
01939 #ifdef EXTRAGCD2
01940         v = GCD2(v,w);
01941 #else
01942         do { u = v%w; v = w; w = u; } while ( w );
01943 #endif
01944         c[0] = (UWORD)v;
01945         if ( ( c[1] = (UWORD)(v » BITSINWORD) ) != 0 ) *nc = 2+ja;
01946         else *nc = 1+ja;
01947         NumberFree(x1,"GCD"); NumberFree(x2,"GCD"); NumberFree(x3,"GCD");
01948         return;
01949     }
01950     if ( na == 1 ) {
01951         UWORD ui, uj;
01952         ui = x1[0]; uj = x2[0];
01953 #ifdef EXTRAGCD2
01954         ui = (UWORD)GCD2((ULONG)ui,(ULONG)uj);
01955 #else
01956         do { nd = ui%uj; ui = uj; uj = nd; } while ( nd );
01957 #endif
01958         c[0] = ui;
01959         *nc = 1 + ja;
01960         NumberFree(x1,"GCD"); NumberFree(x2,"GCD"); NumberFree(x3,"GCD");
01961         return;
01962     }
01963     ia = 1; ib = 0; ic = 0; id = 1;
01964     u = ( ((ULONG)x1[na-1]) « BITSINWORD ) + x1[na-2];
01965     v = ( ((ULONG)x2[nb-1]) « BITSINWORD ) + x2[nb-2];
01966
01967     while ( v+ic != 0 && v+id != 0 &&
01968          ( q = (u+ia)/(v+ic) ) == (u+ib)/(v+id) ) {
01969         T = ia-q*ic; ia = ic; ic = T;
01970         T = ib-q*id; ib = id; id = T;
01971         T = u - q*v; u = v; v = T;
01972     }
01973     if ( ib == 0 ) goto toobad;
01974     Convert(ia,aa,naa);
01975     Convert(ib,bb,nbb);
01976     MulLong(x1,na,aa,naa,x3,&nd);
01977     MulLong(x2,nb,bb,nbb,c,nc);
01978     AddLong(x3,nd,c,*nc,c,nc);
01979     Convert(ic,aa,naa);
01980     Convert(id,bb,nbb);
01981     MulLong(x1,na,aa,naa,x3,&nd);
01982     t = c; na = j = *nc; r = x1;
01983     NCOPY(r,t,j);
01984     MulLong(x2,nb,bb,nbb,c,nc);
01985     AddLong(x3,nd,c,*nc,x2,&nb);
01986     }
01987 }
01988
01989 #endif
01990
01991 /*
01992         #] GCD :
01993         #[ GcdLong :        WORD GcdLong(a,na,b,nb,c,nc)
01994
```

```
01995     Returns the Greatest Common Divider of a and b in c.
01996     If a and or b are zero an error message will be returned.
01997     The answer is always positive.
01998     In principle a and c can be the same.
01999 */
02000
02001 #ifndef NEWTRICK
02002 /*
02003     #[ Old Routine :
02004 */
02005
02006 int GcdLong(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
02007 {
02008     GETBIDENTITY
02009     if ( !na || !nb ) {
02010         if ( !na && !nb ) {
02011 /* INTERNAL_ERROR_EXCL_START */
02012             MLOCK(ErrorMessageLock);
02013             MesPrint("!>Cannot take gcd");
02014             MUNLOCK(ErrorMessageLock);
02015             return(-1);
02016 /* INTERNAL_ERROR_EXCL_STOP */
02017         }
02018
02019         if ( !na ) {
02020             *nc = abs(nb);
02021             NCOPY(c,b,*nc);
02022             *nc = abs(nb);
02023             return(0);
02024         }
02025
02026         *nc = abs(na);
02027         NCOPY(c,a,*nc);
02028         *nc = abs(na);
02029         return(0);
02030     }
02031     if ( na < 0 ) na = -na;
02032     if ( nb < 0 ) nb = -nb;
02033     if ( na == 1 && nb == 1 ) {
02034 #ifdef EXTRAGCD2
02035         *c = (UWORD)GCD2((ULONG)*a,(ULONG)*b);
02036 #else
02037         UWORD x,y,z;
02038         x = *a;
02039         y = *b;
02040         do { z = x % y; x = y; } while ( ( y = z ) != 0 );
02041         *c = x;
02042 #endif
02043         *nc = 1;
02044     }
02045     else if ( na <= 2 && nb <= 2 ) {
02046         RLONG lx,ly,lz;
02047         if ( na == 2 ) { lx = (((RLONG)(a[1]))«BITSINWORD) + *a; }
02048         else { lx = *a; }
02049         if ( nb == 2 ) { ly = (((RLONG)(b[1]))«BITSINWORD) + *b; }
02050         else { ly = *b; }
02051 #ifdef LONGWORD
02052 #ifdef EXTRAGCD2
02053         lx = GCD2(lx,ly);
02054 #else
02055         do { lz = lx % ly; lx = ly; } while ( ( ly = lz ) != 0 );
02056 #endif
02057 #else
02058           if ( lx < ly ) { lz = lx; lx = ly; ly = lz; }
02059         do {
02060             lz = lx % ly; lx = ly;
02061         } while ( ( ly = lz ) != 0 && ( lx & AWORDMASK ) != 0 );
02062         if ( ly ) {
02063             do { *c = ((UWORD)lx)%((UWORD)ly); lx = ly; } while ( ( ly = *c ) != 0 );
02064             *c = (UWORD)lx;
02065             *nc = 1;
02066         }
02067         else
02068 #endif
02069         {
02070             *c++ = (UWORD)lx;
02071             if ( ( *c = (UWORD)(lx » BITSINWORD) ) != 0 ) *nc = 2;
02072             else *nc = 1;
02073         }
02074     }
02075     else {
02076 #ifdef EXTRAGCD
02077         GCD(a,na,b,nb,c,nc);
02078 #else
02079 #ifdef NEWGCD
02080         UWORD *x3,*x1,*x2, *GLscrat7, *GLscrat8;
02081         WORD n1,n2,n3,n4;
```

```
02082         WORD i, j;
02083         x1 = c; x3 = a; n1 = i = na;
02084         NCOPY(x1,x3,i);
02085         GLscrat7 = NumberMalloc("GcdLong"); GLscrat8 = NumberMalloc("GcdLong");
02086         x2 = GLscrat8; x3 = b; n2 = i = nb;
02087         NCOPY(x2,x3,i);
02088         x1 = c; i = 0;
02089         while ( x1[0] == 0 ) { i += BITSINWORD; x1++; n1--; }
02090         while ( ( x1[0] & 1 ) == 0 ) { i++; SCHUIF(x1,n1) }
02091         x2 = GLscrat8; j = 0;
02092         while ( x2[0] == 0 ) { j += BITSINWORD; x2++; n2--; }
02093         while ( ( x2[0] & 1 ) == 0 ) { j++; SCHUIF(x2,n2) }
02094         if ( j > i ) j = i;     /* powers of two in GCD */
02095         for(;;){
02096             if ( n1 > n2 ) {
02097 firstbig:
02098                 SubPLon(x1,n1,x2,n2,x1,&n3);
02099                 n1 = n3;
02100                 if ( n1 == 0 ) {
02101                     x1 = c;
02102                     n1 = i = n2; NCOPY(x1,x2,i);
02103                     break;
02104                 }
02105                 while ( ( x1[0] & 1 ) == 0 ) SCHUIF(x1,n1)
02106                 if ( n1 == 1 ) {
02107                     if ( DivLong(x2,n2,x1,n1,GLscrat7,&n3,x2,&n4) ) goto GcdErr;
02108                     n2 = n4;
02109                     if ( n2 == 0 ) {
02110                         i = n1; x2 = c; NCOPY(x2,x1,i);
02111                         break;
02112                     }
02113 #ifdef EXTRAGCD2
02114                     *c = (UWORD)GCD2((ULONG)x1[0],(ULONG)x2[0]);
02115 #else
02116                     {
02117                         UWORD x,y,z;
02118                         x = x1[0];
02119                         y = x2[0];
02120                         do { z = x % y; x = y; } while ( ( y = z ) != 0 );
02121                         *c = x;
02122                     }
02123 #endif
02124                     n1 = 1;
02125                     break;
02126                 }
02127             }
02128             else if ( n1 < n2 ) {
02129 lastbig:
02130                 SubPLon(x2,n2,x1,n1,x2,&n3);
02131                 n2 = n3;
02132                 if ( n2 == 0 ) {
02133                     i = n1; x2 = c; NCOPY(x2,x1,i);
02134                     break;
02135                 }
02136                 while ( ( x2[0] & 1 ) == 0 ) SCHUIF(x2,n2)
02137                 if ( n2 == 1 ) {
02138                     if ( DivLong(x1,n1,x2,n2,GLscrat7,&n3,x1,&n4) ) goto GcdErr;
02139                     n1 = n4;
02140                     if ( n1 == 0 ) {
02141                         x1 = c;
02142                         n1 = i = n2; NCOPY(x1,x2,i);
02143                         break;
02144                     }
02145 #ifdef EXTRAGCD2
02146                     *c = (UWORD)GCD2((ULONG)x2[0],(ULONG)x1[0]);
02147 #else
02148                     {
02149                         UWORD x,y,z;
02150                         x = x2[0];
02151                         y = x1[0];
02152                         do { z = x % y; x = y; } while ( ( y = z ) != 0 );
02153                         *c = x;
02154                     }
02155 #endif
02156                     n1 = 1;
02157                     break;
02158                 }
02159             }
02160             else {
02161                 for ( i = n1-1; i >= 0; i-- ) {
02162                     if ( x1[i] > x2[i] ) goto firstbig;
02163                     else if ( x1[i] < x2[i] ) goto lastbig;
02164                 }
02165                 i = n1; x2 = c; NCOPY(x2,x1,i);
02166                 break;
02167             }
02168         }
```

```
02169 /*
02170         Now the GCD is in c but still needs j powers of 2.
02171 */
02172         x1 = c;
02173         while ( j >= BITSINWORD ) {
02174             for ( i = n1; i > 0; i-- ) x1[i] = x1[i-1];
02175             x1[0] = 0; n1++;
02176             j -= BITSINWORD;
02177         }
02178         if ( j > 0 ) {
02179             ULONG a1,a2 = 0;
02180             for ( i = 0; i < n1; i++ ) {
02181                 a1 = x1[i]; a1 «= j;
02182                 a2 += a1;
02183                 x1[i] = a2;
02184                 a2 »= BITSINWORD;
02185             }
02186             if ( a2 != 0 ) {
02187                 x1[n1++] = a2;
02188             }
02189         }
02190         *nc = n1;
02191         NumberFree(GLscrat7,"GcdLong"); NumberFree(GLscrat8,"GcdLong");
02192 #else
02193         UWORD *x1,*x2,*x3,*x4,*c1,*c2;
02194         WORD n1,n2,n3,n4,i;
02195         x1 = c; x3 = a; n1 = i = na;
02196         NCOPY(x1,x3,i);
02197         x1 = c; c1 = x2 = NumberMalloc("GcdLong"); x3 = NumberMalloc("GcdLong"); x4 =
    NumberMalloc("GcdLong");
02198         c2 = b; n2 = i = nb;
02199         NCOPY(c1,c2,i);
02200         for(;;){
02201             if ( DivLong(x1,n1,x2,n2,x4,&n4,x3,&n3) ) goto GcdErr;
02202             if ( !n3 ) { x1 = x2; n1 = n2; break; }
02203             if ( DivLong(x2,n2,x3,n3,x4,&n4,x1,&n1) ) goto GcdErr;
02204             if ( !n1 ) { x1 = x3; n1 = n3; break; }
02205             if ( DivLong(x3,n3,x1,n1,x4,&n4,x2,&n2) ) goto GcdErr;
02206             if ( !n2 ) {
02207                 *nc = n1;
02208                 NumberFree(x2,"GcdLong"); NumberFree(x3,"GcdLong"); NumberFree(x4,"GcdLong");
02209                 return(0);
02210             }
02211         }
02212         *nc = i = n1;
02213         NCOPY(c,x1,i);
02214         NumberFree(x2,"GcdLong"); NumberFree(x3,"GcdLong"); NumberFree(x4,"GcdLong");
02215 #endif
02216 #endif
02217     }
02218     return(0);
02219 GcdErr:
02220     MLOCK(ErrorMessageLock);
02221     MesCall("GcdLong");
02222     MUNLOCK(ErrorMessageLock);
02223     SETERROR(-1)
02224 }
02225 /*
02226     #] Old Routine :
02227 */
02228 #else
02229
02230 /*
02231     New routine for GcdLong that uses smart shortcuts.
02232     Algorithm by J. Vermaseren 15-nov-2006.
02233     It runs faster for very big numbers but only by a fixed factor.
02234     There is no improvement in the power behaviour.
02235     Improvement on the whole of hf9 (multiple zeta values at weight 9):
02236         Better than a factor 2 on a 32 bits architecture and 2.76 on a
02237         64 bits architecture.
02238     On hf10 (MZV's at weight 10), 64 bits architecture: factor 7.
02239
02240     If we have two long numbers (na,nb > GCDMAX) we will work in a
02241     truncated way. At the moment of writing (15-nov-2006) it isn't
02242     clear whether this algorithm is an invention or a reinvention.
02243     A short search on the web didn't show anything.
02244
02245     31-jul-2007:
02246     A better search shows that this is an adaptation of the Lehmer-Euclid
02247     algorithm, already described in Knuth. Here we can work without upper
02248     and lower limit because we are only interested in the GCD, not the
02249     extra numbers. Also it takes already some features of the double
02250     digit Lehmer-Euclid algorithm of Jebelean it seems.
02251
02252     Maybe this can be programmed slightly better and we can get another
02253     few percent speed increase. Further improvements for the asymptotic
02254     case come from splitting the calculation as in Karatsuba and working
```

```
02255     with FFT divisions and multiplications etc. But this is when hundreds
02256     of words are involved at the least.
02257
02258     Algorithm
02259
02260     1: while ( na > nb || nb < GCDMAX ) {
02261         if ( nb == 0 ) { result in a }
02262         c = a % b;
02263         a = b;
02264         b = c;
02265        }
02266     2: Make the truncated values in which a and b are the combinations
02267        of the top two words of a and b. The whole numbers are aa and bb now.
02268     3: ma1 = 1; ma2 = 0; mb1 = 0; mb2 = 1;
02269     4: A = a; B = b; m = a/b; c = a - m*b;
02270        c = ma1*a+ma2*b-m*(mb1*a+mb2*b) = (ma1-m*mb1)*a+(ma2-m*mb2)*b
02271        mc1 = ma1-m*mb1; mc2 = ma2-m*mb2;
02272     5: a = b; ma1 = mb1; ma2 = mb2;
02273        b = c; mb1 = mc1; mb2 = mc2;
02274     6: if ( b != 0 && nb >= FULLMAX ) goto 4;
02275     7: Now construct the new quantities
02276         ma1*aa+ma2*bb and mb1*aa+mb2*bb
02277     8: goto 1;
02278
02279     The essence of the above algorithm is that we do the divisions only
02280     on relatively short numbers. Also usually there are many steps 4&5
02281     for each step 7. This eliminates many operations.
02282     The termination at FULLMAX is that we make errors by not considering
02283     the tail of the number. If we run b down all the way, the errors combine
02284     in such a way that the new numbers may be of the same order as the old
02285     numbers. By stopping halfway we don't get the error beyond halfway
02286     either. Unfortunately this means that a >= FULLMAX and hence na > nb
02287     which means that next we will have a complete division. But just once.
02288     Running the steps 4-6 till a < FULLMAX runs already into problems.
02289     It may be necessary to experiment a bit to obtain the optimum value
02290     of GCDMAX.
02291 */
02292
02293 int GcdLong(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
02294 {
02295     GETBIDENTITY
02296     UWORD x,y,z;
02297     UWORD *x1,*x2,*x3,*x4,*x5,*d;
02298     UWORD *GLscrat6, *GLscrat7, *GLscrat8, *GLscrat9, *GLscrat10;
02299     WORD n1,n2,n3,n4,n5,i;
02300     RLONG lx,ly,lz;
02301     LONG ma1, ma2, mb1, mb2, mc1, mc2, m;
02302     if ( !na || !nb ) {
02303         if ( !na && !nb ) {
02304 /* INTERNAL_ERROR_EXCL_START */
02305             MLOCK(ErrorMessageLock);
02306             MesPrint("!>Cannot take gcd");
02307             MUNLOCK(ErrorMessageLock);
02308             return(-1);
02309 /* INTERNAL_ERROR_EXCL_STOP */
02310         }
02311
02312         if ( !na ) {
02313             *nc = abs(nb);
02314             NCOPY(c,b,*nc);
02315             *nc = abs(nb);
02316             return(0);
02317         }
02318
02319         *nc = abs(na);
02320         NCOPY(c,a,*nc);
02321         *nc = abs(na);
02322         return(0);
02323     }
02324     if ( na < 0 ) na = -na;
02325     if ( nb < 0 ) nb = -nb;
02326 /*
02327     #[ GMP stuff :
02328 */
02329 #ifdef WITHGMP
02330     if ( na > 3 && nb > 3 ) {
02331         int ii;
02332         mp_limb_t *upa, *upb, *upc, xx;
02333         UWORD *uw, *u1, *u2;
02334         unsigned int tcounta, tcountb, tcounta1, tcountb1;
02335         mp_size_t ana, anb, anc;
02336
02337         u1 = uw = NumberMalloc("GcdLong");
02338         upa = (mp_limb_t *)u1;
02339         ana = na; tcounta1 = 0;
02340         while ( a[0] == 0 ) { a++; ana--; tcounta1++; }
02341         for ( ii = 0; ii < ana; ii++ ) { *uw++ = *a++; }
```

```
02342         if ( ( ana & 1 ) != 0 ) { *uw = 0; ana++; }
02343         ana /= 2;
02344
02345         u2 = uw = NumberMalloc("GcdLong");
02346         upb = (mp_limb_t *)u2;
02347         anb = nb; tcountb1 = 0;
02348         while ( b[0] == 0 ) { b++; anb--; tcountb1++; }
02349         for ( ii = 0; ii < anb; ii++ ) { *uw++ = *b++; }
02350         if ( ( anb & 1 ) != 0 ) { *uw = 0; anb++; }
02351         anb /= 2;
02352
02353         xx = upa[0]; tcounta = 0;
02354         while ( ( xx & 15 ) == 0 ) { tcounta += 4; xx »= 4; }
02355         while ( ( xx &  1 ) == 0 ) { tcounta += 1; xx »= 1; }
02356         xx = upb[0]; tcountb = 0;
02357         while ( ( xx & 15 ) == 0 ) { tcountb += 4; xx »= 4; }
02358         while ( ( xx &  1 ) == 0 ) { tcountb += 1; xx »= 1; }
02359
02360         if ( tcounta ) {
02361             mpn_rshift(upa,upa,ana,tcounta);
02362             if ( upa[ana-1] == 0 ) ana--;
02363         }
02364         if ( tcountb ) {
02365             mpn_rshift(upb,upb,anb,tcountb);
02366             if ( upb[anb-1] == 0 ) anb--;
02367         }
02368
02369         upc = (mp_limb_t *)(NumberMalloc("GcdLong"));
02370         if ( ( ana > anb ) || ( ( ana == anb ) && ( upa[ana-1] >= upb[ana-1] ) ) ) {
02371             anc = mpn_gcd(upc,upa,ana,upb,anb);
02372         }
02373         else {
02374             anc = mpn_gcd(upc,upb,anb,upa,ana);
02375         }
02376
02377         tcounta = tcounta1*BITSINWORD + tcounta;
02378         tcountb = tcountb1*BITSINWORD + tcountb;
02379         if ( tcountb > tcounta ) tcountb = tcounta;
02380         tcounta = tcountb/BITSINWORD;
02381         tcountb = tcountb%BITSINWORD;
02382
02383         if ( tcountb ) {
02384             xx = mpn_lshift(upc,upc,anc,tcountb);
02385             if ( xx ) { upc[anc] = xx; anc++; }
02386         }
02387
02388         uw = (UWORD *)upc; anc *= 2;
02389         while ( uw[anc-1] == 0 ) anc--;
02390         for ( ii = 0; ii < (int)tcounta; ii++ ) *c++ = 0;
02391         for ( ii = 0; ii < anc; ii++ ) *c++ = *uw++;
02392         *nc = anc + tcounta;
02393         NumberFree(u1,"GcdLong"); NumberFree(u2,"GcdLong"); NumberFree((UWORD *)(upc),"GcdLong");
02394         return(0);
02395     }
02396 #endif
02397 /*
02398     #] GMP stuff :
02399 */
02400 /*
02401     #[ Easy cases :
02402 */
02403     if ( na == 1 && nb == 1 ) {
02404         x = *a;
02405         y = *b;
02406         do { z = x % y; x = y; } while ( ( y = z ) != 0 );
02407         *c = x;
02408         *nc = 1;
02409         return(0);
02410     }
02411     else if ( na <= 2 && nb <= 2 ) {
02412         if ( na == 2 ) { lx = (((RLONG)(a[1]))«BITSINWORD) + *a; }
02413         else { lx = *a; }
02414         if ( nb == 2 ) { ly = (((RLONG)(b[1]))«BITSINWORD) + *b; }
02415         else { ly = *b; }
02416         if ( lx < ly ) { lz = lx; lx = ly; ly = lz; }
02417 #if ( BITSINWORD == 16 )
02418         do {
02419             lz = lx % ly; lx = ly;
02420         } while ( ( ly = lz ) != 0 );
02421 #else
02422         do {
02423             lz = lx % ly; lx = ly;
02424         } while ( ( ly = lz ) != 0 && ( lx & AWORDMASK ) != 0 );
02425         if ( ly ) {
02426             x = (UWORD)lx; y = (UWORD)ly;
02427             do { *c = x % y; x = y; } while ( ( y = *c ) != 0 );
02428             *c = x;
```

```
02429             *nc = 1;
02430         }
02431         else
02432 #endif
02433         {
02434             *c++ = (UWORD)lx;
02435             if ( ( *c = (UWORD)(lx » BITSINWORD) ) != 0 ) *nc = 2;
02436             else *nc = 1;
02437         }
02438         return(0);
02439     }
02440 /*
02441     #] Easy cases :
02442 */
02443     GLscrat6 = NumberMalloc("GcdLong"); GLscrat7 = NumberMalloc("GcdLong");
02444     GLscrat8 = NumberMalloc("GcdLong");
02445     GLscrat9 = NumberMalloc("GcdLong"); GLscrat10 = NumberMalloc("GcdLong");
02446 restart:;
02447 /*
02448     #[ Easy cases :
02449 */
02450     if ( na == 1 && nb == 1 ) {
02451         x = *a;
02452         y = *b;
02453         do { z = x % y; x = y; } while ( ( y = z ) != 0 );
02454         *c = x;
02455         *nc = 1;
02456     }
02457     else if ( na <= 2 && nb <= 2 ) {
02458         if ( na == 2 ) { lx = (((RLONG)(a[1]))«BITSINWORD) + *a; }
02459         else { lx = *a; }
02460         if ( nb == 2 ) { ly = (((RLONG)(b[1]))«BITSINWORD) + *b; }
02461         else { ly = *b; }
02462         if ( lx < ly ) { lz = lx; lx = ly; ly = lz; }
02463 #if ( BITSINWORD == 16 )
02464         do {
02465             lz = lx % ly; lx = ly;
02466         } while ( ( ly = lz ) != 0 );
02467 #else
02468         do {
02469             lz = lx % ly; lx = ly;
02470         } while ( ( ly = lz ) != 0 && ( lx & AWORDMASK ) != 0 );
02471         if ( ly ) {
02472             x = (UWORD)lx; y = (UWORD)ly;
02473             do { *c = x % y; x = y; } while ( ( y = *c ) != 0 );
02474             *c = x;
02475             *nc = 1;
02476         }
02477         else
02478 #endif
02479         {
02480             *c++ = (UWORD)lx;
02481             if ( ( *c = (UWORD)(lx » BITSINWORD) ) != 0 ) *nc = 2;
02482             else *nc = 1;
02483         }
02484     }
02485 /*
02486     #] Easy cases :
02487     #[ Original code :
02488 */
02489     else if ( na < GCDMAX || nb < GCDMAX || na != nb ) {
02490         if ( na < nb ) {
02491             x2 = GLscrat8; x3 = a; n2 = i = na;
02492             NCOPY(x2,x3,i);
02493             x1 = c; x3 = b; n1 = i = nb;
02494             NCOPY(x1,x3,i);
02495         }
02496         else {
02497             x1 = c; x3 = a; n1 = i = na;
02498             NCOPY(x1,x3,i);
02499             x2 = GLscrat8; x3 = b; n2 = i = nb;
02500             NCOPY(x2,x3,i);
02501         }
02502         x1 = c; x2 = GLscrat8; x3 = GLscrat7; x4 = GLscrat6;
02503         for(;;){
02504             if ( DivLong(x1,n1,x2,n2,x4,&n4,x3,&n3) ) goto GcdErr;
02505             if ( !n3 ) { x1 = x2; n1 = n2; break; }
02506             if ( n2 <= 2 ) { a = x2; b = x3; na = n2; nb = n3; goto restart; }
02507             if ( n3 >= GCDMAX && n2 == n3 ) {
02508                 a = GLscrat9; b = GLscrat10; na = n2; nb = n3;
02509                 for ( i = 0; i < na; i++ ) a[i] = x2[i];
02510                 for ( i = 0; i < nb; i++ ) b[i] = x3[i];
02511                 goto newtrick;
02512             }
02513             if ( DivLong(x2,n2,x3,n3,x4,&n4,x1,&n1) ) goto GcdErr;
02514             if ( !n1 ) { x1 = x3; n1 = n3; break; }
02515             if ( n3 <= 2 ) { a = x3; b = x1; na = n3; nb = n1; goto restart; }
```

```
02516             if ( n1 >= GCDMAX && n1 == n3 ) {
02517                 a = GLscrat9; b = GLscrat10; na = n3; nb = n1;
02518                 for ( i = 0; i < na; i++ ) a[i] = x3[i];
02519                 for ( i = 0; i < nb; i++ ) b[i] = x1[i];
02520                 goto newtrick;
02521             }
02522             if ( DivLong(x3,n3,x1,n1,x4,&n4,x2,&n2) ) goto GcdErr;
02523             if ( !n2 ) { *nc = n1; goto normalend; }
02524             if ( n1 <= 2 ) { a = x1; b = x2; na = n1; nb = n2; goto restart; }
02525             if ( n2 >= GCDMAX && n2 == n1 ) {
02526                 a = GLscrat9; b = GLscrat10; na = n1; nb = n2;
02527                 for ( i = 0; i < na; i++ ) a[i] = x1[i];
02528                 for ( i = 0; i < nb; i++ ) b[i] = x2[i];
02529                 goto newtrick;
02530             }
02531         }
02532         *nc = i = n1;
02533         NCOPY(c,x1,i);
02534     }
02535 /*
02536     #] Original code :
02537     #[ New code :
02538 */
02539     else {
02540 /*
02541         This is the new algorithm starting at step 3.
02542
02543     3: ma1 = 1; ma2 = 0; mb1 = 0; mb2 = 1;
02544     4: A = a; B = b; m = a/b; c = a - m*b;
02545        c = ma1*a+ma2*b-m*(mb1*a+mb2*b) = (ma1-m*mb1)*a+(ma2-m*mb2)*b
02546        mc1 = ma1-m*mb1; mc2 = ma2-m*mb2;
02547     5: a = b; ma1 = mb1; ma2 = mb2;
02548        b = c; mb1 = mc1; mb2 = mc2;
02549     6: if ( b != 0 ) goto 4;
02550 */
02551 newtrick:;
02552         ma1 = 1; ma2 = 0; mb1 = 0; mb2 = 1;
02553         lx = (((RLONG)(a[na-1]))«BITSINWORD) + a[na-2];
02554         ly = (((RLONG)(b[nb-1]))«BITSINWORD) + b[nb-2];
02555         if ( ly > lx ) { lz = lx; lx = ly; ly = lz; d = a; a = b; b = d; }
02556         do {
02557             m = lx/ly;
02558             mc1 = ma1-m*mb1; mc2 = ma2-m*mb2;
02559             ma1 = mb1; ma2 = mb2; mb1 = mc1; mb2 = mc2;
02560             lz = lx - m*ly; lx = ly; ly = lz;
02561         } while ( ly >= FULLMAX );
02562 /*
02563         Next the construction of the two new numbers
02564
02565     7: Now construct the new quantities
02566         a = ma1*aa+ma2*bb and b = mb1*aa+mb2*bb
02567 */
02568         x1 = GLscrat6;
02569         x2 = GLscrat7;
02570         x3 = GLscrat8;
02571         x5 = GLscrat10;
02572         if ( ma1 < 0 ) {
02573             ma1 = -ma1;
02574             x1[0] = (UWORD)ma1;
02575             x1[1] = (UWORD)(ma1 » BITSINWORD);
02576             if ( x1[1] ) n1 = -2;
02577             else         n1 = -1;
02578         }
02579         else {
02580             x1[0] = (UWORD)ma1;
02581             x1[1] = (UWORD)(ma1 » BITSINWORD);
02582             if ( x1[1] ) n1 = 2;
02583             else         n1 = 1;
02584         }
02585         if ( MulLong(a,na,x1,n1,x2,&n2) ) goto GcdErr;
02586         if ( ma2 < 0 ) {
02587             ma2 = -ma2;
02588             x1[0] = (UWORD)ma2;
02589             x1[1] = (UWORD)(ma2 » BITSINWORD);
02590             if ( x1[1] ) n1 = -2;
02591             else         n1 = -1;
02592         }
02593         else {
02594             x1[0] = (UWORD)ma2;
02595             x1[1] = (UWORD)(ma2 » BITSINWORD);
02596             if ( x1[1] ) n1 = 2;
02597             else         n1 = 1;
02598         }
02599         if ( MulLong(b,nb,x1,n1,x3,&n3) ) goto GcdErr;
02600         if ( AddLong(x2,n2,x3,n3,c,&n4) ) goto GcdErr;
02601         if ( mb1 < 0 ) {
02602             mb1 = -mb1;
```

```
02603             x1[0] = (UWORD)mb1;
02604             x1[1] = (UWORD)(mb1 » BITSINWORD);
02605             if ( x1[1] ) n1 = -2;
02606             else         n1 = -1;
02607         }
02608         else {
02609             x1[0] = (UWORD)mb1;
02610             x1[1] = (UWORD)(mb1 » BITSINWORD);
02611             if ( x1[1] ) n1 = 2;
02612             else         n1 = 1;
02613         }
02614         if ( MulLong(a,na,x1,n1,x2,&n2) ) goto GcdErr;
02615         if ( mb2 < 0 ) {
02616             mb2 = -mb2;
02617             x1[0] = (UWORD)mb2;
02618             x1[1] = (UWORD)(mb2 » BITSINWORD);
02619             if ( x1[1] ) n1 = -2;
02620             else         n1 = -1;
02621         }
02622         else {
02623             x1[0] = (UWORD)mb2;
02624             x1[1] = (UWORD)(mb2 » BITSINWORD);
02625             if ( x1[1] ) n1 = 2;
02626             else         n1 = 1;
02627         }
02628         if ( MulLong(b,nb,x1,n1,x3,&n3) ) goto GcdErr;
02629         if ( AddLong(x2,n2,x3,n3,x5,&n5) ) goto GcdErr;
02630         a = c; na = n4; b = x5; nb = n5;
02631         if ( nb == 0 ) { *nc = n4; goto normalend; }
02632         x4 = GLscrat9;
02633         for ( i = 0; i < na; i++ ) x4[i] = a[i];
02634         a = x4;
02635         if ( na < 0 ) na = -na;
02636         if ( nb < 0 ) nb = -nb;
02637 /*
02638         The typical case now is that in a we have the last step to go
02639         to loose the leading word, while in b we have lost the leading word.
02640         We could go to DivLong now but we can also add an extra step that
02641         is less wasteful.
02642         In the case that the new leading word of b is extrememly short (like 1)
02643         we make a rather large error of course. In the worst case the whole
02644         will be intercepted by DivLong after all, but that is so rare that
02645         it shouldn't influence any timing in a measurable way.
02646 */
02647         if ( nb >= GCDMAX && na == nb+1 && b[nb-1] >= HALFMAX && b[nb-1] > a[na-1] ) {
02648             lx = (((RLONG)(a[na-1]))«BITSINWORD) + a[na-2];
02649             x1[0] = lx/b[nb-1]; n1 = 1;
02650             MulLong(b,nb,x1,n1,x2,&n2);
02651             n2 = -n2;
02652             AddLong(a,na,x2,n2,x4,&n4);
02653             if ( n4 == 0 ) {
02654                 *nc = nb;
02655                 for ( i = 0; i < nb; i++ ) c[i] = b[i];
02656                 goto normalend;
02657             }
02658             if ( n4 < 0 ) n4 = -n4;
02659             a = b; na = nb; b = x4; nb = n4;
02660         }
02661         goto restart;
02662 /*
02663     #] New code :
02664 */
02665     }
02666 normalend:
02667     NumberFree(GLscrat6,"GcdLong"); NumberFree(GLscrat7,"GcdLong"); NumberFree(GLscrat8,"GcdLong");
02668     NumberFree(GLscrat9,"GcdLong"); NumberFree(GLscrat10,"GcdLong");
02669     return(0);
02670 GcdErr:
02671     MLOCK(ErrorMessageLock);
02672     MesCall("GcdLong");
02673     MUNLOCK(ErrorMessageLock);
02674     NumberFree(GLscrat6,"GcdLong"); NumberFree(GLscrat7,"GcdLong"); NumberFree(GLscrat8,"GcdLong");
02675     NumberFree(GLscrat9,"GcdLong"); NumberFree(GLscrat10,"GcdLong");
02676     SETERROR(-1)
02677 }
02678
02679 #endif
02680
02681 /*
02682         #] GcdLong :
02683         #[ GetBinom :       WORD GetBinom(a,na,i1,i2)
02684 */
02685
02686 int GetBinom(UWORD *a, WORD *na, WORD i1, WORD i2)
02687 {
02688     GETIDENTITY
02689     WORD j, k, l;
```

```
02690     UWORD *GBscrat3, *GBscrat4;
02691     if ( i1-i2 < i2 ) i2 = i1-i2;
02692     if ( i2 == 0 ) { *a = 1; *na = 1; return(0); }
02693     if ( i2 > i1 ) { *a = 0; *na = 0; return(0); }
02694     *a = i1; *na = 1;
02695     GBscrat3 = NumberMalloc("GetBinom"); GBscrat4 = NumberMalloc("GetBinom");
02696     for ( j = 2; j <= i2; j++ ) {
02697         GBscrat3[0] = i1+1-j;
02698         if ( MulLong(a,*na,GBscrat3,(WORD)1,GBscrat4,&k) ) goto CalledFrom;
02699         GBscrat3[0] = j;
02700         if ( DivLong(GBscrat4,k,GBscrat3,(WORD)1,a,na,GBscrat3,&l) ) goto CalledFrom;
02701     }
02702     NumberFree(GBscrat3,"GetBinom"); NumberFree(GBscrat4,"GetBinom");
02703     return(0);
02704 CalledFrom:
02705     MLOCK(ErrorMessageLock);
02706     MesCall("GetBinom");
02707     MUNLOCK(ErrorMessageLock);
02708     NumberFree(GBscrat3,"GetBinom"); NumberFree(GBscrat4,"GetBinom");
02709     SETERROR(-1)
02710 }
02711
02712 /*
02713         #] GetBinom :
02714         #[ LcmLong :        WORD LcmLong(a,na,b,nb)
02715
02716         Computes the LCM of the long numbers a and b and puts the result
02717         in c. c is allowed to be equal to a.
02718 */
02719
02720 int LcmLong(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
02721 {
02722     int error = 0;
02723     UWORD *d = NumberMalloc("LcmLong");
02724     UWORD *e = NumberMalloc("LcmLong");
02725     UWORD *f = NumberMalloc("LcmLong");
02726     WORD nd, ne, nf;
02727     GcdLong(BHEAD a, na, b, nb, d, &nd);
02728     DivLong(a,na,d,nd,e,&ne,f,&nf);
02729     if ( MulLong(b,nb,e,ne,c,nc) ) {
02730         MLOCK(ErrorMessageLock);
02731         MesCall("LcmLong");
02732         MUNLOCK(ErrorMessageLock);
02733         error = -1;
02734     }
02735     NumberFree(f,"LcmLong");
02736     NumberFree(e,"LcmLong");
02737     NumberFree(d,"LcmLong");
02738     return(error);
02739 }
02740
02741 /*
02742         #] LcmLong :
02743         #[ TakeLongRoot:    int TakeLongRoot(a,n,power)
02744
02745     Takes the 'power'-root of the long number in a.
02746     If the root could be taken the return value is zero.
02747     If the root could not be taken, the return value is 1.
02748     The root will be in a if it could be taken, otherwise there will be garbage
02749     Algorithm: (assume b is guess of root, b' better guess)
02750         b' = (a-(power-1)*b^power)/(n*b^(power-1))
02751     Note: power should be positive!
02752 */
02753
02754 int TakeLongRoot(UWORD *a, WORD *n, WORD power)
02755 {
02756     GETIDENTITY
02757     int numbits, guessbits, i, retval = 0;
02758     UWORD x, *b, *c, *d, *e;
02759     WORD na, nb, nc, nd, ne;
02760     if ( *n < 0 && ( power & 1 ) == 0 ) return(1);
02761     if ( power == 1 ) return(0);
02762     if ( *n < 0 ) { na = -*n; }
02763     else          { na =  *n; }
02764     if ( na == 1 ) {
02765 /*          Special cases that are the most frequent */
02766         if ( a[0] == 1 ) return(0);
02767         if ( power < BITSINWORD && na == 1 && a[0] == (UWORD)(1«power) ) {
02768             a[0] = 2; return(0);
02769         }
02770         if ( 2*power < BITSINWORD && na == 1 && a[0] == (UWORD)(1«(2*power)) ) {
02771             a[0] = 4; return(0);
02772         }
02773     }
02774 /*
02775     Step 1: make a guess. We count the number of bits.
02776     numbits will be the 1+2log(a)
```

```
02777 */
02778     numbits = BITSINWORD*(na-1);
02779     x = a[na-1];
02780     while ( ( x » 8 ) != 0 ) { numbits += 8; x »= 8; }
02781     if ( ( x » 4 ) != 0 ) { numbits += 4; x »= 4; }
02782     if ( ( x » 2 ) != 0 ) { numbits += 2; x »= 2; }
02783     if ( ( x » 1 ) != 0 ) numbits++;
02784     guessbits = numbits / power;
02785     if ( guessbits <= 0 ) return(1); /* root < 2 and 1 we did already */
02786     nb = guessbits/BITSINWORD;
02787 /*
02788     The recursion is:
02789     (b'-b) = (a/b^(power-1)-b)/n
02790            = (a/c-b)/n
02791            = (d-b)/n    (remainder of a/c is e)
02792            = c/n  (we reuse the scratch array c)
02793     Termination can be tricky. When a/c has no remainder and = b we have a root.
02794     When d = b but the remainder of a/c != 0, there is definitely no root.
02795 */
02796     b = NumberMalloc("TakeLongRoot"); c = NumberMalloc("TakeLongRoot");
02797     d = NumberMalloc("TakeLongRoot"); e = NumberMalloc("TakeLongRoot");
02798     for ( i = 0; i < nb; i++ ) { b[i] = 0; }
02799     b[nb] = 1 « (guessbits%BITSINWORD);
02800     nb++;
02801     for(;;) {
02802         nc = nb;
02803         for ( i = 0; i < nb; i++ ) c[i] = b[i];
02804         if ( RaisPow(BHEAD c,&nc,power-1) ) goto TLcall;
02805         if ( DivLong(a,na,c,nc,d,&nd,e,&ne) ) goto TLcall;
02806         nb = -nb;
02807         if ( AddLong(d,nd,b,nb,c,&nc) ) goto TLcall;
02808         nb = -nb;
02809         if ( nc == 0 ) {
02810             if ( ne == 0 ) break;
02811             retval = 1; break;
02812 /*
02813             else {
02814                 NumberFree(b,"TakeLongRoot"); NumberFree(c,"TakeLongRoot");
02815                 NumberFree(d,"TakeLongRoot"); NumberFree(e,"TakeLongRoot");
02816                 return(1);
02817             }
02818 */
02819         }
02820         DivLong(c,nc,(UWORD *)(&power),1,d,&nd,e,&ne);
02821         if ( nd == 0 ) {
02822             retval = 1;
02823             break;
02824 /*
02825             NumberFree(b,"TakeLongRoot"); NumberFree(c,"TakeLongRoot");
02826             NumberFree(d,"TakeLongRoot"); NumberFree(e,"TakeLongRoot");
02827             return(1);
02828 */
02829 /*
02830             This code tries b+1 as a final possibility.
02831             We believe this is not needed
02832             UWORD one = 1;
02833             if ( AddLong(b,nb,&one,1,c,&nc) ) goto TLcall;
02834             if ( RaisPow(BHEAD c,&nc,power-1) ) goto TLcall;
02835             if ( DivLong(a,na,c,nc,d,&nd,e,&ne) ) goto TLcall;
02836             if ( ne != 0 ) return(1);
02837             nb = -nb;
02838             if ( SubLong(d,nd,b,nb,c,&nc) ) goto TLcall;
02839             nb = -nb;
02840             if ( nc != 0 ) {
02841                 NumberFree(b,"TakeLongRoot"); NumberFree(c,"TakeLongRoot");
02842                 NumberFree(d,"TakeLongRoot"); NumberFree(e,"TakeLongRoot");
02843                 return(1);
02844             }
02845             break;
02846 */
02847         }
02848         if ( AddLong(b,nb,d,nd,b,&nb) ) goto TLcall;
02849     }
02850     for ( i = 0; i < nb; i++ ) a[i] = b[i];
02851     if ( *n < 0 ) *n = -nb;
02852     else          *n =  nb;
02853     NumberFree(b,"TakeLongRoot"); NumberFree(c,"TakeLongRoot");
02854     NumberFree(d,"TakeLongRoot"); NumberFree(e,"TakeLongRoot");
02855     return(retval);
02856 TLcall:
02857     MLOCK(ErrorMessageLock);
02858     MesCall("TakeLongRoot");
02859     MUNLOCK(ErrorMessageLock);
02860     NumberFree(b,"TakeLongRoot"); NumberFree(c,"TakeLongRoot");
02861     NumberFree(d,"TakeLongRoot"); NumberFree(e,"TakeLongRoot");
02862     Terminate(-1);
02863     return(-1);
```

```
02864 }
02865
02866 /*
02867         #] TakeLongRoot:
02868         #[ MakeRational:
02869
02870         Makes the integer a mod m into a traction b/c with |b|,|c| < sqrt(m)
02871         For the algorithm, see MakeLongRational.
02872 */
02873
02874 int MakeRational(WORD a,WORD m, WORD *b, WORD *c)
02875 {
02876     LONG x1,x2,x3,x4,y1,y2;
02877     if ( a < 0 ) { a = a+m; }
02878     if ( a <= 1 ) {
02879         if ( a > m/2 ) a = a-m;
02880         *b = a; *c = 1; return(0);
02881     }
02882     x1 = m; x2 = a;
02883     if ( x2*x2 >= m ) {
02884         y1 = x1/x2; y2 = x1%x2; x3 = 1; x4 = -y1; x1 = x2; x2 = y2;
02885         while ( x2*x2 >= m ) {
02886             y1 = x1/x2; y2 = x1%x2; x1 = x2; x2 = y2; y2 = x3-y1*x4; x3 = x4; x4 = y2;
02887         }
02888     }
02889     else x4 = 1;
02890     if ( x2 == 0 ) { return(1); }
02891     if ( x2 > m/2 ) *b = x2-m;
02892     else            *b = x2;
02893     if ( x4 > m/2 ) { *c = x4-m; *c = -*c; *b = -*b; }
02894     else if ( x4 <= -m/2 ) { x4 += m; *c = x4; }
02895     else if ( x4 < 0 ) { x4 = -x4; *c = x4; *b = -*b; }
02896     else            *c = x4;
02897     return(0);
02898 }
02899
02900 /*
02901         #] MakeRational:
02902         #[ MakeLongRational:
02903
02904         Converts the long number a mod m into the fraction b
02905         One of the properties of b is that num,den < sqrt(m)
02906         The algorithm: Start with:   m 0
02907                                      a 1
02908         Make now c=m%a, c1=m/a       c c2=0-c1*1
02909         Make now d=a%c  d1=a/c       d d2=1-d1*c2
02910         Make now e=c%d  e1=c/d       e e2=1-e1*d2
02911             etc till in the first column we get a number < sqrt(m)
02912             We have then f,f2 and the fraction is f/f2.
02913         If at any moment we get a zero, m contained an unlucky prime.
02914
02915         Note that this can be made a lot faster when we make the same
02916         improvements as in the GCD routine. That is something for later.
02917 #ifdef WITHMAKERATIONAL
02918 */
02919
02920 #define COPYLONG(x1,nx1,x2,nx2) { int i; for(i=0;i<ABS(nx2);i++)x1[i]=x2[i];nx1=nx2; }
02921
02922 int MakeLongRational(PHEAD UWORD *a, WORD na, UWORD *m, WORD nm, UWORD *b, WORD *nb)
02923 {
02924     UWORD *root = NumberMalloc("MakeRational");
02925     UWORD *x1 = NumberMalloc("MakeRational");
02926     UWORD *x2 = NumberMalloc("MakeRational");
02927     UWORD *x3 = NumberMalloc("MakeRational");
02928     UWORD *x4 = NumberMalloc("MakeRational");
02929     UWORD *y1 = NumberMalloc("MakeRational");
02930     UWORD *y2 = NumberMalloc("MakeRational");
02931     WORD nroot,nx1,nx2,nx3,nx4,ny1,ny2 = 0,retval = 0;
02932     WORD sign = 1;
02933 /*
02934     Step 1: Take the square root of m
02935 */
02936     COPYLONG(root,nroot,m,nm)
02937     TakeLongRoot(root,&nroot,2);
02938 /*
02939     Step 2: Set the start values
02940 */
02941     if ( na < 0 ) { na = -na; sign = -sign; }
02942     COPYLONG(x1,nx1,m,nm)
02943     COPYLONG(x2,nx2,a,na)
02944 /*
02945     x3[0] = 0, nx3 = 0;
02946     x4[0] = 1, nx4 = 1;
02947 */
02948 /*
02949     The start operation needs some special attention because of the zero.
02950 */
```

```
02951     if ( BigLong(x2,nx2,root,nroot) <= 0 ) {
02952         x4[0] = 1, nx4 = 1;
02953         goto gottheanswer;
02954     }
02955     DivLong(x1,nx1,x2,nx2,y1,&ny1,y2,&ny2);
02956     if ( ny2 == 0 ) { retval = 1; goto cleanup; }
02957     COPYLONG(x1,nx1,x2,nx2)
02958     COPYLONG(x2,nx2,y2,ny2)
02959     x3[0] = 1; nx3 = 1;
02960     COPYLONG(x4,nx4,y1,ny1)
02961     nx4 = -nx4;
02962 /*
02963     Now the loop.
02964 */
02965     while ( BigLong(x2,nx2,root,nroot) > 0 ) {
02966         DivLong(x1,nx1,x2,nx2,y1,&ny1,y2,&ny2);
02967         if ( ny2 == 0 ) { retval = 1; goto cleanup; }
02968         COPYLONG(x1,nx1,x2,nx2)
02969         COPYLONG(x2,nx2,y2,ny2)
02970         MulLong(y1,ny1,x4,nx4,y2,&ny2);
02971         ny2 = -ny2;
02972         AddLong(x3,nx3,y2,ny2,y1,&ny1);
02973         COPYLONG(x3,nx3,x4,nx4)
02974         COPYLONG(x4,nx4,y1,ny1)
02975     }
02976 /*
02977     Now we have the answer. It is x2/x4. It has to be packed into b.
02978 */
02979 gottheanswer:
02980     if ( nx4 < 0 ) { sign = -sign; nx4 = -nx4; }
02981     COPYLONG(b,*nb,x2,nx2)
02982     Pack(b,nb,x4,nx4);
02983     if ( sign < 0 ) *nb = -*nb;
02984 cleanup:
02985     NumberFree(y2,"MakeRational");
02986     NumberFree(y1,"MakeRational");
02987     NumberFree(x4,"MakeRational");
02988     NumberFree(x3,"MakeRational");
02989     NumberFree(x2,"MakeRational");
02990     NumberFree(x1,"MakeRational");
02991     NumberFree(root,"MakeRational");
02992     return(retval);
02993 }
02994
02995 /*
02996 #endif
02997         #] MakeLongRational:
02998         #[ ChineseRemainder:
02999 */
03011 #ifdef WITHCHINESEREMAINDER
03012
03013 int ChineseRemainder(PHEAD MODNUM *a1, MODNUM *a2, MODNUM *a)
03014 {
03015     UWORD *inv1 = NumberMalloc("ChineseRemainder");
03016     UWORD *inv2 = NumberMalloc("ChineseRemainder");
03017     UWORD *fac1 = NumberMalloc("ChineseRemainder");
03018     UWORD *fac2 = NumberMalloc("ChineseRemainder");
03019     UWORD two[1];
03020     WORD ninv1, ninv2, nfac1, nfac2;
03021     if ( a1->na < 0 ) {
03022         AddLong(a1->a,a1->na,a1->m,a1->nm,a1->a,&(a1->na));
03023     }
03024     if ( a2->na < 0 ) {
03025         AddLong(a2->a,a2->na,a2->m,a2->nm,a2->a,&(a2->na));
03026     }
03027     MulLong(a1->m,a1->nm,a2->m,a2->nm,a->m,&(a->nm));
03028
03029     GetLongModInverses(BHEAD a1->m,a1->nm,a2->m,a2->nm,inv1,&ninv1,inv2,&ninv2);
03030     MulLong(inv1,ninv1,a1->m,a1->nm,fac1,&nfac1);
03031     MulLong(inv2,ninv2,a2->m,a2->nm,fac2,&nfac2);
03032
03033     MulLong(fac1,nfac1,a2->a,a2->na,inv1,&ninv1);
03034     MulLong(fac2,nfac2,a1->a,a1->na,inv2,&ninv2);
03035     AddLong(inv1,ninv1,inv2,ninv2,a->a,&(a->na));
03036
03037     two[0] = 2;
03038     MulLong(a->a,a->na,two,1,fac1,&nfac1);
03039     if ( BigLong(fac1,nfac1,a->m,a->nm) > 0 ) {
03040         a->nm = -a->nm;
03041         AddLong(a->a,a->na,a->m,a->nm,a->a,&(a->na));
03042         a->nm = -a->nm;
03043     }
03044     NumberFree(fac2,"ChineseRemainder");
03045     NumberFree(fac1,"ChineseRemainder");
03046     NumberFree(inv2,"ChineseRemainder");
03047     NumberFree(inv1,"ChineseRemainder");
03048     return(0);
```

```
03049 }
03050
03051 #endif
03052
03053 /*
03054         #] ChineseRemainder:
03055     #] RekenLong :
03056     #[ RekenTerms :
03057         #[ CompCoef :       WORD CompCoef(term1,term2)
03058
03059     Compares the coefficients of term1 and term2 by subtracting them.
03060     This does more work than needed but this routine is only called
03061     when sorting functions and function arguments.
03062     (and comparing values
03063 */
03064 /* #define 64SAVE */
03065
03066 WORD CompCoef(WORD *term1, WORD *term2)
03067 {
03068     GETIDENTITY
03069     UWORD *c;
03070     WORD n1,n2,n3,*a;
03071     GETCOEF(term1,n1);
03072     GETCOEF(term2,n2);
03073     if ( term1[1] == 0 && n1 == 1 ) {
03074         if ( term2[1] == 0 && n2 == 1 ) return(0);
03075         if ( n2 < 0 ) return(1);
03076         return(-1);
03077     }
03078     else if ( term2[1] == 0 && n2 == 1 ) {
03079         if ( n1 < 0 ) return(-1);
03080         return(1);
03081     }
03082     if ( n1 > 0 ) {
03083         if ( n2 < 0 ) return(1);
03084     }
03085     else {
03086         if ( n2 > 0 ) return(-1);
03087         a = term1; term1 = term2; term2 = a;
03088         n3 = -n1; n1 = -n2; n2 = n3;
03089     }
03090     if ( term1[1] == 1 && term2[1] == 1 && n1 == 1 && n2 == 1 ) {
03091         if ( (UWORD)*term1 > (UWORD)*term2 ) return(1);
03092         else if ( (UWORD)*term1 < (UWORD)*term2 ) return(-1);
03093         else return(0);
03094     }
03095
03096 /*
03097     The next call should get dedicated code, as AddRat does more than
03098     strictly needed. Also more attention should be given to overflow.
03099 */
03100     c = NumberMalloc("CompCoef");
03101     if ( AddRat(BHEAD (UWORD *)term1,n1,(UWORD *)term2,-n2,c,&n3) ) {
03102         MLOCK(ErrorMessageLock);
03103         MesCall("CompCoef");
03104         MUNLOCK(ErrorMessageLock);
03105         NumberFree(c,"CompCoef");
03106         SETERROR(-1)
03107     }
03108     NumberFree(c,"CompCoef");
03109     return(n3);
03110 }
03111
03112 /*
03113         #] CompCoef :
03114         #[ Modulus :        WORD Modulus(term)
03115
03116     Routine takes the coefficient of term modulus b. The answer
03117     is in term again and the length of term is adjusted.
03118
03119 */
03120
03121 int Modulus(WORD *term)
03122 {
03123     WORD *t;
03124     WORD n1;
03125     t = term;
03126     GETCOEF(t,n1);
03127     if ( TakeModulus((UWORD *)t,&n1,AC.cmod,AC.ncmod,UNPACK) ) {
03128         MLOCK(ErrorMessageLock);
03129         MesCall("Modulus");
03130         MUNLOCK(ErrorMessageLock);
03131         SETERROR(-1)
03132     }
03133     if ( !n1 ) {
03134         *term = 0;
03135         return(0);
```

```
03136     }
03137     else if ( n1 > 0 ) {
03138         n1 *= 2;
03139         t += n1;        /* Note that n1 >= 0 */
03140         n1++;
03141     }
03142     else if ( n1 < 0 ) {
03143         n1 *= 2;
03144         t += -n1;
03145         n1--;
03146     }
03147     *t++ = n1;
03148     *term = WORDDIF(t,term);
03149     return(0);
03150 }
03151
03152 /*
03153         #] Modulus :
03154         #[ TakeModulus :    WORD TakeModulus(a,na,cmodvec,ncmod,par)
03155
03156         Routine gets the rational number in a with reduced length na.
03157         It is called when AC.ncmod != 0 and the number in AC.cmod is the
03158         number wrt which we need the modulus.
03159         The result is returned in a and na again.
03160
03161         If par == NOUNPACK we only do a single number, not a fraction.
03162         In addition we don't do fancy. We want a positive number and
03163         the input was supposed to be positive.
03164         We don't pack the result. The calling routine is responsible for that.
03165         This may not be a good idea. To be checked.
03166 */
03167
03168 int TakeModulus(UWORD *a, WORD *na, UWORD *cmodvec, WORD ncmod, WORD par)
03169 {
03170     GETIDENTITY
03171     UWORD *c, *d, *e, *f, *g, *h;
03172     UWORD *x4,*x2;
03173     UWORD *x3,*x1,*x5,*x6,*x7,*x8;
03174     WORD y3,y1,y5,y6;
03175     WORD n1, i, y2, y4;
03176     WORD nh, tdenom, tnumer, nmod;
03177     LONG x;
03178     if ( ncmod == 0 ) return(0);        /* No modulus operation */
03179     nmod = ABS(ncmod);
03180     n1 = *na;
03181     if ( ( par & UNPACK ) != 0 ) UnPack(a,n1,&tdenom,&tnumer);
03182     else { tnumer = n1; }
03183 /*
03184     We fish out the special case that the coefficient is short as well.
03185     There is no need to make lots of calls etc
03186 */
03187     if ( ( ( par & UNPACK ) == 0 ) && nmod == 1 && ( n1 == 1 || n1 == -1 ) ) {
03188         goto simplecase;
03189     }
03190     else if ( nmod == 1 && ( n1 == 1 || n1 == -1 ) ) {
03191         if ( a[1] != 1 ) {
03192             a[1] = a[1] % cmodvec[0];
03193             if ( a[1] == 0 ) {
03194                 MesPrint("Division by zero in short modulus arithmetic");
03195                 return(-1);
03196             }
03197             y1 = 0;
03198             if ( ( AC.modinverses != 0 ) && ( ( par & NOINVERSES ) == 0 ) ) {
03199                 y1 = AC.modinverses[a[1]];
03200             }
03201             else {
03202                 GetModInverses(a[1],cmodvec[0],&y1,&y2);
03203             }
03204             x = a[0];
03205             a[0] = (x*y1) % cmodvec[0];
03206             a[1] = 1;
03207         }
03208         else {
03209 simplecase:
03210             a[0] = a[0] % cmodvec[0];
03211         }
03212         if ( a[0] == 0 ) { *na = 0; return(0); }
03213         if ( ( AC.modmode & POSNEG ) != 0 ) {
03214             if ( a[0] > (UWORD)(cmodvec[0]/2) ) {
03215                 a[0] =  cmodvec[0] - a[0];
03216                 *na = -*na;
03217             }
03218         }
03219         else if ( *na < 0 ) {
03220             *na = 1; a[0] = cmodvec[0] - a[0];
03221         }
03222         return(0);
```

```
03223     }
03224     c = NumberMalloc("TakeModulus"); d = NumberMalloc("TakeModulus"); e = NumberMalloc("TakeModulus");
03225     f = NumberMalloc("TakeModulus"); g = NumberMalloc("TakeModulus"); h = NumberMalloc("TakeModulus");
03226     n1 = ABS(n1);
03227     if ( DivLong(a,tnumer,(UWORD *)cmodvec,nmod,
03228         c,&nh,a,&tnumer) ) goto ModErr;
03229     if ( tnumer == 0 ) { *na = 0; goto normalreturn; }
03230     if ( ( par & UNPACK ) == 0 ) {
03231         if ( ( AC.modmode & POSNEG ) != 0 ) {
03232             NormalModulus(a,&tnumer);
03233         }
03234         else if ( tnumer < 0 ) {
03235             SubPLon((UWORD *)cmodvec,nmod,a,-tnumer,a,&tnumer);
03236         }
03237         *na = tnumer;
03238         goto normalreturn;
03239     }
03240     if ( tdenom == 1 && a[n1] == 1 ) {
03241         if ( ( AC.modmode & POSNEG ) != 0 ) {
03242             NormalModulus(a,&tnumer);
03243         }
03244         else if ( tnumer < 0 ) {
03245             SubPLon((UWORD *)cmodvec,nmod,a,-tnumer,a,&tnumer);
03246         }
03247         *na = tnumer;
03248         i = ABS(tnumer);
03249         a += i;
03250         *a++ = 1;
03251         while ( --i > 0 ) *a++ = 0;
03252         goto normalreturn;
03253     }
03254     if ( DivLong(a+n1,tdenom,(UWORD *)cmodvec,nmod,c,&nh,a+n1,&tdenom) ) goto ModErr;
03255     if ( !tdenom ) {
03256         MLOCK(ErrorMessageLock);
03257         MesPrint("Division by zero in modulus arithmetic");
03258         if ( AP.DebugFlag ) {
03259             AO.OutSkip = 3;
03260             FiniLine();
03261             i = *na;
03262             if ( i < 0 ) i = -i;
03263             while ( --i >= 0 ) { TalToLine((UWORD)(*a++)); TokenToLine((UBYTE *)"  "); }
03264             i = *na;
03265             if ( i < 0 ) i = -i;
03266             while ( --i >= 0 ) { TalToLine((UWORD)(*a++)); TokenToLine((UBYTE *)"  "); }
03267             TalToLine((UWORD)(*na));
03268             AO.OutSkip = 0;
03269             FiniLine();
03270         }
03271         MUNLOCK(ErrorMessageLock);
03272         NumberFree(c,"TakeModulus"); NumberFree(d,"TakeModulus"); NumberFree(e,"TakeModulus");
03273         NumberFree(f,"TakeModulus"); NumberFree(g,"TakeModulus"); NumberFree(h,"TakeModulus");
03274         return(-1);
03275     }
03276     if ( ( AC.modinverses != 0 ) && ( ( par & NOINVERSES ) == 0 )
03277     && ( tdenom == 1 || tdenom == -1 ) ) {
03278         *d = AC.modinverses[a[n1]]; y1 = 1; y2 = tdenom;
03279         if ( MulLong(a,tnumer,d,y1,c,&y3) ) goto ModErr;
03280         if ( DivLong(c,y3,(UWORD *)cmodvec,nmod,d,&y5,a,&tdenom) ) goto ModErr;
03281         if ( y2 < 0 ) tdenom = -tdenom;
03282     }
03283     else {
03284       x2 = (UWORD *)cmodvec; x1 = c; i = nmod; while ( --i >= 0 ) *x1++ = *x2++;
03285       x1 = c; x2 = a+n1; x3 = d; x4 = e; x5 = f; x6 = g;
03286       y1 = nmod; y2 = tdenom; y4 = 0; y5 = 1; *x5 = 1;
03287       for(;;) {
03288         if ( DivLong(x1,y1,x2,y2,h,&nh,x3,&y3) ) goto ModErr;
03289         if ( MulLong(x5,y5,h,nh,x6,&y6) ) goto ModErr;
03290         if ( AddLong(x4,y4,x6,-y6,x6,&y6) ) goto ModErr;
03291         if ( !y3 ) {
03292             if ( y2 != 1 || *x2 != 1 ) {
03293                 MLOCK(ErrorMessageLock);
03294                 MesPrint("Inverse in modulus arithmetic doesn't exist");
03295                 MesPrint("Denominator and modulus are not relative prime");
03296                 MUNLOCK(ErrorMessageLock);
03297                 goto ModErr;
03298             }
03299             break;
03300         }
03301         x7 = x1; x1 = x2; y1 = y2; x2 = x3; y2 = y3; x3 = x7;
03302         x8 = x4; x4 = x5; y4 = y5; x5 = x6; y5 = y6; x6 = x8;
03303       }
03304       if ( y5 < 0 && AddLong((UWORD *)cmodvec,nmod,x5,y5,x5,&y5) ) goto ModErr;
03305       if ( MulLong(a,tnumer,x5,y5,c,&y3) ) goto ModErr;
03306       if ( DivLong(c,y3,(UWORD *)cmodvec,nmod,d,&y5,a,&tdenom) ) goto ModErr;
03307     }
03308     if ( !tdenom ) { *na = 0; goto normalreturn; }
03309     if ( ( ( AC.modmode & POSNEG ) != 0 ) && ( ( par & FROMFUNCTION ) == 0 ) ) {
```

```
03310         NormalModulus(a,&tdenom);
03311     }
03312     else if ( tdenom < 0 ) {
03313         SubPLon((UWORD *)cmodvec,nmod,a,-tdenom,a,&tdenom);
03314     }
03315     *na = tdenom;
03316     i = ABS(tdenom);
03317     a += i;
03318     *a++ = 1;
03319     while ( --i > 0 ) *a++ = 0;
03320 normalreturn:
03321     NumberFree(c,"TakeModulus"); NumberFree(d,"TakeModulus"); NumberFree(e,"TakeModulus");
03322     NumberFree(f,"TakeModulus"); NumberFree(g,"TakeModulus"); NumberFree(h,"TakeModulus");
03323     return(0);
03324 ModErr:
03325     MLOCK(ErrorMessageLock);
03326     MesCall("TakeModulus");
03327     MUNLOCK(ErrorMessageLock);
03328     NumberFree(c,"TakeModulus"); NumberFree(d,"TakeModulus"); NumberFree(e,"TakeModulus");
03329     NumberFree(f,"TakeModulus"); NumberFree(g,"TakeModulus"); NumberFree(h,"TakeModulus");
03330     SETERROR(-1)
03331 }
03332
03333 /*
03334         #] TakeModulus :
03335         #[ TakeNormalModulus :  WORD TakeNormalModulus(a,na,par)
03336
03337         added by Jan [01-09-2010]
03338 */
03339
03340 int TakeNormalModulus (UWORD *a, WORD *na, UWORD *c, WORD nc, WORD par)
03341 {
03342     WORD n;
03343     WORD nhalfc;
03344     UWORD *halfc;
03345
03346     GETIDENTITY;
03347
03348     /* determine c/2 by right shifting */
03349     halfc = NumberMalloc("TakeNormalModulus");
03350     nhalfc=nc;
03351     WCOPY(halfc,c,nc);
03352
03353     for (n=0; n<nhalfc; n++) {
03354         halfc[n] /= 2;
03355         if (n+1<nc) halfc[n] |= c[n+1] « (BITSINWORD-1);
03356     }
03357
03358     if (halfc[nhalfc-1]==0)
03359         nhalfc--;
03360
03361     /* takes care of the number never expanding, e.g., -1(mod 100) -> 99 -> -1 */
03362     if (BigLong(a,ABS(*na),halfc,nhalfc) > 0) {
03363
03364         TakeModulus(a,na,c,nc,par);
03365
03366         n = ABS(*na);
03367         if (BigLong(a,n,halfc,nhalfc) > 0) {
03368             SubPLon(c,nc,a,n,a,&n);
03369             *na = (*na > 0 ? -n : n);
03370         }
03371     }
03372
03373     NumberFree(halfc,"TakeNormalModulus");
03374     return(0);
03375 }
03376
03377 /*
03378         #] TakeNormalModulus :
03379         #[ MakeModTable :   WORD MakeModTable()
03380 */
03381
03382 int MakeModTable(void)
03383 {
03384     LONG size, i, j, n;
03385     n = ABS(AC.ncmod);
03386     if ( AC.modpowers ) {
03387         M_free(AC.modpowers,"AC.modpowers");
03388         AC.modpowers = NULL;
03389     }
03390     if ( n > 2 ) {
03391 /* INTERNAL_ERROR_EXCL_START */
03392         MLOCK(ErrorMessageLock);
03393         MesPrint("!>No memory for modulus generator power table");
03394         MUNLOCK(ErrorMessageLock);
03395         Terminate(-1);
03396 /* INTERNAL_ERROR_EXCL_STOP */
```

```
03397     }
03398     if ( n == 0 ) return(0);
03399     size = (LONG)(*AC.cmod);
03400     if ( n == 2 ) size += (((LONG)AC.cmod[1])«BITSINWORD);
03401     AC.modpowers = (UWORD *)Malloc1(size*n*sizeof(UWORD),"table for powers of modulus");
03402     if ( n == 1 ) {
03403         j = 1;
03404         for ( i = 0; i < size; i++ ) AC.modpowers[i] = 0;
03405         for ( i = 0; i < size; i++ ) {
03406             AC.modpowers[j] = (WORD)i;
03407             j *= *AC.powmod;
03408             j %= *AC.cmod;
03409         }
03410         for ( i = 2; i < size; i++ ) {
03411             if ( AC.modpowers[i] == 0 ) {
03412                 MLOCK(ErrorMessageLock);
03413                 MesPrint("&improper generator for this modulus");
03414                 MUNLOCK(ErrorMessageLock);
03415                 M_free(AC.modpowers,"AC.modpowers");
03416                 return(-1);
03417             }
03418         }
03419         AC.modpowers[1] = 0;
03420     }
03421     else {
03422         GETIDENTITY
03423         WORD nScrat, n2;
03424         UWORD *MMscrat7 = NumberMalloc("MakeModTable"), *MMscratC = NumberMalloc("MakeModTable");
03425         *MMscratC = 1;
03426         nScrat = 1;
03427         j = size * 2;
03428         for ( i = 0; i < j; i+=2 ) { AC.modpowers[i] = 0; AC.modpowers[i+1] = 0; }
03429         for ( i = 0; i < size; i++ ) {
03430             j = *MMscratC + (((LONG)MMscratC[1])«BITSINWORD);
03431             j *= 2;
03432             AC.modpowers[j] = (WORD)(i & WORDMASK);
03433             AC.modpowers[j+1] = (WORD)(i » BITSINWORD);
03434             MulLong((UWORD *)MMscratC,nScrat,(UWORD *)AC.powmod,
03435             AC.npowmod,(UWORD *)MMscrat7,&n2);
03436             TakeModulus(MMscrat7,&n2,AC.cmod,AC.ncmod,NOUNPACK);
03437             *MMscratC = *MMscrat7; MMscratC[1] = MMscrat7[1]; nScrat = n2;
03438         }
03439         NumberFree(MMscrat7,"MakeModTable"); NumberFree(MMscratC,"MakeModTable");
03440         j = size * 2;
03441         for ( i = 4; i < j; i+=2 ) {
03442             if ( AC.modpowers[i] == 0 && AC.modpowers[i+1] == 0 ) {
03443                 MLOCK(ErrorMessageLock);
03444                 MesPrint("&improper generator for this modulus");
03445                 MUNLOCK(ErrorMessageLock);
03446                 M_free(AC.modpowers,"AC.modpowers");
03447                 return(-1);
03448             }
03449         }
03450         AC.modpowers[2] = AC.modpowers[3] = 0;
03451     }
03452     return(0);
03453 }
03454
03455 /*
03456         #] MakeModTable :
03457     #] RekenTerms :
03458     #[ Functions :
03459         #[ Factorial :      WORD Factorial(n,a,na)
03460
03461     Starts with only the value of fac_(0).
03462     Builds up what is needed and remembers it for the next time.
03463
03464     We have:
03465         AT.nfac: the number of the highest stored factorial
03466         AT.pfac: the array of locations in the array of stored factorials
03467         AT.factorials: the array with stored factorials
03468 */
03469
03470 int Factorial(PHEAD WORD n, UWORD *a, WORD *na)
03471 {
03472     GETBIDENTITY
03473     UWORD *b, *c;
03474     WORD nc;
03475     int i, j;
03476     LONG ii;
03477     if ( n > AT.nfac ) {
03478         if ( AT.factorials == 0 ) {
03479             AT.nfac = 0; AT.mfac = 50; AT.sfact = 400;
03480             AT.pfac = (LONG *)Malloc1((AT.mfac+2)*sizeof(LONG),"factorials");
03481             AT.factorials = (UWORD *)Malloc1(AT.sfact*sizeof(UWORD),"factorials");
03482             AT.factorials[0] = 1; AT.pfac[0] = 0; AT.pfac[1] = 1;
03483         }
```

```
03484         b = a;
03485         c = AT.factorials+AT.pfac[AT.nfac];
03486         nc = i = AT.pfac[AT.nfac+1] - AT.pfac[AT.nfac];
03487         while ( --i >= 0 ) *b++ = *c++;
03488         for ( j = AT.nfac+1; j <= n; j++ ) {
03489             Product(a,&nc,j);
03490             if ( nc > AM.MaxTal ) {
03491                 MLOCK(ErrorMessageLock);
03492                 MesPrint("Overflow in factorial. MaxTal = %d",AM.MaxTal);
03493                 MesPrint("Increase MaxTerm in %s",setupfilename);
03494                 MUNLOCK(ErrorMessageLock);
03495                 return(-1);
03496             }
03497             if ( j > AT.mfac ) {  /* double the pfac buffer */
03498                 LONG *p;
03499                 p = (LONG *)Malloc1((AT.mfac*2+2)*sizeof(LONG),"factorials");
03500                 i = AT.mfac;
03501                 for ( i = AT.mfac+1; i >= 0; i-- ) p[i] = AT.pfac[i];
03502                 M_free(AT.pfac,"factorial offsets"); AT.pfac = p; AT.mfac *= 2;
03503             }
03504             if ( AT.pfac[j] + nc >= AT.sfact ) { /* double the factorials buffer */
03505                 UWORD *f;
03506                 f = (UWORD *)Malloc1(AT.sfact*2*sizeof(UWORD),"factorials");
03507                 ii = AT.sfact;
03508                 c = AT.factorials; b = f;
03509                 while ( --ii >= 0 ) *b++ = *c++;
03510                 M_free(AT.factorials,"factorials");
03511                 AT.factorials = f;
03512                 AT.sfact *= 2;
03513             }
03514             b = a; c = AT.factorials + AT.pfac[j]; i = nc;
03515             while ( --i >= 0 ) *c++ = *b++;
03516             AT.pfac[j+1] = AT.pfac[j] + nc;
03517         }
03518         *na = nc;
03519         AT.nfac = n;
03520     }
03521     else if ( n == 0 ) {
03522         *a = 1; *na = 1;
03523     }
03524     else {
03525         *na = i = AT.pfac[n+1] - AT.pfac[n];
03526         b = AT.factorials + AT.pfac[n];
03527         while ( --i >= 0 ) *a++ = *b++;
03528     }
03529     return(0);
03530 }
03531
03532 /*
03533         #] Factorial :
03534         #[ Bernoulli :      WORD Bernoulli(n,a,na)
03535
03536     Starts with only the value of bernoulli_(0).
03537     Builds up what is needed and remembers it for the next time.
03538     b_0 = 1
03539     (n+1)*b_n = -b_{n-1}-sum_(i,1,n-1,b_i*b_{n-i})
03540     The n-1 plays only a role for b_2.
03541     We have hard coded b_0,b_1,b_2 and b_odd. After that:
03542     (2n+1)*b_2n = -sum_(i,1,n-1,b_2i*b_{2n-2i})
03543
03544     We have:
03545         AT.nBer: the number of the highest stored Bernoulli number
03546         AT.pBer: the array of locations in the array of stored Bernoulli numbers
03547         AT.bernoullis: the array with stored Bernoulli numbers
03548 */
03549
03550 int Bernoulli(WORD n, UWORD *a, WORD *na)
03551 {
03552     GETIDENTITY
03553     UWORD *b, *c, *scrib, *ntop, *ntop1;
03554     WORD i, i1, i2, nhalf, nqua, nscrib, nntop, nntop1, *oldworkpointer;
03555     UWORD twee = 2, twonplus1;
03556     int j;
03557     LONG ii;
03558     if ( n <= 1 ) {
03559         if ( n == 0 ) { a[0] = a[1] = 1; *na = 3; }
03560         else if ( n == 1 ) { a[0] = 1; a[1] = 2; *na = 3; }
03561         return(0);
03562     }
03563     if ( ( n & 1 ) != 0 ) { a[0] = a[1] = 0; *na = 0; return(0); }
03564     nhalf = n/2;
03565     if ( nhalf > AT.nBer ) {
03566         oldworkpointer = AT.WorkPointer;
03567         if ( AT.bernoullis == 0 ) {
03568             AT.nBer = 1; AT.mBer = 50; AT.sBer = 400;
03569             AT.pBer = (LONG *)Malloc1((AT.mBer+2)*sizeof(LONG),"bernoullis");
03570             AT.bernoullis = (UWORD *)Malloc1(AT.sBer*sizeof(UWORD),"bernoullis");
```

```
03571             AT.pBer[1] = 0; AT.pBer[2] = 3;
03572             AT.bernoullis[0] = 3; AT.bernoullis[1] = 1; AT.bernoullis[2] = 12;
03573             if ( nhalf == 1 ) {
03574                 a[0] = 1; a[1] = 12; *na = 3; return(0);
03575             }
03576         }
03577         while ( nhalf > AT.mBer ) {
03578             LONG *p;
03579             p = (LONG *)Malloc1((AT.mBer*2+1)*sizeof(LONG),"bernoullis");
03580             i = AT.mBer;
03581             for ( i = AT.mBer; i >= 0; i-- ) p[i] = AT.pBer[i];
03582             M_free(AT.pBer,"factorial pointers"); AT.pBer = p; AT.mBer *= 2;
03583         }
03584         for ( n = AT.nBer+1; n <= nhalf; n++ ) {
03585             scrib = (UWORD *)(AT.WorkPointer);
03586             nqua = n/2;
03587             if ( ( n & 1 ) == 1 ) {
03588                 nscrib = 0; ntop = scrib;
03589             }
03590             else {
03591                 b = AT.bernoullis + AT.pBer[nqua];
03592                 nscrib = *b++;
03593                 i = (WORD)(REDLENG(nscrib));
03594                 MulRat(BHEAD b,i,b,i,scrib,&nscrib);
03595                 ntop = scrib + 2*nscrib;
03596                 nqua--;
03597             }
03598             for ( j = 1; j <= nqua; j++ ) {
03599                 b = AT.bernoullis + AT.pBer[j];
03600                 c = AT.bernoullis + AT.pBer[n-j];
03601                 i1 = (WORD)(*b); i2 = (WORD)(*c);
03602                 i1 = REDLENG(i1);
03603                 i2 = REDLENG(i2);
03604                 MulRat(BHEAD b+1,i1,c+1,i2,ntop,&nntop);
03605                 Mully(BHEAD ntop,&nntop,&twee,1);
03606                 if ( nscrib ) {
03607                     i = (WORD)nntop; if ( i < 0 ) i = -i;
03608                     ntop1 = ntop + 2*i;
03609                     AddRat(BHEAD ntop,nntop,scrib,nscrib,ntop1,&nntop1);
03610                 }
03611                 else {
03612                     ntop1 = ntop; nntop1 = nntop;
03613                 }
03614                 nscrib = i1 = (WORD)nntop1;
03615                 if ( i1 < 0 ) i1 = - i1;
03616                 i1 = 2*i1;
03617                 for ( i = 0; i < i1; i++ ) scrib[i] = ntop1[i];
03618                 ntop = scrib + i1;
03619             }
03620             twonplus1 = 2*n+1;
03621             Divvy(BHEAD scrib,&nscrib,&twonplus1,-1);
03622             i1 = INCLENG(nscrib);
03623             i2 = i1; if ( i2 < 0 ) i2 = -i2;
03624             i = (WORD)(AT.bernoullis[AT.pBer[n-1]]);
03625             if ( i < 0 ) i = -i;
03626             AT.pBer[n] = AT.pBer[n-1]+i;
03627             if ( AT.pBer[n] + i2 >= AT.sBer ) {
03628                 UWORD *f;
03629                 f = (UWORD *)Malloc1(AT.sBer*2*sizeof(UWORD),"bernoullis");
03630                 ii = AT.sBer;
03631                 c = AT.bernoullis; b = f;
03632                 while ( --ii >= 0 ) *b++ = *c++;
03633                 M_free(AT.bernoullis,"bernoullis");
03634                 AT.bernoullis = f;
03635                 AT.sBer *= 2;
03636             }
03637             c = AT.bernoullis + AT.pBer[n]; b = scrib;
03638             *c++ = i1;
03639             for ( i = 1; i < i2; i++ ) *c++ = *b++;
03640         }
03641         AT.nBer = nhalf;
03642         AT.WorkPointer = oldworkpointer;
03643     }
03644     b = AT.bernoullis + AT.pBer[nhalf];
03645     *na = i = (WORD)(*b++);
03646     if ( i < 0 ) i = -i;
03647     i--;
03648     while ( --i >= 0 ) *a++ = *b++;
03649     return(0);
03650 }
03651
03652 /*
03653         #] Bernoulli :
03654         #[ NextPrime :
03655 */
03667 #if ( BITSINWORD == 32 )
03668
```

```
03669 void StartPrimeList(PHEAD0)
03670 {
03671     int i, j;
03672     AR.PrimeList[AR.numinprimelist++] = 3;
03673     for ( i = 5; i < 46340; i += 2 ) {
03674         for ( j = 0; j < AR.numinprimelist && AR.PrimeList[j]*AR.PrimeList[j] <= i; j++ ) {
03675             if ( i % AR.PrimeList[j] == 0 ) goto nexti;
03676         }
03677         AR.PrimeList[AR.numinprimelist++] = i;
03678 nexti:;
03679     }
03680     AR.notfirstprime = 1;
03681 }
03682
03683 #endif
03684
03685 WORD NextPrime(PHEAD WORD num)
03686 {
03687     int i, j;
03688     WORD *newpl;
03689     LONG newsize, x;
03690 #if ( BITSINWORD == 32 )
03691     if ( AR.notfirstprime == 0 ) StartPrimeList(BHEAD0);
03692 #endif
03693     if ( num > AT.inprimelist ) {
03694         while ( AT.inprimelist < num ) {
03695             if ( num >= AT.sizeprimelist ) {
03696                 if ( AT.sizeprimelist == 0 ) newsize = 32;
03697                 else newsize = 2*AT.sizeprimelist;
03698                 while ( num >= newsize ) newsize = newsize*2;
03699                 newpl = (WORD *)Malloc1(newsize*sizeof(WORD),"NextPrime");
03700                 for ( i = 0; i < AT.sizeprimelist; i++ ) {
03701                     newpl[i] = AT.primelist[i];
03702                 }
03703                 if ( AT.sizeprimelist > 0 ) {
03704                     M_free(AT.primelist,"NextPrime");
03705                 }
03706                 AT.sizeprimelist = newsize;
03707                 AT.primelist = newpl;
03708             }
03709             if ( AT.inprimelist < 0 ) { i = MAXPOSITIVE; }
03710             else { i = AT.primelist[AT.inprimelist]; }
03711             while ( i > MAXPOWER ) {
03712                 i -= 2; x = i;
03713 #if ( BITSINWORD == 32 )
03714                 for ( j = 0; j < AR.numinprimelist && AR.PrimeList[j]*(LONG)(AR.PrimeList[j]) <= x;
      j++ ) {
03715                     if ( x % AR.PrimeList[j] == 0 ) goto nexti;
03716                 }
03717 #else
03718                 for ( j = 3; j*((LONG)j) <= x; j += 2 ) {
03719                     if ( x % j == 0 ) goto nexti;
03720                 }
03721 #endif
03722                 AT.inprimelist++;
03723                 AT.primelist[AT.inprimelist] = i;
03724                 break;
03725 nexti:;
03726             }
03727             if ( i < MAXPOWER ) {
03728 /* INTERNAL_ERROR_EXCL_START */
03729                 MLOCK(ErrorMessageLock);
03730                 MesPrint("!>There are not enough short prime numbers for this calculation");
03731                 MesPrint("Try to use a computer with a %d-bits architecture",
03732                     (int)(BITSINWORD*4));
03733                 MUNLOCK(ErrorMessageLock);
03734                 Terminate(-1);
03735 /* INTERNAL_ERROR_EXCL_STOP */
03736             }
03737         }
03738     }
03739     return(AT.primelist[num]);
03740 }
03741
03742 /*
03743         #] NextPrime :
03744         #[ Moebius :
03745
03746         Returns the value of the Moebius function if n fits inside
03747         a WORD.
03748         The method we use is a bit like the sieve of Erathostenes.
03749 */
03750
03751 WORD Moebius(PHEAD WORD nn)
03752 {
03753     WORD i,n = nn, x;
03754     LONG newsize;
```

```
03755     SBYTE *newtable, mu;
03756 #if ( BITSINWORD == 32 )
03757     if ( AR.notfirstprime == 0 ) StartPrimeList(BHEAD0);
03758 #endif
03759 /*
03760     First we make sure that:
03761         a: the table is big enough.
03762         b: the number is not already in the table.
03763 */
03764     if ( nn >= AR.moebiustablesize ) {
03765         if ( AR.moebiustablesize <= 0 ) { newsize = (LONG)nn + 20; }
03766         else { newsize = (LONG)nn*2; }
03767         if ( newsize > MAXPOSITIVE ) newsize = MAXPOSITIVE;
03768         newtable = (SBYTE *)Malloc1(newsize*sizeof(SBYTE),"Moebius");
03769         for ( i = 0; i < AR.moebiustablesize; i++ ) newtable[i] = AR.moebiustable[i];
03770         for ( ; i < newsize; i++ ) newtable[i] = 2;
03771         if ( AR.moebiustablesize > 0 ) M_free(AR.moebiustable,"Moebius");
03772         AR.moebiustable = newtable;
03773         AR.moebiustablesize = newsize;
03774     }
03775     /* NOTE: nn == MAXPOSITIVE never fits in moebiustable. */
03776     if ( nn != MAXPOSITIVE && AR.moebiustable[nn] != 2 ) return((WORD)AR.moebiustable[nn]);
03777     mu = 1;
03778     if ( n == 1 ) goto putvalue;
03779     if ( n % 2 == 0 ) {
03780         n /= 2;
03781         if ( n % 2 == 0 ) { mu = 0; goto putvalue; }
03782         if ( AR.moebiustable[n] != 2 ) { mu = -AR.moebiustable[n]; goto putvalue; }
03783         mu = -mu;
03784         if ( n == 1 ) goto putvalue;
03785     }
03786 #if ( BITSINWORD == 32 )
03787     for ( i = 0; i < AR.numinprimelist; i++ ) {
03788         x = AR.PrimeList[i];
03789 #else
03790     for ( x = 3; x < MAXPOSITIVE; x += 2 ) {
03791 #endif
03792         if ( n % x == 0 ) {
03793             n /= x;
03794             if ( n % x == 0 ) { mu = 0; goto putvalue; }
03795             if ( AR.moebiustable[n] != 2 ) { mu = -AR.moebiustable[n]; goto putvalue; }
03796             mu = -mu;
03797             if ( n == 1 ) goto putvalue;
03798         }
03799         if ( n < x*x ) break; /* notice that x*x always fits inside a WORD */
03800     }
03801     mu = -mu;
03802 putvalue:
03803     if ( nn != MAXPOSITIVE ) AR.moebiustable[nn] = mu;
03804     return((WORD)mu);
03805 }
03806
03807 /*
03808         #] Moebius :
03809         #[ wranf :
03810
03811         A random number generator that generates random WORDs with a very
03812         long sequence. It is based on the Knuth generator.
03813
03814         We take some care that each thread can run its own, but each
03815         uses its own startup. Hence the seed includes the identity of
03816         the thread.
03817
03818         For NPAIR1, NPAIR2 we can use any pair from the table on page 28.
03819         Candidates are 24,55 (the example on the pages 171,172)
03820         or (33,97) or (38,89)
03821         These values are defined in fsizes.h and used in startup.c and threads.c
03822 */
03823
03824 #define WARMUP 6
03825
03826 static void wranfnew(PHEAD0)
03827 {
03828     int i;
03829     LONG j;
03830     for ( i = 0; i < AR.wranfnpair1; i++ ) {
03831         j = AR.wranfia[i] - AR.wranfia[i+(AR.wranfnpair2-AR.wranfnpair1)];
03832         if ( j < 0 ) j += (LONG)1 « (2*BITSINWORD-2);
03833         AR.wranfia[i] = j;
03834     }
03835     for ( i = AR.wranfnpair1; i < AR.wranfnpair2; i++ ) {
03836         j = AR.wranfia[i] - AR.wranfia[i-AR.wranfnpair1];
03837         if ( j < 0 ) j += (LONG)1 « (2*BITSINWORD-2);
03838         AR.wranfia[i] = j;
03839     }
03840 }
03841
```

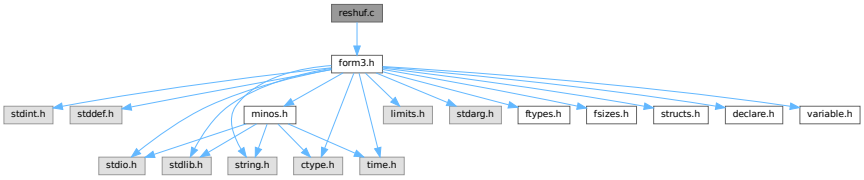
```
03842 void iniwranf(PHEAD0)
03843 {
03844     int imax = AR.wranfnpair2-1;
03845     ULONG i, ii, seed = AR.wranfseed;
03846     LONG j, k;
03847     ULONG offset = 12345;
03848 #ifdef PARALLELCODE
03849     int id;
03850 #if defined(WITHPTHREADS)
03851     id = AT.identity;
03852 #elif defined(WITHMPI)
03853     id = PF.me;
03854 #endif
03855     seed += id;
03856     i = id + 1;
03857     if ( i > 1 ) {
03858         ULONG pow, accu;
03859         pow = offset; accu = 1;
03860         while ( i ) {
03861             if ( ( i & 1 ) != 0 ) accu *= pow;
03862             i /= 2; pow = pow*pow;
03863         }
03864         offset = accu;
03865     }
03866 #endif
03867     if ( seed < ((LONG)1«(BITSINWORD-1)) ) {
03868         j = ( (seed+31459L) « (BITSINWORD-2))+offset;
03869     }
03870     else if ( seed < ((LONG)1«(BITSINWORD+10-1)) ) {
03871         j = ( (seed+31459L) « (BITSINWORD-10-2))+offset;
03872     }
03873     else {
03874         j = ( (seed+31459L) « 1)+offset;
03875     }
03876     if ( ( seed & 1 ) == 1 ) seed++;
03877     j += seed;
03878     AR.wranfia[imax] = j;
03879     k = 1;
03880     for ( i = 0; i <= (ULONG)(imax); i++ ) {
03881         ii = (AR.wranfnpair1*i)%AR.wranfnpair2;
03882         AR.wranfia[ii] = k;
03883         k = ULongToLong((ULONG)j - (ULONG)k);
03884         if ( k < 0 ) k += (LONG)1 « (2*BITSINWORD-2);
03885         j = AR.wranfia[ii];
03886     }
03887     for ( i = 0; i < WARMUP; i++ ) wranfnew(BHEAD0);
03888     AR.wranfcall = 0;
03889 }
03890
03891 UWORD wranf(PHEAD0)
03892 {
03893     UWORD wval;
03894     if ( AR.wranfia == 0 ) {
03895         AR.wranfia = (ULONG *)Malloc1(AR.wranfnpair2*sizeof(ULONG),"wranf");
03896         iniwranf(BHEAD0);
03897     }
03898     if ( AR.wranfcall >= AR.wranfnpair2) {
03899         wranfnew(BHEAD0);
03900         AR.wranfcall = 0;
03901     }
03902     wval = (UWORD)(AR.wranfia[AR.wranfcall++]»(BITSINWORD-1));
03903     return(wval);
03904 }
03905
03906 /*
03907     Returns a random UWORD in the range (0,...,imax-1)
03908 */
03909
03910 UWORD iranf(PHEAD UWORD imax)
03911 {
03912     UWORD i;
03913     ULONG x, xmax;
03914     if (imax < 2) return 0;
03915     x = (LONG)1 « BITSINWORD;
03916     xmax = x - x%imax;
03917     while ( ( i = wranf(BHEAD0) ) >= xmax ) {}
03918     return(i%imax);
03919 }
03920
03921 /*
03922         #] wranf :
03923         #[ PreRandom :
03924
03925         The random number generator of the preprocessor.
03926         This one is completely different from the execution time generator
03927         random_(number). In the preprocessor we generate a floating point
03928         number in a string according to a distribution.
```

```
03929         Currently allowed are:
03930             RANDOM_(log,min,max)
03931             RANDOM_(lin,min,max)
03932         The return value is a string with the floating point number.
03933 */
03934
03935 UBYTE *PreRandom(UBYTE *s)
03936 {
03937     GETIDENTITY
03938     UBYTE *mode,*mins = 0,*maxs = 0, *outval;
03939     float num;
03940     double minval, maxval, value = 0;
03941     int linlog = -1;
03942     mode = s;
03943     while ( FG.cTable[*s] <= 1 ) s++;
03944     if ( *s == ',' ) { *s = 0; s++; }
03945     mins = s;
03946     while ( *s && *s != ',' ) s++;
03947     if ( *s == ',' ) { *s = 0; s++; }
03948     maxs = s;
03949     while ( *s && *s != ',' ) s++;
03950     if ( *s || *maxs == 0 || *mins == 0 ) {
03951         MesPrint("@Illegal arguments in macro RANDOM_");
03952         Terminate(-1);
03953     }
03954     if ( StrICmp(mode,(UBYTE *)"lin") == 0 ) {
03955         linlog = 0;
03956     }
03957     else if ( StrICmp(mode,(UBYTE *)"log") == 0 ) {
03958         linlog = 1;
03959     }
03960     else {
03961         MesPrint("@Illegal mode argument in macro RANDOM_");
03962         Terminate(-1);
03963     }
03964
03965     sscanf((char *)mins,"%f",&num); minval = num;
03966     sscanf((char *)maxs,"%f",&num); maxval = num;
03967
03968     /*
03969      * Note on ParFORM: we should use the same random number on all the
03970      * processes in the complication phase. The random number is generated
03971      * on the master and broadcast to the other processes.
03972      */
03973     {
03974         UWORD x;
03975         double xx;
03976 #ifdef WITHMPI
03977         x = 0;
03978         if ( PF.me == MASTER ) {
03979             x = wranf(BHEAD0);
03980         }
03981         x = (UWORD)PF_BroadcastNumber((LONG)x);
03982 #else
03983         x = wranf(BHEAD0);
03984 #endif
03985         xx = x/pow(2.0,(double)(BITSINWORD-1));
03986         if ( linlog == 0 ) {
03987             value = minval + (maxval-minval)*xx;
03988         }
03989         else if ( linlog == 1 ) {
03990             value = minval * pow(maxval/minval,xx);
03991         }
03992     }
03993
03994     outval = (UBYTE *)Malloc1(64,"PreRandom");
03995     if ( ABS(value) < 0.00001 || ABS(value) > 1000000. ) {
03996         snprintf((char *)outval,64,"%e",value);
03997     }
03998     else if ( ABS(value) < 0.0001 ) { snprintf((char *)outval,64,"%10f",value); }
03999     else if ( ABS(value) < 0.001 ) { snprintf((char *)outval,64,"%9f",value); }
04000     else if ( ABS(value) < 0.01 ) { snprintf((char *)outval,64,"%8f",value); }
04001     else if ( ABS(value) < 0.1 ) { snprintf((char *)outval,64,"%7f",value); }
04002     else if ( ABS(value) < 1. ) { snprintf((char *)outval,64,"%6f",value); }
04003     else if ( ABS(value) < 10. ) { snprintf((char *)outval,64,"%5f",value); }
04004     else if ( ABS(value) < 100. ) { snprintf((char *)outval,64,"%4f",value); }
04005     else if ( ABS(value) < 1000. ) { snprintf((char *)outval,64,"%3f",value); }
04006     else if ( ABS(value) < 10000. ) { snprintf((char *)outval,64,"%2f",value); }
04007     else { snprintf((char *)outval,64,"%1f",value); }
04008     return(outval);
04009 }
04010
04011 /*
04012         #] PreRandom :
04013     #] Functions :
04014 */
```

4.123 reshuf.c File Reference

```
#include "form3.h"
```

Include dependency graph for reshuf.c:



Functions

- WORD ReNumber (PHEAD WORD *term)
- void FunLevel (PHEAD WORD *term)
- WORD DetCurDum (PHEAD WORD *t)
- int FullRenumber (PHEAD WORD *term, WORD par)
- void MoveDummies (PHEAD WORD *term, WORD shift)
- void AdjustRenumScratch (PHEAD0)
- WORD CountDo (WORD *term, WORD *instruct)
- WORD CountFun (WORD *term, WORD *countfun)
- WORD DimensionSubterm (WORD *subterm)
- WORD DimensionTerm (WORD *term)
- WORD DimensionExpression (PHEAD WORD *expr)
- int MultDo (PHEAD WORD *term, WORD *pattern)
- int TryDo (PHEAD WORD *term, WORD *pattern, WORD level)
- int DoDistrib (PHEAD WORD *term, WORD level)
- int EqualArg (WORD *parms, WORD num1, WORD num2)
- int DoDelta3 (PHEAD WORD *term, WORD level)
- int TestPartitions (WORD *tfun, PARTI *parti)
- int DoPartitions (PHEAD WORD *term, WORD level)
- int DoPermutations (PHEAD WORD *term, WORD level)
- int DoShuffle (WORD *term, WORD level, WORD fun, WORD option)
- int Shuffle (WORD *from1, WORD *from2, WORD *to)
- int FinishShuffle (WORD *fini)
- int DoStuffle (WORD *term, WORD level, WORD fun, WORD option)
- int Stuffle (WORD *from1, WORD *from2, WORD *to)
- int FinishStuffle (WORD *fini)
- WORD * StuffRootAdd (WORD *t1, WORD *t2, WORD *to)

4.123.1 Detailed Description

Mixed routines: Routines for relabelling dummy indices. The multiply command The distrib_ function The tryreplace statement

Definition in file reshuf.c.

4.123.2 Macro Definition Documentation

4.123.2.1 NEWCODE

```
#define NEWCODE
```

Definition at line 35 of file reshuf.c.

4.123.3 Function Documentation

4.123.3.1 ReNumber()

```
WORD ReNumber (
            PHEAD WORD * term )
```

Definition at line 80 of file reshuf.c.

4.123.3.2 FunLevel()

```
void FunLevel (
            PHEAD WORD * term )
```

Definition at line 130 of file reshuf.c.

4.123.3.3 DetCurDum()

```
WORD DetCurDum (
            PHEAD WORD * t )
```

Definition at line 254 of file reshuf.c.

4.123.3.4 FullRenumber()

```
int FullRenumber (
            PHEAD WORD * term,
            WORD par )
```

Definition at line 326 of file reshuf.c.

4.123.3.5 MoveDummies()

```
void MoveDummies (
            PHEAD WORD * term,
            WORD shift )
```

Definition at line 438 of file reshuf.c.

4.123.3.6 AdjustRenumScratch()

```
void AdjustRenumScratch (
             PHEAD0  )
```

Definition at line 508 of file reshuf.c.

4.123.3.7 CountDo()

```
WORD CountDo (
             WORD * term,
             WORD * instruct )
```

Definition at line 554 of file reshuf.c.

4.123.3.8 CountFun()

```
WORD CountFun (
             WORD * term,
             WORD * countfun )
```

Definition at line 708 of file reshuf.c.

4.123.3.9 DimensionSubterm()

```
WORD DimensionSubterm (
             WORD * subterm )
```

Definition at line 869 of file reshuf.c.

4.123.3.10 DimensionTerm()

```
WORD DimensionTerm (
             WORD * term )
```

Definition at line 970 of file reshuf.c.

4.123.3.11 DimensionExpression()

```
WORD DimensionExpression (
             PHEAD WORD * expr )
```

Definition at line 1002 of file reshuf.c.

4.123.3.12 MultDo()

```
int MultDo (
            PHEAD WORD * term,
            WORD * pattern )
```

Definition at line 1046 of file reshuf.c.

4.123.3.13 TryDo()

```
int TryDo (
            PHEAD WORD * term,
            WORD * pattern,
            WORD level )
```

Definition at line 1074 of file reshuf.c.

4.123.3.14 DoDistrib()

```
int DoDistrib (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 1121 of file reshuf.c.

4.123.3.15 EqualArg()

```
int EqualArg (
            WORD * parms,
            WORD num1,
            WORD num2 )
```

Definition at line 1381 of file reshuf.c.

4.123.3.16 DoDelta3()

```
int DoDelta3 (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 1407 of file reshuf.c.

4.123.3.17 TestPartitions()

```
int TestPartitions (
            WORD * tfun,
            PARTI * parti )
```

Definition at line 1611 of file reshuf.c.

4.123.3.18 DoPartitions()

```
int DoPartitions (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 1726 of file reshuf.c.

4.123.3.19 DoPermutations()

```
int DoPermutations (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 2011 of file reshuf.c.

4.123.3.20 DoShuffle()

```
int DoShuffle (
            WORD * term,
            WORD level,
            WORD fun,
            WORD option )
```

Definition at line 2099 of file reshuf.c.

4.123.3.21 Shuffle()

```
int Shuffle (
            WORD * from1,
            WORD * from2,
            WORD * to )
```

Definition at line 2233 of file reshuf.c.

4.123.3.22 FinishShuffle()

```
int FinishShuffle (
            WORD * fini )
```

Definition at line 2425 of file reshuf.c.

4.123.3.23 DoStuffle()

```
int DoStuffle (
            WORD * term,
            WORD level,
            WORD fun,
            WORD option )
```

Definition at line 2491 of file reshuf.c.

4.123.3.24 Stuffle()

```
int Stuffle (
              WORD * from1,
              WORD * from2,
              WORD * to )
```

Definition at line 2706 of file reshuf.c.

4.123.3.25 FinishStuffle()

```
int FinishStuffle (
              WORD * fini )
```

Definition at line 2833 of file reshuf.c.

4.123.3.26 StuffRootAdd()

```
WORD * StuffRootAdd (
              WORD * t1,
              WORD * t2,
              WORD * to )
```

Definition at line 2880 of file reshuf.c.

4.124 reshuf.c

Go to the documentation of this file.

```
00001
00009 /* #[ License : */
00010 /*
00011  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00012  *   When using this file you are requested to refer to the publication
00013  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00014  *   This is considered a matter of courtesy as the development was paid
00015  *   for by FOM the Dutch physics granting agency and we would like to
00016  *   be able to track its scientific use to convince FOM of its value
00017  *   for the community.
00018  *
00019  *   This file is part of FORM.
00020  *
00021  *   FORM is free software: you can redistribute it and/or modify it under the
00022  *   terms of the GNU General Public License as published by the Free Software
00023  *   Foundation, either version 3 of the License, or (at your option) any later
00024  *   version.
00025  *
00026  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00027  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00028  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00029  *   details.
00030  *
00031  *   You should have received a copy of the GNU General Public License along
00032  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00033  */
00034 /* #] License : */
00035 #define NEWCODE
00036 /*
00037     #[ Includes : reshuf.c
00038 */
00039
00040 #include "form3.h"
00041
00042 /*
00043     #] Includes :
```

```
00044     #[ Reshuf :
00045
00046     Routines to rearrange dummy indices, so that
00047     a:  The notation becomes reasonably unique (the perfect job
00048         may consume very much time).
00049     b:  The value of AR.CurDum is reset.
00050
00051     Also some routines are needed to aid in the reading of stored
00052     expressions. Also in those expressions there can be dummy
00053     indices, and there should be no conflict with the already
00054     existing dummies.
00055
00056         #[ ReNumber :
00057
00058         Reads the term, tests for dummies, and puts them in order.
00059         Note that this is kind of a first order approximation.
00060         There is quite some room to make this routine 'smart'
00061         First order:
00062             First index will be lowest, second will be next etc.
00063         Second order:
00064             Functions with more than one index and symmetry properties
00065             have some look ahead to see which index is the first to
00066             find its partner.
00067         Third order:
00068             Take the ordering of the functions into account.
00069         Fourth order:
00070             Try all permutations and see which one gives the 'minimal' term.
00071         Currently we use only the first order.
00072
00073         We need a scratch array for the numbers we find, and one for
00074         the addresses at which these numbers are.
00075         We can use the space for the Scrat arrays. There are 13 of those
00076         and each is AM.MaxTal UWORDs long.
00077
00078 */
00079
00080 WORD ReNumber(PHEAD WORD *term)
00081 {
00082     GETBIDENTITY
00083     WORD *d, *e, **p, **f;
00084     WORD n, i, j, old;
00085     AN.DumFound = AN.RenumScratch;
00086     AN.DumPlace = AN.PoinScratch;
00087     AN.DumFunPlace = AN.FunScratch;
00088     AN.NumFound = 0;
00089     FunLevel(BHEAD term);
00090     d = AN.RenumScratch;
00091     p = AN.PoinScratch;
00092     f = AN.FunScratch;
00093 /*
00094     Now the first level sorting.
00095 */
00096     i = AN.IndDum;
00097     n = AN.NumFound;
00098     while ( --n >= 0 ) {
00099         if ( *d > 0 ) {
00100             old = **p;
00101             **p = ++i;
00102             if ( *f ) **f |= DIRTYSYMFLAG;
00103             e = d;
00104             e++;
00105             for ( j = 1; j <= n; j++ ) {
00106                 if ( *e && *(p[j]) == old ) {
00107                     *(p[j]) = i;
00108                     if ( f[j] ) *(f[j]) |= DIRTYSYMFLAG;
00109                     *e = 0;
00110                 }
00111                 e++;
00112             }
00113         }
00114         p++;
00115         d++;
00116         f++;
00117     }
00118     return(i);
00119 }
00120
00121 /*
00122         #] ReNumber :
00123         #[ FunLevel :
00124
00125         Does one term in determining where the dummies are.
00126         Made to work recursively for functions.
00127
00128 */
00129
00130 void FunLevel(PHEAD WORD *term)
```

```
00131 {
00132     GETBIDENTITY
00133     WORD *t, *tstop, *r, *fun;
00134     WORD *m;
00135     t = r = term;
00136     r += *r;
00137     tstop = r - ABS(r[-1]);
00138     t++;
00139     if ( t < tstop ) do {
00140         r = t + t[1];
00141         switch ( *t ) {
00142             case SYMBOL:
00143             case DOTPRODUCT:
00144                 break;
00145             case VECTOR:
00146                 t += 3;
00147                 do {
00148                     if ( *t > AN.IndDum ) {
00149                         if ( AN.NumFound >= AN.MaxRenumScratch ) AdjustRenumScratch(BHEAD0);
00150                         AN.NumFound++;
00151                         *AN.DumFound++ = *t;
00152                         *AN.DumPlace++ = t;
00153                         *AN.DumFunPlace++ = 0;
00154                     }
00155                     t += 2;
00156                 } while ( t < r );
00157                 break;
00158             case SUBEXPRESSION:
00159 /*
00160         Still must hunt down the wildcards(?)
00161 */
00162                 break;
00163             case GAMMA:
00164                 t += FUNHEAD-2;
00165                 /* fall through */
00166             case DELTA:
00167             case INDEX:
00168                 t += 2;
00169                 while ( t < r ) {
00170                     if ( *t > AN.IndDum ) {
00171                         if ( AN.NumFound >= AN.MaxRenumScratch ) AdjustRenumScratch(BHEAD0);
00172                         AN.NumFound++;
00173                         *AN.DumFound++ = *t;
00174                         *AN.DumPlace++ = t;
00175                         *AN.DumFunPlace++ = 0;
00176                     }
00177                     t++;
00178                 }
00179                 break;
00180             case HAAKJE:
00181             case EXPRESSION:
00182             case SNUMBER:
00183             case LNUMBER:
00184                 break;
00185             default:
00186                 if ( *t < FUNCTION ) {
00187 /* INTERNAL_ERROR_EXCL_START */
00188                   MLOCK(ErrorMessageLock);
00189                   MesPrint("!>Unexpected code in ReNumber");
00190                   MUNLOCK(ErrorMessageLock);
00191                   Terminate(-1);
00192 /* INTERNAL_ERROR_EXCL_STOP */
00193                 }
00194                 fun = t+2;
00195                 if ( *t >= FUNCTION && functions[*t-FUNCTION].spec
00196                 >= TENSORFUNCTION ) {
00197                     t += FUNHEAD;
00198                     while ( t < r ) {
00199                         if ( *t > AN.IndDum ) {
00200                             if ( AN.NumFound >= AN.MaxRenumScratch ) AdjustRenumScratch(BHEAD0);
00201                             AN.NumFound++;
00202                             *AN.DumFound++ = *t;
00203                             *AN.DumPlace++ = t;
00204                             *AN.DumFunPlace++ = fun;
00205                         }
00206                         t++;
00207                     }
00208                     break;
00209                 }
00210
00211                 t += FUNHEAD;
00212                 while ( t < r ) {
00213                     if ( *t > 0 ) {
00214
00215                         /* General function. Enter 'recursion'. */
00216
00217                         m = t + *t;
```

```
00218                         t += ARGHEAD;
00219                         while ( t < m ) {
00220                             FunLevel(BHEAD t);
00221                             t += *t;
00222                         }
00223                     }
00224                     else {
00225                         if ( *t == -INDEX ) {
00226                             t++;
00227                             if ( *t > AN.IndDum ) {
00228                                 if ( AN.NumFound >= AN.MaxRenumScratch ) AdjustRenumScratch(BHEAD0);
00229                                 AN.NumFound++;
00230                                 *AN.DumFound++ = *t;
00231                                 *AN.DumPlace++ = t;
00232                                 *AN.DumFunPlace++ = fun;
00233                             }
00234                             t++;
00235                         }
00236                         else if ( *t <= -FUNCTION ) t++;
00237                         else t += 2;
00238                     }
00239                 }
00240                 break;
00241         }
00242         t = r;
00243     } while ( t < tstop );
00244 }
00245
00246 /*
00247         #] FunLevel :
00248         #[ DetCurDum :
00249
00250         We look for indices in the range AM.IndDum to AM.IndDum+MAXDUMMIES.
00251         The maximum value is returned.
00252 */
00253
00254 WORD DetCurDum(PHEAD WORD *t)
00255 {
00256     GETBIDENTITY
00257     WORD maxval = AN.IndDum;
00258     WORD maxtop = AM.IndDum + WILDOFFSET;
00259     WORD *tstop, *m, *r, i;
00260     tstop = t + *t - 1;
00261     tstop -= ABS(*tstop);
00262     t++;
00263     while ( t < tstop ) {
00264         if ( *t == VECTOR ) {
00265             m = t + 3;
00266             t += t[1];
00267             while ( m < t ) {
00268                 if ( *m > maxval && *m < maxtop ) maxval = *m;
00269                 m += 2;
00270             }
00271         }
00272         else if ( *t == DELTA || *t == INDEX ) {
00273             m = t + 2;
00274 Singles:
00275             t += t[1];
00276             while ( m < t ) {
00277                 if ( *m > maxval && *m < maxtop ) maxval = *m;
00278                 m++;
00279             }
00280         }
00281         else if ( *t >= FUNCTION ) {
00282             if ( functions[*t-FUNCTION].spec >= TENSORFUNCTION ) {
00283                 m = t + FUNHEAD;
00284                 goto Singles;
00285             }
00286             r = t + FUNHEAD;
00287             t += t[1];
00288             while ( r < t ) {       /* The arguments */
00289                 if ( *r < 0 ) {
00290                     if ( *r <= -FUNCTION ) r++;
00291                     else if ( *r == -INDEX ) {
00292                         if ( r[1] > maxval && r[1] < maxtop ) maxval = r[1];
00293                         r += 2;
00294                     }
00295                     else r += 2;
00296                 }
00297                 else {
00298                     m = r + ARGHEAD;
00299                     r += *r;
00300                     while ( m < r ) {   /* Terms in the argument */
00301                         i = DetCurDum(BHEAD m);
00302                         if ( i > maxval && i < maxtop ) maxval = i;
00303                         m += *m;
00304                     }
```

```
00305                 }
00306             }
00307         }
00308         else {
00309             t += t[1];
00310         }
00311     }
00312     return(maxval);
00313 }
00314
00315 /*
00316         #] DetCurDum :
00317         #[ FullRenumber :
00318
00319         Does a full renumbering. May be slow if there are many indices.
00320         par = 1 Goes with a factorial!
00321         par = 0 All single exchanges only till there is no more improvement.
00322         Notice that there is a hole in the defense with respect to
00323         arguments inside functions inside functions.
00324 */
00325
00326 int FullRenumber(PHEAD WORD *term, WORD par)
00327 {
00328     GETBIDENTITY
00329     WORD *d, **p, **f, *w, *t, *best, *stac, *perm, a, *termtry;
00330     WORD n, i, j, k, ii;
00331     WORD *oldworkpointer = AT.WorkPointer;
00332     n = ReNumber(BHEAD term) - AM.IndDum;
00333     if ( n <= 1 ) return(0);
00334     Normalize(BHEAD term);
00335     if ( *term == 0 ) return(0);
00336     n = ReNumber(BHEAD term) - AM.IndDum;
00337     d = AN.RenumScratch;
00338     p = AN.PoinScratch;
00339     f = AN.FunScratch;
00340     if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
00341     k = AN.NumFound;
00342     best = w = AT.WorkPointer; t = term;
00343     for ( i = *term; i > 0; i-- ) *w++ = *t++;
00344     AT.WorkPointer = w;
00345     Normalize(BHEAD best);
00346     AT.WorkPointer = w = best + *best;
00347     stac = w+100;
00348     perm = stac + n + 1;
00349     termtry = perm + n + 1;
00350     for ( i = 1; i <= n; i++ ) perm[i] = i + AM.IndDum;
00351     for ( i = 1; i <= n; i++ ) stac[i] = i;
00352     for ( i = 0; i < k; i++ ) d[i] = *(p[i]) - AM.IndDum;
00353     if ( par == 0 ) {   /* All single exchanges */
00354         for ( i = 1; i < n; i++ ) {
00355             for ( j = i+1; j <= n; j++ ) {
00356                 a = perm[j]; perm[j] = perm[i]; perm[i] = a;
00357                 for ( ii = 0; ii < k; ii++ ) {
00358                     *(p[ii]) = perm[d[ii]];
00359                     if ( f[ii] ) *(f[ii]) |= DIRTYSYMFLAG;
00360                 }
00361                 t = term; w = termtry;
00362                 for ( ii = 0; ii < *term; ii++ ) *w++ = *t++;
00363                 AT.WorkPointer = w;
00364                 if ( Normalize(BHEAD termtry) == 0 ) {
00365                     if ( *termtry == 0 ) goto Return0;
00366                     if ( ( ii = CompareTerms(BHEAD termtry,best,0) ) > 0 ) {
00367                         t = termtry; w = best;
00368                         for ( ii = 0; ii < *termtry; ii++ ) *w++ = *t++;
00369                         i = 0; break; /* restart from beginning */
00370                     }
00371                     else if ( ii == 0 && CompCoef(termtry,best) != 0 )
00372                         goto Return0;
00373                 }
00374                 /* if no success, set back */
00375                 a = perm[j]; perm[j] = perm[i]; perm[i] = a;
00376             }
00377         }
00378     }
00379     else if ( par == 1 ) {  /* all permutations */
00380         j = n-1;
00381         for(;;) {
00382             if ( stac[j] == n ) {
00383                 a = perm[j]; perm[j] = perm[n]; perm[n] = a;
00384                 stac[j] = j;
00385                 j--;
00386                 if ( j <= 0 ) break;
00387                 continue;
00388             }
00389             if ( j != stac[j] ) {
00390                 a = perm[j]; perm[j] = perm[stac[j]]; perm[stac[j]] = a;
00391             }
```

```
00392             (stac[j])++;
00393             a = perm[j]; perm[j] = perm[stac[j]]; perm[stac[j]] = a;
00394             {
00395                 for ( i = 0; i < k; i++ ) {
00396                     *(p[i]) = perm[d[i]];
00397                     if ( f[i] ) *(f[i]) |= DIRTYSYMFLAG;
00398                 }
00399                 t = term; w = termtry;
00400                 for ( i = 0; i < *term; i++ ) *w++ = *t++;
00401                 AT.WorkPointer = w;
00402                 if ( Normalize(BHEAD termtry) == 0 ) {
00403                     if ( *termtry == 0 ) goto Return0;
00404                     if ( ( ii = CompareTerms(BHEAD termtry,best,0) ) > 0 ) {
00405                         t = termtry; w = best;
00406                         for ( i = 0; i < *termtry; i++ ) *w++ = *t++;
00407                     }
00408                     else if ( ii == 0 && CompCoef(termtry,best) != 0 )
00409                         goto Return0;
00410                 }
00411             }
00412             if ( j < n-1 ) { j = n-1; }
00413         }
00414     }
00415     t = term; w = best;
00416     n = *best;
00417     for ( i = 0; i < n; i++ ) *t++ = *w++;
00418     AT.WorkPointer = oldworkpointer;
00419     return(0);
00420 Return0:
00421     *term = 0;
00422     AT.WorkPointer = oldworkpointer;
00423     return(0);
00424 }
00425
00426 /*
00427         #] FullRenumber :
00428         #[ MoveDummies :
00429
00430         Routine shifts the dummy indices by an amount 'shift'.
00431         It is an adaptation of DetCurDum.
00432         Needed for  = ...*expression^power*...
00433         in which expression contains dummy indices.
00434         Note that this code should have been in ver1 already and has
00435         always been missing. Routine made 29-jan-2007.
00436 */
00437
00438 void MoveDummies(PHEAD WORD *term, WORD shift)
00439 {
00440     GETBIDENTITY
00441     WORD maxval = AN.IndDum;
00442     WORD maxtop = AM.IndDum + WILDOFFSET;
00443     WORD *tstop, *m, *r;
00444     tstop = term + *term - 1;
00445     tstop -= ABS(*tstop);
00446     term++;
00447     while ( term < tstop ) {
00448         if ( *term == VECTOR ) {
00449             m = term + 3;
00450             term += term[1];
00451             while ( m < term ) {
00452                 if ( *m > maxval && *m < maxtop ) *m += shift;
00453                 m += 2;
00454             }
00455         }
00456         else if ( *term == DELTA || *term == INDEX ) {
00457             m = term + 2;
00458 Singles:
00459             term += term[1];
00460             while ( m < term ) {
00461                 if ( *m > maxval && *m < maxtop ) *m += shift;
00462                 m++;
00463             }
00464         }
00465         else if ( *term >= FUNCTION ) {
00466             if ( functions[*term-FUNCTION].spec >= TENSORFUNCTION ) {
00467                 m = term + FUNHEAD;
00468                 goto Singles;
00469             }
00470             r = term + FUNHEAD;
00471             term += term[1];
00472             while ( r < term ) {        /* The arguments */
00473                 if ( *r < 0 ) {
00474                     if ( *r <= -FUNCTION ) r++;
00475                     else if ( *r == -INDEX ) {
00476                         if ( r[1] > maxval && r[1] < maxtop ) r[1] += shift;
00477                         r += 2;
00478                     }
```

```
00479                   else r += 2;
00480               }
00481               else {
00482                   m = r + ARGHEAD;
00483                   r += *r;
00484                   while ( m < r ) {   /* Terms in the argument */
00485                       MoveDummies(BHEAD m,shift);
00486                       m += *m;
00487                   }
00488               }
00489           }
00490       }
00491       else {
00492           term += term[1];
00493       }
00494   }
00495 }
00496
00497 /*
00498       #] MoveDummies :
00499       #[ AdjustRenumScratch :
00500
00501       Extends the buffer for number of dummies that can be found in
00502       a term. Originally we had a fixed buffer at size 300 in the AN
00503       struct. Thomas Hahn ran out of that. Hence we have now made it
00504       a dynamical buffer.
00505       Note that the pointers used in FunLevel need adjustment as well.
00506 */
00507
00508 void AdjustRenumScratch(PHEAD0)
00509 {
00510     GETBIDENTITY
00511     WORD newsize;
00512     int i;
00513     WORD **newpoin, *newnum;
00514     if ( AN.MaxRenumScratch == 0 ) newsize = 100;
00515     else newsize = AN.MaxRenumScratch*2;
00516     if ( newsize > MAXPOSITIVE/2 ) newsize = MAXPOSITIVE/2+1;
00517
00518     newpoin = (WORD **)Malloc1(newsize*sizeof(WORD *),"PoinScratch");
00519     for ( i = 0; i < AN.NumFound; i++ ) newpoin[i] = AN.PoinScratch[i];
00520     for ( ; i < newsize; i++ ) newpoin[i] = 0;
00521     if ( AN.PoinScratch ) M_free(AN.PoinScratch,"PoinScratch");
00522     AN.PoinScratch = newpoin;
00523     AN.DumPlace = newpoin + AN.NumFound;
00524
00525     newpoin = (WORD **)Malloc1(newsize*sizeof(WORD *),"FunScratch");
00526     for ( i = 0; i < AN.NumFound; i++ ) newpoin[i] = AN.FunScratch[i];
00527     for ( ; i < newsize; i++ ) newpoin[i] = 0;
00528     if ( AN.FunScratch ) M_free(AN.FunScratch,"FunScratch");
00529     AN.FunScratch = newpoin;
00530     AN.DumFunPlace = newpoin + AN.NumFound;
00531
00532     newnum = (WORD *)Malloc1(newsize*sizeof(WORD),"RenumScratch");
00533     for ( i = 0; i < AN.NumFound; i++ ) newnum[i] = AN.RenumScratch[i];
00534     for ( ; i < newsize; i++ ) newnum[i] = 0;
00535     if ( AN.RenumScratch ) M_free(AN.RenumScratch,"RenumScratch");
00536     AN.RenumScratch = newnum;
00537     AN.DumFound = newnum + AN.NumFound;
00538
00539     AN.MaxRenumScratch = newsize;
00540 }
00541
00542 /*
00543       #] AdjustRenumScratch :
00544     #] Reshuf :
00545     #[ Count :
00546       #[ CountDo :
00547
00548       This function executes the counting action in a count
00549       operation. The return value is the count of the term.
00550       Input is the term and a pointer to the instruction.
00551
00552 */
00553
00554 WORD CountDo(WORD *term, WORD *instruct)
00555 {
00556     WORD *m, *r, i, j, count = 0;
00557     WORD *stopper, *tstop, *r1 = 0, *r2 = 0;
00558     m = instruct;
00559     stopper = m + m[1];
00560     instruct += 3;
00561     tstop = term + *term; tstop -= ABS(tstop[-1]); term++;
00562     while ( term < tstop ) {
00563         switch ( *term ) {
00564             case SYMBOL:
00565                 i = term[1] - 2;
```

```
00566                 term += 2;
00567                 while ( i > 0 ) {
00568                     m = instruct;
00569                     while ( m < stopper ) {
00570                         if ( *m == SYMBOL && m[2] == *term ) {
00571                             count += m[3] * term[1];
00572                         }
00573                         m += m[1];
00574                     }
00575                     term += 2;
00576                     i -= 2;
00577                 }
00578                 break;
00579             case DOTPRODUCT:
00580                 i = term[1] - 2;
00581                 term += 2;
00582                 while ( i > 0 ) {
00583                     m = instruct;
00584                     while ( m < stopper ) {
00585                         if ( *m == DOTPRODUCT && (( m[2] == *term &&
00586                         m[3] == term[1]) || ( m[2] == term[1] &&
00587                         m[3] == *term )) ) {
00588                             count += m[4] * term[2];
00589                             break;
00590                         }
00591                         m += m[1];
00592                     }
00593                     m = instruct;
00594                     while ( m < stopper ) {
00595                         if ( *m == VECTOR && m[2] == *term &&
00596                         ( m[3] & DOTPBIT ) != 0 ) {
00597                             count += m[m[1]-1] * term[2];
00598                         }
00599                         m += m[1];
00600                     }
00601                     m = instruct;
00602                     while ( m < stopper ) {
00603                         if ( *m == VECTOR && m[2] == term[1] &&
00604                         ( m[3] & DOTPBIT ) != 0 ) {
00605                             count += m[m[1]-1] * term[2];
00606                         }
00607                         m += m[1];
00608                     }
00609                     term += 3;
00610                     i -= 3;
00611                 }
00612                 break;
00613             case INDEX:
00614                 j = 1;
00615                 goto VectInd;
00616             case VECTOR:
00617                 j = 2;
00618 VectInd:        i = term[1] - 2;
00619                 term += 2;
00620                 while ( i > 0 ) {
00621                     m = instruct;
00622                     while ( m < stopper ) {
00623                         if ( *m == VECTOR && m[2] == *term &&
00624                         ( m[3] & VECTBIT ) != 0 ) {
00625                             count += m[m[1]-1];
00626                         }
00627                         m += m[1];
00628                     }
00629                     term += j;
00630                     i -= j;
00631                 }
00632                 break;
00633             default:
00634                 if ( *term >= FUNCTION ) {
00635                     i = *term;
00636                     m = instruct;
00637                     while ( m < stopper ) {
00638                         if ( *m == FUNCTION && m[2] == i ) count += m[3];
00639                         m += m[1];
00640                     }
00641                     if ( functions[i-FUNCTION].spec >= TENSORFUNCTION ) {
00642                         i = term[1] - FUNHEAD;
00643                         term += FUNHEAD;
00644                         while ( i > 0 ) {
00645                             if ( *term < 0 ) {
00646                                 m = instruct;
00647                                 while ( m < stopper ) {
00648                                     if ( *m == VECTOR && m[2] == *term &&
00649                                     ( m[3] & FUNBIT ) != 0 ) {
00650                                         count += m[m[1]-1];
00651                                     }
00652                                     m += m[1];
```

```
00653                                   }
00654                               }
00655                               term++;
00656                               i--;
00657                           }
00658                       }
00659                       else {
00660                           r = term + term[1];
00661                           term += FUNHEAD;
00662                           while ( term < r ) {
00663                               if ( ( *term == -INDEX || *term == -VECTOR
00664                               || *term == -MINVECTOR ) && term[1] < MINSPEC ) {
00665                                   m = instruct;
00666                                   while ( m < stopper ) {
00667                                       if ( *m == VECTOR && term[1] == m[2]
00668                                       && ( m[3] & SETBIT ) != 0 ) {
00669                                           r1 = SetElements + Sets[m[4]].first;
00670                                           r2 = SetElements + Sets[m[4]].last;
00671                                           while ( r1 < r2 ) {
00672                                               if ( *r1 == i ) {
00673                                                   count += m[m[1]-1];
00674                                                   goto NextFF;
00675                                               }
00676                                               r1++;
00677                                           }
00678                                       }
00679                                       m += m[1];
00680                                   }
00681 NextFF:
00682                                   term += 2;
00683                               }
00684                               else { NEXTARG(term) }
00685                           }
00686                       }
00687                       break;
00688                   }
00689                   else {
00690                       term += term[1];
00691                   }
00692                   break;
00693           }
00694       }
00695       return(count);
00696 }
00697
00698 /*
00699           #] CountDo :
00700           #[ CountFun :
00701
00702           This is the count function.
00703           The return value is the count of the term.
00704           Input is the term and a pointer to the count function.
00705
00706 */
00707
00708 WORD CountFun(WORD *term, WORD *countfun)
00709 {
00710     WORD *m, *r, i, j, count = 0, *instruct, *stopper, *tstop;
00711     m = countfun;
00712     stopper = m + m[1];
00713     instruct = countfun + FUNHEAD;
00714     tstop = term + *term; tstop -= ABS(tstop[-1]); term++;
00715     while ( term < tstop ) {
00716         switch ( *term ) {
00717             case SYMBOL:
00718                 i = term[1] - 2;
00719                 term += 2;
00720                 while ( i > 0 ) {
00721                     m = instruct;
00722                     while ( m < stopper ) {
00723                         if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
00724                         if ( *m == -SYMBOL && m[1] == *term
00725                         && m[2] == -SNUMBER && ( m + 2 ) < stopper ) {
00726                             count += m[3] * term[1]; m += 4;
00727                         }
00728                         else { NEXTARG(m) }
00729                     }
00730                     term += 2;
00731                     i -= 2;
00732                 }
00733                 break;
00734             case DOTPRODUCT:
00735                 i = term[1] - 2;
00736                 term += 2;
00737                 while ( i > 0 ) {
00738                     m = instruct;
00739                     while ( m < stopper ) {
```

```
00740                         if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
00741                         if ( *m == 9+ARGHEAD && m[ARGHEAD] == 9
00742                         && m[ARGHEAD+1] == DOTPRODUCT
00743                         && m[ARGHEAD+9] == -SNUMBER && ( m + ARGHEAD+9 ) < stopper
00744                         && (( m[ARGHEAD+3] == *term &&
00745                         m[ARGHEAD+4] == term[1]) ||
00746                          ( m[ARGHEAD+3] == term[1] &&
00747                         m[ARGHEAD+4] == *term )) ) {
00748                             count += m[ARGHEAD+10] * term[2];
00749                             m += ARGHEAD+11;
00750                         }
00751                         else { NEXTARG(m) }
00752                     }
00753                     m = instruct;
00754                     while ( m < stopper ) {
00755                         if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
00756                         if ( ( *m == -VECTOR || *m == -MINVECTOR )
00757                         && m[1] == *term &&
00758                         m[2] == -SNUMBER && ( m+2 ) < stopper ) {
00759                             count += m[3] * term[2]; m += 4;
00760                         }
00761                         NEXTARG(m)
00762                     }
00763                     m = instruct;
00764                     while ( m < stopper ) {
00765                         if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
00766                         if ( ( *m == -VECTOR || *m == -MINVECTOR )
00767                         && m[1] == term[1] &&
00768                         m[2] == -SNUMBER && ( m+2 ) < stopper ) {
00769                             count += m[3] * term[2];
00770                             m += 4;
00771                         }
00772                         NEXTARG(m)
00773                     }
00774                     term += 3;
00775                     i -= 3;
00776                 }
00777                 break;
00778             case INDEX:
00779                 j = 1;
00780                 goto VectInd;
00781             case VECTOR:
00782                 j = 2;
00783 VectInd:        i = term[1] - 2;
00784                 term += 2;
00785                 while ( i > 0 ) {
00786                     m = instruct;
00787                     while ( m < stopper ) {
00788                         if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
00789                         if ( ( *m == -VECTOR || *m == -MINVECTOR )
00790                         && m[1] == *term &&
00791                         m[2] == -SNUMBER && (m+2) < stopper ) {
00792                             count += m[3]; m += 4;
00793                         }
00794                         NEXTARG(m)
00795                     }
00796                     term += j;
00797                     i -= j;
00798                 }
00799                 break;
00800             default:
00801                 if ( *term >= FUNCTION ) {
00802                     i = *term;
00803                     m = instruct;
00804                     while ( m < stopper ) {
00805                         if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
00806                         if ( *m == -i && m[1] == -SNUMBER && (m+1) < stopper ) {
00807                             count += m[2]; m += 3;
00808                         }
00809                         NEXTARG(m)
00810                     }
00811                     if ( functions[i-FUNCTION].spec >= TENSORFUNCTION ) {
00812                         i = term[1] - FUNHEAD;
00813                         term += FUNHEAD;
00814                         while ( i > 0 ) {
00815                             if ( *term < 0 ) {
00816                                 m = instruct;
00817                                 while ( m < stopper ) {
00818                                     if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
00819                                     if ( ( *m == -VECTOR || *m == -INDEX
00820                                     || *m == -MINVECTOR ) && m[1] == *term &&
00821                                     m[2] == -SNUMBER && (m+2) < stopper ) {
00822                                         count += m[3]; m += 4;
00823                                     }
00824                                     else { NEXTARG(m) }
00825                                 }
00826                             }
```

```
00827                               term++;
00828                               i--;
00829                           }
00830                       }
00831                       else {
00832                           r = term + term[1];
00833                           term += FUNHEAD;
00834                           while ( term < r ) {
00835                               if ( ( *term == -INDEX || *term == -VECTOR
00836                               || *term == -MINVECTOR ) && term[1] < MINSPEC ) {
00837                                   m = instruct;
00838                                   while ( m < stopper ) {
00839                                       if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
00840                                       if ( *m == -VECTOR && m[1] == term[1]
00841                                       && m[2] == -SNUMBER && (m+2) < stopper ) {
00842                                           count += m[3];
00843                                           m += 4;
00844                                       }
00845                                       else { NEXTARG(m) }
00846                                   }
00847                                   term += 2;
00848                               }
00849                               else { NEXTARG(term) }
00850                           }
00851                       }
00852                       break;
00853                   }
00854                   else {
00855                       term += term[1];
00856                   }
00857                   break;
00858           }
00859       }
00860       return(count);
00861   }
00862
00863   /*
00864           #] CountFun :
00865       #] Count :
00866       #[ DimensionSubterm :
00867   */
00868
00869   WORD DimensionSubterm(WORD *subterm)
00870   {
00871       WORD *r, *rstop, dim, i;
00872       LONG x = 0;
00873       rstop = subterm + subterm[1];
00874       if ( *subterm == SYMBOL ) {
00875           r = subterm + 2;
00876           while ( r < rstop ) {
00877               if ( *r <= NumSymbols && *r > -MAXPOWER ) {
00878                   dim = symbols[*r].dimension;
00879                   if ( dim == MAXPOSITIVE ) goto undefined;
00880                   x += dim * r[1];
00881                   if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
00882                   r += 2;
00883               }
00884               else if ( *r <= MAXVARIABLES ) {
00885   /*
00886                   Here we have an extra symbol. Store dimension in the compiler buffer
00887   */
00888                   i = MAXVARIABLES - *r;
00889                   dim = cbuf[AM.sbufnum].dimension[i];
00890                   if ( dim ==  MAXPOSITIVE ) goto undefined;
00891                   if ( dim == -MAXPOSITIVE ) goto outofrange;
00892                   x += dim * r[1];
00893                   if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
00894                   r += 2;
00895               }
00896               else { r += 2; }
00897           }
00898       }
00899       else if ( *subterm == DOTPRODUCT ) {
00900           r = subterm + 2;
00901           while ( r < rstop ) {
00902               dim = vectors[*r-AM.OffsetVector].dimension;
00903               if ( dim == MAXPOSITIVE ) goto undefined;
00904               x += dim * r[2];
00905               if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
00906               dim = vectors[r[1]-AM.OffsetVector].dimension;
00907               if ( dim == MAXPOSITIVE ) goto undefined;
00908               x += dim * r[2];
00909               if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
00910               r += 3;
00911           }
00912       }
00913       else if ( *subterm == VECTOR ) {
```

```
00914         r = subterm + 2;
00915         while ( r < rstop ) {
00916             dim = vectors[*r-AM.OffsetVector].dimension;
00917             if ( dim == MAXPOSITIVE ) goto undefined;
00918             x += dim;
00919             if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
00920             r += 2;
00921         }
00922     }
00923     else if ( *subterm == INDEX ) {
00924         r = subterm + 2;
00925         while ( r < rstop ) {
00926             if ( *r < 0 ) {
00927                 dim = vectors[*r-AM.OffsetVector].dimension;
00928                 if ( dim == MAXPOSITIVE ) goto undefined;
00929                 x += dim;
00930                 if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
00931             }
00932             r++;
00933         }
00934     }
00935     else if ( *subterm >= FUNCTION ) {
00936         dim = functions[*subterm-FUNCTION].dimension;
00937         if ( dim == MAXPOSITIVE ) goto undefined;
00938         x += dim;
00939         if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
00940         if ( functions[*subterm-FUNCTION].spec > 0 ) {  /* tensor */
00941             r = subterm + FUNHEAD;
00942             while ( r < rstop ) {
00943                 if ( *r < MINSPEC ) {
00944                     dim = vectors[*r-AM.OffsetVector].dimension;
00945                     if ( dim == MAXPOSITIVE ) goto undefined;
00946                     x += dim;
00947                     if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
00948                 }
00949                 r++;
00950             }
00951         }
00952     }
00953     return((WORD)x);
00954 undefined:
00955     return((WORD)MAXPOSITIVE);
00956 outofrange:
00957     return(-(WORD)MAXPOSITIVE);
00958 }
00959
00960 /*
00961     #] DimensionSubterm :
00962     #[ DimensionTerm :
00963
00964     Returns the dimension of the given term.
00965     If there is any variable of which the dimension is not defined
00966     we return the code for undefined which is MAXPOSITIVE
00967     When the value is out of range we return -MAXPOSITIVE
00968 */
00969
00970 WORD DimensionTerm(WORD *term)
00971 {
00972     WORD *t, *tstop, dim;
00973     LONG x = 0;
00974     tstop = term + *term; tstop -= ABS(tstop[-1]);
00975     t = term+1;
00976     while ( t < tstop ) {
00977         dim = DimensionSubterm(t);
00978         if ( dim ==  MAXPOSITIVE ) goto undefined;
00979         if ( dim == -MAXPOSITIVE ) goto outofrange;
00980         x += dim;
00981         if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
00982         t += t[1];
00983     }
00984     return((WORD)x);
00985 undefined:
00986     return((WORD)MAXPOSITIVE);
00987 outofrange:
00988     return(-(WORD)MAXPOSITIVE);
00989 }
00990
00991 /*
00992     #] DimensionTerm :
00993     #[ DimensionExpression :
00994
00995     Returns the dimension of the given expression.
00996     If there is any variable of which the dimension is not defined
00997     we return the code for undefined which is MAXPOSITIVE
00998     When the value is out of range we return -MAXPOSITIVE
00999     When the value is not consistent we return -MAXPOSITIVE.
01000 */
```

```
01001
01002 WORD DimensionExpression(PHEAD WORD *expr)
01003 {
01004     WORD dim, *term, *old, x = 0;
01005     int first = 1;
01006     term = expr;
01007     while ( *term ) {
01008         dim = DimensionTerm(term);
01009         if ( dim ==  MAXPOSITIVE ) goto undefined;
01010         if ( dim == -MAXPOSITIVE ) goto outofrange;
01011         if ( first ) { x = dim; }
01012         else if ( x != dim ) {
01013             old  = AN.currentTerm;
01014             MLOCK(ErrorMessageLock);
01015             MesPrint("Dimension is not the same in the terms of the expression");
01016             term = expr;
01017             while ( *term ) {
01018                 AN.currentTerm = term;
01019                 MesPrint("   %T");
01020             }
01021             MUNLOCK(ErrorMessageLock);
01022             AN.currentTerm = old;
01023             return(-(WORD)MAXPOSITIVE);
01024         }
01025         term += *term;
01026     }
01027     return((WORD)x);
01028 undefined:
01029     return((WORD)MAXPOSITIVE);
01030 outofrange:
01031     old  = AN.currentTerm;
01032     AN.currentTerm = term;
01033     MLOCK(ErrorMessageLock);
01034     MesPrint("Dimension out of range in %t in subexpression");
01035     MUNLOCK(ErrorMessageLock);
01036     AN.currentTerm = old;
01037     return(-(WORD)MAXPOSITIVE);
01038 }
01039
01040 /*
01041     #] DimensionExpression :
01042     #[ Multiply Term :
01043         #[ MultDo :
01044 */
01045
01046 int MultDo(PHEAD WORD *term, WORD *pattern)
01047 {
01048     GETBIDENTITY
01049     WORD *t, *r, i;
01050     t = term + *term;
01051     if ( pattern[2] > 0 ) {         /* Left multiply */
01052         i = *term - 1;
01053     }
01054     else {                          /* Right multiply */
01055         i = ABS(t[-1]);
01056     }
01057     *term += SUBEXPSIZE;
01058     r = t + SUBEXPSIZE;
01059     do { *--r = *--t; } while ( --i > 0 );
01060     r = pattern + 3;
01061     i = r[1];
01062     while ( --i >= 0 ) *t++ = *r++;
01063     AT.WorkPointer = term + *term;
01064     return(0);
01065 }
01066
01067 /*
01068         #] MultDo :
01069     #] Multiply Term :
01070     #[ Try Term(s) :
01071         #[ TryDo :
01072 */
01073
01074 int TryDo(PHEAD WORD *term, WORD *pattern, WORD level)
01075 {
01076     GETBIDENTITY
01077     WORD *t, *r, *m, i, j;
01078     ReNumber(BHEAD term);
01079     Normalize(BHEAD term);
01080     m = r = term + *term;
01081     m++;
01082     i = pattern[2];
01083     t = pattern + 3;
01084     NCOPY(m,t,i)
01085     j = *term - 1;
01086     t = term + 1;
01087     NCOPY(m,t,j)
```

```
01088     *r = WORDDIF(m,r);
01089     AT.WorkPointer = m;
01090     if ( ( j = Normalize(BHEAD r) ) == 0 || j == 1 ) {
01091         if ( *r == 0 ) return(0);
01092         ReNumber(BHEAD r); Normalize(BHEAD r);
01093         if ( *r == 0 ) return(0);
01094         if ( ( i = CompareTerms(BHEAD term,r,0) ) < 0 ) {
01095             *AN.RepPoint = 1;
01096             AR.expchanged = 1;
01097             return(Generator(BHEAD r,level));
01098         }
01099         if ( i == 0 && CompCoef(term,r) != 0 ) { return(0); }
01100     }
01101     AT.WorkPointer = r;
01102     return(Generator(BHEAD term,level));
01103 }
01104
01105 /*
01106         #] TryDo :
01107     #] Try Term(s) :
01108     #[ Distribute :
01109         #[ DoDistrib :
01110
01111         The routine that generates the terms ordered by a distrib_ function.
01112         The presence of a replaceable distrib_ function has been sensed
01113         in the routine TestSub and has been passed on to Generator.
01114         It is then Generator that calls this function in a way that is
01115         similar to calling the trace routines, except for that for the
01116         trace routines and the Levi-Civita tensors the arguments are put
01117         in temporary storage and here we leave them inside the term,
01118         because there is no knowing how long the field will be.
01119 */
01120
01121 int DoDistrib(PHEAD WORD *term, WORD level)
01122 {
01123     GETBIDENTITY
01124     WORD *t, *m, *r = 0, *stop, *tstop, *termout, *endhead, *starttail, *parms;
01125     WORD i, j, k, n, nn, ntype, fun1 = 0, fun2 = 0, typ1 = 0, typ2 = 0;
01126     WORD *arg, *oldwork, *mf, ktype = 0, atype = 0;
01127     WORD sgn, dirtyflag;
01128     AN.TeInFun = AR.TePos = 0;
01129     t = term;
01130     tstop = t + *t;
01131     stop = tstop - ABS(tstop[-1]);
01132     t++;
01133     while ( t < stop ) {
01134         r = t + t[1];
01135         if ( *t == DISTRIBUTION && t[FUNHEAD] == -SNUMBER
01136         && t[FUNHEAD+1] >= -2 && t[FUNHEAD+1] <= 2
01137         && t[FUNHEAD+2] == -SNUMBER
01138         && t[FUNHEAD+4] <= -FUNCTION
01139         && t[FUNHEAD+5] <= -FUNCTION ) {
01140             WORD *ttt = t+FUNHEAD+6, *tttstop = t+t[1];
01141             while ( ttt < tttstop ) {
01142                 if ( *ttt == -DOLLAREXPRESSION ) break;
01143                 NEXTARG(ttt);
01144             }
01145             if ( ttt >= tttstop ) {
01146                 fun1 = -t[FUNHEAD+4];
01147                 fun2 = -t[FUNHEAD+5];
01148                 typ1 = functions[fun1-FUNCTION].spec;
01149                 typ2 = functions[fun2-FUNCTION].spec;
01150                 if ( typ1 > 0 || typ2 > 0 ) {
01151                     m = t + FUNHEAD+6;
01152                     r = t + t[1];
01153                     while ( m < r ) {
01154                         if ( *m != -INDEX && *m != -VECTOR && *m != -MINVECTOR )
01155                             break;
01156                         m += 2;
01157                     }
01158                     if ( m < r ) {
01159                         MLOCK(ErrorMessageLock);
01160                         MesPrint("Incompatible function types and arguments in distrib_");
01161                         MUNLOCK(ErrorMessageLock);
01162                         SETERROR(-1)
01163                     }
01164                 }
01165                 break;
01166             }
01167         }
01168         t = r;
01169     }
01170     dirtyflag = t[2];
01171     ntype = t[FUNHEAD+1];
01172     n = t[FUNHEAD+3];
01173 /*
01174     t points at the distrib_ function to be expanded.
```

```
01175     fun1,fun2 and typ1,typ2 are the two functions and their types.
01176     ntype indicates the action:
01177         0:  Make all possible divisions:  2^nargs
01178         1:  fun1 should get n arguments:  nargs! / ( n! (nargs-n)! )
01179         2:  fun2 should get n arguments:  nargs! / ( n! (nargs-n)! )
01180     The distinction between 1 and two is for noncommuting objects.
01181         3:  fun1 should get n arguments. Super symmetric option.
01182         4:  fun2 idem
01183     The super symmetric option involves:
01184         a:  arguments get sorted
01185         b:  identical arguments are seen as such. Hence not all their
01186             distributions are taken into account. It is as if after the
01187             distrib there is a symmetrize fun1; symmetrize fun2;
01188         c:  Hence if the occurrence of each argument is a,b,c,...
01189             and their occurrence in fun1 is a1,b1,c1,... and in fun2
01190             is a2,b2,c2,... then each term is generated (a1+a2)!/a1!/a2!
01191             (b1+b2)!/b1!/b2! (c1+c2)!/c1!/c2! ... times.
01192         d:  We have to make an array of occurrences and positions.
01193         e:  Then we sort the arguments indirectly.
01194         f:  Next we generate the argument lists in the same way as we
01195             generate powers of expressions with binomials. Hence we need
01196             a third array to keep track of the `powers'
01197 */
01198     endhead = t;
01199     starttail = r;
01200     parms = m = t + FUNHEAD+6;
01201     i = 0;
01202     while ( m < r ) {   /* Count arguments */
01203         i++;
01204         NEXTARG(m);
01205     }
01206     oldwork = AT.WorkPointer;
01207     arg = AT.WorkPointer + 1;
01208     arg[-1] = 0;
01209     termout = arg + i;
01210     switch ( ntype ) {
01211         case  0: ktype = 1; atype = n < 0 ? 1: 0; n = 0; break;
01212         case  1: ktype = 1; atype = 0; break;
01213         case  2: ktype = 0; atype = 0; break;
01214         case -1: ktype = 1; atype = 1; break;
01215         case -2: ktype = 0; atype = 1; break;
01216     }
01217     do {
01218 /*
01219         All distributions with n elements. We generate the array arg with
01220         all possible 1 and 0 patterns. 1 means in fun1 and 0 means in fun2.
01221 */
01222         if ( n > i ) return(0);     /* 0 elements */
01223
01224         for ( j = 0; j < n; j++ ) arg[j] = 1;
01225         for ( j = n; j < i; j++ ) arg[j] = 0;
01226         for(;;) {
01227             sgn = 0;
01228             t = term;
01229             m = termout;
01230             while ( t < endhead ) *m++ = *t++;
01231             mf = m;
01232             *m++ = fun1;
01233             *m++ = FUNHEAD;
01234             *m++ = dirtyflag;
01235 #if FUNHEAD > 3
01236             k = FUNHEAD -3;
01237             while ( k-- > 0 ) *m++ = 0;
01238 #endif
01239             r = parms;
01240             for ( k = 0; k < i; k++ ) {
01241                 if ( arg[k] == ktype ) {
01242                     if ( *r <= -FUNCTION ) *m++ = *r++;
01243                     else if ( *r < 0 ) {
01244                         if ( typ1 > 0 ) {
01245                             if ( *r == -MINVECTOR ) sgn ^= 1;
01246                             r++;
01247                             *m++ = *r++;
01248                         }
01249                         else { *m++ = *r++; *m++ = *r++; }
01250                     }
01251                     else {
01252                         nn = *r;
01253                         NCOPY(m,r,nn);
01254                     }
01255                 }
01256                 else { NEXTARG(r) }
01257             }
01258             mf[1] = WORDDIF(m,mf);
01259             mf = m;
01260             *m++ = fun2;
01261             *m++ = FUNHEAD;
```

```
01262             *m++ = dirtyflag;
01263 #if FUNHEAD > 3
01264             k = FUNHEAD -3;
01265             while ( k-- > 0 ) *m++ = 0;
01266 #endif
01267             r = parms;
01268             for ( k = 0; k < i; k++ ) {
01269                 if ( arg[k] != ktype ) {
01270                     if ( *r <= -FUNCTION ) *m++ = *r++;
01271                     else if ( *r < 0 ) {
01272                         if ( typ2 > 0 ) {
01273                             if ( *r == -MINVECTOR ) sgn ^= 1;
01274                             r++;
01275                             *m++ = *r++;
01276                         }
01277                         else { *m++ = *r++; *m++ = *r++; }
01278                     }
01279                     else {
01280                         nn = *r;
01281                         NCOPY(m,r,nn);
01282                     }
01283                 }
01284                 else { NEXTARG(r) }
01285             }
01286             mf[1] = WORDDIF(m,mf);
01287 #ifndef NUOVO
01288             if ( atype == 0 ) {
01289                 WORD k1,k2;
01290                 for ( k = 0; k < i-1; k++ ) {
01291                     if ( arg[k] == 0 ) continue;
01292                     k1 = 1; k2 = k;
01293                     while ( k < i-1 && EqualArg(parms,k,k+1) ) { k++; k1++; }
01294                     while ( k2 <= k && arg[k2] == 1 ) k2++;
01295                     k2 = k-k2+1;
01296 /*
01297                     Now we need k1!/(k2! (k1-k2)!)
01298 */
01299                     if ( k2 != k1 && k2 != 0 ) {
01300                         if ( GetBinom((UWORD *)m+3,m+2,k1,k2) ) {
01301                             MLOCK(ErrorMessageLock);
01302                             MesCall("DoDistrib");
01303                             MUNLOCK(ErrorMessageLock);
01304                             SETERROR(-1)
01305                         }
01306                         m[1] = ( m[2] < 0 ? -m[2]: m[2] ) + 3;
01307                         *m = LNUMBER;
01308                         m += m[1];
01309                     }
01310                 }
01311             }
01312 #endif
01313             r = starttail;
01314             while ( r < tstop ) *m++ = *r++;
01315
01316             if ( atype ) {      /* antisymmetric field */
01317                 k = n;
01318                 nn = 0;
01319                 for ( j = 0; j < i && k > 0; j++ ) {
01320                     if ( arg[j] == 1 ) k--;
01321                     else nn += k;
01322                 }
01323                 sgn ^= nn & 1;
01324             }
01325
01326             if ( sgn ) m[-1] = -m[-1];
01327             *termout = WORDDIF(m,termout);
01328             AT.WorkPointer = m;
01329             if ( AT.WorkPointer > AT.WorkTop ) {
01330                 MLOCK(ErrorMessageLock);
01331                 MesWork();
01332                 MUNLOCK(ErrorMessageLock);
01333                 return(-1);
01334             }
01335             *AN.RepPoint = 1;
01336             AR.expchanged = 1;
01337             if ( Generator(BHEAD termout,level) ) Terminate(-1);
01338 #ifndef NUOVO
01339             {
01340                 WORD k1;
01341             j = i - 1;
01342             k = 0;
01343 redok:      while ( arg[j] == 1 && j >= 0 ) { j--; k++; }
01344             while ( arg[j] == 0 && j >= 0 ) j--;
01345             if ( j < 0 ) break;
01346             k1 = j;
01347             arg[j] = 0;
01348             while ( !atype && EqualArg(parms,j,j+1) ) {
```

```
01349                 j++;
01350                 if ( j >= i - k - 1 ) { j = k1; k++; goto redok; }
01351                 arg[j] = 0;
01352             }
01353             while ( k >= 0 ) { j++; arg[j] = 1; k--; }
01354             j++;
01355             while ( j < i ) { arg[j] = 0; j++; }
01356             }
01357 #else
01358             j = i - 1;
01359             k = 0;
01360             while ( arg[j] == 1 && j >= 0 ) { j--; k++; }
01361             while ( arg[j] == 0 && j >= 0 ) j--;
01362             if ( j < 0 ) break;
01363             arg[j] = 0;
01364             while ( k >= 0 ) { j++; arg[j] = 1; k--; }
01365             j++;
01366             while ( j < i ) { arg[j] = 0; j++; }
01367 #endif
01368         }
01369     } while ( ntype == 0 && ++n <= i );
01370     AT.WorkPointer = oldwork;
01371     return(0);
01372 }
01373
01374 /*
01375         #] DoDistrib :
01376         #[ EqualArg :
01377
01378         Returns 1 if the arguments in the field are identical.
01379 */
01380
01381 int EqualArg(WORD *parms, WORD num1, WORD num2)
01382 {
01383     WORD *t1, *t2;
01384     WORD i;
01385     t1 = parms;
01386     while ( --num1 >= 0 ) { NEXTARG(t1); }
01387     t2 = parms;
01388     while ( --num2 >= 0 ) { NEXTARG(t2); }
01389     if ( *t1 != *t2 ) return(0);
01390     if ( *t1 < 0 ) {
01391         if ( *t1 <= -FUNCTION || t1[1] == t2[1] ) return(1);
01392         return(0);
01393     }
01394     i = *t1;
01395     while ( --i >= 0 ) {
01396         if ( *t1 != *t2 ) return(0);
01397         t1++; t2++;
01398     }
01399     return(1);
01400 }
01401
01402 /*
01403         #] EqualArg :
01404         #[ DoDelta3 :
01405 */
01406
01407 int DoDelta3(PHEAD WORD *term, WORD level)
01408 {
01409     GETBIDENTITY
01410     WORD *t, *m, *m1, *m2, *stopper, *tstop, *termout, *dels, *taken;
01411     WORD *ic, *jc, *factors;
01412     WORD num, num2, i, j, k, knum, a;
01413     AN.TeInFun = AR.TePos = 0;
01414     tstop = term + *term;
01415     stopper = tstop - ABS(tstop[-1]);
01416     t = term+1;
01417     while ( ( *t != DELTA3 || ((t[1]-FUNHEAD) & 1 ) != 0 ) && t < stopper )
01418         t += t[1];
01419     if ( t >= stopper ) {
01420 /* INTERNAL_ERROR_EXCL_START */
01421         MLOCK(ErrorMessageLock);
01422         MesPrint("!>Internal error with dd_ function");
01423         MUNLOCK(ErrorMessageLock);
01424         Terminate(-1);
01425 /* INTERNAL_ERROR_EXCL_STOP */
01426     }
01427     m1 = t; m2 = t + t[1];
01428     num = t[1] - FUNHEAD;
01429     if ( num == 0 ) {
01430         termout = t = AT.WorkPointer;
01431         m = term;
01432         while ( m < m1 ) *t++ = *m++;
01433         m = m2; while ( m < tstop ) *t++ = *m++;
01434         *termout = WORDDIF(t,termout);
01435         AT.WorkPointer = t;
```

```
01436         *AN.RepPoint = 1;
01437         AR.expchanged = 1;
01438         if ( Generator(BHEAD termout,level) ) {
01439             MLOCK(ErrorMessageLock);
01440             MesCall("Do dd_");
01441             MUNLOCK(ErrorMessageLock);
01442             SETERROR(-1)
01443         }
01444         AT.WorkPointer = termout;
01445         return(0);
01446     }
01447     t += FUNHEAD;
01448 /*
01449     Step 1: sort the arguments
01450 */
01451     for ( i = 1; i < num; i++ ) {
01452         if ( t[i] < t[i-1] ) {
01453             a = t[i]; t[i] = t[i-1]; t[i-1] = a;
01454             j = i - 1;
01455             while ( j > 0 ) {
01456                 if ( t[j] >= t[j-1] ) break;
01457                 a = t[j]; t[j] = t[j-1]; t[j-1] = a;
01458                 j--;
01459             }
01460         }
01461     }
01462 /*
01463     Step 2: Order them by occurrence
01464     In 'taken' we have the array with the number of occurrences.
01465     in 'dels' is the type of object.
01466 */
01467     m = taken = AT.WorkPointer;
01468     for ( i = 0; i < num; i++ ) *m++ = 0;
01469     dels = m; knum = 0;
01470     for ( i = 0; i < num; knum++ ) {
01471         *m++ = t[i]; i++; taken[knum] = 1;
01472         while ( i < num ) {
01473             if ( t[i] != t[i-1] ) break;
01474             i++; (taken[knum])++;
01475         }
01476     }
01477     for ( i = 0; i < knum; i++ ) *m++ = taken[i];
01478     ic = m; num2 = num/2;
01479     jc = ic + num2;
01480     factors = jc + num2;
01481     termout = factors + num2;
01482 /*
01483     The recursion has num/2 steps
01484 */
01485     k = 0;
01486     while ( k >= 0 ) {
01487         if ( k >= num2 ) {
01488             t = termout; m = term;
01489             while ( m < m1 ) *t++ = *m++;
01490             *t++ = DELTA; *t++ = num+2;
01491             for ( i = 0; i < num2; i++ ) {
01492                 *t++ = dels[ic[i]]; *t++ = dels[jc[i]];
01493             }
01494             for ( i = 0; i < num2; i++ ) {
01495                 if ( ic[i] == jc[i] ) {
01496                     j = 1;
01497                     while ( i < num2-1 && ic[i] == ic[i+1] && ic[i] == jc[i+1] )
01498                         { i++; j++; }
01499                     for ( a = 1; a < j; a++ ) {
01500                         *t++ = SNUMBER; *t++ = 4; *t++ = 2*a+1; *t++ = 1;
01501                     }
01502                     for ( a = 0; a+1+i < num2; a++ ) {
01503                         if ( ic[a+i] != ic[a+i+1] ) break;
01504                     }
01505                     if ( a > 0 ) {
01506                         if ( GetBinom((UWORD *)(t+3),t+2,2*j+a,a) ) {
01507                             MLOCK(ErrorMessageLock);
01508                             MesCall("Do dd_");
01509                             MUNLOCK(ErrorMessageLock);
01510                             SETERROR(-1)
01511                         }
01512                         t[1] = ( t[2] < 0 ? -t[2]: t[2] ) + 3;
01513                         *t = LNUMBER;
01514                         t += t[1];
01515                     }
01516                 }
01517                 else if ( factors[i] != 1 ) {
01518                     *t++ = SNUMBER; *t++ = 4; *t++ = factors[i]; *t++ = 1;
01519                 }
01520             }
01521             for ( i = 0; i < num2-1; i++ ) {
01522                 if ( ic[i] == jc[i] ) continue;
```

```
01523                 j = 1;
01524                 while ( i < num2-1 && jc[i] == jc[i+1] && ic[i] == ic[i+1] ) {
01525                     i++; j++;
01526                 }
01527                 for ( a = 0; a+i < num2-1; a++ ) {
01528                     if ( ic[i+a] != ic[i+a+1] ) break;
01529                 }
01530                 if ( a > 0 ) {
01531                     if ( GetBinom((UWORD *)(t+3),t+2,j+a,a) ) {
01532                         MLOCK(ErrorMessageLock);
01533                         MesCall("Do dd_");
01534                         MUNLOCK(ErrorMessageLock);
01535                         SETERROR(-1)
01536                     }
01537                     t[1] = ( t[2] < 0 ? -t[2]: t[2] ) + 3;
01538                     *t = LNUMBER;
01539                     t += t[1];
01540                 }
01541             }
01542             m = m2;
01543             while ( m < tstop ) *t++ = *m++;
01544             *termout = WORDDIF(t,termout);
01545             AT.WorkPointer = t;
01546             *AN.RepPoint = 1;
01547             AR.expchanged = 1;
01548             if ( Generator(BHEAD termout,level) ) {
01549                 MLOCK(ErrorMessageLock);
01550                 MesCall("Do dd_");
01551                 MUNLOCK(ErrorMessageLock);
01552                 SETERROR(-1)
01553             }
01554             k--;
01555             if ( k >= 0 ) goto nextj;
01556             else break;
01557         }
01558         for ( ic[k] = 0; ic[k] < knum; ic[k]++ ) {
01559             if ( taken[ic[k]] > 0 ) break;
01560         }
01561         if ( k > 0 && ic[k-1] == ic[k] ) jc[k] = jc[k-1];
01562         else jc[k] = ic[k];
01563         for ( ; jc[k] < knum; jc[k]++ ) {
01564             if ( taken[jc[k]] <= 0 ) continue;
01565             if ( ic[k] == jc[k] ) {
01566                 if ( taken[jc[k]] <= 1 ) continue;
01567 /*
01568                 factors[k] = taken[ic[k]];
01569                 if ( ( factors[k] & 1 ) == 0 ) (factors[k])--;
01570 */
01571                 taken[ic[k]] -= 2;
01572             }
01573             else {
01574                 factors[k] = taken[jc[k]];
01575                 (taken[ic[k]])--; (taken[jc[k]])--;
01576             }
01577             k++;
01578             goto nextk;  /* This is the simulated recursion */
01579 nextj:;
01580             (taken[ic[k]])++; (taken[jc[k]])++;
01581         }
01582         k--;
01583         if ( k >= 0 ) goto nextj;
01584 nextk:;
01585     }
01586     AT.WorkPointer = taken;
01587     return(0);
01588 }
01589
01590 /*
01591         #] DoDelta3 :
01592         #[ TestPartitions :
01593
01594         Checks whether the function in tfun is a partitions_ function
01595         that can be expanded. If it can a number of relevant objects is
01596         inside the struct parti.
01597         This test is not entirely trivial because there are many restrictions
01598         w.r.t. the arguments.
01599         Syntax (still to be implemented)
01600         partitions_(number_of_partition_entries,[function,number,]^nope,arguments)
01601         [function,number,]: can be
01602             f,3     for a partition of 3 arguments
01603             f,0     for the remaining arguments (should be last)
01604             num1,f,num2 with num1 effectively a number of partitions but this
01605                     counts as num1 entries.
01606             0,f,num2: all partitions have num2 arguments. No number of partition
01607                     entries needed. If num2 does not divide the number of
01608                     arguments there will be no action.
01609 */
```

```
01610
01611 int TestPartitions(WORD *tfun, PARTI *parti)
01612 {
01613     WORD *tnext = tfun + tfun[1];
01614     WORD *t, *tt;
01615     WORD argcount = 0, sum = 0, i, ipart, argremain;
01616     WORD tensorflag = 0;
01617     parti->psize = parti->nfun = parti->args = parti->nargs = 0;
01618     parti->numargs = parti->numpart = parti->where = 0;
01619     tt = t = tfun + FUNHEAD;
01620     while ( t < tnext ) { argcount++; NEXTARG(t); }
01621     if ( argcount < 1 ) goto No;
01622     t = tt;
01623     if ( *t != -SNUMBER ) goto No;
01624     if ( t[1] == 0 ) {
01625         t += 2;
01626         if ( *t <= -FUNCTION && t[1] == -SNUMBER && t[2] > 0 ) {
01627             if ( functions[-*t-FUNCTION].spec > 0 ) tensorflag = 1;
01628             if ( argcount-3 < 0 ) goto No;
01629             if ( ( (argcount-3) % t[2] ) != 0 ) goto No;
01630         }
01631         else goto No;
01632         parti->numpart = (argcount-3)/t[2];
01633         parti->numargs = argcount - 3;
01634         parti->psize = (WORD *)Malloc1((parti->numpart*2+parti->numargs*2+2)
01635                         *sizeof(WORD),"partitions");
01636         parti->nfun = parti->psize + parti->numpart;
01637         parti->args = parti->nfun + parti->numpart;
01638         parti->nargs = parti->args + parti->numargs;
01639         for ( i = 0; i < parti->numpart; i++ ) {
01640             parti->psize[i] =  t[2];
01641             parti->nfun[i]  = -t[0];
01642         }
01643         t += 3;
01644     }
01645     else if ( t[1] > 0 ) { /* Number of partitions */
01646 /*
01647         We can have sequences of function,number for one partition
01648         or number1,function,number2 for number1 partitions of size number2.
01649         The last partition can have number=0. It must be a single partition
01650         and it will take all remaining arguments.
01651         If any of the functions is a tensor, all arguments must be either
01652         vector or index.
01653 */
01654         parti->numpart = t[1]; t += 2;
01655         ipart = sum = 0; argremain = argcount - 1;
01656 /*
01657         At this point is seems better to make an allocation already that
01658         may be too big. The alternative is having to pass this code twice.
01659 */
01660         parti->psize = (WORD *)Malloc1((argcount*4+2)*sizeof(WORD),"partitions");
01661         parti->nfun = parti->psize+argcount;
01662         parti->args = parti->nfun+argcount;
01663         parti->nargs = parti->args+argcount;
01664         while ( ipart < parti->numpart ) {
01665             if ( *t <= -FUNCTION && t[1] == -SNUMBER && t[2] >= 0 ) {
01666                 if ( functions[-*t-FUNCTION].spec > 0 ) tensorflag = 1;
01667                 if ( t[2] == 0 ) {
01668                     if ( ipart+1 != parti->numpart ) goto WhatAPity;
01669                     argremain -= 2;
01670                     parti->nfun[ipart] = -*t;
01671                     parti->psize[ipart++] = argremain-sum;
01672                     ipart++;
01673                     sum = argremain;
01674                 }
01675                 else {
01676                     parti->nfun[ipart] = -*t;
01677                     parti->psize[ipart++] = t[2];
01678                     argremain -= 2;
01679                     sum += t[2];
01680                 }
01681                 t += 3;
01682             }
01683             else if ( *t == -SNUMBER && t[1] > 0 && ipart+t[1] <= parti->numpart
01684             && t[2] <= -FUNCTION && t[3] == -SNUMBER && t[4] > 0 ) {
01685                 if ( functions[-t[2]-FUNCTION].spec > 0 ) tensorflag = 1;
01686                 argremain -= 3;
01687                 for ( i = 0; i < t[1]; i++ ) {
01688                     parti->nfun[ipart] = -t[2];
01689                     parti->psize[ipart++] = t[4];
01690                     sum += t[4];
01691                 }
01692                 if ( sum > argremain ) goto WhatAPity;
01693                 t += 5;
01694             }
01695             else goto WhatAPity;
01696         }
```

```
01697         if ( sum != argremain ) goto WhatAPity;
01698         parti->numargs = argremain;
01699     }
01700     else goto No;
01701 /*
01702     Now load the offsets of the arguments and check if needed whether OK with tensor
01703 */
01704     for ( i = 0; i < parti->numargs; i++ ) {
01705         parti->args[i] = t - tfun;
01706         if ( tensorflag && ( *t != -VECTOR && *t != -INDEX ) ) goto WhatAPity;
01707         NEXTARG(t);
01708     }
01709     return(1);
01710 WhatAPity:
01711     M_free(parti->psize,"partitions");
01712     parti->psize = parti->nfun = parti->args = parti->nargs = 0;
01713     parti->numargs = parti->numpart = parti->where = 0;
01714 No:
01715     return(0);
01716 }
01717
01718 /*
01719         #] TestPartitions :
01720         #[ DoPartitions :
01721
01722     As we have only one AT.partitions we need to copy it locally
01723     if we keep needing it.
01724 */
01725
01726 int DoPartitions(PHEAD WORD *term, WORD level)
01727 {
01728     WORD x, i, j, im, *fun, ndiff, siz, tensorflag = 0;
01729     PARTI part = AT.partitions;
01730     WORD *array, **j3, **j3fill, **j3where;
01731     WORD a, pfill, *j2, *j2fill, j3size, ncoeff, ncoeffnum, nfac, ncoeff2, ncoeff3, n;
01732     UWORD *coeff, *coeffnum, *cfac, *coeff2, *coeff3, *c;
01733     /* Make AT.partitions ready for future use (if there is another function) */
01734     AT.partitions.psize = AT.partitions.nfun = AT.partitions.args = AT.partitions.nargs = 0;
01735     AT.partitions.numargs = AT.partitions.numpart = AT.partitions.where = 0;
01736 /*
01737     Start with bubble sorting the list of arguments. And the list of partitions.
01738 */
01739     fun = term + part.where;
01740     if ( functions[*fun-FUNCTION].spec > 0 ) tensorflag = 1;
01741     for ( i = 1; i < part.numargs; i++ ) {
01742         for ( j = i-1; j >= 0; j-- ) {
01743             if ( CompArg(fun+part.args[j+1],fun+part.args[j]) >= 0 ) break;
01744             x = part.args[j+1]; part.args[j+1] = part.args[j]; part.args[j] = x;
01745         }
01746     }
01747     for ( i = 1; i < part.numpart; i++ ) {
01748         for ( j = i-1; j >= 0; j-- ) {
01749             if ( part.psize[j+1] < part.psize[j] ) break;
01750             if ( part.psize[j+1] == part.psize[j] && part.nfun[j+1] <= part.nfun[j] ) break;
01751             x = part.psize[j+1]; part.psize[j+1] = part.psize[j]; part.psize[j] = x;
01752             x = part.nfun[j+1]; part.nfun[j+1] = part.nfun[j]; part.nfun[j] = x;
01753         }
01754     }
01755 /*
01756     Now we have the partitions sorted from high to low and the arguments
01757     have been sorted the regular way arguments are sorted in a symmetrize.
01758     The important thing is that identical arguments are adjacent.
01759     Assign the numbers (identical arguments have identical numbers).
01760 */
01761     ndiff = 1; part.nargs[0] = ndiff;
01762     for ( i = 1; i < part.numargs; i++ ) {
01763         if ( CompArg(fun+part.args[i],fun+part.args[i-1]) != 0 ) ndiff++;
01764         part.nargs[i] = ndiff;
01765     }
01766     part.nargs[part.numargs] = 0;
01767     coeffnum = NumberMalloc("partitionsn");
01768     coeff = NumberMalloc("partitions");
01769     coeff2 = NumberMalloc("partitions2");
01770     coeff3 = NumberMalloc("partitions3");
01771     cfac = NumberMalloc("partitions!");
01772     ncoeffnum = 1; coeffnum[0] = 1;
01773 /*
01774     The numerator of the coefficient will be n1!*n2!*...*n(ndiff)!
01775     We compute it only once.
01776 */
01777     j = 0;
01778     for ( i = 1; i <= ndiff; i++ ) {
01779         n = 0;
01780         while ( part.nargs[j] == i ) { n++; j++; }
01781         if ( n > 1 ) { /* 1! needs no attention */
01782             if ( Factorial(BHEAD n, cfac, &nfac) ) Terminate(-1);
01783             if ( MulLong(coeffnum,ncoeffnum,cfac,nfac,coeff2,&ncoeff2) ) Terminate(-1);
```

```
01784             c = coeffnum; coeffnum = coeff2; coeff2 = c;
01785             n = ncoeffnum; ncoeffnum = ncoeff2; ncoeff2 = n;
01786         }
01787     }
01788 /*
01789     Now comes the part where we have to make sure that
01790     a: we generate all partitions.
01791     b: we generate only different partitions.
01792     c: we get the proper combinatorics factor.
01793     Method:
01794     Suppose the largest partition needs n objects and there are m partitions.
01795     We allocate m arrays of n 'digits'. Make in the smaller partitions the
01796     appropriate leading digits zero.
01797     Divide the largest numbers (of the arguments) over the partitions as
01798     leftmost digits (after possible zeroes). The arrays, seen as numbers,
01799     should be such that each is less or equal to its left neighbour. Take the
01800     next largest numbers, etc. This generates unique partitions and all of
01801     them. Because we have a formula for the multiplicity, this should do it.
01802
01803     The general case. At a later stage we might put in a more economical
01804     version for special cases.
01805 */
01806     siz = part.psize[0];
01807     j3size = 2*(part.numpart+1)+2*(part.numargs+1);
01808     array = (WORD *)Malloc1((part.numpart+1)*siz*sizeof(WORD),"parts");
01809     j3 = (WORD **)Malloc1(j3size*sizeof(WORD *),"parts3");
01810     j2 = (WORD *)Malloc1((part.numpart+part.numargs+2)*sizeof(WORD),"parts2");
01811     j3fill = j3+(part.numpart+1);
01812     j3where = j3fill+(part.numpart+1);
01813     for ( i = 0; i < j3size; i++ ) j3[i] = 0;
01814     j2fill = j2+(part.numpart+1);
01815     for ( i = 0; i < part.numargs; i++ ) j2fill[i] = 0;
01816     for ( i = 0; i < part.numpart; i++ ) {
01817         j3[i] = array+i*siz;
01818         for ( j = 0; j < siz; j++ ) j3[i][j] = 0;
01819         j3fill[i] = j3[i]+(siz-part.psize[i]);
01820         j2[i] = part.psize[i];  /* Number of places still available */
01821     }
01822     j3[part.numpart] = array+part.numpart*siz;
01823     j2[part.numpart] = 0;
01824 /*
01825     Now comes a complicated two-level recursion in a and pfill.
01826 */
01827     a = part.numargs-1;
01828     pfill = 0;
01829 /*
01830     We start putting the last number in part.nargs in the first partition in array.
01831     For backtracking we need to know where we put this number. Hence j3where.
01832 */
01833     while ( a < part.numargs ) {
01834         while ( j2[pfill] <= 0 ) {
01835             pfill++;
01836             while ( pfill >= part.numpart ) { /* we have to pop */
01837                 a++;
01838                 if ( a >= part.numargs ) goto Done;
01839                 pfill = j2fill[a];
01840                 j2[pfill]++;
01841                 j3where[a][0] = 0;
01842                 j3fill[pfill]--;
01843                 pfill++;
01844             }
01845         }
01846         j3where[a] = j3fill[pfill];
01847         *(j3fill[pfill])++ = part.nargs[a];
01848         j2[pfill]--; j2fill[a] = pfill;
01849 /*
01850         Now test whether this is allowed.
01851 */
01852         if ( pfill > 0 && part.psize[pfill] == part.psize[pfill-1]
01853              && part.nfun[pfill] == part.nfun[pfill-1] ) { /* First check whether allowed */
01854             for ( im = 0; im < siz; im++ ) {
01855                 if ( j3[pfill-1][im] < j3[pfill][im] ) break;
01856                 if ( j3[pfill-1][im] > j3[pfill][im] ) im = siz;
01857             }
01858             if ( im < siz ) { /* not ordered. undo and raise pfill */
01859                 pfill = j2fill[a];
01860                 j2[pfill]++;
01861                 j3where[a][0] = 0;
01862                 j3fill[pfill]--;
01863                 pfill++;
01864                 continue; /* Note that j2[part.numpart] = 0 */
01865             }
01866         }
01867         a--;
01868         if ( a < 0 ) {  /* Solution */
01869 /*
01870             #[ Solution :
```

```
01871
01872             Now we compose the output term. The input term contains
01873             three parts: head, partitions_, tail.
01874             partitions_ starts at term+part.where.
01875             We first put the function parts and worry about the coefficient later.
01876 */
01877             WORD *t, *to, *twhere = term+part.where, *t2, *tend = term+*term, *termout;
01878             WORD num, jj, *targ, *tfun;
01879             t2 = twhere+twhere[1];
01880             to = termout = AT.WorkPointer;
01881             if ( termout + *term + part.numpart*FUNHEAD + AM.MaxTal >= AT.WorkTop ) {
01882                 MesWork();
01883             }
01884             for ( i = 0; i < ncoeffnum; i++ ) coeff[i] = coeffnum[i];
01885             ncoeff = ncoeffnum;
01886             t = term; while ( t < twhere ) *to++ = *t++;
01887 /*
01888             Now the partitions
01889 */
01890             for ( i = 0; i < part.numpart; i++ ) {
01891                 tfun = to;
01892                 *to++ = part.nfun[i]; to++; FILLFUN(to);
01893                 for ( j = 1; j <= part.psize[i]; j++ ) {
01894                     num = j3[i][siz-j]; /* now we need an argument with this number */
01895                     for ( jj = num-1; jj < part.numargs; jj++ ) {
01896                         if ( part.nargs[jj] == num ) break;
01897                     }
01898                     targ = part.args[jj]+twhere;
01899                     if ( *targ < 0 ) {
01900                         if ( tensorflag ) targ++;
01901                         else if ( *targ > -FUNCTION ) *to++ = *targ++;
01902                         *to++ = *targ++;
01903                     }
01904                     else { jj = *targ; NCOPY(to,targ,jj); }
01905                 }
01906                 tfun[1] = to - tfun;
01907             }
01908 /*
01909             Now the denominators of the coefficient
01910             First identical functions/partitions
01911 */
01912             j = 1; n = 1;
01913             while ( j < part.numpart ) {
01914                 for ( im = 0; im < siz; im++ ) {
01915                     if ( part.nfun[j-1] != part.nfun[j] ) break;
01916                     if ( j3[j-1][im] < j3[j][im] ) break;
01917                     if ( j3[j-1][im] > j3[j][im] ) im = 2*siz+2;
01918                 }
01919                 if ( im == siz ) { n++; j++; continue; }
01920                 if ( n > 1 ) {
01921 div1:               if ( Factorial(BHEAD n, cfac, &nfac) ) Terminate(-1);
01922                     if ( DivLong(coeff,ncoeff,cfac,nfac,coeff2,&ncoeff2,coeff3,&ncoeff3) )
    Terminate(-1);
01923                     c = coeff; coeff = coeff2; coeff2 = c;
01924                     n = ncoeff; ncoeff = ncoeff2; ncoeff2 = n;
01925                 }
01926                 n = 1; j++;
01927             }
01928             if ( n > 1 ) goto div1;
01929 /*
01930             Now identical elements inside the partitions
01931 */
01932             for ( i = 0; i < part.numpart; i++ ) {
01933                 j = 0; while ( j3[i][j] == 0 ) j++;
01934                 n = 1; j++;
01935                 while ( j < siz ) {
01936                     if ( j3[i][j-1] == j3[i][j] ) { n++; j++; }
01937                     else {
01938                         if ( n > 1 ) {
01939 div2:                       if ( Factorial(BHEAD n, cfac, &nfac) ) Terminate(-1);
01940                             if ( DivLong(coeff,ncoeff,cfac,nfac,coeff2,&ncoeff2,coeff3,&ncoeff3) )
    Terminate(-1);
01941                             c = coeff; coeff = coeff2; coeff2 = c;
01942                             n = ncoeff; ncoeff = ncoeff2; ncoeff2 = n;
01943                         }
01944                         n = 1; j++;
01945                     }
01946                 }
01947                 if ( n > 1 ) goto div2;
01948             }
01949 /*
01950             And put this inside the term. Normalize will take care of it.
01951 */
01952             if ( ncoeff != 1 || coeff[0] > 1 ) {
01953                 if ( ncoeff == 1 && coeff[0] <= MAXPOSITIVE ) {
01954                     *to++ = SNUMBER; *to++ = 4; *to++ = (WORD)(coeff[0]); *to++ = 1;
01955                 }
```

```
01956                 else {
01957                     *to++ = LNUMBER; *to++ = ncoeff+3; *to++ = ncoeff;
01958                     for ( i = 0; i < ncoeff; i++ ) *to++ = ((WORD *)coeff)[i];
01959                 }
01960             }
01961 /*
01962             And the tail
01963 */
01964             while ( t2 < tend ) *to++ = *t2++;
01965             *termout = to-termout;
01966             AT.WorkPointer = to;
01967             if ( Generator(BHEAD termout,level) ) Terminate(-1);
01968             AT.WorkPointer = termout;
01969 /*
01970             #] Solution :
01971
01972             Now we can pop all a with the lowest value and one more.
01973 */
01974             a = 0;
01975             while ( part.nargs[a] == 1 ) {
01976                 pfill = j2fill[a]; j2[pfill]++; j3where[a][0] = 0; j3fill[pfill]--; a++;
01977             }
01978             if ( a < part.numargs ) {
01979                 pfill = j2fill[a]; j2[pfill]++; j3where[a][0] = 0; j3fill[pfill]--; a++;
01980             }
01981             a--;
01982             pfill++;
01983         }
01984         else if ( part.nargs[a] == part.nargs[a+1] ) {}
01985         else { pfill = 0; }
01986     }
01987 Done:
01988     M_free(j2,"parts2");
01989     M_free(j3,"parts3");
01990     M_free(array,"parts");
01991     NumberFree(cfac,"partitions!");
01992     NumberFree(coeff3,"partitions3");
01993     NumberFree(coeff2,"partitions2");
01994     NumberFree(coeff,"partitions");
01995     NumberFree(coeffnum,"partitionsn");
01996     M_free(part.psize,"partitions");
01997     part.psize = part.nfun = part.args = part.nargs = 0;
01998     part.numargs = part.numpart = part.where = 0;
01999     return(0);
02000 }
02001
02002 /*
02003         #] DoPartitions :
02004     #] Distribute :
02005     #[ DoPermutations :
02006
02007     Routine replaces the function perm_(f,args) by occurrences of f with
02008     all permutations of the args. This should always fit!
02009 */
02010
02011 int DoPermutations(PHEAD WORD *term, WORD level)
02012 {
02013     PERMP perm;
02014     WORD *oldworkpointer = AT.WorkPointer, *termout = AT.WorkPointer;
02015     WORD *t, *tstop, *tt, *ttstop, odd = 0;
02016     WORD *args[MAXMATCH], nargs, i, first, skip, *to, *from;
02017 /*
02018     Find function and count arguments. Check for odd/even
02019 */
02020     tstop = term+*term; tstop -= ABS(tstop[-1]);
02021     t = term+1;
02022     while ( t < tstop ) {
02023         if ( *t == PERMUTATIONS ) {
02024             if ( t[1] >= FUNHEAD+1 && t[FUNHEAD] <= -FUNCTION ) {
02025                 odd = 0; skip = 1;
02026             }
02027             else if ( t[1] >= FUNHEAD+3 && t[FUNHEAD] == -SNUMBER && t[FUNHEAD+2] <= -FUNCTION ) {
02028                 if ( t[FUNHEAD+1] % 2 == 1 ) odd = -1;
02029                 else odd = 0;
02030                 skip = 3;
02031             }
02032             else { t += t[1]; continue; }
02033             tt = t+FUNHEAD+skip; ttstop = t + t[1];
02034             nargs = 0;
02035             while ( tt < ttstop ) { NEXTARG(tt); nargs++; }
02036             tt = t+FUNHEAD+skip;
02037             if ( nargs > MAXMATCH ) {
02038                 MLOCK(ErrorMessageLock);
02039                 MesPrint("Too many arguments in function perm_. %d! is way too big",(WORD)MAXMATCH);
02040                 MUNLOCK(ErrorMessageLock);
02041                 SETERROR(-1)
02042             }
```

```
02043             i = 0;
02044             while ( tt < ttstop ) { args[i++] = tt; NEXTARG(tt); }
02045             perm.n = nargs;
02046             perm.sign = 0;
02047             perm.objects = args;
02048             first = 1;
02049             while ( (first = PermuteP(&perm,first) ) == 0 ) {
02050 /*
02051                 Compose the output term
02052 */
02053                 to = termout; from = term;
02054                 while ( from < t ) *to++ = *from++;
02055                 *to++ = -t[FUNHEAD+skip-1];
02056                 *to++ = t[1] - skip;
02057                 for ( i = 2; i < FUNHEAD; i++ ) *to++ = t[i];
02058                 for ( i = 0; i < nargs; i++ ) {
02059                     from = args[i];
02060                     COPY1ARG(to,from);
02061                 }
02062                 from = t+t[1];
02063                 tstop = term + *term;
02064                 while ( from < tstop ) *to++ = *from++;
02065                 if ( odd && ( ( perm.sign & 1 ) != 0 ) ) to[-1] = -to[-1];
02066                 *termout = to - termout;
02067                 AT.WorkPointer = to;
02068                 if ( Generator(BHEAD termout,level) ) Terminate(-1);
02069                 AT.WorkPointer = oldworkpointer;
02070             }
02071             return(0);
02072         }
02073         t += t[1];
02074     }
02075     return(0);
02076 }
02077
02078 /*
02079     #] DoPermutations :
02080     #[ DoShuffle :
02081
02082     Merges the arguments of all occurrences of function fun into a
02083     single occurrence of fun. The opposite of Distrib_
02084     Syntax:
02085         Shuffle[,once|all],fun;
02086         Shuffle[,once|all],$fun;
02087     The expansion of the dollar should give a single function.
02088     The dollar is indicated as usual with a negative value.
02089     option = 1 (once): generate identical results only once
02090     option = 0 (all): generate identical results with combinatorics (default)
02091 */
02092
02093 /*
02094     We use the Shuffle routine which has a large amount of combinatorics.
02095     It doesn't have grouped combinatorics as in (0,1,2)*(0,1,3) where the
02096     groups (0,1) also cause double terms.
02097 */
02098
02099 int DoShuffle(WORD *term, WORD level, WORD fun, WORD option)
02100 {
02101     GETIDENTITY
02102     SHvariables SHback, *SH = &(AN.SHvar);
02103     WORD *t1, *t2, *tstop, ncoef, n = fun, *to, *from;
02104     int i, error;
02105     LONG k;
02106     UWORD *newcombi;
02107
02108     if ( n < 0 ) {
02109         if ( ( n = DolToFunction(BHEAD -n) ) == 0 ) {
02110             MLOCK(ErrorMessageLock);
02111             MesPrint("$-variable in merge statement did not evaluate to a function.");
02112             MUNLOCK(ErrorMessageLock);
02113             return(1);
02114         }
02115     }
02116     if ( AT.WorkPointer + 3*(*term) + AM.MaxTal > AT.WorkTop ) {
02117         MLOCK(ErrorMessageLock);
02118         MesWork();
02119         MUNLOCK(ErrorMessageLock);
02120         return(-1);
02121     }
02122
02123     tstop = term + *term;
02124     ncoef = tstop[-1];
02125     tstop -= ABS(ncoef);
02126     t1 = term + 1;
02127     while ( t1 < tstop ) {
02128         if ( ( *t1 == n ) && ( t1+t1[1] < tstop ) && ( t1[1] > FUNHEAD ) ) {
02129             t2 = t1 + t1[1];
```

```
02130             if ( t2 >= tstop ) {
02131                 return(Generator(BHEAD term,level));
02132             }
02133             while ( t2 < tstop ) {
02134                 if ( ( *t2 == n ) && ( t2[1] > FUNHEAD ) ) break;
02135                 t2 += t2[1];
02136             }
02137             if ( t2 < tstop ) break;
02138         }
02139         t1 += t1[1];
02140     }
02141     if ( t1 >= tstop ) {
02142         return(Generator(BHEAD term,level));
02143     }
02144     *AN.RepPoint = 1;
02145 /*
02146     Now we have two occurrences of the function.
02147     Back up all relevant variables and load all the stuff that needs to be
02148     passed on.
02149 */
02150     SHback = AN.SHvar;
02151     SH->finishuf = &FinishShuffle;
02152     SH->do_uffle = &DoShuffle;
02153     SH->outterm = AT.WorkPointer;
02154     AT.WorkPointer += *term;
02155     SH->stop1 = t1 + t1[1];
02156     SH->stop2 = t2 + t2[1];
02157     SH->thefunction = n;
02158     SH->option = option;
02159     SH->level = level;
02160     SH->incoef = tstop;
02161     SH->nincoef = ncoef;
02162
02163     if ( AN.SHcombi == 0 || AN.SHcombisize == 0 ) {
02164         AN.SHcombisize = 200;
02165         AN.SHcombi = (UWORD *)Malloc1(AN.SHcombisize*sizeof(UWORD),"AN.SHcombi");
02166         SH->combilast = 0;
02167         SHback.combilast = 0;
02168     }
02169     else {
02170         SH->combilast += AN.SHcombi[SH->combilast]+1;
02171         if ( SH->combilast >= AN.SHcombisize - 100 ) {
02172             newcombi = (UWORD *)Malloc1(2*AN.SHcombisize*sizeof(UWORD),"AN.SHcombi");
02173             for ( k = 0; k < AN.SHcombisize; k++ ) newcombi[k] = AN.SHcombi[k];
02174             M_free(AN.SHcombi,"AN.SHcombi");
02175             AN.SHcombi = newcombi;
02176             AN.SHcombisize *= 2;
02177         }
02178     }
02179     AN.SHcombi[SH->combilast] = 1;
02180     AN.SHcombi[SH->combilast+1] = 1;
02181
02182     i = t1-term; to = SH->outterm; from = term;
02183     NCOPY(to,from,i)
02184     SH->outfun = to;
02185     for ( i = 0; i < FUNHEAD; i++ ) { *to++ = t1[i]; }
02186
02187     error = Shuffle(t1+FUNHEAD,t2+FUNHEAD,to);
02188
02189     AT.WorkPointer = SH->outterm;
02190     AN.SHvar = SHback;
02191     if ( error ) {
02192         MesCall("DoShuffle");
02193         return(-1);
02194     }
02195     return(0);
02196 }
02197
02198 /*
02199     #] DoShuffle :
02200     #[ Shuffle :
02201
02202     How to make shuffles:
02203
02204     We have two lists of arguments. We have to make a single
02205     shuffle of them. All combinations. Doubles should have as
02206     much as possible a combinatorics factor. Sometimes this is
02207     very difficult as in:
02208         (0,1,2)x(0,1,3) -> (0,1) is a repeated pattern and the
02209         factor on that is difficult
02210     Simple way: (without combinatorics)
02211         repeat id  f0(?c)*f(x1?,?a)*f(x2?,?b) =
02212                         +f0(?c,x1)*f(?a)*f(x2,?b)
02213                         +f0(?c,x2)*f(x1,?a)*f(?b);
02214     Refinement:
02215             if ( x1 == x2 ) check how many more there are of the same.
02216             --> (n1,x) and (n2,x)
```

```
02217             id  f0(?c)*f1((n1,x),?b)*f2((n2,x),?c) =
02218                     +binom_(n1+n2,n1)*f0(?c,(n1+n2,x))*f1(?a)*f2(?b)
02219                     +sum_(j,0,n1-1,binom_(n2+j,j)*f0(?c,(j+n2,x))
02220                             *f1((n1-j),?a)*f2(?b))*force2
02221                     +sum_(j,0,n2-1,binom_(n1+j,j)*f0(?c,(j+n1,x))
02222                             *f1(?a)*f2((n2-j),?b))*force1
02223     The force operation can be executed directly
02224
02225     The next question is how to program this: recursively or linearly
02226     which would require simulation of a recursion. Recursive is clearest
02227     but we need to pass a number of arguments from the calling routine
02228     to the final routine. This is done with AN.SHvar.
02229
02230     We need space for the accumulation of the combinatoric factors.
02231 */
02232
02233 int Shuffle(WORD *from1, WORD *from2, WORD *to)
02234 {
02235     GETIDENTITY
02236     WORD *t, *fr, *next1, *next2, na, *fn1, *fn2, *tt;
02237     int i, n, n1, n2, j;
02238     LONG combilast;
02239     SHvariables *SH = &(AN.SHvar);
02240     if ( from1 == SH->stop1 && from2 == SH->stop2 ) {
02241         return(FiniShuffle(to));
02242     }
02243     else if ( from1 == SH->stop1 ) {
02244         i = SH->stop2 - from2; t = to; tt = from2; NCOPY(t,tt,i)
02245         return(FiniShuffle(t));
02246     }
02247     else if ( from2 == SH->stop2 ) {
02248         i = SH->stop1 - from1; t = to; tt = from1; NCOPY(t,tt,i)
02249         return(FiniShuffle(t));
02250     }
02251 /*
02252     Compare lead arguments
02253 */
02254     if ( AreArgsEqual(from1,from2) ) {
02255 /*
02256         First find out how many of each
02257 */
02258         next1 = from1; n1 = 1; NEXTARG(next1)
02259         while ( ( next1 < SH->stop1 ) && AreArgsEqual(from1,next1) ) {
02260             n1++; NEXTARG(next1)
02261         }
02262         next2 = from2; n2 = 1; NEXTARG(next2)
02263         while ( ( next2 < SH->stop2 ) && AreArgsEqual(from2,next2) ) {
02264             n2++; NEXTARG(next2)
02265         }
02266         combilast = SH->combilast;
02267 /*
02268             +binom_(n1+n2,n1)*f0(?c,(n1+n2,x))*f1(?a)*f2(?b)
02269 */
02270         t = to;
02271         n = n1 + n2;
02272         while ( --n >= 0 ) { fr = from1; CopyArg(t,fr) }
02273         if ( GetBinom((UWORD *)(t),&na,n1+n2,n1) ) goto shuffcall;
02274         if ( combilast + AN.SHcombi[combilast] + na + 2 >= AN.SHcombisize ) {
02275 /*
02276             We need more memory in this stack. Fortunately this is the
02277             only place where we have to do this, because the other factors
02278             are definitely smaller.
02279             Layout:   size, LongInteger, size, LongInteger, .....
02280             We start pointing at the last one.
02281 */
02282             UWORD *combi = (UWORD *)Malloc1(2*AN.SHcombisize*2,"AN.SHcombi");
02283             LONG jj;
02284             for ( jj = 0; jj < AN.SHcombisize; jj++ ) combi[jj] = AN.SHcombi[jj];
02285             AN.SHcombisize *= 2;
02286             M_free(AN.SHcombi,"AN.SHcombi");
02287             AN.SHcombi = combi;
02288         }
02289         if ( MulLong((UWORD *)(AN.SHcombi+combilast+1),AN.SHcombi[combilast],
02290                     (UWORD *)(t),na,
02291                     (UWORD *)(AN.SHcombi+combilast+AN.SHcombi[combilast]+2),
02292                     (WORD *)(AN.SHcombi+combilast+AN.SHcombi[combilast]+1)) ) goto shuffcall;
02293         SH->combilast = combilast + AN.SHcombi[combilast] + 1;
02294         if ( next1 >= SH->stop1 ) {
02295             fr = next2; i = SH->stop2 - fr;
02296             NCOPY(t,fr,i)
02297             if ( FiniShuffle(t) ) goto shuffcall;
02298         }
02299         else if ( next2 >= SH->stop2 ) {
02300             fr = next1; i = SH->stop1 - fr;
02301             NCOPY(t,fr,i)
02302             if ( FiniShuffle(t) ) goto shuffcall;
02303         }
```

```
02304         else {
02305             if ( Shuffle(next1,next2,t) ) goto shuffcall;
02306         }
02307         SH->combilast = combilast;
02308 /*
02309             +sum_(j,0,n1-1,binom_(n2+j,j)*f0(?c,(j+n2,x))
02310                     *f1((n1-j),?a)*f2(?b))*force2
02311 */
02312         if ( next2 < SH->stop2 ) {
02313          t = to;
02314          n = n2;
02315          while ( --n >= 0 ) { fr = from1; CopyArg(t,fr) }
02316          for ( j = 0; j < n1; j++ ) {
02317           if ( GetBinom((UWORD *)(t),&na,n2+j,j) ) goto shuffcall;
02318           if ( MulLong((UWORD *)(AN.SHcombi+combilast+1),AN.SHcombi[combilast],
02319                        (UWORD *)(t),na,
02320                        (UWORD *)(AN.SHcombi+combilast+AN.SHcombi[combilast]+2),
02321                        (WORD *)(AN.SHcombi+combilast+AN.SHcombi[combilast]+1)) ) goto shuffcall;
02322           SH->combilast = combilast + AN.SHcombi[combilast] + 1;
02323           if ( j > 0 ) { fr = from1; CopyArg(t,fr) }
02324           fn2 = next2; tt = t;
02325           CopyArg(tt,fn2)
02326
02327           if ( fn2 >= SH->stop2 ) {
02328             n = n1-j;
02329             while ( --n >= 0 ) { fr = from1; CopyArg(tt,fr) }
02330             fr = next1; i = SH->stop1 - fr;
02331             NCOPY(tt,fr,i)
02332             if ( FiniShuffle(tt) ) goto shuffcall;
02333           }
02334           else {
02335             n = j; fn1 = from1; while ( --n >= 0 ) { NEXTARG(fn1) }
02336             if ( Shuffle(fn1,fn2,tt) ) goto shuffcall;
02337           }
02338           SH->combilast = combilast;
02339          }
02340         }
02341 /*
02342             +sum_(j,0,n2-1,binom_(n1+j,j)*f0(?c,(j+n1,x))
02343                     *f1(?a)*f2((n2-j),?b))*force1
02344 */
02345         if ( next1 < SH->stop1 ) {
02346          t = to;
02347          n = n1;
02348          while ( --n >= 0 ) { fr = from1; CopyArg(t,fr) }
02349          for ( j = 0; j < n2; j++ ) {
02350           if ( GetBinom((UWORD *)(t),&na,n1+j,j) ) goto shuffcall;
02351           if ( MulLong((UWORD *)(AN.SHcombi+combilast+1),AN.SHcombi[combilast],
02352                        (UWORD *)(t),na,
02353                        (UWORD *)(AN.SHcombi+combilast+AN.SHcombi[combilast]+2),
02354                        (WORD *)(AN.SHcombi+combilast+AN.SHcombi[combilast]+1)) ) goto shuffcall;
02355           SH->combilast = combilast + AN.SHcombi[combilast] + 1;
02356           if ( j > 0 ) { fr = from1; CopyArg(t,fr) }
02357           fn1 = next1; tt = t;
02358           CopyArg(tt,fn1)
02359
02360           if ( fn1 >= SH->stop1 ) {
02361             n = n2-j;
02362             while ( --n >= 0 ) { fr = from1; CopyArg(tt,fr) }
02363             fr = next2; i = SH->stop2 - fr;
02364             NCOPY(tt,fr,i)
02365             if ( FiniShuffle(tt) ) goto shuffcall;
02366           }
02367           else {
02368             n = j; fn2 = from2; while ( --n >= 0 ) { NEXTARG(fn2) }
02369             if ( Shuffle(fn1,fn2,tt) ) goto shuffcall;
02370           }
02371           SH->combilast = combilast;
02372          }
02373         }
02374     }
02375     else {
02376 /*
02377         Argument from first list
02378 */
02379         t = to;
02380         fr = from1;
02381         CopyArg(t,fr)
02382         if ( fr >= SH->stop1 ) {
02383             fr = from2; i = SH->stop2 - fr;
02384             NCOPY(t,fr,i)
02385             if ( FiniShuffle(t) ) goto shuffcall;
02386         }
02387         else {
02388             if ( Shuffle(fr,from2,t) ) goto shuffcall;
02389         }
02390 /*
```

```
02391         Argument from second list
02392 */
02393         t = to;
02394         fr = from2;
02395         CopyArg(t,fr)
02396         if ( fr >= SH->stop2 ) {
02397             fr = from1; i = SH->stop1 - fr;
02398             NCOPY(t,fr,i)
02399             if ( FiniShuffle(t) ) goto shuffcall;
02400         }
02401         else {
02402             if ( Shuffle(from1,fr,t) ) goto shuffcall;
02403         }
02404     }
02405     return(0);
02406 shuffcall:
02407     MesCall("Shuffle");
02408     return(-1);
02409 }
02410
02411 /*
02412     #] Shuffle :
02413     #[ FinishShuffle :
02414
02415     The complications here are:
02416     1: We want to save space. We put the output term in 'out' straight
02417        on top of what we produced thusfar. We have to copy the early
02418        piece because once the term goes back to Generator, Normalize can
02419        change it in situ
02420     2: There can be other occurrence of the function between the two
02421        that we did. For shuffles that isn't likely, but we use this
02422        routine also for the stuffles and there it can happen.
02423 */
02424
02425 int FinishShuffle(WORD *fini)
02426 {
02427     GETIDENTITY
02428     WORD *t, *t1, *oldworkpointer = AT.WorkPointer, *tcoef, ntcoef, *out;
02429     int i;
02430     SHvariables *SH = &(AN.SHvar);
02431     SH->outfun[1] = fini - SH->outfun;
02432     if ( functions[SH->outfun[0]-FUNCTION].symmetric != 0 )
02433                     SH->outfun[2] |= DIRTYSYMFLAG;
02434     out = fini; i = fini - SH->outterm; t = SH->outterm;
02435     NCOPY(fini,t,i)
02436     t = SH->stop1;
02437     t1 = t + t[1];
02438     while ( t1 < SH->stop2 ) { t = t1; t1 = t + t[1]; }
02439     t1 = SH->stop1;
02440     while ( t1 < t ) *fini++ = *t1++;
02441     t = SH->stop2;
02442     while ( t < SH->incoef ) *fini++ = *t++;
02443     tcoef = fini;
02444     ntcoef = SH->nincoef;
02445     i = ABS(ntcoef);
02446     NCOPY(fini,t,i);
02447     ntcoef = REDLENG(ntcoef);
02448     Mully(BHEAD (UWORD *)tcoef,&ntcoef,
02449         (UWORD *)(AN.SHcombi+SH->combilast+1),AN.SHcombi[SH->combilast]);
02450     ntcoef = INCLENG(ntcoef);
02451     fini = tcoef + ABS(ntcoef);
02452     if ( ( ( SH->option & 2 ) != 0 ) && ( ( SH->option & 256 ) != 0 ) ) ntcoef = -ntcoef;
02453     fini[-1] = ntcoef;
02454     i = *out = fini - out;
02455 /*
02456     Now check whether we have to do more
02457 */
02458     AT.WorkPointer = out + *out;
02459     if ( ( SH->option & 1 ) == 1 ) {
02460         if ( Generator(BHEAD out,SH->level) ) goto Finicall;
02461     }
02462     else {
02463         if ( DoShtuffle(out,SH->level,SH->thefunction,SH->option) ) goto Finicall;
02464     }
02465     AT.WorkPointer = oldworkpointer;
02466     return(0);
02467 Finicall:
02468     AT.WorkPointer = oldworkpointer;
02469     MesCall("FinishShuffle");
02470     return(-1);
02471 }
02472
02473 /*
02474     #] FinishShuffle :
02475     #[ DoStuffle :
02476
02477     Stuffling is a variation of shuffling.
```

```
02478     In the stuffling we insist that the arguments are (short) integers. nonzero.
02479     The stuffle sum is x st y = sig_(x)*sig_(y)*(abs(x)+abs(y))
02480     The way we do this is:
02481         1: count the arguments in each function: n1, n2
02482         2: take the minimum minval = min(n1,n2).
02483         3: for ( j = 0; j <= min; j++ ) take j elements in each of the lists.
02484         4: the j+1 groups of remaining arguments have to each be shuffled
02485         5: the j selected pairs have to be stuffle added.
02486     We can use many of the shuffle things.
02487     Considering the recursive nature of the generation we actually don't
02488     need to know n1, n2, minval.
02489 */
02490
02491 int DoStuffle(WORD *term, WORD level, WORD fun, WORD option)
02492 {
02493     GETIDENTITY
02494     SHvariables SHback, *SH = &(AN.SHvar);
02495     WORD *t1, *t2, *tstop, *t1stop, *t2stop, ncoef, n = fun, *to, *from;
02496     WORD *r1, *r2;
02497     int i, error;
02498     LONG k;
02499     UWORD *newcombi;
02500 #ifdef NEWCODE
02501     WORD *rr1, *rr2, i1, i2;
02502 #endif
02503     if ( n < 0 ) {
02504         if ( ( n = DolToFunction(BHEAD -n) ) == 0 ) {
02505             MLOCK(ErrorMessageLock);
02506             MesPrint("$-variable in merge statement did not evaluate to a function.");
02507             MUNLOCK(ErrorMessageLock);
02508             return(1);
02509         }
02510     }
02511     if ( AT.WorkPointer + 3*(*term) + AM.MaxTal > AT.WorkTop ) {
02512         MLOCK(ErrorMessageLock);
02513         MesWork();
02514         MUNLOCK(ErrorMessageLock);
02515         return(-1);
02516     }
02517
02518     tstop = term + *term;
02519     ncoef = tstop[-1];
02520     tstop -= ABS(ncoef);
02521     t1 = term + 1;
02522 retry1:;
02523     while ( t1 < tstop ) {
02524         if ( ( *t1 == n ) && ( t1+t1[1] < tstop ) && ( t1[1] > FUNHEAD ) ) {
02525             t2 = t1 + t1[1];
02526             if ( t2 >= tstop ) {
02527                 return(Generator(BHEAD term,level));
02528             }
02529 retry2:;
02530             while ( t2 < tstop ) {
02531                 if ( ( *t2 == n ) && ( t2[1] > FUNHEAD ) ) break;
02532                 t2 += t2[1];
02533             }
02534             if ( t2 < tstop ) break;
02535         }
02536         t1 += t1[1];
02537     }
02538     if ( t1 >= tstop ) {
02539         return(Generator(BHEAD term,level));
02540     }
02541 /*
02542     Next we have to check that the arguments are of the correct type
02543     At the same time we can count them.
02544 */
02545 #ifndef NEWCODE
02546     t1stop = t1 + t1[1];
02547     r1 = t1 + FUNHEAD;
02548     while ( r1 < t1stop ) {
02549         if ( *r1 != -SNUMBER ) break;
02550         if ( r1[1] == 0 ) break;
02551         r1 += 2;
02552     }
02553     if ( r1 < t1stop ) { t1 = t2; goto retry1; }
02554     t2stop = t2 + t2[1];
02555     r2 = t2 + FUNHEAD;
02556     while ( r2 < t2stop ) {
02557         if ( *r2 != -SNUMBER ) break;
02558         if ( r2[1] == 0 ) break;
02559         r2 += 2;
02560     }
02561     if ( r2 < t2stop ) { t2 = t2 + t2[1]; goto retry2; }
02562 #else
02563     t1stop = t1 + t1[1];
02564     r1 = t1 + FUNHEAD;
```

```
02565     while ( r1 < t1stop ) {
02566         if ( *r1 == -SNUMBER ) {
02567             if ( r1[1] == 0 ) break;
02568             r1 += 2; continue;
02569         }
02570         else if ( *r1 == -SYMBOL ) {
02571             if ( ( symbols[r1[1]].complex & VARTYPEROOTOFUNITY ) != VARTYPEROOTOFUNITY )
02572                         break;
02573             r1 += 2; continue;
02574         }
02575         if ( *r1 > 0 && *r1 == r1[ARGHEAD]+ARGHEAD ) {
02576             if ( ABS(r1[r1[0]-1]) == r1[0]-ARGHEAD-1 ) {}
02577             else if ( r1[ARGHEAD+1] == SYMBOL ) {
02578                 rr1 = r1 + ARGHEAD + 3;
02579                 i1 = rr1[-1]-2;
02580                 while ( i1 > 0 ) {
02581                     if ( ( symbols[*rr1].complex & VARTYPEROOTOFUNITY ) != VARTYPEROOTOFUNITY )
02582                         break;
02583                     i1 -= 2; rr1 += 2;
02584                 }
02585                 if ( i1 > 0 ) break;
02586             }
02587             else break;
02588             rr1 = r1+*r1-1;
02589             i1 = (ABS(*rr1)-1)/2;
02590             while ( i1 > 1 ) {
02591                 if ( rr1[-1] ) break;
02592                 i1--; rr1--;
02593             }
02594             if ( i1 > 1 || rr1[-1] != 1 ) break;
02595             r1 += *r1;
02596         }
02597         else break;
02598     }
02599     if ( r1 < t1stop ) { t1 = t2; goto retry1; }
02600     t2stop = t2 + t2[1];
02601     r2 = t2 + FUNHEAD;
02602
02603     while ( r2 < t2stop ) {
02604         if ( *r2 == -SNUMBER ) {
02605             if ( r2[1] == 0 ) break;
02606             r2 += 2; continue;
02607         }
02608         else if ( *r2 == -SYMBOL ) {
02609             if ( ( symbols[r2[1]].complex & VARTYPEROOTOFUNITY ) != VARTYPEROOTOFUNITY )
02610                         break;
02611             r2 += 2; continue;
02612         }
02613         if ( *r2 > 0 && *r2 == r2[ARGHEAD]+ARGHEAD ) {
02614             if ( ABS(r2[r2[0]-1]) == r2[0]-ARGHEAD-1 ) {}
02615             else if ( r2[ARGHEAD+1] == SYMBOL ) {
02616                 rr2 = r2 + ARGHEAD + 3;
02617                 i2 = rr2[-1]-2;
02618                 while ( i2 > 0 ) {
02619                     if ( ( symbols[*rr2].complex & VARTYPEROOTOFUNITY ) != VARTYPEROOTOFUNITY )
02620                         break;
02621                     i2 -= 2; rr2 += 2;
02622                 }
02623                 if ( i2 > 0 ) break;
02624             }
02625             else break;
02626             rr2 = r2+*r2-1;
02627             i2 = (ABS(*rr2)-1)/2;
02628             while ( i2 > 1 ) {
02629                 if ( rr2[-1] ) break;
02630                 i2--; rr2--;
02631             }
02632             if ( i2 > 1 || rr2[-1] != 1 ) break;
02633             r2 += *r2;
02634         }
02635         else break;
02636     }
02637     if ( r2 < t2stop ) { t2 = t2 + t2[1]; goto retry2; }
02638 #endif
02639 /*
02640     OK, now we got two objects that can be used.
02641 */
02642     *AN.RepPoint = 1;
02643
02644     SHback = AN.SHvar;
02645     SH->finishuf = &FinishStuffle;
02646     SH->do_uffle = &DoStuffle;
02647     SH->outterm = AT.WorkPointer;
02648     AT.WorkPointer += *term;
02649     SH->ststop1 = t1 + t1[1];
02650     SH->ststop2 = t2 + t2[1];
02651     SH->thefunction = n;
```

```
02652     SH->option = option;
02653     SH->level = level;
02654     SH->incoef = tstop;
02655     SH->nincoef = ncoef;
02656     if ( AN.SHcombi == 0 || AN.SHcombisize == 0 ) {
02657         AN.SHcombisize = 200;
02658         AN.SHcombi = (UWORD *)Malloc1(AN.SHcombisize*sizeof(UWORD),"AN.SHcombi");
02659         SH->combilast = 0;
02660         SHback.combilast = 0;
02661     }
02662     else {
02663         SH->combilast += AN.SHcombi[SH->combilast]+1;
02664         if ( SH->combilast >= AN.SHcombisize - 100 ) {
02665             newcombi = (UWORD *)Malloc1(2*AN.SHcombisize*sizeof(UWORD),"AN.SHcombi");
02666             for ( k = 0; k < AN.SHcombisize; k++ ) newcombi[k] = AN.SHcombi[k];
02667             M_free(AN.SHcombi,"AN.SHcombi");
02668             AN.SHcombi = newcombi;
02669             AN.SHcombisize *= 2;
02670         }
02671     }
02672     AN.SHcombi[SH->combilast] = 1;
02673     AN.SHcombi[SH->combilast+1] = 1;
02674
02675     i = t1-term; to = SH->outterm; from = term;
02676     NCOPY(to,from,i)
02677     SH->outfun = to;
02678     for ( i = 0; i < FUNHEAD; i++ ) { *to++ = t1[i]; }
02679
02680     error = Stuffle(t1+FUNHEAD,t2+FUNHEAD,to);
02681
02682     AT.WorkPointer = SH->outterm;
02683     AN.SHvar = SHback;
02684     if ( error ) {
02685         MesCall("DoStuffle");
02686         return(-1);
02687     }
02688     return(0);
02689 }
02690
02691 /*
02692     #] DoStuffle :
02693     #[ Stuffle :
02694
02695     The way to generate the stuffles
02696     1: select an argument in the first  list (for(j1=0;j1<last;j1++))
02697     2: select an argument in the second list (for(j2=0;j2<last;j2++))
02698     3: put values for SH->ststop1 and SH->ststop2 at these arguments.
02699     4: generate all shuffles of the arguments in front.
02700     5: Then put the stuffle sum of arg(j1) and arg(j2)
02701     6: Then continue calling Stuffle
02702     7: Once one gets exhausted, we can clean up the list and call FinishShuffle
02703     8: if ( ( SH->option & 2 ) != 0 ) the stuffle sum is negative.
02704 */
02705
02706 int Stuffle(WORD *from1, WORD *from2, WORD *to)
02707 {
02708     GETIDENTITY
02709     WORD *t, *tf, *next1, *next2, *st1, *st2, *save1, *save2;
02710     SHvariables *SH = &(AN.SHvar);
02711     int i, retval;
02712 /*
02713     First the special cases (exhausted list(s)):
02714 */
02715     save1 = SH->stop1; save2 = SH->stop2;
02716     if ( from1 >= SH->ststop1 && from2 == SH->ststop2 ) {
02717         SH->stop1 = SH->ststop1;
02718         SH->stop2 = SH->ststop2;
02719         retval = FinishShuffle(to);
02720         SH->stop1 = save1; SH->stop2 = save2;
02721         return(retval);
02722     }
02723     else if ( from1 >= SH->ststop1 ) {
02724         i = SH->ststop2 - from2; t = to; tf = from2; NCOPY(t,tf,i)
02725         SH->stop1 = SH->ststop1;
02726         SH->stop2 = SH->ststop2;
02727         retval = FinishShuffle(t);
02728         SH->stop1 = save1; SH->stop2 = save2;
02729         return(retval);
02730     }
02731     else if ( from2 >= SH->ststop2 ) {
02732         i = SH->ststop1 - from1; t = to; tf = from1; NCOPY(t,tf,i)
02733         SH->stop1 = SH->ststop1;
02734         SH->stop2 = SH->ststop2;
02735         retval = FinishShuffle(t);
02736         SH->stop1 = save1; SH->stop2 = save2;
02737         return(retval);
02738     }
```

```
02739 /*
02740     Now the case that we have no stuffle sums.
02741 */
02742     SH->stop1 = SH->ststop1;
02743     SH->stop2 = SH->ststop2;
02744     SH->finishuf = &FinishShuffle;
02745     if ( Shuffle(from1,from2,to) ) goto stuffcall;
02746     SH->finishuf = &FinishStuffle;
02747 /*
02748     Now we have to select a pair, one from 1 and one from 2.
02749 */
02750 #ifndef NEWCODE
02751     st1 = from1; next1 = st1+2;       /* <----- */
02752 #else
02753     st1 = next1 = from1;
02754     NEXTARG(next1)
02755 #endif
02756     while ( next1 <= SH->ststop1 ) {
02757 #ifndef NEWCODE
02758         st2 = from2; next2 = st2+2;       /* <----- */
02759 #else
02760         next2 = st2 = from2;
02761         NEXTARG(next2)
02762 #endif
02763         while ( next2 <= SH->ststop2 ) {
02764             SH->stop1 = st1;
02765             SH->stop2 = st2;
02766             if ( st1 == from1 && st2 == from2 ) {
02767                 t = to;
02768 #ifndef NEWCODE
02769                 *t++ = -SNUMBER; *t++ = StuffAdd(st1[1],st2[1]);
02770 #else
02771                 t = StuffRootAdd(st1,st2,t);
02772 #endif
02773                 SH->option ^= 256;
02774                 if ( Stuffle(next1,next2,t) ) goto stuffcall;
02775                 SH->option ^= 256;
02776             }
02777             else if ( st1 == from1 ) {
02778                 i = st2-from2;
02779                 t = to; tf = from2; NCOPY(t,tf,i)
02780 #ifndef NEWCODE
02781                 *t++ = -SNUMBER; *t++ = StuffAdd(st1[1],st2[1]);
02782 #else
02783                 t = StuffRootAdd(st1,st2,t);
02784 #endif
02785                 SH->option ^= 256;
02786                 if ( Stuffle(next1,next2,t) ) goto stuffcall;
02787                 SH->option ^= 256;
02788             }
02789             else if ( st2 == from2 ) {
02790                 i = st1-from1;
02791                 t = to; tf = from1; NCOPY(t,tf,i)
02792 #ifndef NEWCODE
02793                 *t++ = -SNUMBER; *t++ = StuffAdd(st1[1],st2[1]);
02794 #else
02795                 t = StuffRootAdd(st1,st2,t);
02796 #endif
02797                 SH->option ^= 256;
02798                 if ( Stuffle(next1,next2,t) ) goto stuffcall;
02799                 SH->option ^= 256;
02800             }
02801             else {
02802                 if ( Shuffle(from1,from2,to) ) goto stuffcall;
02803             }
02804 #ifndef NEWCODE
02805             st2 = next2; next2 += 2;       /* <----- */
02806 #else
02807             st2 = next2;
02808             NEXTARG(next2)
02809 #endif
02810         }
02811 #ifndef NEWCODE
02812         st1 = next1; next1 += 2;       /* <----- */
02813 #else
02814         st1 = next1;
02815         NEXTARG(next1)
02816 #endif
02817     }
02818     SH->stop1 = save1; SH->stop2 = save2;
02819     return(0);
02820 stuffcall:;
02821     MesCall("Stuffle");
02822     return(-1);
02823 }
02824
02825 /*
```

```
02826     #] Stuffle :
02827     #[ FinishStuffle :
02828
02829     The program only comes here from the Shuffle routine.
02830     It should add the stuffle sum and then call Stuffle again.
02831 */
02832
02833 int FinishStuffle(WORD *fini)
02834 {
02835     GETIDENTITY
02836     SHvariables *SH = &(AN.SHvar);
02837 #ifdef NEWCODE
02838     WORD *next1 = SH->stop1, *next2 = SH->stop2;
02839     fini = StuffRootAdd(next1,next2,fini);
02840 #else
02841     *fini++ = -SNUMBER; *fini++ = StuffAdd(SH->stop1[1],SH->stop2[1]);
02842 #endif
02843     SH->option ^= 256;
02844 #ifdef NEWCODE
02845     NEXTARG(next1)
02846     NEXTARG(next2)
02847     if ( Stuffle(next1,next2,fini) ) goto stuffcall;
02848 #else
02849     if ( Stuffle(SH->stop1+2,SH->stop2+2,fini) ) goto stuffcall;
02850 #endif
02851     SH->option ^= 256;
02852     return(0);
02853 stuffcall:;
02854     MesCall("FinishStuffle");
02855     return(-1);
02856 }
02857
02858 /*
02859     #] FinishStuffle :
02860     #[ StuffRootAdd :
02861
02862     Makes the stuffle sum of two arguments.
02863     The arguments can be of one of three types:
02864     1: -SNUMBER,num
02865     2: -SYMBOL,symbol
02866     3: Numerical (long) argument.
02867     4: Generic argument with (only) symbols that are roots of unity and
02868        a coefficient.
02869     We have excluded the case that both t1 and t2 are of type 1:
02870     The output should be written to 'to' and the new fill position should
02871     be the return value.
02872     `to' is inside the workspace.
02873
02874     The stuffle sum is sig_(t2)*t1+sig_(t1)*t2
02875     or sig_(t1)*sig_(t2)*(abs_(t1)+abs_(t2))
02876 */
02877
02878 #ifdef NEWCODE
02879
02880 WORD *StuffRootAdd(WORD *t1, WORD *t2, WORD *to)
02881 {
02882     int type1, type2, type3, sgn, sgn1, sgn2, sgn3, pow, root, nosymbols, i;
02883     WORD *tt1, *tt2, it1, it2, *t3, *r, size1, size2, size3;
02884     WORD scratch[2];
02885     LONG x;
02886     if ( *t1 == -SNUMBER ) { type1 = 1; if ( t1[1] < 0 ) sgn1 = -1; else sgn1 = 1; }
02887     else if ( *t1 == -SYMBOL ) { type1 = 2; sgn1 = 1; }
02888     else if ( ABS(t1[*t1-1]) == *t1-ARGHEAD-1 ) {
02889         type1 = 3; if ( t1[*t1-1] < 0 ) sgn1 = -1; else sgn1 = 1; }
02890     else { type1 = 4; if ( t1[*t1-1] < 0 ) sgn1 = -1; else sgn1 = 1; }
02891     if ( *t2 == -SNUMBER ) { type2 = 1; if ( t2[1] < 0 ) sgn2 = -1; else sgn2 = 1; }
02892     else if ( *t2 == -SYMBOL ) { type2 = 2; sgn2 = 1; }
02893     else if ( ABS(t2[*t2-1]) == *t2-ARGHEAD-1 ) {
02894         type2 = 3; if ( t2[*t2-1] < 0 ) sgn2 = -1; else sgn2 = 1; }
02895     else { type2 = 4; if ( t2[*t2-1] < 0 ) sgn2 = -1; else sgn2 = 1; }
02896     if ( type1 > type2 ) {
02897         t3 = t1; t1 = t2; t2 = t3;
02898         type3 = type1; type1 = type2; type2 = type3;
02899         sgn3 = sgn1; sgn1 = sgn2; sgn2 = sgn3;
02900     }
02901     nosymbols = 1; sgn3 = 1;
02902     switch ( type1 ) {
02903         case 1:
02904             if ( type2 == 1 ) {
02905                 x  = sgn2 * t1[1];
02906                 x += sgn1 * t2[1];
02907                 if ( x > MAXPOSITIVE || x < -(MAXPOSITIVE+1) ) {
02908                     if ( x < 0 ) { sgn1 = -3; x = -x; }
02909                     else sgn1 = 3;
02910                     *to++ = ARGHEAD+4;
02911                     *to++ = 0;
02912                     FILLARG(to)
```

```
02913                   *to++ = 4; *to++ = (UWORD)x; *to++ = 1; *to++ = sgn1;
02914               }
02915               else { *to++ = -SNUMBER; *to++ = (WORD)x; }
02916           }
02917           else if ( type2 == 2 ) {
02918               *to++ = ARGHEAD+8; *to++ = 0; FILLARG(to)
02919               *to++ = 8; *to++ = SYMBOL; *to++ = 4; *to++ = t2[1]; *to++ = 1;
02920               *to++ = ABS(t1[1])+1;
02921               *to++ = 1;
02922               *to++ = 3*sgn1;
02923           }
02924           else if ( type2 == 3 ) {
02925               tt1 = (WORD *)scratch; tt1[0] = ABS(t1[1]); size1 = 1;
02926               tt2 = t2+ARGHEAD+1; size2 = (ABS(t2[*t2-1])-1)/2;
02927               t3 = to;
02928               *to++ = 0; *to++ = 0; FILLARG(to) *to++ = 0;
02929               goto DoCoeffi;
02930           }
02931           else {
02932 /*
02933               t1 is (short) numeric, t2 has the symbol(s).
02934 */
02935               tt1 = (WORD *)scratch; tt1[0] = ABS(t1[1]); size1 = 1;
02936               tt2 = t2+ARGHEAD+1; tt2 += tt2[1]; size2 = (ABS(t2[*t2-1])-1)/2;
02937               t3 = to; i = tt2 - t2; r = t2;
02938               NCOPY(to,r,i)
02939               nosymbols = 0;
02940               goto DoCoeffi;
02941           }
02942       break;
02943       case 2:
02944           if ( type2 == 2 ) {
02945               if ( t1[1] == t2[1] ) {
02946                   if ( ( symbols[t1[1]].maxpower == 4 )
02947                   && ( ( symbols[t1[1]].complex & VARTYPEMINUS ) == VARTYPEMINUS ) ) {
02948                       *to++ = -SNUMBER; *to++ = -2;
02949                   }
02950                   else if ( symbols[t1[1]].maxpower == 2 ) {
02951                       *to++ = -SNUMBER; *to++ = 2;
02952                   }
02953                   else {
02954                       *to++ = ARGHEAD+8; *to++ = 0; FILLARG(to)
02955                       *to++ = 8; *to++ = SYMBOL; *to++ = 4;
02956                       *to++ = t1[1]; *to++ = 2;
02957                       *to++ = 2; *to++ = 1; *to++ = 3;
02958                   }
02959               }
02960               else {
02961                   *to++ = ARGHEAD+10; *to++ = 0; FILLARG(to)
02962                   *to++ = 10; *to++ = SYMBOL; *to++ = 6;
02963                   if ( t1[1] < t2[1] ) {
02964                       *to++ = t1[1]; *to++ = 1; *to++ = t2[1]; *to++ = 1;
02965                   }
02966                   else {
02967                       *to++ = t2[1]; *to++ = 1; *to++ = t1[1]; *to++ = 1;
02968                   }
02969                   *to++ = 2; *to++ = 1; *to++ = 3;
02970               }
02971           }
02972           else if ( type2 == 3 ) {
02973               t3 = to;
02974               *to++ = 0; *to++ = 0; FILLARG(to) *to++ = 0;
02975               *to++ = SYMBOL; *to++ = 4; *to++ = t1[1]; *to++ = 1;
02976               tt1 = scratch; tt1[1] = 1; size1 = 1;
02977               tt2 = t2+ARGHEAD+1; size2 = (ABS(t2[*t2-1])-1)/2;
02978               nosymbols = 0;
02979               goto DoCoeffi;
02980           }
02981           else {
02982               tt1 = scratch; tt1[0] = 1; size1 = 1;
02983               t3 = to;
02984               *to++ = 0; *to++ = 0; FILLARG(to) *to++ = 0;
02985               *to++ = SYMBOL; *to++ = 0;
02986               tt2 = t2 + ARGHEAD+3; it2 = tt2[-1]-2;
02987               while ( it2 > 0 ) {
02988                   if ( *tt2 == t1[1] ) {
02989                       pow = tt2[1]+1;
02990                       root = symbols[*tt2].maxpower;
02991                       if ( pow >= root ) pow -= root;
02992                       if ( ( symbols[*tt2].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
02993                           if ( ( root & 1 ) == 0 && pow >= root/2 ) {
02994                               pow -= root/2; sgn3 = -sgn3;
02995                           }
02996                       }
02997                       if ( pow != 0 ) {
02998                           *to++ = *tt2; *to++ = pow;
02999                       }
```

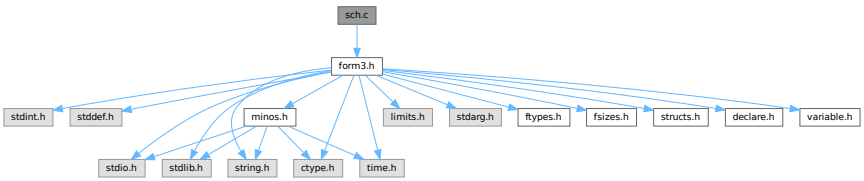
```
03000                       tt2 += 2; it2 -= 2;
03001                       break;
03002                   }
03003                   else if ( t1[1] < *tt2 ) {
03004                       *to++ = t1[1]; *to++ = 1; break;
03005                   }
03006                   else {
03007                       *to++ = *tt2++; *to++ = *tt2++; it2 -= 2;
03008                       if ( it2 <= 0 ) { *to++ = t1[1]; *to++ = 1; }
03009                   }
03010               }
03011               while ( it2 > 0 ) { *to++ = *tt2++; *to++ = *tt2++; it2 -= 2; }
03012               if ( (to - t3) > ARGHEAD+3 ) {
03013                   t3[ARGHEAD+2] = (to-t3)-ARGHEAD-1; /* size of the SYMBOL field */
03014                   nosymbols = 0;
03015               }
03016               else {
03017                   to = t3+ARGHEAD+1; /* no SYMBOL field */
03018               }
03019               size2 = (ABS(t2[*t2-1])-1)/2;
03020               goto DoCoeffi;
03021           }
03022       break;
03023       case 3:
03024           if ( type2 == 3 ) {
03025 /*
03026               Both are numeric
03027 */
03028               tt1 = t1+ARGHEAD+1; size1 = (ABS(t1[*t1-1])-1)/2;
03029               tt2 = t2+ARGHEAD+1; size2 = (ABS(t2[*t2-1])-1)/2;
03030               t3 = to;
03031               *to++ = 0; *to++ = 0; FILLARG(to) *to++ = 0;
03032               goto DoCoeffi;
03033           }
03034           else {
03035 /*
03036               t1 is (long) numeric, t2 has the symbol(s).
03037 */
03038               tt1 = t1+ARGHEAD+1; size1 = (ABS(t1[*t1-1])-1)/2;
03039               tt2 = t2+ARGHEAD+1; tt2 += tt2[1]; size2 = (ABS(t2[*t2-1])-1)/2;
03040               t3 = to; i = tt2 - t2; r = t2;
03041               NCOPY(to,r,i)
03042               nosymbols = 0;
03043               goto DoCoeffi;
03044           }
03045       break;
03046       case 4:
03047 /*
03048           Both have roots of unity
03049           1: Merge the lists and simplify if possible
03050 */
03051           tt1 = t1+ARGHEAD+3; it1 = tt1[-1]-2;
03052           tt2 = t2+ARGHEAD+3; it2 = tt2[-1]-2;
03053           t3 = to;
03054           *to++ = 0; *to++ = 0; FILLARG(to)
03055           *to++ = 0; *to++ = SYMBOL; *to++ = 0;
03056           while ( it1 > 0 && it2 > 0 ) {
03057               if ( *tt1 == *tt2 ) {
03058                   pow = tt1[1]+tt2[1];
03059                   root = symbols[*tt1].maxpower;
03060                   if ( pow >= root ) pow -= root;
03061                   if ( ( symbols[*tt1].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
03062                       if ( ( root & 1 ) == 0 && pow >= root/2 ) {
03063                           pow -= root/2; sgn3 = -sgn3;
03064                       }
03065                   }
03066                   if ( pow != 0 ) {
03067                       *to++ = *tt1; *to++ = pow;
03068                   }
03069                   tt1 += 2; tt2 += 2; it1 -= 2; it2 -= 2;
03070               }
03071               else if ( *tt1 < *tt2 ) {
03072                   *to++ = *tt1++; *to++ = *tt1++; it1 -= 2;
03073               }
03074               else {
03075                   *to++ = *tt2++; *to++ = *tt2++; it2 -= 2;
03076               }
03077           }
03078           while ( it1 > 0 ) { *to++ = *tt1++; *to++ = *tt1++; it1 -= 2; }
03079           while ( it2 > 0 ) { *to++ = *tt2++; *to++ = *tt2++; it2 -= 2; }
03080           if ( (to - t3) > ARGHEAD+3 ) {
03081               t3[ARGHEAD+2] = (to-t3)-ARGHEAD-1; /* size of the SYMBOL field */
03082               nosymbols = 0;
03083           }
03084           else {
03085               to = t3+ARGHEAD+1; /* no SYMBOL field */
03086           }
```

```
03087             size1 = (ABS(t1[*t1-1])-1)/2;
03088             size2 = (ABS(t2[*t2-1])-1)/2;
03089 /*
03090             Now tt1 and tt2 are pointing at their coefficients.
03091             sgn1 is the sign of 1, sgn2 is the sign of 2 and sgn3 is an extra
03092             overall sign.
03093 */
03094 DoCoeffi:
03095             if ( AddLong((UWORD *)tt1,size1,(UWORD *)tt2,size2,(UWORD *)to,&size3) ) {
03096 /* INTERNAL_ERROR_EXCL_START */
03097                 MLOCK(ErrorMessageLock);
03098                 MesPrint("!>Called from StuffRootAdd");
03099                 MUNLOCK(ErrorMessageLock);
03100                 Terminate(-1);
03101 /* INTERNAL_ERROR_EXCL_STOP */
03102             }
03103             sgn = sgn1*sgn2*sgn3;
03104             if ( nosymbols && size3 == 1 ) {
03105                 if ( (UWORD)(to[0]) <= MAXPOSITIVE && sgn > 0 ) {
03106                     sgn1 = to[0];
03107                     to = t3; *to++ = -SNUMBER; *to++ = sgn1;
03108                 }
03109                 else if ( (UWORD)(to[0]) <= (MAXPOSITIVE+1) && sgn < 0 ) {
03110                     sgn1 = to[0];
03111                     to = t3; *to++ = -SNUMBER; *to++ = -sgn1;
03112                 }
03113                 else goto genericcoef;
03114             }
03115             else {
03116 genericcoef:
03117                 to += size3;
03118                 sgn = sgn*(2*size3+1);
03119                 *to++ = 1;
03120                 while ( size3 > 1 ) { *to++ = 0; size3--; }
03121                 *to++ = sgn;
03122                 t3[0] = to - t3;
03123                 t3[ARGHEAD] = t3[0] - ARGHEAD;
03124             }
03125         break;
03126     }
03127     return(to);
03128 }
03129
03130 #endif
03131
03132 /*
03133     #] StuffRootAdd :
03134 */
```

4.125 sch.c File Reference

`#include "form3.h"`

Include dependency graph for sch.c:



Functions

- UBYTE * StrCopy (UBYTE *from, UBYTE *to)
- void AddToLine (UBYTE *s)
- void FiniLine (void)
- void IniLine (WORD extrablank)

- void LongToLine (UWORD *a, WORD na)
- void RatToLine (UWORD *a, WORD na)
- void TalToLine (UWORD x)
- void TokenToLine (UBYTE *s)
- UBYTE * CodeToLine (WORD number, UBYTE *Out)
- void MultiplyToLine (void)
- UBYTE * AddArrayIndex (WORD num, UBYTE *out)
- void PrtTerms (void)
- UBYTE * WrtPower (UBYTE *Out, WORD Power)
- void PrintTime (UBYTE *mess)
- void WriteLists (void)
- void WriteDictionary (DICTIONARY *dict)
- void WriteArgument (WORD *t)
- int WriteSubTerm (WORD *sterm, WORD first)
- int WriteInnerTerm (WORD *term, WORD first)
- int WriteTerm (WORD *term, WORD *lbrac, WORD first, WORD prtf, WORD br)
- int WriteExpression (WORD *terms, LONG ltot)
- int WriteAll (void)
- int WriteOne (UBYTE *name, int alreadyinline, int nosemi, WORD plus)

4.125.1 Detailed Description

Contains the functions that deal with the writing of expressions/terms in a textual representation. (Dutch schrijven = to write)

Definition in file sch.c.

4.125.2 Function Documentation

4.125.2.1 StrCopy()

```
UBYTE * StrCopy (
            UBYTE * from,
            UBYTE * to )
```

Definition at line 49 of file sch.c.

4.125.2.2 AddToLine()

```
void AddToLine (
            UBYTE * s )
```

Definition at line 64 of file sch.c.

4.125.2.3 FiniLine()

```
void FiniLine (
            void  )
```

Definition at line 164 of file sch.c.

4.125.2.4 IniLine()

```
void IniLine (
             WORD extrablank )
```

Definition at line 249 of file sch.c.

4.125.2.5 LongToLine()

```
void LongToLine (
             UWORD * a,
             WORD na )
```

Definition at line 285 of file sch.c.

4.125.2.6 RatToLine()

```
void RatToLine (
             UWORD * a,
             WORD na )
```

Definition at line 319 of file sch.c.

4.125.2.7 TalToLine()

```
void TalToLine (
             UWORD x )
```

Definition at line 504 of file sch.c.

4.125.2.8 TokenToLine()

```
void TokenToLine (
             UBYTE * s )
```

Definition at line 533 of file sch.c.

4.125.2.9 CodeToLine()

```
UBYTE * CodeToLine (
             WORD number,
             UBYTE * Out )
```

Definition at line 640 of file sch.c.

4.125.2.10 MultiplyToLine()

```
void MultiplyToLine (
            void  )
```

Definition at line 653 of file sch.c.

4.125.2.11 AddArrayIndex()

```
UBYTE * AddArrayIndex (
            WORD num,
            UBYTE * out )
```

Definition at line 678 of file sch.c.

4.125.2.12 PrtTerms()

```
void PrtTerms (
            void  )
```

Definition at line 698 of file sch.c.

4.125.2.13 WrtPower()

```
UBYTE * WrtPower (
            UBYTE * Out,
            WORD Power )
```

Definition at line 720 of file sch.c.

4.125.2.14 PrintTime()

```
void PrintTime (
            UBYTE * mess )
```

Definition at line 766 of file sch.c.

4.125.2.15 WriteLists()

```
void WriteLists (
            void  )
```

Definition at line 792 of file sch.c.

4.125.2.16 WriteDictionary()

```
void WriteDictionary (
            DICTIONARY * dict )
```

Definition at line 1289 of file sch.c.

4.125.2.17 WriteArgument()

```
void WriteArgument (
             WORD * t )
```

Definition at line 1406 of file sch.c.

4.125.2.18 WriteSubTerm()

```
int WriteSubTerm (
             WORD * sterm,
             WORD first )
```

Definition at line 1526 of file sch.c.

4.125.2.19 WriteInnerTerm()

```
int WriteInnerTerm (
             WORD * term,
             WORD first )
```

Definition at line 1937 of file sch.c.

4.125.2.20 WriteTerm()

```
int WriteTerm (
             WORD * term,
             WORD * lbrac,
             WORD first,
             WORD prtf,
             WORD br )
```

Definition at line 2137 of file sch.c.

4.125.2.21 WriteExpression()

```
int WriteExpression (
             WORD * terms,
             LONG ltot )
```

Definition at line 2459 of file sch.c.

4.125.2.22 WriteAll()

```
int WriteAll (
             void  )
```

Definition at line 2490 of file sch.c.

4.125.2.23 WriteOne()

```
int WriteOne (
              UBYTE * name,
              int alreadyinline,
              int nosemi,
              WORD plus )
```

Definition at line 2716 of file sch.c.

4.126 sch.c

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032 /*
00033     #[ Includes : sch.c
00034 */
00035
00036 #include "form3.h"
00037
00038 static int startinline = 0;
00039 static char fcontchar = '&';
00040 static int noextralinefeed = 0;
00041 static int lowestlevel = 1;
00042
00043 /*
00044     #] Includes :
00045     #[ schryf-Utilities :
00046         #[ StrCopy :            UBYTE *StrCopy(from,to)
00047 */
00048
00049 UBYTE *StrCopy(UBYTE *from, UBYTE *to)
00050 {
00051     while( ( *to++ = *from++ ) != 0 );
00052     return(to-1);
00053 }
00054
00055 /*
00056         #] StrCopy :
00057         #[ AddToLine :          void AddToLine(s)
00058
00059     Puts the characters of s in the outputline. If the line becomes
00060     filled it is written.
00061
00062 */
00063
00064 void AddToLine(UBYTE *s)
00065 {
00066     UBYTE *Out;
00067     LONG num;
```

```
00068     int i;
00069     if ( AO.OutInBuffer ) { AddToDollarBuffer(s); return; }
00070     Out = AO.OutFill;
00071     while ( *s ) {
00072         if ( Out >= AO.OutStop ) {
00073             if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90 ) {
00074                 *Out++ = fcontchar;
00075             }
00076 #ifdef WITHRETURN
00077             *Out++ = CARRIAGERETURN;
00078 #endif
00079             *Out++ = LINEFEED;
00080             AO.FortFirst = 0;
00081             num = Out - AO.OutputLine;
00082
00083             if ( AC.LogHandle >= 0 ) {
00084                 if ( WriteFile(AC.LogHandle,AO.OutputLine+startinline
00085                                 ,num-startinline) != (num-startinline) ) {
00086 /*
00087                     We cannot write to an otherwise open log file.
00088                     The disk could be full of course.
00089 */
00090 #ifdef DEBUGGER
00091                     if ( BUG.logfileflag == 0 ) {
00092                         fprintf(stderr,"Panic: Cannot write to log file! Disk full?\n");
00093                         BUG.logfileflag = 1;
00094                     }
00095                     BUG.eflag = 1; BUG.printflag = 1;
00096 #else
00097                     Terminate(-1);
00098 #endif
00099                 }
00100             }
00101
00102             if ( ( AO.PrintType & PRINTLFILE ) == 0 ) {
00103 #ifdef WITHRETURN
00104                 if ( num > 1 && AO.OutputLine[num-2] == CARRIAGERETURN ) {
00105                     AO.OutputLine[num-2] = LINEFEED;
00106                     num--;
00107                 }
00108 #endif
00109                 if ( WriteFile(AM.StdOut,AO.OutputLine+startinline
00110                         ,num-startinline) != (num-startinline) ) {
00111 #ifdef DEBUGGER
00112                     if ( BUG.stdoutflag == 0 ) {
00113                         fprintf(stderr,"Panic: Cannot write to standard output!\n");
00114                         BUG.stdoutflag = 1;
00115                     }
00116                     BUG.eflag = 1; BUG.printflag = 1;
00117 #else
00118                     Terminate(-1);
00119 #endif
00120                 }
00121             }
00122             /* thomasr 23/04/09: A continuation line has been started.
00123              * In Fortran90 we do not want a space after the initial
00124              * '&' character otherwise we might end up with something
00125              * like:
00126              *    ...  2.&
00127              *  & 0 ...
00128              */
00129             startinline = 0;
00130             for ( i = 0; i < AO.OutSkip; i++ ) AO.OutputLine[i] = ' ';
00131             Out = AO.OutputLine + AO.OutSkip;
00132             if ( ( AC.OutputMode == FORTRANMODE
00133              || AC.OutputMode == PFORTRANMODE ) && AO.OutSkip == 7 ) {
00134                 /* thomasr 23/04/09: fix leading blank in fortran90 mode */
00135                 if(AC.IsFortran90 == ISFORTRAN90) {
00136                     Out[-1] = fcontchar;
00137                 }
00138                 else {
00139                     Out[-2] = fcontchar;
00140                     Out[-1] = ' ';
00141                 }
00142             }
00143             if ( AO.IsBracket ) { *Out++ = ' ';
00144                 if ( AC.OutputSpaces == NORMALFORMAT ) {
00145                      *Out++ = ' '; *Out++ = ' '; }
00146             }
00147             *Out = '\0';
00148             if ( AC.OutputMode == FORTRANMODE
00149              || ( AC.OutputMode == CMODE && AO.FactorMode == 0 )
00150              || AC.OutputMode == PFORTRANMODE )
00151                 AO.InFbrack++;
00152         }
00153         *Out++ = *s++;
00154     }
```

```
00155     *Out = '\0';
00156     AO.OutFill = Out;
00157 }
00158
00159 /*
00160         #] AddToLine :
00161         #[ FiniLine :           void FiniLine()
00162 */
00163
00164 void FiniLine(void)
00165 {
00166     UBYTE *Out;
00167     WORD i;
00168     LONG num;
00169     if ( AO.OutInBuffer ) return;
00170     Out = AO.OutFill;
00171     while ( Out > AO.OutputLine ) {
00172         if ( Out[-1] == ' ' ) Out--;
00173         else break;
00174     }
00175     i = (WORD)(Out-AO.OutputLine);
00176     if ( noextralinefeed == 0 ) {
00177         if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90
00178             && Out > AO.OutputLine ) {
00179 /*
00180             *Out++ = fcontchar;
00181 */
00182         }
00183 #ifdef WITHRETURN
00184         *Out++ = CARRIAGERETURN;
00185 #endif
00186         *Out++ = LINEFEED;
00187         AO.FortFirst = 0;
00188     }
00189     num = Out - AO.OutputLine;
00190
00191     if ( AC.LogHandle >= 0 ) {
00192         if ( WriteFile(AC.LogHandle,AO.OutputLine+startinline
00193                 ,num-startinline) != (num-startinline) ) {
00194 #ifdef DEBUGGER
00195             if ( BUG.logfileflag == 0 ) {
00196                 fprintf(stderr,"Panic: Cannot write to log file! Disk full?\n");
00197                 BUG.logfileflag = 1;
00198             }
00199             BUG.eflag = 1; BUG.printflag = 1;
00200 #else
00201             Terminate(-1);
00202 #endif
00203         }
00204     }
00205
00206     if ( ( AO.PrintType & PRINTLFILE ) == 0 ) {
00207 #ifdef WITHRETURN
00208         if ( num > 1 && AO.OutputLine[num-2] == CARRIAGERETURN ) {
00209             AO.OutputLine[num-2] = LINEFEED;
00210             num--;
00211         }
00212 #endif
00213         if ( WriteFile(AM.StdOut,AO.OutputLine+startinline,
00214                 num-startinline) != (num-startinline) ) {
00215 #ifdef DEBUGGER
00216             if ( BUG.stdoutflag == 0 ) {
00217                 fprintf(stderr,"Panic: Cannot write to standard output!\n");
00218                 BUG.stdoutflag = 1;
00219             }
00220             BUG.eflag = 1; BUG.printflag = 1;
00221 #else
00222             Terminate(-1);
00223 #endif
00224         }
00225     }
00226     startinline = 0;
00227     if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
00228          || ( AC.OutputMode == CMODE && AO.FactorMode == 0 ) ) AO.InFbrack++;
00229     Out = AO.OutputLine;
00230     AO.OutStop = Out + AC.LineLength;
00231     i = AO.OutSkip;
00232     while ( --i >= 0 ) *Out++ = ' ';
00233     if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
00234      && AO.OutSkip == 7 ) {
00235         Out[-2] = fcontchar;
00236         Out[-1] = ' ';
00237     }
00238     AO.OutFill = Out;
00239 }
00240
00241 /*
```

```
00242         #] FiniLine :
00243         #[ IniLine :            void IniLine(extrablank)
00244
00245     Initializes the output line for the type of output
00246
00247 */
00248
00249 void IniLine(WORD extrablank)
00250 {
00251     UBYTE *Out;
00252     Out = AO.OutputLine;
00253     AO.OutStop = Out + AC.LineLength;
00254     *Out++ = ' ';
00255     *Out++ = ' ';
00256     *Out++ = ' ';
00257     *Out++ = ' ';
00258     *Out++ = ' ';
00259     if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {
00260         *Out++ = fcontchar;
00261         AO.OutSkip = 7;
00262     }
00263     else
00264         AO.OutSkip = 6;
00265     *Out++ = ' ';
00266     while ( extrablank > 0 ) {
00267         *Out++ = ' ';
00268         extrablank--;
00269     }
00270     AO.OutFill = Out;
00271 }
00272
00273 /*
00274         #] IniLine :
00275         #[ LongToLine :         void LongToLine(a,na)
00276
00277     Puts a Long integer in the output line. If it is only a single
00278     word long it is put in the line as a single token.
00279     The sign of a is ignored.
00280
00281 */
00282
00283 static UBYTE *LLscratch = 0;
00284
00285 void LongToLine(UWORD *a, WORD na)
00286 {
00287     UBYTE *OutScratch;
00288     if ( LLscratch == 0 ) {
00289         LLscratch = (UBYTE *)Malloc1(4*(AM.MaxTal*sizeof(WORD)+2)*sizeof(UBYTE),"LongToLine");
00290     }
00291     OutScratch = LLscratch;
00292     if ( na < 0 ) na = -na;
00293     if ( na > 1 ) {
00294         PrtLong(a,na,OutScratch);
00295         if ( AO.NoSpacesInNumbers || AC.OutputMode == REDUCEMODE ) {
00296             AO.BlockSpaces = 1;
00297             TokenToLine(OutScratch);
00298             AO.BlockSpaces = 0;
00299         }
00300         else {
00301             TokenToLine(OutScratch);
00302         }
00303     }
00304     else if ( !na ) TokenToLine((UBYTE *)"0");
00305     else TalToLine(*a);
00306 }
00307
00308 /*
00309         #] LongToLine :
00310         #[ RatToLine :          void RatToLine(a,na)
00311
00312     Puts a rational number in the output line. The sign is ignored.
00313
00314 */
00315
00316 static UBYTE *RLscratch = 0;
00317 static UWORD *RLscratE = 0;
00318
00319 void RatToLine(UWORD *a, WORD na)
00320 {
00321     GETIDENTITY
00322     WORD adenom, anumer;
00323     ULONG maxInt;
00324
00325     if ( AC.OutputMode == CMODE ) {
00326         // In C, integer literals over 2^32-1 are automatically promoted to longer types up to
00327         // unsigned long long, however FORM has always given integers over 2^32-1 a floating suffix
00328         // in C mode. Retain that behaviour here for now. In the future, maxInt could be a user-
```

```
00329         // configurable parameter.
00330         maxInt = 4294967295;
00331     }
00332     else {
00333         // In Fortran modes, literals over 2^31-1 should be printed with a float suffix
00334         maxInt = 2147483647;
00335     }
00336
00337     if ( na < 0 ) na = -na;
00338
00339     if ( AC.OutNumberType == RATIONALMODE ) {
00340
00341         // Work out what is to be printed:
00342         WORD isOne = 0, isOneOver = 0, isIntegral = 0;
00343         WORD isLongNum = 0, isLongDen = 0;
00344         UnPack(a,na,&adenom,&anumer);
00345         if ( na == 1 && a[0] == 1 && a[1] == 1 ) { isOne = 1; isIntegral = 1; }
00346         else if ( adenom == 1 && a[na] == 1 ) { isIntegral = 1; }
00347         else if ( anumer == 1 && a[0] == 1 ) { isOneOver = 1; }
00348         if ( anumer > 1 || ( anumer == 1 && a[0] > maxInt ) ) { isLongNum = 1; };
00349         if ( adenom > 1 || ( adenom == 1 && a[na] > maxInt ) ) { isLongDen = 1; };
00350
00351         // Now sort out the float suffix for the numerator:
00352         UBYTE* suffNum = (UBYTE*)"";
00353         UBYTE* suffDen = (UBYTE*)"";
00354         if ( isLongNum || !isIntegral || AC.Fortran90Kind || ( AO.DoubleFlag & 4 ) == 4 ) {
00355             if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90 ) {
00356                 if ( AC.Fortran90Kind ) { suffNum = AC.Fortran90Kind; }
00357                 else { suffNum = (UBYTE*)"."; }
00358             }
00359             else if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == CMODE ) {
00360                 if ( ( AO.DoubleFlag & 2 ) == 2 ) { suffNum = (UBYTE*)".Q0"; }
00361                 else if ( ( AO.DoubleFlag & 1 ) == 1 ) { suffNum = (UBYTE*)".D0"; }
00362                 else { suffNum = (UBYTE*)"."; }
00363             }
00364         }
00365         // The same again, for the denominator:
00366         if ( isLongDen || !isIntegral || AC.Fortran90Kind || ( AO.DoubleFlag & 4 ) == 4 ) {
00367             if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90 ) {
00368                 if ( AC.Fortran90Kind ) { suffDen = AC.Fortran90Kind; }
00369                 else { suffDen = (UBYTE*)"."; }
00370             }
00371             else if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == CMODE ) {
00372                 if ( ( AO.DoubleFlag & 2 ) == 2 ) { suffDen = (UBYTE*)".Q0"; }
00373                 else if ( ( AO.DoubleFlag & 1 ) == 1 ) { suffDen = (UBYTE*)".D0"; }
00374                 else { suffDen = (UBYTE*)"."; }
00375             }
00376         }
00377         // In PFORTRAN mode, rationals don't get a suffix if the integers are not long
00378         // Also the suffix is at least ".D0" (and never ".").
00379         if ( AC.OutputMode == PFORTRANMODE ) {
00380             if ( isLongNum ) {
00381                 if ( ( AO.DoubleFlag & 2 ) == 2 ) { suffNum = (UBYTE*)".Q0"; }
00382                 else { suffNum = (UBYTE*)".D0"; }
00383             }
00384             if ( isLongDen ) {
00385                 if ( ( AO.DoubleFlag & 2 ) == 2 ) { suffDen = (UBYTE*)".Q0"; }
00386                 else { suffDen = (UBYTE*)".D0"; }
00387             }
00388         }
00389
00390         // Finally, print the number:
00391         // In PFORTRAN we use
00392         //    one     if denom = numerator = 1
00393         //    integer     if denom = 1
00394         //    (one/integer) if numerator = 1
00395         //    ((one*integer)/integer) in the general case
00396         if ( AC.OutputMode == PFORTRANMODE ) {
00397             if ( isOne ) {
00398                 AddToLine((UBYTE *)"one");
00399             }
00400             else if ( isOneOver ) {
00401                 AddToLine((UBYTE *)"(one/");
00402                 LongToLine(a+na,adenom);
00403                 AddToLine(suffDen);
00404                 AddToLine((UBYTE*)")");
00405             }
00406             else if ( isIntegral ) {
00407                 LongToLine(a,anumer);
00408                 AddToLine(suffNum);
00409             }
00410             else {
00411                 // rational
00412                 AddToLine((UBYTE *)"((one*");
00413                 LongToLine(a,anumer);
00414                 AddToLine(suffNum);
00415                 AddToLine((UBYTE*)")/");
```

```
00416                   LongToLine(a+na,adenom);
00417                   AddToLine(suffDen);
00418                   AddToLine((UBYTE*)")");
00419               }
00420           }
00421           // All other modes use the same printing code:
00422           else {
00423               // Numerator:
00424               LongToLine(a,anumer);
00425               AddToLine(suffNum);
00426               // Denominator, if it is not 1:
00427               if (!isIntegral) {
00428                   AddToLine((UBYTE*)"/");
00429                   LongToLine(a+na,adenom);
00430                   AddToLine(suffDen);
00431               }
00432           }
00433       }
00434
00435       else {
00436   /*
00437           This is the float mode
00438   */
00439           UBYTE *OutScratch;
00440           WORD exponent = 0, i, ndig, newl;
00441           UWORD *c, *den, b = 10, dig[10];
00442           UBYTE *o, *out, cc;
00443   /*
00444           First we have to adjust the numerator and denominator
00445   */
00446           if ( RLscratch == 0 ) {
00447               RLscratch = (UBYTE *)Malloc1(4*(AM.MaxTal+2)*sizeof(UBYTE),"RatToLine");
00448               RLscratE = (UWORD *)Malloc1(2*(AM.MaxTal+2)*sizeof(UWORD),"RatToLine");
00449           }
00450           out = OutScratch = RLscratch;
00451           c = RLscratE; for ( i = 0; i < 2*na; i++ ) c[i] = a[i];
00452           UnPack(c,na,&adenom,&anumer);
00453           while ( BigLong(c,anumer,c+na,adenom) >= 0 ) {
00454               Divvy(BHEAD c,&na,&b,1);
00455               UnPack(c,na,&adenom,&anumer);
00456               exponent++;
00457           }
00458           while ( BigLong(c,anumer,c+na,adenom) < 0 ) {
00459               Mully(BHEAD c,&na,&b,1);
00460               UnPack(c,na,&adenom,&anumer);
00461               exponent--;
00462           }
00463   /*
00464           Now division will give a number between 1 and 9
00465   */
00466           den = c + na; i = 1;
00467           DivLong(c,anumer,den,adenom,dig,&ndig,c,&newl);
00468           *out++ = (UBYTE)(dig[0]+'0'); *out++ = '.';
00469           while ( newl && i < AC.OutNumberType ) {
00470               Pack(c,&newl,den,adenom);
00471               Mully(BHEAD c,&newl,&b,1);
00472               na = newl;
00473               UnPack(c,na,&adenom,&anumer);
00474               den = c + na;
00475               DivLong(c,anumer,den,adenom,dig,&ndig,c,&newl);
00476               if ( ndig == 0 ) *out++ = '0';
00477               else *out++ = (UBYTE)(dig[0]+'0');
00478               i++;
00479           }
00480           *out++ = 'E';
00481           if ( exponent < 0 ) { exponent = -exponent; *out++ = '-'; }
00482           else { *out++ = '+'; }
00483           o = out;
00484           do {
00485               *out++ = (UBYTE)((exponent % 10)+'0');
00486               exponent /= 10;
00487           } while ( exponent );
00488           *out = 0; out--;
00489           while ( o < out ) { cc = *o; *o = *out; *out = cc; o++; out--; }
00490           TokenToLine(OutScratch);
00491       }
00492   }
00493
00494   /*
00495           #] RatToLine :
00496           #[ TalToLine :          void TalToLine(x)
00497
00498       Writes the unsigned number x to the output as a single token.
00499       Par indicates the number of leading blanks in the line.
00500       This parameter is needed here for the WriteLists routine.
00501
00502   */
```

```
00503
00504 void TalToLine(UWORD x)
00505 {
00506     UBYTE t[BITSINWORD/3+1];
00507     UBYTE *s;
00508     WORD i = 0, j;
00509     s = t;
00510     do { *s++ = (UBYTE)((x % 10)+'0'); i++; } while ( ( x /= 10 ) != 0 );
00511     *s-- = '\0';
00512     j = ( i - 1 ) » 1;
00513     while ( j >= 0 ) {
00514         i = t[j]; t[j] = s[-j]; s[-j] = (UBYTE)i; j--;
00515     }
00516     TokenToLine(t);
00517 }
00518
00519 /*
00520         #] TalToLine :
00521         #[ TokenToLine :        void TokenToLine(s)
00522
00523     Puts s in the output buffer. If it doesn't fit the buffer is
00524     flushed first. This routine keeps tokens as one unit.
00525     Par indicates the number of leading blanks in the line.
00526     This parameter is needed here for the WriteLists routine.
00527
00528     Remark (27-oct-2007): i and j must be longer than WORD!
00529     It can happen that a number is so long that it has more than 2^15 or 2^31
00530     digits!
00531 */
00532
00533 void TokenToLine(UBYTE *s)
00534 {
00535     UBYTE *t, *Out;
00536     LONG num, i = 0, j;
00537     if ( AO.OutInBuffer ) { AddToDollarBuffer(s); return; }
00538     t = s; Out = AO.OutFill;
00539     while ( *t++ ) i++;
00540     while ( i > 0 ) {
00541         if ( ( Out + i ) >= AO.OutStop && ( ( i < ((AC.LineLength-AO.OutSkip)»1) )
00542         || ( (AO.OutStop-Out) < (i»2) ) ) ) {
00543             if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90 ) {
00544                 *Out++ = fcontchar;
00545             }
00546 #ifdef WITHRETURN
00547             *Out++ = CARRIAGERETURN;
00548 #endif
00549             *Out++ = LINEFEED;
00550             AO.FortFirst = 0;
00551             num = Out - AO.OutputLine;
00552             if ( AC.LogHandle >= 0 ) {
00553                 if ( WriteFile(AC.LogHandle,AO.OutputLine+startinline,
00554                     num-startinline) != (num-startinline) ) {
00555 #ifdef DEBUGGER
00556                     if ( BUG.logfileflag == 0 ) {
00557                         fprintf(stderr,"Panic: Cannot write to log file! Disk full?\n");
00558                         BUG.logfileflag = 1;
00559                     }
00560                     BUG.eflag = 1; BUG.printflag = 1;
00561 #else
00562                     Terminate(-1);
00563 #endif
00564                 }
00565             }
00566             if ( ( AO.PrintType & PRINTLFILE ) == 0 ) {
00567 #ifdef WITHRETURN
00568                 if ( num > 1 && AO.OutputLine[num-2] == CARRIAGERETURN ) {
00569                     AO.OutputLine[num-2] = LINEFEED;
00570                     num--;
00571                 }
00572 #endif
00573                 if ( WriteFile(AM.StdOut,AO.OutputLine+startinline,
00574                     num-startinline) != (num-startinline) ) {
00575 #ifdef DEBUGGER
00576                     if ( BUG.stdoutflag == 0 ) {
00577                         fprintf(stderr,"Panic: Cannot write to standard output!\n");
00578                         BUG.stdoutflag = 1;
00579                     }
00580                     BUG.eflag = 1; BUG.printflag = 1;
00581 #else
00582                     Terminate(-1);
00583 #endif
00584                 }
00585             }
00586             startinline = 0;
00587             Out = AO.OutputLine;
00588             if ( AO.BlockSpaces == 0 ) {
00589                 for ( j = 0; j < AO.OutSkip; j++ ) { *Out++ = ' '; }
```

```
00590                 if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) ) {
00591                   if ( AO.OutSkip == 7 ) {
00592                     Out[-2] = fcontchar;
00593                     Out[-1] = ' ';
00594                   }
00595                 }
00596             }
00597 /*
00598             Out = AO.OutputLine + AO.OutSkip;
00599             if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
00600              && AO.OutSkip == 7 ) {
00601                 Out[-2] = fcontchar;
00602                 Out[-1] = ' ';
00603             }
00604             else {
00605                 for ( j = 0; j < AO.OutSkip; j++ ) { AO.OutputLine[j] = ' '; }
00606             }
00607 */
00608             if ( AO.IsBracket ) { *Out++ = ' '; *Out++ = ' '; *Out++ = ' '; }
00609             *Out = '\0';
00610             if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
00611              || ( AC.OutputMode == CMODE && AO.FactorMode == 0 ) ) AO.InFbrack++;
00612         }
00613         if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {
00614                 /* Very long numbers */
00615             if ( i > (WORD)(AO.OutStop-Out) ) j = (WORD)(AO.OutStop - Out);
00616             else                              j = i;
00617             i -= j;
00618             NCOPYB(Out,s,j);
00619         }
00620         else {
00621             if ( i > (WORD)(AO.OutStop-Out) ) j = (WORD)(AO.OutStop - Out - 1);
00622             else                              j = i;
00623             i -= j;
00624             NCOPYB(Out,s,j);
00625             if ( i > 0 ) *Out++ = '\\';
00626         }
00627     }
00628     *Out = '\0';
00629     AO.OutFill = Out;
00630 }
00631
00632 /*
00633         #] TokenToLine :
00634         #[ CodeToLine :         void CodeToLine(name,number,mode)
00635
00636     Writes a name and possibly its number to output as a single token.
00637
00638 */
00639
00640 UBYTE *CodeToLine(WORD number, UBYTE *Out)
00641 {
00642     Out = StrCopy((UBYTE *)"(",Out);
00643     Out = NumCopy(number,Out);
00644     Out = StrCopy((UBYTE *)")",Out);
00645     return(Out);
00646 }
00647
00648 /*
00649         #] CodeToLine :
00650         #[ MultiplyToLine :
00651 */
00652
00653 void MultiplyToLine(void)
00654 {
00655     int i;
00656     if ( AO.CurrentDictionary > 0 && AO.CurDictSpecials > 0
00657      && AO.CurDictSpecials == DICT_DOSPECIALS ) {
00658         DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];
00659 /*
00660         Find the star:
00661 */
00662         for ( i = 0; i < dict->numelements; i++ ) {
00663             if ( dict->elements[i]->type != DICT_SPECIALCHARACTER ) continue;
00664             if ( (UBYTE)dict->elements[i]->lhs[0] == (UBYTE)('*') ) {
00665                 TokenToLine((UBYTE *)(dict->elements[i]->rhs));
00666                 return;
00667             }
00668         }
00669     }
00670     TokenToLine((UBYTE *)"*");
00671 }
00672
00673 /*
00674         #] MultiplyToLine :
00675         #[ AddArrayIndex :
00676 */
```

```
00677
00678 UBYTE *AddArrayIndex(WORD num,UBYTE *out)
00679 {
00680     if ( AC.OutputMode == CMODE ) {
00681         out = StrCopy((UBYTE *)"[",out);
00682         out = NumCopy(num,out);
00683         out = StrCopy((UBYTE *)"]",out);
00684     }
00685     else {
00686         out = StrCopy((UBYTE *)"(",out);
00687         out = NumCopy(num,out);
00688         out = StrCopy((UBYTE *)")",out);
00689     }
00690     return(out);
00691 }
00692
00693 /*
00694         #] AddArrayIndex :
00695         #[ PrtTerms :           void PrtTerms()
00696 */
00697
00698 void PrtTerms(void)
00699 {
00700     UWORD a[2];
00701     WORD na;
00702     a[0] = (UWORD)AO.NumInBrack;
00703     a[1] = (UWORD)(AO.NumInBrack » BITSINWORD);
00704     if ( a[1] ) na = 2;
00705     else na = 1;
00706     TokenToLine((UBYTE *)" ");
00707     LongToLine(a,na);
00708     if ( a[0] == 1 && na == 1 ) {
00709         TokenToLine((UBYTE *)" term");
00710     }
00711     else TokenToLine((UBYTE *)" terms");
00712     AO.NumInBrack = 0;
00713 }
00714
00715 /*
00716         #] PrtTerms :
00717         #[ WrtPower :
00718 */
00719
00720 UBYTE *WrtPower(UBYTE *Out, WORD Power)
00721 {
00722     if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
00723          || AC.OutputMode == REDUCEMODE ) {
00724         *Out++ = '*'; *Out++ = '*';
00725     }
00726     else if ( AC.OutputMode == CMODE ) *Out++ = ',';
00727     else {
00728         UBYTE *Out1 = IsExponentSign();
00729         if ( Out1 == 0 ) *Out++ = '^';
00730         else {
00731             while ( *Out1 ) *Out++ = *Out1++;
00732             *Out = 0;
00733         }
00734     }
00735     if ( Power >= 0 ) {
00736         if ( Power < 2*MAXPOWER )
00737             Out = NumCopy(Power,Out);
00738         else
00739             Out = StrCopy(FindSymbol((WORD)((LONG)Power-2*MAXPOWER)),Out);
00740 /*          Out = StrCopy(VARNAME(symbols,(LONG)Power-2*MAXPOWER),Out); */
00741         if ( AC.OutputMode == CMODE ) *Out++ = ')';
00742         *Out = 0;
00743     }
00744     else {
00745         if ( ( AC.OutputMode >= FORTRANMODE || AC.OutputMode >= PFORTRANMODE
00746          || AC.OutputMode >= REDUCEMODE ) && AC.OutputMode != CMODE )
00747             *Out++ = '(';
00748         *Out++ = '-';
00749         if ( Power > -2*MAXPOWER )
00750             Out = NumCopy(-Power,Out);
00751         else
00752             Out = StrCopy(FindSymbol((WORD)((LONG)Power-2*MAXPOWER)),Out);
00753 /*          Out = StrCopy(VARNAME(symbols,(LONG)(-Power)-2*MAXPOWER),Out); */
00754         if ( AC.OutputMode >= FORTRANMODE || AC.OutputMode >= PFORTRANMODE
00755          || AC.OutputMode >= REDUCEMODE) *Out++ = ')';
00756         *Out = 0;
00757     }
00758     return(Out);
00759 }
00760
00761 /*
00762         #] WrtPower :
00763         #[ PrintTime :
```

```
00764 */
00765
00766 void PrintTime(UBYTE *mess)
00767 {
00768     LONG millitime = TimeCPU(1);
00769     WORD timepart = (WORD)(millitime%1000);
00770     millitime /= 1000;
00771     timepart /= 10;
00772     MesPrint("At %s: Time = %7l.%2i sec",mess,millitime,timepart);
00773 }
00774
00775 /*
00776         #] PrintTime :
00777     #] schryf-Utilities :
00778     #[ schryf-Writes :
00779         #[ WriteLists :          void WriteLists()
00780
00781     Writes the namelists. If mode > 0 also the internal codes are given.
00782
00783 */
00784
00785 static UBYTE *symname[] = {
00786       (UBYTE *)"(cyclic)",(UBYTE *)"(reversecyclic)"
00787     ,(UBYTE *)"(symmetric)",(UBYTE *)"(antisymmetric)" };
00788 static UBYTE *rsymname[] = {
00789       (UBYTE *)"(-cyclic)",(UBYTE *)"(-reversecyclic)"
00790     ,(UBYTE *)"(-symmetric)",(UBYTE *)"(-antisymmetric)" };
00791
00792 void WriteLists(void)
00793 {
00794     GETIDENTITY
00795     WORD i, j, k, *skip;
00796     int first, startvalue;
00797     UBYTE *OutScr, *Out;
00798     EXPRESSIONS e;
00799     CBUF *C = cbuf+AC.cbufnum;
00800     int olddict = AO.CurrentDictionary;
00801     skip = &AO.OutSkip;
00802     *skip = 0;
00803     AO.OutputLine = AO.OutFill = (UBYTE *)AT.WorkPointer;
00804     AO.CurrentDictionary = 0;
00805     FiniLine();
00806     OutScr = (UBYTE *)AT.WorkPointer + ( TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer) ) /2;
00807     if ( AC.CodesFlag || AC.NamesFlag > 1 ) startvalue = 0;
00808     else startvalue = FIRSTUSERSYMBOL;
00809 /*
00810         #[ Symbols :
00811 */
00812     if ( ( j = NumSymbols ) > startvalue ) {
00813         TokenToLine((UBYTE *)" Symbols");
00814         *skip = 3;
00815         FiniLine();
00816         for ( i = startvalue; i < j; i++ ) {
00817             if ( i >= BUILTINSYMBOLS && i < FIRSTUSERSYMBOL ) continue;
00818             Out = StrCopy(VARNAME(symbols,i),OutScr);
00819             if ( symbols[i].minpower > -MAXPOWER || symbols[i].maxpower < MAXPOWER ) {
00820                 Out = StrCopy((UBYTE *)"(",Out);
00821                 if ( symbols[i].minpower > -MAXPOWER )
00822                     Out = NumCopy(symbols[i].minpower,Out);
00823                 Out = StrCopy((UBYTE *)":",Out);
00824                 if ( symbols[i].maxpower < MAXPOWER )
00825                     Out = NumCopy(symbols[i].maxpower,Out);
00826                 Out = StrCopy((UBYTE *)")",Out);
00827             }
00828             if ( ( symbols[i].complex & VARTYPEIMAGINARY ) == VARTYPEIMAGINARY ) {
00829                 Out = StrCopy((UBYTE *)"#i",Out);
00830             }
00831             else if ( ( symbols[i].complex & VARTYPECOMPLEX ) == VARTYPECOMPLEX ) {
00832                 Out = StrCopy((UBYTE *)"#c",Out);
00833             }
00834             else if ( ( symbols[i].complex & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY ) {
00835                 Out = StrCopy((UBYTE *)"#",Out);
00836                 if ( ( symbols[i].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
00837                     Out = StrCopy((UBYTE *)"-",Out);
00838                 }
00839                 else {
00840                     Out = StrCopy((UBYTE *)"+",Out);
00841                 }
00842                 Out = NumCopy(symbols[i].maxpower,Out);
00843             }
00844             if ( AC.CodesFlag ) Out = CodeToLine(i,Out);
00845             if ( ( symbols[i].complex & VARTYPECOMPLEX ) == VARTYPECOMPLEX ) i++;
00846             StrCopy((UBYTE *)" ",Out);
00847             TokenToLine(OutScr);
00848         }
00849         *skip = 0;
00850         FiniLine();
```

```
00851     }
00852 /*
00853         #] Symbols :
00854         #[ Indices :
00855 */
00856     if ( AC.CodesFlag || AC.NamesFlag > 1 ) startvalue = 0;
00857     else startvalue = BUILTININDICES;
00858     if ( ( j = NumIndices ) > startvalue ) {
00859         TokenToLine((UBYTE *)" Indices");
00860         *skip = 3;
00861         FiniLine();
00862         for ( i = startvalue; i < j; i++ ) {
00863             Out = StrCopy(FindIndex(i+AM.OffsetIndex),OutScr);
00864             Out = StrCopy(VARNAME(indices,i),OutScr);
00865             if ( indices[i].dimension >= 0 ) {
00866                 if ( indices[i].dimension != AC.lDefDim ) {
00867                     Out = StrCopy((UBYTE *)"=",Out);
00868                     Out = NumCopy(indices[i].dimension,Out);
00869                 }
00870             }
00871             else if ( indices[i].dimension < 0 ) {
00872                 Out = StrCopy((UBYTE *)"=",Out);
00873                 Out = StrCopy(VARNAME(symbols,-indices[i].dimension),Out);
00874                 if ( indices[i].nmin4 < -NMIN4SHIFT ) {
00875                     Out = StrCopy((UBYTE *)":",Out);
00876                     Out = StrCopy(VARNAME(symbols,-indices[i].nmin4-NMIN4SHIFT),Out);
00877                 }
00878             }
00879             if ( AC.CodesFlag ) Out = CodeToLine(i+AM.OffsetIndex,Out);
00880             StrCopy((UBYTE *)" ",Out);
00881             TokenToLine(OutScr);
00882         }
00883         *skip = 0;
00884         FiniLine();
00885     }
00886 /*
00887         #] Indices :
00888         #[ Vectors :
00889 */
00890     if ( AC.CodesFlag || AC.NamesFlag > 1 ) startvalue = 0;
00891     else startvalue = BUILTINVECTORS;
00892     if ( ( j = NumVectors ) > startvalue ) {
00893         TokenToLine((UBYTE *)" Vectors");
00894         *skip = 3;
00895         FiniLine();
00896         for ( i = startvalue; i < j; i++ ) {
00897             Out = StrCopy(VARNAME(vectors,i),OutScr);
00898             if ( AC.CodesFlag ) Out = CodeToLine(i+AM.OffsetVector,Out);
00899             StrCopy((UBYTE *)" ",Out);
00900             TokenToLine(OutScr);
00901         }
00902         *skip = 0;
00903         FiniLine();
00904     }
00905 /*
00906         #] Vectors :
00907         #[ Functions :
00908 */
00909     if ( AC.CodesFlag || AC.NamesFlag > 1 ) startvalue = 0;
00910     else startvalue = AM.NumFixedFunctions;
00911     for ( k = 0; k < 2; k++ ) {
00912         first = 1;
00913         j = NumFunctions;
00914         for ( i = startvalue; i < j; i++ ) {
00915             if ( i > MAXBUILTINFUNCTION-FUNCTION
00916              && i < FIRSTUSERFUNCTION-FUNCTION ) continue;
00917             if ( ( k == 0 && functions[i].commute )
00918             || ( k != 0 && !functions[i].commute ) ) {
00919                 if ( first ) {
00920                     TokenToLine((UBYTE *)(FG.FunNam[k]));
00921                     *skip = 3;
00922                     FiniLine();
00923                     first = 0;
00924                 }
00925                 Out = StrCopy(VARNAME(functions,i),OutScr);
00926                 if ( ( functions[i].complex & VARTYPEIMAGINARY ) == VARTYPEIMAGINARY ) {
00927                     Out = StrCopy((UBYTE *)"#i",Out);
00928                 }
00929                 else if ( ( functions[i].complex & VARTYPECOMPLEX ) == VARTYPECOMPLEX ) {
00930                     Out = StrCopy((UBYTE *)"#c",Out);
00931                 }
00932                 if ( functions[i].spec == VERTEXFUNCTION ) {
00933                     Out = StrCopy((UBYTE *)"(Particle)",Out);
00934                 }
00935                 else if ( functions[i].spec >= TENSORFUNCTION ) {
00936                     Out = StrCopy((UBYTE *)"(Tensor)",Out);
00937                 }
```

```
00938                 if ( functions[i].symmetric > 0 ) {
00939                     if ( ( functions[i].symmetric & REVERSEORDER ) != 0 ) {
00940                         Out = StrCopy((UBYTE *)(rsymname[(functions[i].symmetric &
      ~REVERSEORDER)-1]),Out);
00941                     }
00942                     else {
00943                         Out = StrCopy((UBYTE *)(symname[functions[i].symmetric-1]),Out);
00944                     }
00945                 }
00946                 if ( AC.CodesFlag ) Out = CodeToLine(i+FUNCTION,Out);
00947                 if ( ( functions[i].complex & VARTYPECOMPLEX ) == VARTYPECOMPLEX ) i++;
00948                 StrCopy((UBYTE *)" ",Out);
00949                 TokenToLine(OutScr);
00950             }
00951         }
00952         *skip = 0;
00953         if ( first == 0 ) FiniLine();
00954     }
00955 /*
00956         #] Functions :
00957         #[ Sets :
00958 */
00959     if ( AC.CodesFlag || AC.NamesFlag > 1 ) startvalue = 0;
00960     else startvalue = AM.NumFixedSets;
00961     if ( ( j = AC.SetList.num ) > startvalue ) {
00962         WORD element, LastElement, type, number;
00963         TokenToLine((UBYTE *)" Sets");
00964         for ( i = startvalue; i < j; i++ ) {
00965             *skip = 3;
00966             FiniLine();
00967             if ( Sets[i].name < 0 ) {
00968                 Out = StrCopy((UBYTE *)"{}",OutScr);
00969             }
00970             else {
00971                 Out = StrCopy(VARNAME(Sets,i),OutScr);
00972             }
00973             if ( AC.CodesFlag ) Out = CodeToLine(i,Out);
00974             StrCopy((UBYTE *)":",Out);
00975             TokenToLine(OutScr);
00976             if ( i < AM.NumFixedSets ) {
00977                 TokenToLine((UBYTE *)" ");
00978                 TokenToLine((UBYTE *)fixedsets[i].description);
00979             }
00980             else if ( Sets[i].type == CRANGE ) {
00981                 int iflag = 0;
00982                 if ( Sets[i].first == 3*MAXPOWER ) {
00983                 }
00984                 else if ( Sets[i].first >= MAXPOWER ) {
00985                     TokenToLine((UBYTE *)"<=");
00986                     NumCopy(Sets[i].first-2*MAXPOWER,OutScr);
00987                     TokenToLine(OutScr);
00988                     iflag = 1;
00989                 }
00990                 else {
00991                     TokenToLine((UBYTE *)"<");
00992                     NumCopy(Sets[i].first,OutScr);
00993                     TokenToLine(OutScr);
00994                     iflag = 1;
00995                 }
00996                 if ( Sets[i].last == -3*MAXPOWER ) {
00997                 }
00998                 else if ( Sets[i].last <= -MAXPOWER ) {
00999                     if ( iflag ) TokenToLine((UBYTE *)",");
01000                     TokenToLine((UBYTE *)">=");
01001                     NumCopy(Sets[i].last+2*MAXPOWER,OutScr);
01002                     TokenToLine(OutScr);
01003                 }
01004                 else {
01005                     if ( iflag ) TokenToLine((UBYTE *)",");
01006                     TokenToLine((UBYTE *)">");
01007                     NumCopy(Sets[i].last,OutScr);
01008                     TokenToLine(OutScr);
01009                 }
01010             }
01011             else {
01012                 element = Sets[i].first;
01013                 LastElement = Sets[i].last;
01014                 type = Sets[i].type;
01015                 while ( element < LastElement ) {
01016                     TokenToLine((UBYTE *)" ");
01017                     number = SetElements[element++];
01018                     switch ( type ) {
01019                         case CSYMBOL:
01020                             if ( number < 0 ) {
01021                                 StrCopy(VARNAME(symbols,-number),OutScr);
01022                                 StrCopy((UBYTE *)"?",Out);
01023                                 TokenToLine(OutScr);
```

```
01024                           }
01025                           else if ( number < MAXPOWER )
01026                               TokenToLine(VARNAME(symbols,number));
01027                           else {
01028                               NumCopy(number-2*MAXPOWER,OutScr);
01029                               TokenToLine(OutScr);
01030                           }
01031                           break;
01032                       case CINDEX:
01033                           if ( number >= AM.IndDum ) {
01034                               Out = StrCopy((UBYTE *)"N",OutScr);
01035                               Out = NumCopy(number-(AM.IndDum),Out);
01036                               StrCopy((UBYTE *)"_?",Out);
01037                               TokenToLine(OutScr);
01038                           }
01039                           else if ( number >= AM.OffsetIndex + (WORD)WILDMASK ) {
01040                               Out = StrCopy(VARNAME(indices,number
01041                               -AM.OffsetIndex-WILDMASK),OutScr);
01042                               StrCopy((UBYTE *)"?",Out);
01043                               TokenToLine(OutScr);
01044                           }
01045                           else if ( number >= AM.OffsetIndex ) {
01046                               TokenToLine(VARNAME(indices,number-AM.OffsetIndex));
01047                           }
01048                           else {
01049                               NumCopy(number,OutScr);
01050                               TokenToLine(OutScr);
01051                           }
01052                           break;
01053                       case CVECTOR:
01054                           Out = OutScr;
01055                           if ( number < AM.OffsetVector ) {
01056                               number += WILDMASK;
01057                               Out = StrCopy((UBYTE *)"-",Out);
01058                           }
01059                           if ( number >= AM.OffsetVector + WILDOFFSET ) {
01060                               Out = StrCopy(VARNAME(vectors,number
01061                               -AM.OffsetVector-WILDOFFSET),Out);
01062                               StrCopy((UBYTE *)"?",Out);
01063                           }
01064                           else {
01065                               Out = StrCopy(VARNAME(vectors,number-AM.OffsetVector),Out);
01066                           }
01067                           TokenToLine(OutScr);
01068                           break;
01069                       case CFUNCTION:
01070                           if ( number >= FUNCTION + (WORD)WILDMASK ) {
01071                               Out = StrCopy(VARNAME(functions,number
01072                               -FUNCTION-WILDMASK),OutScr);
01073                               StrCopy((UBYTE *)"?",Out);
01074                               TokenToLine(OutScr);
01075                           }
01076                           TokenToLine(VARNAME(functions,number-FUNCTION));
01077                           break;
01078                       default:
01079                           NumCopy(number,OutScr);
01080                           TokenToLine(OutScr);
01081                           break;
01082                   }
01083               }
01084           }
01085       }
01086       *skip = 0;
01087       FiniLine();
01088   }
01089 /*
01090       #] Sets :
01091       #[ Expressions :
01092 */
01093   if ( AS.ExecMode ) {
01094       e = Expressions;
01095       j = NumExpressions;
01096       first = 1;
01097       for ( i = 0; i < j; i++, e++ ) {
01098           if ( e->status >= 0 ) {
01099               if ( first ) {
01100                   TokenToLine((UBYTE *)" Expressions");
01101                   *skip = 3;
01102                   FiniLine();
01103                   first = 0;
01104               }
01105               Out = StrCopy(AC.exprnames->namebuffer+e->name,OutScr);
01106               Out = StrCopy((UBYTE *)(FG.ExprStat[e->status]),Out);
01107               if ( AC.CodesFlag ) Out = CodeToLine(i,Out);
01108               StrCopy((UBYTE *)" ",Out);
01109               TokenToLine(OutScr);
01110           }
```

```
01111         }
01112         if ( !first ) {
01113             *skip = 0;
01114             FiniLine();
01115         }
01116     }
01117     e = Expressions;
01118     j = NumExpressions;
01119     first = 1;
01120     for ( i = 0; i < j; i++ ) {
01121         if ( e->printflag && ( e->status == LOCALEXPRESSION ||
01122         e->status == GLOBALEXPRESSION || e->status == UNHIDELEXPRESSION
01123         || e->status == UNHIDEGEXPRESSION ) ) {
01124             if ( first ) {
01125                 TokenToLine((UBYTE *)" Expressions to be printed");
01126                 *skip = 3;
01127                 FiniLine();
01128                 first = 0;
01129             }
01130             Out = StrCopy(AC.exprnames->namebuffer+e->name,OutScr);
01131             StrCopy((UBYTE *)" ",Out);
01132             TokenToLine(OutScr);
01133         }
01134         e++;
01135     }
01136     if ( !first ) {
01137         *skip = 0;
01138         FiniLine();
01139     }
01140 /*
01141         #] Expressions :
01142         #[ Dollars :
01143 */
01144
01145     if ( AC.CodesFlag || AC.NamesFlag > 1 ) startvalue = 0;
01146     else startvalue = BUILTINDOLLARS;
01147     if ( ( j = NumDollars ) > startvalue ) {
01148         TokenToLine((UBYTE *)" Dollar variables");
01149         *skip = 3;
01150         FiniLine();
01151         for ( i = startvalue; i < j; i++ ) {
01152             Out = StrCopy((UBYTE *)"$", OutScr);
01153             Out = StrCopy(DOLLARNAME(Dollars, i), Out);
01154             if ( AC.CodesFlag ) Out = CodeToLine(i, Out);
01155             StrCopy((UBYTE *)" ", Out);
01156             TokenToLine(OutScr);
01157         }
01158         *skip = 0;
01159         FiniLine();
01160     }
01161
01162     if ( ( j = NumPotModdollars ) > 0 ) {
01163         TokenToLine((UBYTE *)" Dollar variables to be modified");
01164         *skip = 3;
01165         FiniLine();
01166         for ( i = 0; i < j; i++ ) {
01167             Out = StrCopy((UBYTE *)"$", OutScr);
01168             Out = StrCopy(DOLLARNAME(Dollars, PotModdollars[i]), Out);
01169             for ( k = 0; k < NumModOptdollars; k++ )
01170                 if ( ModOptdollars[k].number == PotModdollars[i] ) break;
01171             if ( k < NumModOptdollars ) {
01172                 switch ( ModOptdollars[k].type ) {
01173                     case MODSUM:
01174                         Out = StrCopy((UBYTE *)"(sum)", Out);
01175                         break;
01176                     case MODMAX:
01177                         Out = StrCopy((UBYTE *)"(maximum)", Out);
01178                         break;
01179                     case MODMIN:
01180                         Out = StrCopy((UBYTE *)"(minimum)", Out);
01181                         break;
01182                     case MODLOCAL:
01183                         Out = StrCopy((UBYTE *)"(local)", Out);
01184                         break;
01185                     default:
01186                         Out = StrCopy((UBYTE *)"(?)", Out);
01187                         break;
01188                 }
01189             }
01190             StrCopy((UBYTE *)" ", Out);
01191             TokenToLine(OutScr);
01192         }
01193         *skip = 0;
01194         FiniLine();
01195     }
01196 /*
01197         #] Dollars :
```

```
01198 */
01199
01200     if ( AC.ncmod != 0 ) {
01201         TokenToLine((UBYTE *)"All arithmetic is modulus ");
01202         LongToLine((UWORD *)AC.cmod,ABS(AC.ncmod));
01203         if ( AC.ncmod > 0 ) TokenToLine((UBYTE *)" with powerreduction");
01204         else                TokenToLine((UBYTE *)" without powerreduction");
01205         if ( ( AC.modmode & POSNEG ) != 0 ) TokenToLine((UBYTE *)" centered around 0");
01206         else                                TokenToLine((UBYTE *)" positive numbers only");
01207         FiniLine();
01208     }
01209     if ( AC.lDefDim != 4 ) {
01210         TokenToLine((UBYTE *)"The default dimension is ");
01211         if ( AC.lDefDim >= 0 ) {
01212             NumCopy(AC.lDefDim,OutScr);
01213             TokenToLine(OutScr);
01214         }
01215         else {
01216             TokenToLine(VARNAME(symbols,-AC.lDefDim));
01217             if ( AC.lDefDim4 != -NMIN4SHIFT ) {
01218                 TokenToLine((UBYTE *)":");
01219                 if ( AC.lDefDim4 >= -NMIN4SHIFT ) {
01220                     NumCopy(AC.lDefDim4,OutScr);
01221                     TokenToLine(OutScr);
01222                 }
01223                 else {
01224                     TokenToLine(VARNAME(symbols,-AC.lDefDim4-NMIN4SHIFT));
01225                 }
01226             }
01227         }
01228         FiniLine();
01229     }
01230     if ( AC.lUnitTrace != 4 ) {
01231         TokenToLine((UBYTE *)"The trace of the unit matrix is ");
01232         if ( AC.lUnitTrace >= 0 ) {
01233             NumCopy(AC.lUnitTrace,OutScr);
01234             TokenToLine(OutScr);
01235         }
01236         else {
01237             TokenToLine(VARNAME(symbols,-AC.lUnitTrace));
01238         }
01239         FiniLine();
01240     }
01241     if ( AO.NumDictionaries > 0 ) {
01242         for ( i = 0; i < AO.NumDictionaries; i++ ) {
01243             WriteDictionary(AO.Dictionaries[i]);
01244         }
01245         if ( olddict > 0 )
01246             MesPrint("\nCurrently dictionary %s is active\n",
01247                 AO.Dictionaries[olddict-1]->name);
01248         else
01249             MesPrint("\nCurrently there is no active dictionary\n");
01250     }
01251     if ( AC.CodesFlag ) {
01252         if ( C->numlhs > 0 ) {
01253             TokenToLine((UBYTE *)" Left Hand Sides:");
01254             AO.OutSkip = 3;
01255             for ( i = 1; i <= C->numlhs; i++ ) {
01256                 FiniLine();
01257                 skip = C->lhs[i];
01258                 j = skip[1];
01259                 while ( --j >= 0 ) { TalToLine((UWORD)(*skip++)); TokenToLine((UBYTE *)"  "); }
01260             }
01261             AO.OutSkip = 0;
01262             FiniLine();
01263         }
01264         if ( C->numrhs > 0 ) {
01265             TokenToLine((UBYTE *)" Right Hand Sides:");
01266             AO.OutSkip = 3;
01267             for ( i = 1; i <= C->numrhs; i++ ) {
01268                 FiniLine();
01269                 skip = C->rhs[i];
01270                 while ( ( j = skip[0] ) != 0 ) {
01271                     while ( --j >= 0 ) { TalToLine((UWORD)(*skip++)); TokenToLine((UBYTE *)"  "); }
01272                 }
01273                 FiniLine();
01274             }
01275             AO.OutSkip = 0;
01276             FiniLine();
01277         }
01278     }
01279     AO.CurrentDictionary = olddict;
01280 }
01281
01282 /*
01283         #] WriteLists :
01284         #[ WriteDictionary :
```

```
01285
01286     This routine is part of WriteLists and should be called from there.
01287 */
01288
01289 void WriteDictionary(DICTIONARY *dict)
01290 {
01291     GETIDENTITY
01292     int i, first;
01293     WORD *skip, na, *a, spec, *t, *tstop, j;
01294     UBYTE str[2], *OutScr, *Out;
01295     WORD oldoutputmode = AC.OutputMode, oldoutputspaces = AC.OutputSpaces;
01296     WORD oldoutskip = AO.OutSkip;
01297     AC.OutputMode = NORMALFORMAT;
01298     AC.OutputSpaces = NOSPACEFORMAT;
01299     MesPrint("===Contents of dictionary %s===",dict->name);
01300     skip = &AO.OutSkip;
01301     *skip = 3;
01302     AO.OutputLine = AO.OutFill = (UBYTE *)AT.WorkPointer;
01303     for ( j = 0; j < *skip; j++ ) *(AO.OutFill)++ = ' ';
01304
01305     OutScr = (UBYTE *)AT.WorkPointer + ( TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer) ) /2;
01306     for ( i = 0; i < dict->numelements; i++ ) {
01307         switch ( dict->elements[i]->type ) {
01308             case DICT_INTEGERNUMBER:
01309                 LongToLine((UWORD *)(dict->elements[i]->lhs),dict->elements[i]->size);
01310                 Out = OutScr; *Out = 0;
01311                 break;
01312             case DICT_RATIONALNUMBER:
01313                 a = dict->elements[i]->lhs;
01314                 na = a[a[0]-1]; na = (ABS(na)-1)/2;
01315                 RatToLine((UWORD *)(a+1),na);
01316                 Out = OutScr; *Out = 0;
01317                 break;
01318             case DICT_SYMBOL:
01319                 na = dict->elements[i]->lhs[0];
01320                 Out = StrCopy(VARNAME(symbols,na),OutScr);
01321                 break;
01322             case DICT_VECTOR:
01323                 na = dict->elements[i]->lhs[0]-AM.OffsetVector;
01324                 Out = StrCopy(VARNAME(vectors,na),OutScr);
01325                 break;
01326             case DICT_INDEX:
01327                 na = dict->elements[i]->lhs[0]-AM.OffsetIndex;
01328                 Out = StrCopy(VARNAME(indices,na),OutScr);
01329                 break;
01330             case DICT_FUNCTION:
01331                 na = dict->elements[i]->lhs[0]-FUNCTION;
01332                 Out = StrCopy(VARNAME(functions,na),OutScr);
01333                 break;
01334             case DICT_FUNCTION_WITH_ARGUMENTS:
01335                 t = dict->elements[i]->lhs;
01336                 na = *t-FUNCTION;
01337                 Out = StrCopy(VARNAME(functions,na),OutScr);
01338                 spec = functions[*t - FUNCTION].spec;
01339                 tstop = t + t[1];
01340                 first = 1;
01341                 if ( t[1] <= FUNHEAD ) {}
01342                 else if ( spec >= TENSORFUNCTION ) {
01343                     t += FUNHEAD; *Out++ = (UBYTE)'(';
01344                     while ( t < tstop ) {
01345                         if ( first == 0 ) *Out++ = (UBYTE)(',');
01346                         else first = 0;
01347                         j = *t++;
01348                         if ( j >= 0 ) {
01349                             if ( j < AM.OffsetIndex ) { Out = NumCopy(j,Out); }
01350                             else if ( j < AM.IndDum ) {
01351                                 Out = StrCopy(VARNAME(indices,j-AM.OffsetIndex),Out);
01352                             }
01353                             else {
01354                                 MesPrint("Currently wildcards are not allowed in dictionary
      elements");
01355                                 Terminate(-1);
01356                             }
01357                         }
01358                         else {
01359                             Out = StrCopy(VARNAME(vectors,j-AM.OffsetVector),Out);
01360                         }
01361                     }
01362                     *Out++ = (UBYTE)')'; *Out = 0;
01363                 }
01364                 else {
01365                     t += FUNHEAD; *Out++ = (UBYTE)'('; *Out = 0;
01366                     TokenToLine(OutScr);
01367                     while ( t < tstop ) {
01368                         if ( !first ) TokenToLine((UBYTE *)",");
01369                         WriteArgument(t);
01370                         NEXTARG(t)
```

```
01371                       first = 0;
01372                   }
01373                   Out = OutScr;
01374                   *Out++ = (UBYTE)')'; *Out = 0;
01375               }
01376               break;
01377           case DICT_SPECIALCHARACTER:
01378               str[0] = (UBYTE)(dict->elements[i]->lhs[0]);
01379               str[1] = 0;
01380               Out = StrCopy(str,OutScr);
01381               break;
01382           default:
01383               Out = OutScr; *Out = 0;
01384               break;
01385       }
01386       Out = StrCopy((UBYTE *)": \"",Out);
01387       Out = StrCopy((UBYTE *)(dict->elements[i]->rhs),Out);
01388       Out = StrCopy((UBYTE *)"\"",Out);
01389       TokenToLine(OutScr);
01390       FiniLine();
01391   }
01392   MesPrint("========End of dictionary %s===",dict->name);
01393   AC.OutputMode = oldoutputmode;
01394   AC.OutputSpaces = oldoutputspaces;
01395   AO.OutSkip = oldoutskip;
01396 }
01397
01398 /*
01399       #] WriteDictionary :
01400       #[ WriteArgument :      void WriteArgument(WORD *t)
01401
01402       Write a single argument field. The general field goes to
01403       WriteExpression and the fast field is dealt with here.
01404 */
01405
01406 void WriteArgument(WORD *t)
01407 {
01408   UBYTE buffer[180];
01409   UBYTE *Out;
01410   WORD i;
01411   int oldoutsidefun, oldlowestlevel = lowestlevel;
01412   lowestlevel = 0;
01413   if ( *t > 0 ) {
01414       oldoutsidefun = AC.outsidefun; AC.outsidefun = 0;
01415       WriteExpression(t+ARGHEAD,(LONG)(*t-ARGHEAD));
01416       AC.outsidefun = oldoutsidefun;
01417       goto CleanUp;
01418   }
01419   Out = buffer;
01420   if ( *t == -SNUMBER) {
01421       NumCopy(t[1],Out);
01422   }
01423   else if ( *t == -SYMBOL ) {
01424       if ( t[1] >= MAXVARIABLES-cbuf[AM.sbufnum].numrhs ) {
01425           Out = StrCopy(FindExtraSymbol(MAXVARIABLES-t[1]),Out);
01426 /*
01427           Out = StrCopy((UBYTE *)AC.extrasym,Out);
01428           if ( AC.extrasymbols == 0 ) {
01429               Out = NumCopy((MAXVARIABLES-t[1]),Out);
01430               Out = StrCopy((UBYTE *)"_",Out);
01431           }
01432           else if ( AC.extrasymbols == 1 ) {
01433               Out = AddArrayIndex((MAXVARIABLES-t[1]),Out);
01434           }
01435 */
01436 /*
01437           else if ( AC.extrasymbols == 2 ) {
01438               Out = NumCopy((MAXVARIABLES-t[1]),Out);
01439           }
01440 */
01441       }
01442       else {
01443           StrCopy(FindSymbol(t[1]),Out);
01444 /*        StrCopy(VARNAME(symbols,t[1]),Out); */
01445       }
01446   }
01447   else if ( *t == -VECTOR ) {
01448       if ( t[1] == FUNNYVEC ) { *Out++ = '?'; *Out = 0; }
01449       else
01450           StrCopy(FindVector(t[1]),Out);
01451 /*        StrCopy(VARNAME(vectors,t[1] - AM.OffsetVector),Out); */
01452   }
01453   else if ( *t == -MINVECTOR ) {
01454       *Out++ = '-';
01455       StrCopy(FindVector(t[1]),Out);
01456 /*    StrCopy(VARNAME(vectors,t[1] - AM.OffsetVector),Out); */
01457   }
```

```
01458     else if ( *t == -INDEX ) {
01459         if ( t[1] >= 0 ) {
01460             if ( t[1] < AM.OffsetIndex ) { NumCopy(t[1],Out); }
01461             else {
01462                 i = t[1];
01463                 if ( i >= AM.IndDum ) {
01464                     i -= AM.IndDum;
01465                     *Out++ = 'N';
01466                     Out = NumCopy(i,Out);
01467                     *Out++ = '_';
01468                     *Out++ = '?';
01469                     *Out = 0;
01470                 }
01471                 else {
01472                     i -= AM.OffsetIndex;
01473                     Out = StrCopy(FindIndex(i%WILDOFFSET+AM.OffsetIndex),Out);
01474 /*                  Out = StrCopy(VARNAME(indices,i%WILDOFFSET),Out); */
01475                     if ( i >= WILDOFFSET ) { *Out++ = '?'; *Out = 0; }
01476                 }
01477             }
01478         }
01479         else if ( t[1] == FUNNYVEC ) { *Out++ = '?'; *Out = 0; }
01480         else
01481             StrCopy(FindVector(t[1]),Out);
01482 /*          StrCopy(VARNAME(vectors,t[1] - AM.OffsetVector),Out); */
01483     }
01484     else if ( *t == -DOLLAREXPRESSION ) {
01485         DOLLARS d = Dollars + t[1];
01486         *Out++ = '$';
01487         StrCopy(AC.dollarnames->namebuffer+d->name,Out);
01488     }
01489     else if ( *t == -EXPRESSION ) {
01490         StrCopy(EXPRNAME(t[1]),Out);
01491     }
01492     else if ( *t == -SETSET ) {
01493         StrCopy(VARNAME(Sets,t[1]),Out);
01494     }
01495     else if ( *t <= -FUNCTION ) {
01496         StrCopy(FindFunction(-*t),Out);
01497 /*      StrCopy(VARNAME(functions,-*t-FUNCTION),Out); */
01498     }
01499     else {
01500 /* INTERNAL_ERROR_EXCL_START */
01501         MesPrint("!>Illegal function argument while writing");
01502         goto CleanUp;
01503 /* INTERNAL_ERROR_EXCL_STOP */
01504     }
01505     TokenToLine(buffer);
01506 CleanUp:
01507     lowestlevel = oldlowestlevel;
01508     return;
01509 }
01510
01511 /*
01512         #] WriteArgument :
01513         #[ WriteSubTerm :       WORD WriteSubTerm(sterm,first)
01514
01515     Writes a single subterm field to the output line.
01516     There is a recursion for functions.
01517
01518
01519 #define NUMSPECS 8
01520 UBYTE *specfunnames[NUMSPECS] = {
01521       (UBYTE *)"fac" , (UBYTE *)"nargs", (UBYTE *)"binom"
01522     , (UBYTE *)"sign", (UBYTE *)"mod", (UBYTE *)"min", (UBYTE *)"max"
01523     , (UBYTE *)"invfac" };
01524 */
01525
01526 int WriteSubTerm(WORD *sterm, WORD first)
01527 {
01528     UBYTE buffer[80];
01529     UBYTE *Out, closepar[2] = { (UBYTE)')', 0};
01530     WORD *stopper, *t, *tt, i, j, po = 0;
01531     int oldoutsidefun;
01532     stopper = sterm + sterm[1];
01533     t = sterm + 2;
01534     switch ( *sterm ) {
01535         case SYMBOL :
01536             while ( t < stopper ) {
01537                 if ( lowestlevel && ( ( AO.PrintType & PRINTALL ) != 0 ) ) {
01538                     FiniLine();
01539                     if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
01540                         else IniLine(3);
01541                     if ( first ) TokenToLine((UBYTE *)" ");
01542                 }
01543                 if ( !first ) MultiplyToLine();
01544                 if ( AC.OutputMode == CMODE && t[1] != 1 ) {
```

```
01545                   if ( AC.Cnumpows >= t[1] && t[1] > 0 ) {
01546                       po = t[1];
01547                       Out = StrCopy((UBYTE *)"POW",buffer);
01548                       Out = NumCopy(po,Out);
01549                       Out = StrCopy((UBYTE *)"(",Out);
01550                       TokenToLine(buffer);
01551                   }
01552                   else {
01553                       TokenToLine((UBYTE *)"pow(");
01554                   }
01555               }
01556               if ( *t < NumSymbols ) {
01557                   Out = StrCopy(FindSymbol(*t),buffer); t++;
01558 /*                Out = StrCopy(VARNAME(symbols,*t),buffer); t++; */
01559               }
01560               else {
01561 /*
01562                   see also routine PrintSubtermList.
01563 */
01564                   Out = StrCopy(FindExtraSymbol(MAXVARIABLES-*t),buffer);
01565 /*
01566                   Out = StrCopy((UBYTE *)AC.extrasym,buffer);
01567                   if ( AC.extrasymbols == 0 ) {
01568                       Out = NumCopy((MAXVARIABLES-*t),Out);
01569                       Out = StrCopy((UBYTE *)"_",Out);
01570                   }
01571                   else if ( AC.extrasymbols == 1 ) {
01572                       Out = AddArrayIndex((MAXVARIABLES-*t),Out);
01573                   }
01574 */
01575 /*
01576                   else if ( AC.extrasymbols == 2 ) {
01577                       Out = NumCopy((MAXVARIABLES-*t),Out);
01578                   }
01579 */
01580                   t++;
01581               }
01582               if ( AC.OutputMode == CMODE && po > 1
01583                 && AC.Cnumpows >= po ) {
01584                   Out = StrCopy((UBYTE *)")",Out);
01585                   po = 0;
01586               }
01587               else if ( *t != 1 ) WrtPower(Out,*t);
01588               TokenToLine(buffer);
01589               t++;
01590               first = 0;
01591           }
01592           break;
01593       case VECTOR :
01594           while ( t < stopper ) {
01595               if ( lowestlevel && ( ( AO.PrintType & PRINTALL ) != 0 ) ) {
01596                   FiniLine();
01597                   if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
01598                       else IniLine(3);
01599                   if ( first ) TokenToLine((UBYTE *)" ");
01600               }
01601               if ( !first ) MultiplyToLine();
01602
01603               Out = StrCopy(FindVector(*t),buffer);
01604 /*            Out = StrCopy(VARNAME(vectors,*t - AM.OffsetVector),buffer); */
01605               t++;
01606               if ( AC.OutputMode == MATHEMATICAMODE ) *Out++ = '[';
01607               else *Out++ = '(';
01608               if ( *t >= AM.OffsetIndex ) {
01609                   i = *t++;
01610                   if ( i >= AM.IndDum ) {
01611                       i -= AM.IndDum;
01612                       *Out++ = 'N';
01613                       Out = NumCopy(i,Out);
01614                       *Out++ = '_';
01615                       *Out++ = '?';
01616                       *Out = 0;
01617                   }
01618                   else
01619                       Out = StrCopy(FindIndex(i),Out);
01620 /*                    Out = StrCopy(VARNAME(indices,i - AM.OffsetIndex),Out); */
01621               }
01622               else if ( *t == FUNNYVEC ) { *Out++ = '?'; *Out = 0; }
01623               else {
01624                   Out = NumCopy(*t++,Out);
01625               }
01626               if ( AC.OutputMode == MATHEMATICAMODE ) *Out++ = ']';
01627               else *Out++ = ')';
01628               *Out = 0;
01629               TokenToLine(buffer);
01630               first = 0;
01631           }
```

```
01632            break;
01633        case INDEX :
01634            while ( t < stopper ) {
01635                if ( lowestlevel && ( ( AO.PrintType & PRINTALL ) != 0 ) ) {
01636                    FiniLine();
01637                    if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
01638                        else IniLine(3);
01639                    if ( first ) TokenToLine((UBYTE *)" ");
01640                }
01641                if ( !first ) MultiplyToLine();
01642                if ( *t >= 0 ) {
01643                    if ( *t < AM.OffsetIndex ) {
01644                        TalToLine((UWORD)(*t++));
01645                    }
01646                    else {
01647                        i = *t++;
01648                        if ( i >= AM.IndDum ) {
01649                            i -= AM.IndDum;
01650                            Out = buffer;
01651                            *Out++ = 'N';
01652                            Out = NumCopy(i,Out);
01653                            *Out++ = '_';
01654                            *Out++ = '?';
01655                            *Out = 0;
01656                        }
01657                        else {
01658                            i -= AM.OffsetIndex;
01659                            Out = StrCopy(FindIndex(i%WILDOFFSET+AM.OffsetIndex),buffer);
01660 /*                         Out = StrCopy(VARNAME(indices,i%WILDOFFSET),buffer); */
01661                            if ( i >= WILDOFFSET ) { *Out++ = '?'; *Out = 0; }
01662                        }
01663                        TokenToLine(buffer);
01664                    }
01665                }
01666                else {
01667                    TokenToLine(FindVector(*t)); t++;
01668 /*                 TokenToLine(VARNAME(vectors,*t - AM.OffsetVector)); t++; */
01669                }
01670                first = 0;
01671                }
01672                break;
01673            case DOLLAREXPRESSION:
01674                {
01675                    DOLLARS d = Dollars + sterm[2];
01676                    Out = StrCopy((UBYTE *)"$",buffer);
01677                    Out = StrCopy(AC.dollarnames->namebuffer+d->name,Out);
01678                    if ( sterm[3] != 1 ) WrtPower(Out,sterm[3]);
01679                    TokenToLine(buffer);
01680                }
01681                first = 0;
01682                break;
01683            case DELTA :
01684                while ( t < stopper ) {
01685                    if ( lowestlevel && ( ( AO.PrintType & PRINTALL ) != 0 ) ) {
01686                        FiniLine();
01687                        if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
01688                            else IniLine(3);
01689                        if ( first ) TokenToLine((UBYTE *)" ");
01690                    }
01691                    if ( !first ) MultiplyToLine();
01692                    Out = StrCopy((UBYTE *)"d_(",buffer);
01693                    if ( *t >= AM.OffsetIndex ) {
01694                        if ( *t < AM.IndDum ) {
01695                            Out = StrCopy(FindIndex(*t),Out);
01696 /*                         Out = StrCopy(VARNAME(indices,*t - AM.OffsetIndex),Out); */
01697                            t++;
01698                        }
01699                        else {
01700                            *Out++ = 'N';
01701                            Out = NumCopy( *t++ - AM.IndDum, Out);
01702                            *Out++ = '_';
01703                            *Out++ = '?';
01704                            *Out = 0;
01705                        }
01706                    }
01707                    else if ( *t == FUNNYVEC ) { *Out++ = '?'; *Out = 0; }
01708                    else {
01709                        Out = NumCopy(*t++,Out);
01710                    }
01711                    *Out++ = ',';
01712                    if ( *t >= AM.OffsetIndex ) {
01713                        if ( *t < AM.IndDum ) {
01714                            Out = StrCopy(FindIndex(*t),Out);
01715 /*                         Out = StrCopy(VARNAME(indices,*t - AM.OffsetIndex),Out); */
01716                            t++;
01717                        }
01718                        else {
```

```
01719                       *Out++ = 'N';
01720                       Out = NumCopy(*t++ - AM.IndDum,Out);
01721                       *Out++ = '_';
01722                       *Out++ = '?';
01723                   }
01724               }
01725               else {
01726                   Out = NumCopy(*t++,Out);
01727               }
01728               *Out++ = ')';
01729               *Out = 0;
01730               TokenToLine(buffer);
01731               first = 0;
01732           }
01733           break;
01734       case DOTPRODUCT :
01735           while ( t < stopper ) {
01736               if ( lowestlevel && ( ( AO.PrintType & PRINTALL ) != 0 ) ) {
01737                   FiniLine();
01738                   if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
01739                       else IniLine(3);
01740                   if ( first ) TokenToLine((UBYTE *)" ");
01741               }
01742               if ( !first ) MultiplyToLine();
01743               if ( AC.OutputMode == CMODE && t[2] != 1 )
01744                   TokenToLine((UBYTE *)"pow(");
01745               if ( AC.OutputMode == MATHEMATICAMODE )
01746                   TokenToLine((UBYTE *)"(");
01747               Out = StrCopy(FindVector(*t),buffer);
01748 /*            Out = StrCopy(VARNAME(vectors,*t - AM.OffsetVector),buffer); */
01749               t++;
01750               if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
01751                || AC.OutputMode == CMODE )
01752                   *Out++ = AO.FortDotChar;
01753               else *Out++ = '.';
01754               Out = StrCopy(FindVector(*t),Out);
01755 /*            Out = StrCopy(VARNAME(vectors,*t - AM.OffsetVector),Out); */
01756               if ( AC.OutputMode == MATHEMATICAMODE ) {
01757                   *Out++ = ')';
01758                   *Out = 0;
01759               }
01760               t++;
01761               if ( *t != 1 ) WrtPower(Out,*t);
01762               t++;
01763               TokenToLine(buffer);
01764               first = 0;
01765           }
01766           break;
01767       case EXPONENT :
01768 #if FUNHEAD != 2
01769           t += FUNHEAD - 2;
01770 #endif
01771           if ( !first ) MultiplyToLine();
01772           if ( AC.OutputMode == CMODE ) TokenToLine((UBYTE *)"pow(");
01773           else TokenToLine((UBYTE *)"(");
01774           WriteArgument(t);
01775           if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
01776            || AC.OutputMode == REDUCEMODE )
01777               TokenToLine((UBYTE *)")**(");
01778           else if ( AC.OutputMode == CMODE ) TokenToLine((UBYTE *)",");
01779           else {
01780               UBYTE *Out1 = IsExponentSign();
01781               if ( Out1 ) {
01782                   TokenToLine((UBYTE *)")");
01783                   TokenToLine(Out1);
01784                   TokenToLine((UBYTE *)"(");
01785               }
01786               else TokenToLine((UBYTE *)")^(");
01787           }
01788           NEXTARG(t)
01789           WriteArgument(t);
01790           TokenToLine((UBYTE *)")");
01791           break;
01792       case DENOMINATOR :
01793 #if FUNHEAD != 2
01794           t += FUNHEAD - 2;
01795 #endif
01796           if ( first ) TokenToLine((UBYTE *)"1/(");
01797           else TokenToLine((UBYTE *)"/(");
01798           WriteArgument(t);
01799           TokenToLine((UBYTE *)")");
01800           break;
01801       case SUBEXPRESSION:
01802           if ( !first ) MultiplyToLine();
01803           TokenToLine((UBYTE *)"(");
01804           t = cbuf[sterm[4]].rhs[sterm[2]];
01805           tt = t;
```

```
01806             while ( *tt ) tt += *tt;
01807             oldoutsidefun = AC.outsidefun; AC.outsidefun = 0;
01808             if ( *t ) {
01809                 WriteExpression(t,(LONG)(tt-t));
01810             }
01811             else {
01812                 TokenToLine((UBYTE *)"0");
01813             }
01814             AC.outsidefun = oldoutsidefun;
01815             TokenToLine((UBYTE *)")");
01816             if ( sterm[3] != 1 ) {
01817                 UBYTE *Out1 = IsExponentSign();
01818                 if ( Out1 ) TokenToLine(Out1);
01819                 else TokenToLine((UBYTE *)"^");
01820                 Out = buffer;
01821                 NumCopy(sterm[3],Out);
01822                 TokenToLine(buffer);
01823             }
01824             break;
01825         default :
01826             if ( lowestlevel && ( ( AO.PrintType & PRINTALL ) != 0 ) ) {
01827                 FiniLine();
01828                 if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
01829                     else IniLine(3);
01830                 if ( first ) TokenToLine((UBYTE *)" ");
01831             }
01832             if ( *sterm < FUNCTION ) {
01833 /* INTERNAL_ERROR_EXCL_START */
01834             return(MesPrint("!>Illegal subterm while writing"));
01835 /* INTERNAL_ERROR_EXCL_STOP */
01836             }
01837             if ( !first ) MultiplyToLine();
01838             first = 1;
01839             { UBYTE *tmp;
01840                 if ( ( tmp = FindFunWithArgs(sterm) ) != 0 ) {
01841                     TokenToLine(tmp);
01842                     break;
01843                 }
01844             }
01845             t += FUNHEAD-2;
01846
01847             if ( *sterm == GAMMA && t[-FUNHEAD+1] == FUNHEAD+1 ) {
01848                 TokenToLine((UBYTE *)"gi_(");
01849             }
01850             else {
01851                 if ( *sterm != DUMFUN ) {
01852                     Out = StrCopy(FindFunction(*sterm),buffer);
01853 /*                  Out = StrCopy(VARNAME(functions,*sterm - FUNCTION),buffer); */
01854                 }
01855                 else { Out = buffer; *Out = 0; }
01856                 if ( t >= stopper ) {
01857                     TokenToLine(buffer);
01858                     break;
01859                 }
01860                 if ( AC.OutputMode == MATHEMATICAMODE ) { *Out++ = '['; closepar[0] = (UBYTE)']'; }
01861                 else { *Out++ = '('; }
01862                 *Out = 0;
01863                 TokenToLine(buffer);
01864             }
01865             i = functions[*sterm - FUNCTION].spec;
01866             if ( i >= TENSORFUNCTION ) {
01867                 int curdict = AO.CurrentDictionary;
01868                 if ( AO.CurrentDictionary && AO.CurDictNotInFunctions > 0 )
01869                     AO.CurrentDictionary = 0;
01870                 t = sterm + FUNHEAD;
01871                 while ( t < stopper ) {
01872                     if ( !first ) TokenToLine((UBYTE *)",");
01873                     else first = 0;
01874                     j = *t++;
01875                     if ( j >= 0 ) {
01876                         if ( j < AM.OffsetIndex ) TalToLine((UWORD)(j));
01877                         else if ( j < AM.IndDum ) {
01878                             i = j - AM.OffsetIndex;
01879                             Out = StrCopy(FindIndex(i%WILDOFFSET+AM.OffsetIndex),buffer);
01880 /*                          Out = StrCopy(VARNAME(indices,i%WILDOFFSET),buffer); */
01881                             if ( i >= WILDOFFSET ) { *Out++ = '?'; *Out = 0; }
01882                             TokenToLine(buffer);
01883                         }
01884                         else {
01885                             Out = buffer;
01886                             *Out++ = 'N';
01887                             Out = NumCopy(j - AM.IndDum,Out);
01888                             *Out++ = '_';
01889                             *Out++ = '?';
01890                             *Out = 0;
01891                             TokenToLine(buffer);
01892                         }
```

```
01893                     }
01894                     else if ( j == FUNNYVEC ) { TokenToLine((UBYTE *)"?"); }
01895                     else if ( j > -WILDOFFSET ) {
01896                         Out = buffer;
01897                         Out = NumCopy((UWORD)(-j + 4),Out);
01898                         *Out++ = '_';
01899                         *Out = 0;
01900                         TokenToLine(buffer);
01901                     }
01902                     else {
01903                         TokenToLine(FindVector(j));
01904 /*                      TokenToLine(VARNAME(vectors,j - AM.OffsetVector)); */
01905                     }
01906                 }
01907                 AO.CurrentDictionary = curdict;
01908             }
01909             else {
01910                 int curdict = AO.CurrentDictionary;
01911                 if ( AO.CurrentDictionary && AO.CurDictNotInFunctions > 0 )
01912                     AO.CurrentDictionary = 0;
01913                 while ( t < stopper ) {
01914                     if ( !first ) TokenToLine((UBYTE *)",");
01915                     WriteArgument(t);
01916                     NEXTARG(t)
01917                     first = 0;
01918                 }
01919                 AO.CurrentDictionary = curdict;
01920             }
01921             TokenToLine(closepar);
01922             closepar[0] = (UBYTE)')';
01923             break;
01924     }
01925     return(0);
01926 }
01927
01928 /*
01929         #] WriteSubTerm :
01930         #[ WriteInnerTerm :     WORD WriteInnerTerm(term,first)
01931
01932     Writes the contents of term to the output.
01933     Only the part that is inside parentheses is written.
01934
01935 */
01936
01937 int WriteInnerTerm(WORD *term, WORD first)
01938 {
01939     WORD *t, *s, *s1, *s2, n, i, pow;
01940 #ifdef WITHFLOAT
01941     int FloatChars = 0;
01942     GETIDENTITY
01943 #endif
01944     t = term;
01945     s = t+1;
01946     GETCOEF(t,n);
01947     while ( s < t ) {
01948         if ( *s == HAAKJE ) break;
01949         s += s[1];
01950     }
01951     if ( s < t ) { s += s[1]; }
01952     else { s = term+1; }
01953
01954     if ( n < 0 || !first ) {
01955         if ( n > 0 ) { TOKENTOLINE(" + ","+") }
01956         else if ( n < 0 ) { n = -n; TOKENTOLINE(" - ","-") }
01957     }
01958     if ( AC.modpowers ) {
01959         if ( n == 1 && *t == 1 && t > s ) first = 1;
01960         else if ( ABS(AC.ncmod) == 1 ) {
01961             UBYTE *Out1 = IsExponentSign();
01962             LongToLine((UWORD *)AC.powmod,AC.npowmod);
01963             if ( Out1 ) TokenToLine(Out1);
01964             else TokenToLine((UBYTE *)"^");
01965             TalToLine(AC.modpowers[(LONG)((UWORD)*t)]);
01966             first = 0;
01967         }
01968         else {
01969             LONG jj;
01970             UBYTE *Out1 = IsExponentSign();
01971             LongToLine((UWORD *)AC.powmod,AC.npowmod);
01972             if ( Out1 ) TokenToLine(Out1);
01973             else TokenToLine((UBYTE *)"^");
01974             jj = (UWORD)*t;
01975             if ( n == 2 ) jj += ((LONG)t[1])«BITSINWORD;
01976             if ( AC.modpowers[jj+1] == 0 ) {
01977                 TalToLine(AC.modpowers[jj]);
01978             }
01979             else {
```

```
01980                 LongToLine(AC.modpowers+jj,2);
01981             }
01982             first = 0;
01983         }
01984     }
01985     else if ( n != 1 || *t != 1 || t[1] != 1 || t <= s ) {
01986         if ( lowestlevel && ( ( AO.PrintType & PRINTONEFUNCTION ) != 0 ) ) {
01987             FiniLine();
01988             if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
01989             else IniLine(3);
01990         }
01991         if ( AO.CurrentDictionary > 0 ) TransformRational((UWORD *)t,n);
01992         else                            RatToLine((UWORD *)t,n);
01993         first = 0;
01994     }
01995 #ifdef WITHFLOAT
01996 /*
01997     Check whether there is a 'proper' float_ function and no raw mode.
01998     If so, print as float.
01999     Raw mode is indicated as AO.FloatPrec < 0.
02000     AO.FloatPrec == 0 indicated as many digits as the precision allows.
02001 */
02002     else if ( AO.FloatPrec >= 0 && AT.aux_ != 0 ) {
02003         WORD *ss = s;
02004         while ( ss < t ) {
02005             if ( *ss == FLOATFUN ) {
02006                 if ( ( FloatChars = PrintFloat(ss,AO.FloatPrec) ) != 0 ) {
02007                     TokenToLine(AO.floatspace);
02008                     if ( AC.IsFortran90 == ISFORTRAN90 && AC.Fortran90Kind ) {
02009                         AddToLine(AC.Fortran90Kind);
02010                     }
02011                     first = 0;
02012                 }
02013                 break;
02014             }
02015             ss += ss[1];
02016         }
02017         if ( ss >= t ) first = 1;
02018     }
02019 #endif
02020     else first = 1;
02021     while ( s < t ) {
02022         if ( lowestlevel && ( (AO.PrintType & (PRINTONEFUNCTION | PRINTALL)) == PRINTONEFUNCTION ) ) {
02023             FiniLine();
02024             if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
02025             else IniLine(3);
02026         }
02027
02028 /*
02029         #[ NEWGAMMA :
02030 */
02031 #ifdef NEWGAMMA
02032         if ( *s == GAMMA ) {    /* String them up */
02033             WORD *tt,*ss;
02034             ss = AT.WorkPointer;
02035             *ss++ = GAMMA;
02036             *ss++ = s[1];
02037             FILLFUN(ss)
02038             *ss++ = s[FUNHEAD];
02039             tt = s + FUNHEAD + 1;
02040             n = s[1] - FUNHEAD-1;
02041             do {
02042                 while ( --n >= 0 ) *ss++ = *tt++;
02043                 tt = s + s[1];
02044                 while ( *tt == GAMMA && tt[FUNHEAD] == s[FUNHEAD] && tt < t ) {
02045                     s = tt;
02046                     tt += FUNHEAD + 1;
02047                     n = s[1] - FUNHEAD-1;
02048                     if ( n > 0 ) break;
02049                 }
02050             } while ( n > 0 );
02051             tt = AT.WorkPointer;
02052             AT.WorkPointer = ss;
02053             tt[1] = WORDDIF(ss,tt);
02054             if ( WriteSubTerm(tt,first) ) {
02055                 MesCall("WriteInnerTerm");
02056                 SETERROR(-1)
02057             }
02058             AT.WorkPointer = tt;
02059         }
02060         else
02061 #endif
02062 /*
02063         #] NEWGAMMA :
02064 */
02065 #ifdef WITHFLOAT
02066         if ( *s == FLOATFUN && AO.FloatPrec >= 0 && AT.aux_ != 0 ) {
```

```
02067         }
02068         else
02069 #endif
02070         {
02071             if ( *s >= FUNCTION && AC.funpowers > 0
02072             && functions[*s-FUNCTION].spec == 0 && ( AC.funpowers == ALLFUNPOWERS ||
02073             ( AC.funpowers == COMFUNPOWERS && functions[*s-FUNCTION].commute == 0 ) ) ) {
02074                 pow = 1;
02075                 for(;;) {
02076                     s1 = s; s2 = s + s[1]; i = s[1];
02077                     if ( s2 < t ) {
02078                         while ( --i >= 0 && *s1 == *s2 ) { s1++; s2++; }
02079                         if ( i < 0 ) {
02080                             pow++; s = s+s[1];
02081                         }
02082                         else break;
02083                     }
02084                     else break;
02085                 }
02086                 if ( pow > 1 ) {
02087                     if ( AC.OutputMode == CMODE ) {
02088                         if ( !first ) MultiplyToLine();
02089                         TokenToLine((UBYTE *)"pow(");
02090                         first = 1;
02091                     }
02092                     if ( WriteSubTerm(s,first) ) {
02093                         MesCall("WriteInnerTerm");
02094                         SETERROR(-1)
02095                     }
02096                     if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
02097                      || AC.OutputMode == REDUCEMODE ) { TokenToLine((UBYTE *)"**"); }
02098                     else if ( AC.OutputMode == CMODE ) { TokenToLine((UBYTE *)","); }
02099                     else {
02100                         UBYTE *Out1 = IsExponentSign();
02101                         if ( Out1 ) TokenToLine(Out1);
02102                         else TokenToLine((UBYTE *)"^");
02103                     }
02104                     TalToLine(pow);
02105                     if ( AC.OutputMode == CMODE ) TokenToLine((UBYTE *)")");
02106                 }
02107                 else if ( WriteSubTerm(s,first) ) {
02108                     MesCall("WriteInnerTerm");
02109                     SETERROR(-1)
02110                 }
02111             }
02112             else if ( WriteSubTerm(s,first) ) {
02113                 MesCall("WriteInnerTerm");
02114                 SETERROR(-1)
02115             }
02116         }
02117         first = 0;
02118         s += s[1];
02119     }
02120     return(0);
02121 }
02122
02123 /*
02124         #] WriteInnerTerm :
02125         #[ WriteTerm :          WORD WriteTerm(term,lbrac,first,prtf,br)
02126
02127     Writes a term to output. It tests the bracket information first.
02128     If there are no brackets or the bracket is the same all is passed
02129     to WriteInnerTerm. If there are brackets and the bracket is not
02130     the same as for the predecessor the old bracket is closed and
02131     a new one is opened.
02132     br indicates whether we are in a subexpression, barring zeroing
02133     AO.IsBracket
02134
02135 */
02136
02137 int WriteTerm(WORD *term, WORD *lbrac, WORD first, WORD prtf, WORD br)
02138 {
02139     WORD *t, *stopper, *b, n;
02140     int oldIsFortran90 = AC.IsFortran90, i;
02141     if ( *lbrac >= 0 ) {
02142         t = term + 1;
02143         stopper = (term + *term - 1);
02144         stopper -= ABS(*stopper) - 1;
02145         while ( t < stopper ) {
02146             if ( *t == HAAKJE ) {
02147                 stopper = t;
02148                 t = term+1;
02149                 if ( *lbrac == ( n = WORDDIF(stopper,t) ) ) {
02150                     b = AO.bracket + 1;
02151                     t = term + 1;
02152                     while ( n > 0 && ( *b++ == *t++ ) ) { n--; }
02153                     if ( n <= 0 && ( ( AM.FortranCont <= 0 || AO.InFbrack < AM.FortranCont )
```

```
02154                     || ( lowestlevel == 0 ) ) ) {
02155 /*
02156                         We continue inside a bracket.
02157 */
02158                         AO.IsBracket = 1;
02159                         if ( ( prtf & PRINTCONTENTS ) != 0 ) {
02160                             AO.NumInBrack++;
02161                         }
02162                         else {
02163                             if ( WriteInnerTerm(term,0) ) goto WrtTmes;
02164                             if ( ( AO.PrintType & PRINTONETERM ) != 0 ) {
02165                                 FiniLine();
02166                                 TokenToLine((UBYTE *)"    ");
02167                             }
02168                         }
02169                         return(0);
02170                     }
02171                     t = term + 1;
02172                     n = WORDDIF(stopper,t);
02173                 }
02174 /*
02175                 Close the bracket
02176 */
02177                 if ( *lbrac ) {
02178                     if ( ( prtf & PRINTCONTENTS ) ) PrtTerms();
02179                     TOKENTOLINE(" )",")")
02180                     if ( AC.OutputMode == CMODE && AO.FactorMode == 0 )
02181                         TokenToLine((UBYTE *)";");
02182                     else if ( AO.FactorMode && ( n == 0 ) ) {
02183 /*
02184                         This should not happen.
02185 */
02186                         return(0);
02187                     }
02188                     AC.IsFortran90 = ISNOTFORTRAN90;
02189                     FiniLine();
02190                     AC.IsFortran90 = oldIsFortran90;
02191                     if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE
02192                         && AC.OutputSpaces == NORMALFORMAT
02193                         && AO.FactorMode == 0 ) FiniLine();
02194                 }
02195                 else {
02196                     if ( AC.OutputMode == CMODE && AO.FactorMode == 0 )
02197                         TokenToLine((UBYTE *)";");
02198                     if ( AO.FortFirst == 0 ) {
02199                         if ( !first ) {
02200                             AC.IsFortran90 = ISNOTFORTRAN90;
02201                             FiniLine();
02202                             AC.IsFortran90 = oldIsFortran90;
02203                         }
02204                     }
02205                 }
02206                 if ( AO.FactorMode == 0 ) {
02207                   if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
02208                    && !first ) {
02209                     WORD oldmode = AC.OutputMode;
02210                     AC.OutputMode = 0;
02211                     IniLine(0);
02212                     AC.OutputMode = oldmode;
02213                     AO.OutSkip = 7;
02214
02215                     if ( AO.FortFirst == 0 ) {
02216                         TokenToLine(AO.CurBufWrt);
02217                         TOKENTOLINE(" = ","=")
02218                         TokenToLine(AO.CurBufWrt);
02219                     }
02220                     else {
02221                         AO.FortFirst = 0;
02222                         TokenToLine(AO.CurBufWrt);
02223                         TOKENTOLINE(" = ","=")
02224                     }
02225                   }
02226                   else if ( AC.OutputMode == CMODE && !first ) {
02227                     IniLine(0);
02228                     if ( AO.FortFirst == 0 ) {
02229                         TokenToLine(AO.CurBufWrt);
02230                         TOKENTOLINE(" += ","+=")
02231                     }
02232                     else {
02233                         AO.FortFirst = 0;
02234                         TokenToLine(AO.CurBufWrt);
02235                         TOKENTOLINE(" = ","=")
02236                     }
02237                   }
02238                   else if ( startinline == 0 ) {
02239                     IniLine(0);
02240                   }
```

```
02241                   AO.InFbrack = 0;
02242                   if ( ( *lbrac = n ) > 0 ) {
02243                     b = AO.bracket;
02244                     *b++ = n + 4;
02245                     while ( --n >= 0 ) *b++ = *t++;
02246                     *b++ = 1; *b++ = 1; *b = 3;
02247                     AO.IsBracket = 0;
02248                     if ( WriteInnerTerm(AO.bracket,0) ) {
02249                         /* Error message */
02250                         WORD i;
02251 WrtTmes:                t = term;
02252                         AO.OutSkip = 3;
02253                         FiniLine();
02254                         i = *t;
02255                         while ( --i >= 0 ) { TalToLine((UWORD)(*t++));
02256                             if ( AC.OutputSpaces == NORMALFORMAT )
02257                                         TokenToLine((UBYTE *)"  "); }
02258                         AO.OutSkip = 0;
02259                         FiniLine();
02260                         MesCall("WriteTerm");
02261                         SETERROR(-1)
02262                     }
02263                     TOKENTOLINE(" * ( ","*(")
02264                     AO.NumInBrack = 0;
02265                     AO.IsBracket = 1;
02266                     if ( ( prtf & PRINTONETERM ) != 0 ) {
02267                         first = 0;
02268                         FiniLine();
02269                         TokenToLine((UBYTE *)"   ");
02270                     }
02271                     else first = 1;
02272                   }
02273                   else {
02274                     AO.IsBracket = 0;
02275                     first = 0;
02276                   }
02277                 }
02278                 else {
02279 /*
02280                     Here is the code that writes the glue between two factors.
02281                     We should not forget factors that are zero!
02282 */
02283                     if ( ( *lbrac = n ) > 0 ) {
02284                         b = AO.bracket;
02285                         *b++ = n + 4;
02286                         while ( --n >= 0 ) *b++ = *t++;
02287                         *b++ = 1; *b++ = 1; *b = 3;
02288                         for ( i = AO.FactorNum+1; i < AO.bracket[4]; i++ ) {
02289                             if ( first ) {
02290                                 TOKENTOLINE("   ( 0 )"," (0)")
02291                                 first = 0;
02292                             }
02293                             else {
02294                                 TOKENTOLINE(" * ( 0 )","*(0)")
02295                             }
02296                             FiniLine();
02297                             IniLine(0);
02298                         }
02299                         AO.FactorNum = AO.bracket[4];
02300                     }
02301                     else {
02302                         AO.NumInBrack = 0;
02303                         return(0);
02304                     }
02305                     if ( first == 0 ) { TOKENTOLINE(" * ( ","*(") }
02306                     else              { TOKENTOLINE("   ( "," (") }
02307                     AO.NumInBrack = 0;
02308                     first = 1;
02309                 }
02310                 if ( ( prtf & PRINTCONTENTS ) != 0 ) AO.NumInBrack++;
02311                 else if ( WriteInnerTerm(term,first) ) goto WrtTmes;
02312                 if ( ( AO.PrintType & PRINTONETERM ) != 0 ) {
02313                     FiniLine();
02314                     TokenToLine((UBYTE *)"   ");
02315                 }
02316                 return(0);
02317             }
02318             else t += t[1];
02319         }
02320         if ( *lbrac > 0 ) {
02321             if ( ( prtf & PRINTCONTENTS ) != 0 ) PrtTerms();
02322             TokenToLine((UBYTE *)" )");
02323             if ( AC.OutputMode == CMODE ) TokenToLine((UBYTE *)";");
02324             if ( AO.FortFirst == 0 ) {
02325                 AC.IsFortran90 = ISNOTFORTRAN90;
02326                 FiniLine();
02327                 AC.IsFortran90 = oldIsFortran90;
```

```
02328             }
02329             if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE
02330                  && AC.OutputSpaces == NORMALFORMAT ) FiniLine();
02331             if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
02332              && !first ) {
02333                 WORD oldmode = AC.OutputMode;
02334                 AC.OutputMode = 0;
02335                 IniLine(0);
02336                 AC.OutputMode = oldmode;
02337                 AO.OutSkip = 7;
02338                 if ( AO.FortFirst == 0 ) {
02339                     TokenToLine(AO.CurBufWrt);
02340                     TOKENTOLINE(" = ","=")
02341                     TokenToLine(AO.CurBufWrt);
02342                 }
02343                 else {
02344                     AO.FortFirst = 0;
02345                     TokenToLine(AO.CurBufWrt);
02346                     TOKENTOLINE(" = ","=")
02347                 }
02348 /*
02349                 TokenToLine(AO.CurBufWrt);
02350                 TOKENTOLINE(" = ","=")
02351                 if ( AO.FortFirst == 0 )
02352                     TokenToLine(AO.CurBufWrt);
02353                 else AO.FortFirst = 0;
02354 */
02355             }
02356             else if ( AC.OutputMode == CMODE && !first ) {
02357                 IniLine(0);
02358                 if ( AO.FortFirst == 0 ) {
02359                     TokenToLine(AO.CurBufWrt);
02360                     TOKENTOLINE(" += ","+=")
02361                 }
02362                 else {
02363                     AO.FortFirst = 0;
02364                     TokenToLine(AO.CurBufWrt);
02365                     TOKENTOLINE(" = ","=")
02366                 }
02367 /*
02368                 TokenToLine(AO.CurBufWrt);
02369                 if ( AO.FortFirst == 0 ) { TOKENTOLINE(" += ","+=") }
02370                 else {
02371                     TOKENTOLINE(" = ","=")
02372                     AO.FortFirst = 0;
02373                 }
02374 */
02375             }
02376             else IniLine(0);
02377             *lbrac = 0;
02378             first = 1;
02379         }
02380     }
02381     if ( !br ) AO.IsBracket = 0;
02382     if ( ( AM.FortranCont > 0 && AO.InFbrack >= AM.FortranCont ) && lowestlevel ) {
02383         if ( AC.OutputMode == CMODE ) TokenToLine((UBYTE *)";");
02384         if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
02385          && !first ) {
02386             WORD oldmode = AC.OutputMode;
02387             if ( AO.FortFirst == 0 ) {
02388                 AC.IsFortran90 = ISNOTFORTRAN90;
02389                 FiniLine();
02390                 AC.IsFortran90 = oldIsFortran90;
02391                 AC.OutputMode = 0;
02392                 IniLine(0);
02393                 AC.OutputMode = oldmode;
02394                 AO.OutSkip = 7;
02395                 TokenToLine(AO.CurBufWrt);
02396                 TOKENTOLINE(" = ","=")
02397                 TokenToLine(AO.CurBufWrt);
02398             }
02399             else {
02400                 AO.FortFirst = 0;
02401 /*
02402                 TokenToLine(AO.CurBufWrt);
02403                 TOKENTOLINE(" = ","=")
02404 */
02405             }
02406 /*
02407             TokenToLine(AO.CurBufWrt);
02408             TOKENTOLINE(" = ","=")
02409             if ( AO.FortFirst == 0 )
02410                 TokenToLine(AO.CurBufWrt);
02411             else AO.FortFirst = 0;
02412 */
02413         }
02414         else if ( AC.OutputMode == CMODE && !first ) {
```

```
02415             FiniLine();
02416             IniLine(0);
02417             if ( AO.FortFirst == 0 ) {
02418                 TokenToLine(AO.CurBufWrt);
02419                 TOKENTOLINE(" += ","+=")
02420             }
02421             else {
02422                 AO.FortFirst = 0;
02423                 TokenToLine(AO.CurBufWrt);
02424                 TOKENTOLINE(" = ","=")
02425             }
02426 /*
02427             TokenToLine(AO.CurBufWrt);
02428             if ( AO.FortFirst == 0 ) { TOKENTOLINE(" += ","+=") }
02429             else {
02430                 TOKENTOLINE(" = ","=")
02431                 AO.FortFirst = 0;
02432             }
02433 */
02434         }
02435         else {
02436             FiniLine();
02437             IniLine(0);
02438         }
02439         AO.InFbrack = 0;
02440     }
02441     if ( WriteInnerTerm(term,first) ) goto WrtTmes;
02442     if ( ( AO.PrintType & PRINTONETERM ) != 0 ) {
02443         FiniLine();
02444         IniLine(0);
02445     }
02446     return(0);
02447 }
02448
02449 /*
02450         #] WriteTerm :
02451         #[ WriteExpression :    WORD WriteExpression(terms,ltot)
02452
02453     Writes a subexpression to output.
02454     The subexpression is in terms and contains ltot words.
02455     This is only used for function arguments.
02456
02457 */
02458
02459 int WriteExpression(WORD *terms, LONG ltot)
02460 {
02461     WORD *stopper;
02462     WORD first, btot;
02463     WORD OldIsBracket = AO.IsBracket, OldPrintType = AO.PrintType;
02464     if ( !AC.outsidefun ) { AO.PrintType &= ~PRINTONETERM; first = 1; }
02465     else first = 0;
02466     stopper = terms + ltot;
02467     btot = -1;
02468     while ( terms < stopper ) {
02469         AO.IsBracket = OldIsBracket;
02470         if ( WriteTerm(terms,&btot,first,0,1) ) {
02471             MesCall("WriteExpression");
02472             SETERROR(-1)
02473         }
02474         first = 0;
02475         terms += *terms;
02476     }
02477 /*  AO.IsBracket = 0; */
02478     AO.IsBracket = OldIsBracket;
02479     AO.PrintType = OldPrintType;
02480     return(0);
02481 }
02482
02483 /*
02484         #] WriteExpression :
02485         #[ WriteAll :           WORD WriteAll()
02486
02487         Writes all expressions that should be written
02488 */
02489
02490 int WriteAll(void)
02491 {
02492     GETIDENTITY
02493     WORD lbrac, first;
02494     WORD *t, *stopper, n, prtf;
02495     int oldIsFortran90 = AC.IsFortran90, i;
02496     POSITION pos;
02497     FILEHANDLE *f;
02498     EXPRESSIONS e;
02499     if ( AM.exitflag ) return(0);
02500 #ifdef WITHMPI
02501     if ( PF.me != MASTER ) {
```

```
02502         /*
02503          * For the slaves, we need to call Optimize() the same number of times
02504          * as the master. The first argument doesn't have any important role.
02505          */
02506         for ( n = 0; n < NumExpressions; n++ ) {
02507             e = &Expressions[n];
02508             if ( (!e->printflag) & PRINTON ) continue;
02509             switch ( e->status ) {
02510                 case LOCALEXPRESSION:
02511                 case GLOBALEXPRESSION:
02512                 case UNHIDELEXPRESSION:
02513                 case UNHIDEGEXPRESSION:
02514                     break;
02515                 default:
02516                     continue;
02517             }
02518             e->printflag = 0;
02519             PutPreVar(AM.oldnumextrasymbols, GetPreVar((UBYTE *)"EXTRASYMBOLS_", 0), 0, 1);
02520             if ( AO.OptimizationLevel > 0 ) {
02521                 if ( Optimize(0, 1) ) return(-1);
02522             }
02523         }
02524         return(0);
02525     }
02526 #endif
02527     SeekScratch(AR.outfile,&pos);
02528     if ( ResetScratch() ) {
02529         MesCall("WriteAll");
02530         SETERROR(-1)
02531     }
02532     AO.termbuf = AT.WorkPointer;
02533     AO.bracket = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
02534     AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer*2);
02535     AO.OutFill = AO.OutputLine = (UBYTE *)AT.WorkPointer;
02536     AT.WorkPointer += 2*AC.LineLength;
02537     *(AR.CompressBuffer) = 0;
02538     first = 0;
02539     for ( n = 0; n < NumExpressions; n++ ) {
02540         if ( ( Expressions[n].printflag & PRINTON ) != 0 ) { first = 1; break; }
02541     }
02542     if ( !first ) goto EndWrite;
02543     AO.IsBracket = 0;
02544     AO.OutSkip = 3;
02545     AR.DeferFlag = 0;
02546     while ( GetTerm(BHEAD AO.termbuf) ) {
02547         t = AO.termbuf + 1;
02548         e = Expressions + AO.termbuf[3];
02549         n = e->status;
02550         if ( ( n == LOCALEXPRESSION || n == GLOBALEXPRESSION
02551         || n == UNHIDELEXPRESSION || n == UNHIDEGEXPRESSION ) &&
02552         ( ( prtf = e->printflag ) & PRINTON ) != 0 ) {
02553             e->printflag = 0;
02554             AO.NumInBrack = 0;
02555             PutPreVar(AM.oldnumextrasymbols,
02556                     GetPreVar((UBYTE *)"EXTRASYMBOLS_",0),0,1);
02557             if ( ( prtf & PRINTLFILE ) != 0 ) {
02558                 if ( AC.LogHandle < 0 ) prtf &= ~PRINTLFILE;
02559             }
02560             AO.PrintType = prtf;
02561 /*
02562             if ( AC.OutputMode == VORTRANMODE ) {
02563                 UBYTE *oldOutFill = AO.OutFill, *oldOutputLine = AO.OutputLine;
02564                 AO.OutSkip = 6;
02565                 if ( Optimize(AO.termbuf[3], 1) ) goto AboWrite;
02566                 AO.OutSkip = 3;
02567                 AO.OutFill = oldOutFill; AO.OutputLine = oldOutputLine;
02568                 FiniLine();
02569                 continue;
02570             }
02571             else
02572 */
02573             if ( AO.OptimizationLevel > 0 ) {
02574                 UBYTE *oldOutFill = AO.OutFill, *oldOutputLine = AO.OutputLine;
02575                 AO.OutSkip = 6;
02576                 if ( Optimize(AO.termbuf[3], 1) ) goto AboWrite;
02577                 AO.OutSkip = 3;
02578                 AO.OutFill = oldOutFill; AO.OutputLine = oldOutputLine;
02579                 FiniLine();
02580                 continue;
02581             }
02582             if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
02583                     AO.OutSkip = 6;
02584             FiniLine();
02585             AO.CurBufWrt = EXPRNAME(AO.termbuf[3]);
02586             TokenToLine(AO.CurBufWrt);
02587             stopper = t + t[1];
02588             t += SUBEXPSIZE;
```

```
02589             if ( t < stopper ) {
02590                 TokenToLine((UBYTE *)"(");
02591                 first = 1;
02592                 while ( t < stopper ) {
02593                     n = *t;
02594                     if ( !first ) TokenToLine((UBYTE *)",");
02595                     switch ( n ) {
02596                         case SYMTOSYM :
02597                             TokenToLine(FindSymbol(t[2]));
02598 /*                          TokenToLine(VARNAME(symbols,t[2])); */
02599                             break;
02600                         case VECTOVEC :
02601                             TokenToLine(FindVector(t[2]));
02602 /*                          TokenToLine(VARNAME(vectors,t[2] - AM.OffsetVector)); */
02603                             break;
02604                         case INDTOIND :
02605                             TokenToLine(FindIndex(t[2]));
02606 /*                          TokenToLine(VARNAME(indices,t[2] - AM.OffsetIndex)); */
02607                             break;
02608                         default :
02609                             TokenToLine(FindFunction(t[2]));
02610 /*                          TokenToLine(VARNAME(functions,t[2] - FUNCTION)); */
02611                             break;
02612                     }
02613                     t += t[1];
02614                     first = 0;
02615                 }
02616                 TokenToLine((UBYTE *)")");
02617             }
02618             TOKENTOLINE(" =","=");
02619             if ( AC.OutputMode == MATHEMATICAMODE ) {
02620                 TOKENTOLINE(" (","(");
02621             }
02622             lbrac = 0;
02623             AO.InFbrack = 0;
02624             if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
02625                 AO.FortFirst = 1;
02626             else
02627                 AO.FortFirst = 0;
02628             first = 1;
02629             if ( ( e->vflags & ISFACTORIZED ) != 0 ) {
02630                 AO.FactorMode = 1+e->numfactors;
02631                 AO.FactorNum = 0; /* Which factor are we doing. For factors that are zero */
02632             }
02633             else {
02634                 AO.FactorMode = 0;
02635             }
02636             while ( GetTerm(BHEAD AO.termbuf) ) {
02637                 WORD *m;
02638                 GETSTOP(AO.termbuf,m);
02639                 if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
02640                 && ( ( prtf & PRINTONETERM ) != 0 ) ) {}
02641                 else {
02642                     if ( first ) {
02643                         FiniLine();
02644                         IniLine(0);
02645                     }
02646                 }
02647                 if ( ( prtf & PRINTONETERM ) != 0 ) first = 0;
02648                 if ( WriteTerm(AO.termbuf,&lbrac,first,prtf,0) )
02649                     goto AboWrite;
02650                 first = 0;
02651             }
02652             if ( AO.FactorMode ) {
02653                 if ( first ) { AO.FactorNum = 1; TOKENTOLINE("   ( 0 )","  (0)") }
02654                 else TOKENTOLINE(" )",")");
02655                 for ( i = AO.FactorNum+1; i <= e->numfactors; i++ ) {
02656                     FiniLine();
02657                     IniLine(0);
02658                     TOKENTOLINE(" * ( 0 )","*(0)");
02659                 }
02660                 AO.FactorNum = e->numfactors;
02661                 if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE )
02662                     TokenToLine((UBYTE *)";");
02663             }
02664             else if ( AO.FactorMode == 0 || first ) {
02665                 if ( first ) { TOKENTOLINE(" 0","0") }
02666                 else if ( lbrac ) {
02667                     if ( ( prtf & PRINTCONTENTS ) != 0 ) PrtTerms();
02668                     TOKENTOLINE(" )",")")
02669                 }
02670                 else if ( ( prtf & PRINTCONTENTS ) != 0 ) {
02671                     TOKENTOLINE(" + 1 * ( ","+1*(")
02672                     PrtTerms();
02673                     TOKENTOLINE(" )",")")
02674                 }
02675                 if ( AC.OutputMode == MATHEMATICAMODE ) {
```

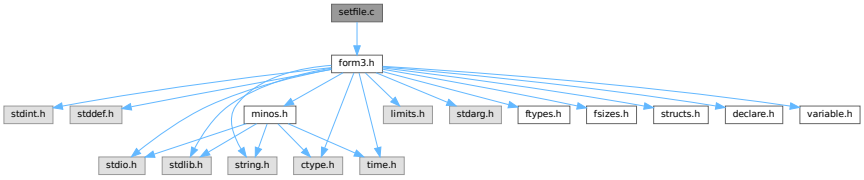
```
02676                       TokenToLine((UBYTE *)")");
02677                   }
02678                   if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE )
02679                       TokenToLine((UBYTE *)";");
02680               }
02681               AO.OutSkip = 3;
02682               AC.IsFortran90 = ISNOTFORTRAN90;
02683               FiniLine();
02684               AC.IsFortran90 = oldIsFortran90;
02685               AO.FactorMode = 0;
02686           }
02687           else {
02688               do { } while ( GetTerm(BHEAD AO.termbuf) );
02689           }
02690       }
02691       if ( AC.OutputSpaces == NORMALFORMAT ) FiniLine();
02692   EndWrite:
02693       if ( AR.infile->handle >= 0 ) {
02694           SeekFile(AR.infile->handle,&(AR.infile->filesize),SEEK_SET);
02695       }
02696       AO.IsBracket = 0;
02697       AT.WorkPointer = AO.termbuf;
02698       SetScratch(AR.infile,&pos);
02699       f = AR.outfile; AR.outfile = AR.infile; AR.infile = f;
02700       return(0);
02701   AboWrite:
02702       SetScratch(AR.infile,&pos);
02703       f = AR.outfile; AR.outfile = AR.infile; AR.infile = f;
02704       MesCall("WriteAll");
02705       Terminate(-1);
02706       return(-1);
02707   }
02708
02709   /*
02710           #] WriteAll :
02711           #[ WriteOne :           WORD WriteOne(name,alreadyinline)
02712
02713           Writes one expression from the preprocessor
02714   */
02715
02716   int WriteOne(UBYTE *name, int alreadyinline, int nosemi, WORD plus)
02717   {
02718       GETIDENTITY
02719       WORD number;
02720       WORD lbrac, first;
02721       POSITION pos;
02722       FILEHANDLE *f;
02723       WORD prf;
02724
02725       if ( GetName(AC.exprnames,name,&number,NOAUTO) != CEXPRESSION ) {
02726           MesPrint("@%s is not an expression",name);
02727           return(-1);
02728       }
02729       switch ( Expressions[number].status ) {
02730           case HIDDENLEXPRESSION:
02731           case HIDDENGEXPRESSION:
02732           case HIDELEXPRESSION:
02733           case HIDEGEXPRESSION:
02734           case UNHIDELEXPRESSION:
02735           case UNHIDEGEXPRESSION:
02736   /*
02737           case DROPHLEXPRESSION:
02738           case DROPHGEXPRESSION:
02739   */
02740               AR.GetFile = 2;
02741               break;
02742           case LOCALEXPRESSION:
02743           case GLOBALEXPRESSION:
02744           case SKIPLEXPRESSION:
02745           case SKIPGEXPRESSION:
02746   /*
02747           case DROPLEXPRESSION:
02748           case DROPGEXPRESSION:
02749   */
02750               AR.GetFile = 0;
02751               break;
02752           default:
02753               MesPrint("@expressions %s is not active. It cannot be written",name);
02754               return(-1);
02755       }
02756       SeekScratch(AR.outfile,&pos);
02757
02758       f = AR.outfile; AR.outfile = AR.infile; AR.infile = f;
02759   /*
02760       if ( ResetScratch() ) {
02761           MesCall("WriteOne");
02762           SETERROR(-1)
```

```
02763     }
02764 */
02765     if ( AR.GetFile == 2 ) f = AR.hidefile;
02766     else f = AR.infile;
02767     prf = Expressions[number].printflag;
02768     if ( plus ) prf |= PRINTONETERM;
02769 /*
02770         Now position the file
02771 */
02772     if ( f->handle >= 0 ) {
02773         SetScratch(f,&(Expressions[number].onfile));
02774     }
02775     else {
02776         f->POfill = (WORD *)((UBYTE *)(f->PObuffer)
02777                 + BASEPOSITION(Expressions[number].onfile));
02778     }
02779     AO.termbuf = AT.WorkPointer;
02780     AO.bracket = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
02781     AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer*2);
02782
02783     AO.OutFill = AO.OutputLine = (UBYTE *)AT.WorkPointer;
02784     AT.WorkPointer += 2*AC.LineLength;
02785     *(AR.CompressBuffer) = 0;
02786
02787     AO.IsBracket = 0;
02788     AO.OutSkip = 3;
02789     AR.DeferFlag = 0;
02790
02791     if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
02792             AO.OutSkip = 6;
02793     if ( GetTerm(BHEAD AO.termbuf) <= 0 ) {
02794 /* INTERNAL_ERROR_EXCL_START */
02795         MesPrint("!>ReadError in expression %s",name);
02796         goto AboWrite;
02797 /* INTERNAL_ERROR_EXCL_STOP */
02798     }
02799 /*
02800     PutPreVar(AM.oldnumextrasymbols,
02801             GetPreVar((UBYTE *)"EXTRASYMBOLS_",0),0,1);
02802 */
02803     /*
02804      * Currently WriteOne() is called only from writeToChannel() with setting
02805      * AO.OptimizationLevel = 0, which means Optimize() is never called here.
02806      * So we don't need to think about how to ensure that the master and the
02807      * slaves call Optimize() at the same time. (TU 26 Jul 2013)
02808      */
02809     if ( AO.OptimizationLevel > 0 ) {
02810         AO.OutSkip = 6;
02811         if ( Optimize(AO.termbuf[3], 1) ) goto AboWrite;
02812         AO.OutSkip = 3;
02813         FiniLine();
02814     }
02815     else {
02816         lbrac = 0;
02817         AO.InFbrack = 0;
02818         AO.FortFirst = 0;
02819         first = 1;
02820         while ( GetTerm(BHEAD AO.termbuf) ) {
02821             WORD *m;
02822             GETSTOP(AO.termbuf,m);
02823             if ( first ) {
02824                 IniLine(0);
02825                 startinline = alreadyinline;
02826                 AO.OutFill = AO.OutputLine + startinline;
02827                 if ( WriteTerm(AO.termbuf,&lbrac,first,0,0) )
02828                     goto AboWrite;
02829                 first = 0;
02830             }
02831             else {
02832                 if ( ( prf & PRINTONETERM ) != 0 ) first = 1;
02833                 if ( first ) {
02834                     FiniLine();
02835                     IniLine(0);
02836                 }
02837                 first = 0;
02838                 if ( WriteTerm(AO.termbuf,&lbrac,first,0,0) )
02839                     goto AboWrite;
02840             }
02841         }
02842         if ( first ) {
02843             IniLine(0);
02844             startinline = alreadyinline;
02845             AO.OutFill = AO.OutputLine + startinline;
02846             TOKENTOLINE(" 0","0");
02847         }
02848         else if ( lbrac ) {
02849             TOKENTOLINE(" )",")");
```

```
02850         }
02851         if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE
02852              && nosemi == 0 ) TokenToLine((UBYTE *)";");
02853         AO.OutSkip = 3;
02854         if ( AC.OutputSpaces == NORMALFORMAT && nosemi == 0 ) {
02855             FiniLine();
02856         }
02857         else {
02858             noextralinefeed = 1;
02859             FiniLine();
02860             noextralinefeed = 0;
02861         }
02862     }
02863     AO.IsBracket = 0;
02864     AT.WorkPointer = AO.termbuf;
02865     SetScratch(f,&pos);
02866     f = AR.outfile; AR.outfile = AR.infile; AR.infile = f;
02867     AO.InFbrack = 0;
02868     return(0);
02869 AboWrite:
02870     SetScratch(AR.infile,&pos);
02871     f->POposition = pos;
02872     f = AR.outfile; AR.outfile = AR.infile; AR.infile = f;
02873     MesCall("WriteOne");
02874     Terminate(-1);
02875     return(-1);
02876 }
02877
02878 /*
02879         #] WriteOne :
02880     #] schryf-Writes :
02881 */
```

4.127 setfile.c File Reference

`#include "form3.h"`
Include dependency graph for setfile.c:



Macros

- #define NUMERICALVALUE 0
- #define STRINGVALUE 1
- #define PATHVALUE 2
- #define ONOFFVALUE 3
- #define DEFINEVALUE 4
- #define SETBUFSIZE 257

Functions

- int DoSetups (void)
- int ProcessOption (UBYTE *s1, UBYTE *s2, int filetype)
- SETUPPARAMETERS * GetSetupPar (UBYTE *s)
- int RecalcSetups (void)

- int AllocSetups (void)
- void WriteSetup (void)
- SORTING * AllocSort (LONG inLargeSize, LONG inSmallSize, LONG inSmallEsize, LONG inTermsInSmall, int inMaxPatches, int inMaxFpatches, LONG inIOsize, int level)
- void AllocSortFileName (SORTING *sort)
- FILEHANDLE * AllocFileHandle (WORD par, char *name)
- void DeAllocFileHandle (FILEHANDLE *fh)
- int MakeSetupAllocs (void)
- NORMDATA * AllocNormData (void)
- int TryFileSetups (void)
- int TryEnvironment (void)

Variables

- char curdirp [ ] = "."
- char cursortdirp [ ] = "."
- char commentchar [ ] = "*"
- char dotchar [ ] = "_"
- char highfirst [ ] = "highfirst"
- char lowfirst [ ] = "lowfirst"
- char procedureextension [ ] = "prc"
- SETUPPARAMETERS setupparameters [ ]

4.127.1 Detailed Description

The routines that deal with the setup parameters.

Definition in file setfile.c.

4.127.2 Macro Definition Documentation

4.127.2.1 NUMERICALVALUE

```
#define NUMERICALVALUE 0
```

Definition at line 47 of file setfile.c.

4.127.2.2 STRINGVALUE

```
#define STRINGVALUE 1
```

Definition at line 48 of file setfile.c.

4.127.2.3 PATHVALUE

```
#define PATHVALUE 2
```

Definition at line 49 of file setfile.c.

4.127.2.4 ONOFFVALUE

```
#define ONOFFVALUE 3
```

Definition at line 50 of file setfile.c.

4.127.2.5 DEFINEVALUE

```
#define DEFINEVALUE 4
```

Definition at line 51 of file setfile.c.

4.127.2.6 SETBUFSIZE

```
#define SETBUFSIZE 257
```

Definition at line 1167 of file setfile.c.

4.127.3 Function Documentation

4.127.3.1 DoSetups()

```
int DoSetups (
            void  )
```

Definition at line 133 of file setfile.c.

4.127.3.2 ProcessOption()

```
int ProcessOption (
            UBYTE * s1,
            UBYTE * s2,
            int filetype )
```

Definition at line 183 of file setfile.c.

4.127.3.3 GetSetupPar()

```
SETUPPARAMETERS * GetSetupPar (
            UBYTE * s )
```

Definition at line 338 of file setfile.c.

4.127.3.4 RecalcSetups()

```
int RecalcSetups (
            void  )
```

Definition at line 359 of file setfile.c.

4.127.3.5 AllocSetups()

```
int AllocSetups (
            void  )
```

Definition at line 415 of file setfile.c.

4.127.3.6 WriteSetup()

```
void WriteSetup (
            void  )
```

Definition at line 812 of file setfile.c.

4.127.3.7 AllocSort()

```
SORTING * AllocSort (
            LONG inLargeSize,
            LONG inSmallSize,
            LONG inSmallEsize,
            LONG inTermsInSmall,
            int inMaxPatches,
            int inMaxFpatches,
            LONG inIOsize,
            int level )
```

Definition at line 870 of file setfile.c.

4.127.3.8 AllocSortFileName()

```
void AllocSortFileName (
            SORTING * sort )
```

Definition at line 1020 of file setfile.c.

4.127.3.9 AllocFileHandle()

```
FILEHANDLE * AllocFileHandle (
            WORD par,
            char * name )
```

Definition at line 1045 of file setfile.c.

4.127.3.10 DeAllocFileHandle()

```
void DeAllocFileHandle (
            FILEHANDLE * fh )
```

Definition at line 1106 of file setfile.c.

4.127.3.11 MakeSetupAllocs()

```
int MakeSetupAllocs (
            void  )
```

Definition at line 1123 of file setfile.c.

4.127.3.12 AllocNormData()

```
NORMDATA * AllocNormData (
            void  )
```

Definition at line 1137 of file setfile.c.

4.127.3.13 TryFileSetups()

```
int TryFileSetups (
            void  )
```

Definition at line 1169 of file setfile.c.

4.127.3.14 TryEnvironment()

```
int TryEnvironment (
            void  )
```

Definition at line 1258 of file setfile.c.

4.127.4 Variable Documentation

4.127.4.1 curdirp

```
char curdirp[] = "."
```

Definition at line 39 of file setfile.c.

4.127.4.2 cursortdirp

```
char cursortdirp[] = "."
```

Definition at line 40 of file setfile.c.

4.127.4.3 commentchar

```
char commentchar[] = "*"
```

Definition at line 41 of file setfile.c.

4.127.4.4 dotchar

```
char dotchar[] = "_"
```

Definition at line 42 of file setfile.c.

4.127.4.5 highfirst

```
char highfirst[] = "highfirst"
```

Definition at line 43 of file setfile.c.

4.127.4.6 lowfirst

```
char lowfirst[] = "lowfirst"
```

Definition at line 44 of file setfile.c.

4.127.4.7 procedureextension

```
char procedureextension[] = "prc"
```

Definition at line 45 of file setfile.c.

4.127.4.8 setupparameters

```
SETUPPARAMETERS setupparameters[]
```

Definition at line 53 of file setfile.c.

4.128 setfile.c

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
00026  *
```

```
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
00031 /*
00032     #[ Includes :
00033
00034     Routines that deal with settings and the setup file
00035 */
00036
00037 #include "form3.h"
00038
00039 char curdirp[] = ".";
00040 char cursortdirp[] = ".";
00041 char commentchar[] = "*";
00042 char dotchar[] = "_";
00043 char highfirst[] = "highfirst";
00044 char lowfirst[] = "lowfirst";
00045 char procedureextension[] = "prc";
00046
00047 #define NUMERICALVALUE 0
00048 #define STRINGVALUE 1
00049 #define PATHVALUE 2
00050 #define ONOFFVALUE 3
00051 #define DEFINEVALUE 4
00052
00053 SETUPPARAMETERS setupparameters[] =
00054 {
00055      {(UBYTE *)"bracketindexsize",      NUMERICALVALUE, 0, (LONG)MAXBRACKETBUFFERSIZE}
00056     ,{(UBYTE *)"commentchar",             STRINGVALUE, 0, (LONG)commentchar}
00057     ,{(UBYTE *)"compresssize",          NUMERICALVALUE, 0, (LONG)COMPRESSBUFFER}
00058     ,{(UBYTE *)"constindex",            NUMERICALVALUE, 0, (LONG)NUMFIXED}
00059     ,{(UBYTE *)"continuationlines",     NUMERICALVALUE, 0, (LONG)FORTRANCONTINUATIONLINES}
00060 #ifdef WITHFLOAT
00061     ,{(UBYTE *)"defaultprecision",      NUMERICALVALUE, 0, (LONG)DEFAULTPRECISION}
00062 #endif
00063     ,{(UBYTE *)"define",                 DEFINEVALUE, 0, (LONG)0}
00064     ,{(UBYTE *)"dotchar",                STRINGVALUE, 0, (LONG)dotchar}
00065     ,{(UBYTE *)"factorizationcache",    NUMERICALVALUE, 0, (LONG)FBUFFERSIZE}
00066     ,{(UBYTE *)"filepatches",           NUMERICALVALUE, 0, (LONG)MAXFPATCHES}
00067     ,{(UBYTE *)"functionlevels",        NUMERICALVALUE, 0, (LONG)MAXFLEVELS}
00068     ,{(UBYTE *)"hidesize",              NUMERICALVALUE, 0, (LONG)0}
00069     ,{(UBYTE *)"incdir",                   PATHVALUE, 0, (LONG)curdirp}
00070     ,{(UBYTE *)"indentspace",           NUMERICALVALUE, 0, (LONG)INDENTSPACE}
00071     ,{(UBYTE *)"insidefirst",             ONOFFVALUE, 0, (LONG)1}
00072     ,{(UBYTE *)"jumpratio",             NUMERICALVALUE, 0, (LONG)JUMPRATIO}
00073     ,{(UBYTE *)"largepatches",          NUMERICALVALUE, 0, (LONG)MAXPATCHES}
00074     ,{(UBYTE *)"largesize",             NUMERICALVALUE, 0, (LONG)LARGEBUFFER}
00075     ,{(UBYTE *)"maxnumbersize",         NUMERICALVALUE, 0, (LONG)0}
00076 /*  ,{(UBYTE *)"maxnumbersize",         NUMERICALVALUE, 0, (LONG)MAXNUMBERSIZE} */
00077     ,{(UBYTE *)"maxtermsize",           NUMERICALVALUE, 0, (LONG)MAXTER}
00078 #ifdef WITHFLOAT
00079     ,{(UBYTE *)"maxweight",             NUMERICALVALUE, 0, (LONG)MAXWEIGHT}
00080 #endif
00081     ,{(UBYTE *)"maxwildcards",          NUMERICALVALUE, 0, (LONG)MAXWILDC}
00082     ,{(UBYTE *)"nospacesinnumbers",       ONOFFVALUE, 0, (LONG)0}
00083     ,{(UBYTE *)"numstorecaches",        NUMERICALVALUE, 0, (LONG)NUMSTORECACHES}
00084     ,{(UBYTE *)"nwritefinalstatistics",   ONOFFVALUE, 0, (LONG)0}
00085     ,{(UBYTE *)"nwriteprocessstatistics", ONOFFVALUE, 0, (LONG)0}
00086     ,{(UBYTE *)"nwritestatistics",        ONOFFVALUE, 0, (LONG)0}
00087     ,{(UBYTE *)"nwritethreadstatistics",  ONOFFVALUE, 0, (LONG)0}
00088     ,{(UBYTE *)"oldfactarg",              ONOFFVALUE, 0, (LONG)NEWFACTARG}
00089     ,{(UBYTE *)"oldgcd",                  ONOFFVALUE, 0, (LONG)1}
00090     ,{(UBYTE *)"oldorder",                ONOFFVALUE, 0, (LONG)0}
00091     ,{(UBYTE *)"oldparallelstatistics",   ONOFFVALUE, 0, (LONG)0}
00092     ,{(UBYTE *)"oldprfsign",              ONOFFVALUE, 0, (LONG)0}
00093     ,{(UBYTE *)"parentheses",           NUMERICALVALUE, 0, (LONG)MAXPARLEVEL}
00094     ,{(UBYTE *)"path",                     PATHVALUE, 0, (LONG)curdirp}
00095     ,{(UBYTE *)"procedureextension",      STRINGVALUE, 0, (LONG)procedureextension}
00096     ,{(UBYTE *)"processbucketsize",     NUMERICALVALUE, 0, (LONG)DEFAULTPROCESSBUCKETSIZE}
00097     ,{(UBYTE *)"resettimeonclear",        ONOFFVALUE, 0, (LONG)1}
00098     ,{(UBYTE *)"scratchsize",           NUMERICALVALUE, 0, (LONG)SCRATCHSIZE}
00099     ,{(UBYTE *)"shmwinsize",            NUMERICALVALUE, 0, (LONG)SHMWINSIZE}
00100     ,{(UBYTE *)"sizestorecache",        NUMERICALVALUE, 0, (LONG)SIZESTORECACHE}
00101     ,{(UBYTE *)"smallextension",        NUMERICALVALUE, 0, (LONG)SMALLOVERFLOW}
00102     ,{(UBYTE *)"smallsize",             NUMERICALVALUE, 0, (LONG)SMALLBUFFER}
00103     ,{(UBYTE *)"sortiosize",            NUMERICALVALUE, 0, (LONG)SORTIOSIZE}
00104     ,{(UBYTE *)"sorttype",                STRINGVALUE, 0, (LONG)lowfirst}
00105     ,{(UBYTE *)"spectatorsize",         NUMERICALVALUE, 0, (LONG)SPECTATORSIZE}
00106     ,{(UBYTE *)"subfilepatches",        NUMERICALVALUE, 0, (LONG)SMAXFPATCHES}
00107     ,{(UBYTE *)"sublargepatches",       NUMERICALVALUE, 0, (LONG)SMAXPATCHES}
00108     ,{(UBYTE *)"sublargesize",          NUMERICALVALUE, 0, (LONG)SLARGEBUFFER}
00109     ,{(UBYTE *)"subsmallextension",     NUMERICALVALUE, 0, (LONG)SSMALLOVERFLOW}
00110     ,{(UBYTE *)"subsmallsize",          NUMERICALVALUE, 0, (LONG)SSMALLBUFFER}
00111     ,{(UBYTE *)"subsortiosize",         NUMERICALVALUE, 0, (LONG)SSORTIOSIZE}
00112     ,{(UBYTE *)"subtermsinsmall",       NUMERICALVALUE, 0, (LONG)STERMSSMALL}
00113     ,{(UBYTE *)"tempdir",                 STRINGVALUE, 0, (LONG)curdirp}
```

```
00114     ,{(UBYTE *)"tempsortdir",              STRINGVALUE, 0, (LONG)cursortdirp}
00115     ,{(UBYTE *)"termsinsmall",           NUMERICALVALUE, 0, (LONG)TERMSSMALL}
00116     ,{(UBYTE *)"threadbucketsize",       NUMERICALVALUE, 0, (LONG)DEFAULTTHREADBUCKETSIZE}
00117     ,{(UBYTE *)"threadloadbalancing",        ONOFFVALUE, 0, (LONG)DEFAULTTHREADLOADBALANCING}
00118     ,{(UBYTE *)"threads",                NUMERICALVALUE, 0, (LONG)DEFAULTTHREADS}
00119     ,{(UBYTE *)"threadscratchoutsize",   NUMERICALVALUE, 0, (LONG)THREADSCRATCHOUTSIZE}
00120     ,{(UBYTE *)"threadscratchsize",      NUMERICALVALUE, 0, (LONG)THREADSCRATCHSIZE}
00121     ,{(UBYTE *)"threadsortfilesynch",        ONOFFVALUE, 0, (LONG)0}
00122     ,{(UBYTE *)"totalsize",                  ONOFFVALUE, 0, (LONG)2}
00123     ,{(UBYTE *)"workspace",              NUMERICALVALUE, 0, (LONG)WORKBUFFER}
00124     ,{(UBYTE *)"wtimestats",                 ONOFFVALUE, 0, (LONG)2}
00125 };
00126
00127 /*
00128     #] Includes :
00129     #[ Setups :
00130         #[ DoSetups :
00131 */
00132
00133 int DoSetups(void)
00134 {
00135     UBYTE *setbuffer, *s, *t, *u /*, c */;
00136     int errors = 0;
00137     setbuffer = LoadInputFile((UBYTE *)setupfilename,SETUPFILE);
00138     if ( setbuffer ) {
00139 /*
00140         The contents of the file are now in setbuffer.
00141         Each line is commentary or a single command.
00142         The buffer is terminated with a zero.
00143 */
00144         s = setbuffer;
00145         while ( *s ) {
00146             if ( *s == ' ' || *s == '\t' || *s == '*' || *s == '#' || *s == '\n' ) {
00147                 while ( *s && *s != '\n' ) s++;
00148             }
00149             else if ( tolower(*s) < 'a' || tolower(*s) > 'z' ) {
00150                 t = s;
00151                 while ( *s && *s != '\n' ) s++;
00152 /*
00153                 c = *s; *s = 0;
00154                 Error1("Setup file: Illegal statement: ",t);
00155                 errors++; *s = c;
00156 */
00157             }
00158             else {
00159                 t = s; /* name of the option */
00160                 while ( tolower(*s) >= 'a' && tolower(*s) <= 'z' ) s++;
00161                 *s++ = 0;
00162                 while ( *s == ' ' || *s == '\t' ) s++;
00163                 u = s; /* 'value' of the option */
00164                 while ( *s && *s != '\n' && *s != '\r' ) s++;
00165                 if ( *s ) *s++ = 0;
00166                 errors += ProcessOption(t,u,0);
00167             }
00168             while ( *s == '\n' || *s == '\r' ) s++;
00169         }
00170         M_free(setbuffer,"setup file buffer");
00171     }
00172     if ( errors ) return(1);
00173     else return(0);
00174 }
00175
00176 /*
00177         #] DoSetups :
00178         #[ ProcessOption :
00179 */
00180
00181 static char *proop1[3] = { "Setup file", "Setups in .frm file", "Setup in environment" };
00182
00183 int ProcessOption(UBYTE *s1, UBYTE *s2, int filetype)
00184 {
00185     SETUPPARAMETERS *sp;
00186     int n, giveback = 0, error = 0;
00187     UBYTE *s, *t, *s2ret;
00188     LONG x;
00189     sp = GetSetupPar(s1);
00190     if ( sp ) {
00191 /*
00192         We check now whether there are `' variables to be looked up in the
00193         environment. This is new (30-may-2008). This is only allowed in s2.
00194 */
00195 restart:;
00196         {
00197             UBYTE *s3,*s4,*s5,*s6, c, *start;
00198             int n1,n2,n3;
00199             s = s2;
00200             while ( *s ) {
```

```
00201                 if ( *s == '\\' ) s += 2;
00202                 else if ( *s == '`' ) {
00203                     start = s; s++;
00204                     while ( *s && *s != '\'' ) {
00205                         if ( *s == '\\' ) s++;
00206                         s++;
00207                     }
00208                     if ( *s == 0 ) {
00209                         MesPrint("%s: Illegal use of ` character for parameter %s"
00210                         ,proop1[filetype],s1);
00211                         return(1);
00212                     }
00213                     c = *s; *s = 0;
00214                     s3 = (UBYTE *)getenv((char *)(start+1));
00215                     if ( s3 == 0 ) {
00216                         MesPrint("%s: Cannot find environment variable %s for parameter %s"
00217                         ,proop1[filetype],start+1,s1);
00218                         return(1);
00219 
00220                     }
00221                     *s = c; s++;
00222                     n1 = start - s2; s4 = s3; n2 = 0;
00223                     while ( *s4 ) {
00224                         if ( *s4 == '\\' ) { s4++; n2++; }
00225                         s4++; n2++;
00226                     }
00227                     s4 = s; n3 = 0;
00228                     while ( *s4 ) {
00229                         if ( *s4 == '\\' ) { s4++; n3++; }
00230                         s4++; n3++;
00231                     }
00232                     s4 = (UBYTE *)Malloc1((n1+n2+n3+1)*sizeof(UBYTE),"environment in setup");
00233                     s5 = s2; s6 = s4;
00234                     while ( n1-- > 0 ) *s6++ = *s5++;
00235                     s5 = s3;
00236                     while ( n2-- > 0 ) *s6++ = *s5++;
00237                     s5 = s;
00238                     while ( n3-- > 0 ) *s6++ = *s5++;
00239                     *s6 = 0;
00240                     if ( giveback ) M_free(s2,"environment in setup");
00241                     s2 = s4;
00242                     giveback = 1;
00243                     goto restart;
00244                 }
00245                 else s++;
00246             }
00247         }
00248         n = sp->type;
00249         s2ret = s2;
00250         switch ( n ) {
00251             case NUMERICALVALUE:
00252                     ParseNumber(x,s2);
00253                 if ( *s2 == 'K' ) { x = x * 1000; s2++; }
00254                 else if ( *s2 == 'M' ) { x = x * 1000000; s2++; }
00255                 else if ( *s2 == 'G' ) { x = x * 1000000000; s2++; }
00256                 else if ( *s2 == 'T' ) { x = x * 1000000000000; s2++; }
00257                 if ( *s2 && *s2 != ' ' && *s2 != '\t' ) {
00258                     MesPrint("%s: Numerical value expected for parameter %s"
00259                     ,proop1[filetype],s1);
00260                     error = 1; break;
00261                 }
00262                 sp->value = x;
00263                 sp->flags = USEDFLAG;
00264                 break;
00265             case STRINGVALUE:
00266                 if ( StrICmp(s1,(UBYTE *)"tempsortdir") == 0 ) AM.havesortdir = 1;
00267                 s = s2; t = s2;
00268                 while ( *s ) {
00269                     if ( *s == ' ' || *s == '\t' ) break;
00270                     if ( *s == '\\' ) s++;
00271                     *t++ = *s++;
00272                 }
00273                 *t = 0;
00274                 if ( sp->flags == USEDFLAG && sp->value != 0 )
00275                         M_free((void *)(sp->value),"Process option");
00276                 sp->value = (LONG)strDup1(s2,"Process option");
00277                 sp->flags = USEDFLAG;
00278                 break;
00279             case PATHVALUE:
00280                 if ( StrICmp(s1,(UBYTE *)"incdir") == 0 ) {
00281                     AM.IncDir = NULL;
00282                 }
00283                 else if ( StrICmp(s1,(UBYTE *)"path") == 0 ) {
00284                     if ( AM.Path ) M_free(AM.Path,"path");
00285                     AM.Path = NULL;
00286                 }
00287                 else {
```

```
00288                     MesPrint("Setups: %s not yet implemented",s1);
00289                     error = 1;
00290                     break;
00291                 }
00292                 if ( sp->flags == USEDFLAG && sp->value != 0 )
00293                     M_free((void *)(sp->value),"Process option");
00294                 sp->value = (LONG)strDup1(s2,"Process option");
00295                 sp->flags = USEDFLAG;
00296                 break;
00297             case ONOFFVALUE:
00298                 if ( tolower(*s2) == 'o' && tolower(s2[1]) == 'n'
00299                 && ( s2[2] == 0 || s2[2] == ' ' || s2[2] == '\t' ) )
00300                     sp->value = 1;
00301                 else if ( tolower(*s2) == 'o' && tolower(s2[1]) == 'f'
00302                 && tolower(s2[2]) == 'f'
00303                 && ( s2[3] == 0 || s2[3] == ' ' || s2[3] == '\t' ) )
00304                     sp->value = 0;
00305                 else {
00306                     MesPrint("%s: Unrecognized option for parameter %s: %s"
00307                     ,proop1[filetype],s1,s2);
00308                     error = 1; break;
00309                 }
00310                 sp->flags = USEDFLAG;
00311                 break;
00312             case DEFINEVALUE:
00313 /*
00314                 if ( sp->value ) M_free((UBYTE *)(sp->value),"Process option");
00315                 sp->value = (LONG)strDup1(s2,"Process option");
00316 */
00317                 if ( TheDefine(s2,2) ) error = 1;
00318                 break;
00319             default:
00320                 Error1("Error in setupparameter table for:",s1);
00321                 error = 1;
00322                 break;
00323         }
00324     }
00325     else {
00326         MesPrint("%s: Keyword not recognized: %s",proop1[filetype],s1);
00327         error = 1;
00328     }
00329     if ( giveback ) M_free(s2ret,"environment in setup");
00330     return(error);
00331 }
00332
00333 /*
00334         #] ProcessOption :
00335         #[ GetSetupPar :
00336 */
00337
00338 SETUPPARAMETERS *GetSetupPar(UBYTE *s)
00339 {
00340     int hi, med, lo, i;
00341     lo = 0;
00342     // -1: remove possibility of out-of-bounds read in StrICmp
00343     hi = sizeof(setupparameters)/sizeof(SETUPPARAMETERS) - 1;
00344     do {
00345         med = ( hi + lo ) / 2;
00346         i = StrICmp(s,(UBYTE *)setupparameters[med].parameter);
00347         if ( i == 0 ) return(setupparameters+med);
00348         if ( i < 0 ) hi = med-1;
00349         else         lo = med+1;
00350     } while ( hi >= lo );
00351     return(0);
00352 }
00353
00354 /*
00355         #] GetSetupPar :
00356         #[ RecalcSetups :
00357 */
00358
00359 int RecalcSetups(void)
00360 {
00361     SETUPPARAMETERS *sp, *sp1;
00362
00363     sp1 = GetSetupPar((UBYTE *)"threads");
00364     if ( AM.totalnumberofthreads > 1 ) sp1->value = AM.totalnumberofthreads - 1;
00365     else sp1->value = 0;
00366 /*
00367     if ( sp1->value > 0 ) AM.totalnumberofthreads = sp1->value+1;
00368     if ( AM.totalnumberofthreads == 0 ) AM.totalnumberofthreads = 1;
00369 */
00370     sp  = GetSetupPar((UBYTE *)"filepatches");
00371     if ( sp->value < AM.totalnumberofthreads-1 )
00372         sp->value = AM.totalnumberofthreads - 1;
00373
00374     sp  = GetSetupPar((UBYTE *)"smallsize");
```

```
00375     sp1 = GetSetupPar((UBYTE *)"smallextension");
00376     if ( 6*sp1->value < 7*sp->value ) sp1->value = (7*sp->value)/6;
00377     sp = GetSetupPar((UBYTE *)"termsinsmall");
00378     sp->value = ( sp->value + 15 ) & (-16L);
00379 #ifdef WITHPTHREADS
00380     {
00381     SETUPPARAMETERS *sp2;
00382     LONG totalsize, minimumsize;
00383     sp = GetSetupPar((UBYTE *)"largesize");
00384     totalsize = sp1->value+sp->value;
00385     sp2 = GetSetupPar((UBYTE *)"maxtermsize");
00386     AM.MaxTer = sp2->value*sizeof(WORD);
00387     if ( AM.MaxTer < 200*(LONG)(sizeof(WORD)) ) AM.MaxTer = 200*(LONG)(sizeof(WORD));
00388     if ( AM.MaxTer > MAXPOSITIVE - 200*(LONG)(sizeof(WORD)) ) AM.MaxTer = MAXPOSITIVE -
      200*(LONG)(sizeof(WORD));
00389     AM.MaxTer /= sizeof(WORD);
00390     AM.MaxTer *= sizeof(WORD);
00391 #ifdef WITHSORTBOTS
00392     if ( AM.totalnumberofthreads-1 > 2 ) {
00393         minimumsize = (2*(AM.totalnumberofthreads-1)-2)*(AM.MaxTer+
00394             NUMBEROFBLOCKSINSORT*MINIMUMNUMBEROFTERMS/2*AM.MaxTer);
00395     }
00396     else
00397 #endif
00398     {
00399         minimumsize = (AM.totalnumberofthreads-1)*(AM.MaxTer+
00400             NUMBEROFBLOCKSINSORT*MINIMUMNUMBEROFTERMS*AM.MaxTer);
00401     }
00402     if ( totalsize < minimumsize ) {
00403         sp->value = minimumsize - sp1->value;
00404     }
00405     }
00406 #endif
00407     return(0);
00408 }
00409
00410 /*
00411         #] RecalcSetups :
00412         #[ AllocSetups :
00413 */
00414
00415 int AllocSetups(void)
00416 {
00417     SETUPPARAMETERS *sp;
00418     LONG LargeSize, SmallSize, SmallEsize, TermsInSmall, IOsize;
00419     int MaxPatches, MaxFpatches, error = 0, i, size;
00420     UBYTE *s;
00421 #ifndef WITHPTHREADS
00422     int j;
00423 #endif
00424     sp = GetSetupPar((UBYTE *)"threads");
00425     if ( sp->value > 0 ) AM.totalnumberofthreads = sp->value+1;
00426
00427     AM.OutBuffer = (UBYTE *)Malloc1(AM.OutBufSize+1,"OutputBuffer");
00428     AP.PreAssignStack =(LONG *)Malloc1(AP.MaxPreAssignLevel*sizeof(LONG *),"PreAssignStack");
00429     for ( i = 0; i < AP.MaxPreAssignLevel; i++ ) AP.PreAssignStack[i] = 0;
00430     AC.iBuffer = (UBYTE *)Malloc1(AC.iBufferSize+1,"statement buffer");
00431     AC.iStop = AC.iBuffer + AC.iBufferSize-2;
00432     AP.preStart = (UBYTE *)Malloc1(AP.pSize,"instruction buffer");
00433     AP.preStop = AP.preStart + AP.pSize - 3;
00434     /* AP.PreIfStack is already allocated in StartPrepro(), but to be sure we
00435        "if" the freeing */
00436     if ( AP.PreIfStack ) M_free(AP.PreIfStack,"PreIfStack");
00437     AP.PreIfStack = (int *)Malloc1(AP.MaxPreIfLevel*sizeof(int),
00438                 "Preprocessor if stack");
00439     AP.PreIfStack[0] = EXECUTINGIF;
00440     sp = GetSetupPar((UBYTE *)"insidefirst");
00441     AM.ginsidefirst = AC.minsidefirst = AC.insidefirst = sp->value;
00442 /*
00443     We need to consider eliminating this variable
00444 */
00445     sp = GetSetupPar((UBYTE *)"maxtermsize");
00446     AM.MaxTer = sp->value*sizeof(WORD);
00447     if ( AM.MaxTer < 200*(LONG)(sizeof(WORD)) ) AM.MaxTer = 200*(LONG)(sizeof(WORD));
00448     if ( AM.MaxTer > MAXPOSITIVE - 200*(LONG)(sizeof(WORD)) ) AM.MaxTer = MAXPOSITIVE -
      200*(LONG)(sizeof(WORD));
00449     AM.MaxTer /= (LONG)sizeof(WORD);
00450     AM.MaxTer *= (LONG)sizeof(WORD);
00451 /*
00452     Allocate workspace.
00453 */
00454     sp = GetSetupPar((UBYTE *)"workspace");
00455     AM.WorkSize = sp->value;
00456 #ifdef WITHPTHREADS
00457 #else
00458     AT.WorkSpace = (WORD *)Malloc1(AM.WorkSize*sizeof(WORD),(char *)(sp->parameter));
00459     AT.WorkTop = AT.WorkSpace + AM.WorkSize;
```

```
00460     AT.WorkPointer = AT.WorkSpace;
00461 #endif
00462 /*
00463     Fixed indices
00464 */
00465     sp = GetSetupPar((UBYTE *)"constindex");
00466     if ( ( sp->value+100+5*WILDOFFSET ) > MAXPOSITIVE ) {
00467         MesPrint("Setting of %s in setupfile too large","constindex");
00468         AM.OffsetIndex = MAXPOSITIVE - 5*WILDOFFSET - 100;
00469         MesPrint("value corrected to maximum allowed: %d",AM.OffsetIndex);
00470     }
00471     else AM.OffsetIndex = sp->value + 1;
00472     AC.FixIndices = (WORD *)Malloc1((AM.OffsetIndex)*sizeof(WORD),(char *)(sp->parameter));
00473     AM.WilInd = AM.OffsetIndex + WILDOFFSET;
00474     AM.DumInd = AM.OffsetIndex + 2*WILDOFFSET;
00475     AM.IndDum = AM.DumInd + WILDOFFSET;
00476 #ifndef WITHPTHREADS
00477     AR.CurDum = AN.IndDum = AM.IndDum;
00478 #endif
00479     AM.mTraceDum = AM.IndDum + 2*WILDOFFSET;
00480
00481     sp = GetSetupPar((UBYTE *)"parentheses");
00482     AM.MaxParLevel = sp->value+1;
00483     AC.tokenarglevel = (WORD *)Malloc1((sp->value+1)*sizeof(WORD),(char *)(sp->parameter));
00484 /*
00485     Space during calculations
00486 */
00487     sp = GetSetupPar((UBYTE *)"maxnumbersize");
00488 /*
00489     size = ( sp->value + 11 ) & (-4);
00490     AM.MaxTal = size - 2;
00491     if ( AM.MaxTal > (AM.MaxTer/sizeof(WORD)-2)/2 )
00492                 AM.MaxTal = (AM.MaxTer/sizeof(WORD)-2)/2;
00493     if ( AM.MaxTal < (AM.MaxTer/sizeof(WORD)-2)/4 )
00494                 AM.MaxTal = (AM.MaxTer/sizeof(WORD)-2)/4;
00495 */
00496 /*
00497     There is too much confusion about MaxTal cq maxnumbersize.
00498     It seems better to fix it at its maximum value. This way we only worry
00499     about maxtermsize. This can be understood better by the 'innocent' user.
00500 */
00501     if ( sp->value == 0 ) {
00502         AM.MaxTal = (AM.MaxTer/sizeof(WORD)-2)/2;
00503     }
00504     else {
00505         size = ( sp->value + 11 ) & (-4);
00506         AM.MaxTal = size - 2;
00507         if ( (size_t)AM.MaxTal > (size_t)((AM.MaxTer/sizeof(WORD)-2)/2) )
00508                     AM.MaxTal = (AM.MaxTer/sizeof(WORD)-2)/2;
00509     }
00510     AM.MaxTal &= -sizeof(WORD)*2;
00511
00512     sp->value = AM.MaxTal;
00513     AC.cmod = (UWORD *)Malloc1(AM.MaxTal*4*sizeof(UWORD),(char *)(sp->parameter));
00514     AM.gcmod = AC.cmod + AM.MaxTal;
00515     AC.powmod = AM.gcmod + AM.MaxTal;
00516     AM.gpowmod = AC.powmod + AM.MaxTal;
00517 /*
00518     The IO buffers for the input and output expressions.
00519     Fscr[2] will be assigned in a later stage for hiding expressions from
00520     the regular action. That will make the program faster.
00521 */
00522     sp = GetSetupPar((UBYTE *)"scratchsize");
00523     AM.ScratSize = sp->value/sizeof(WORD);
00524     if ( AM.ScratSize < 4*AM.MaxTer ) AM.ScratSize = 4*AM.MaxTer;
00525     AM.HideSize = AM.ScratSize;
00526     sp = GetSetupPar((UBYTE *)"hidesize");
00527     if ( sp->value > 0 ) {
00528         AM.HideSize = sp->value/sizeof(WORD);
00529         if ( AM.HideSize < 4*AM.MaxTer ) AM.HideSize = 4*AM.MaxTer;
00530     }
00531     sp = GetSetupPar((UBYTE *)"factorizationcache");
00532     AM.fbuffersize = sp->value;
00533 #ifdef WITHPTHREADS
00534     sp = GetSetupPar((UBYTE *)"threadscratchsize");
00535     AM.ThreadScratSize = sp->value/sizeof(WORD);
00536     sp = GetSetupPar((UBYTE *)"threadscratchoutsize");
00537     AM.ThreadScratOutSize = sp->value/sizeof(WORD);
00538 #endif
00539 #ifndef WITHPTHREADS
00540     for ( j = 0; j < 2; j++ ) {
00541         WORD *ScratchBuf;
00542         ScratchBuf = (WORD *)Malloc1(AM.ScratSize*sizeof(WORD),"scratchsize");
00543         AR.Fscr[j].POsize = AM.ScratSize * sizeof(WORD);
00544         AR.Fscr[j].POfull = AR.Fscr[j].POfill = AR.Fscr[j].PObuffer = ScratchBuf;
00545         AR.Fscr[j].POstop = AR.Fscr[j].PObuffer + AM.ScratSize;
00546         PUTZERO(AR.Fscr[j].POposition);
```

```
00547     }
00548     AR.Fscr[2].PObuffer = 0;
00549 #endif
00550     sp = GetSetupPar((UBYTE *)"threadbucketsize");
00551     AC.ThreadBucketSize = AM.gThreadBucketSize = AM.ggThreadBucketSize = sp->value;
00552     sp = GetSetupPar((UBYTE *)"threadloadbalancing");
00553     AC.ThreadBalancing = AM.gThreadBalancing = AM.ggThreadBalancing = sp->value;
00554     sp = GetSetupPar((UBYTE *)"threadsortfilesynch");
00555     AC.ThreadSortFileSynch = AM.gThreadSortFileSynch = AM.ggThreadSortFileSynch = sp->value;
00556 /*
00557      The size for shared memory window for oneside MPI2 communications (unused)
00558 */
00559     sp = GetSetupPar((UBYTE *)"shmwinsize");
00560     AM.shmWinSize = sp->value/sizeof(WORD);
00561     if ( AM.shmWinSize < 4*AM.MaxTer ) AM.shmWinSize = 4*AM.MaxTer;
00562 /*
00563     The sort buffer
00564 */
00565     sp = GetSetupPar((UBYTE *)"smallsize");
00566     SmallSize = sp->value;
00567     sp = GetSetupPar((UBYTE *)"smallextension");
00568     SmallEsize = sp->value;
00569     sp = GetSetupPar((UBYTE *)"largesize");
00570     LargeSize = sp->value;
00571     sp = GetSetupPar((UBYTE *)"termsinsmall");
00572     TermsInSmall = sp->value;
00573     sp = GetSetupPar((UBYTE *)"largepatches");
00574     MaxPatches = sp->value;
00575     sp = GetSetupPar((UBYTE *)"filepatches");
00576     MaxFpatches = sp->value;
00577     sp = GetSetupPar((UBYTE *)"sortiosize");
00578     IOsize = sp->value;
00579     if ( IOsize < AM.MaxTer ) { IOsize = AM.MaxTer; sp->value = IOsize; }
00580 #ifndef WITHPTHREADS
00581 #ifdef WITHZLIB
00582     for ( j = 0; j < 2; j++ ) { AR.Fscr[j].ziosize = IOsize; }
00583 #endif
00584 #endif
00585     AM.S0 = 0;
00586     AM.S0 = AllocSort(LargeSize,SmallSize,SmallEsize,TermsInSmall
00587                     ,MaxPatches,MaxFpatches,IOsize,0);
00588     /* AM.S0->file.ziosize was already set to a (larger) value by AllocSort, here it is re-set. */
00589 #ifdef WITHZLIB
00590     AM.S0->file.ziosize = IOsize;
00591 #ifndef WITHPTHREADS
00592     AR.FoStage4[0].ziosize = IOsize;
00593     AR.FoStage4[1].ziosize = IOsize;
00594     AT.S0 = AM.S0;
00595 #endif
00596 #else
00597 #ifndef WITHPTHREADS
00598     AT.S0 = AM.S0;
00599 #endif
00600 #endif
00601 #ifndef WITHPTHREADS
00602     AR.FoStage4[0].POsize   = ((IOsize+sizeof(WORD)-1)/sizeof(WORD))*sizeof(WORD);
00603     AR.FoStage4[1].POsize   = ((IOsize+sizeof(WORD)-1)/sizeof(WORD))*sizeof(WORD);
00604 #endif
00605     sp = GetSetupPar((UBYTE *)"subsmallsize");
00606     AM.SSmallSize = sp->value;
00607     sp = GetSetupPar((UBYTE *)"subsmallextension");
00608     AM.SSmallEsize = sp->value;
00609     sp = GetSetupPar((UBYTE *)"sublargesize");
00610     AM.SLargeSize = sp->value;
00611     sp = GetSetupPar((UBYTE *)"subtermsinsmall");
00612     AM.STermsInSmall = sp->value;
00613     sp = GetSetupPar((UBYTE *)"sublargepatches");
00614     AM.SMaxPatches = sp->value;
00615     sp = GetSetupPar((UBYTE *)"subfilepatches");
00616     AM.SMaxFpatches = sp->value;
00617     sp = GetSetupPar((UBYTE *)"subsortiosize");
00618     AM.SIOsize = sp->value;
00619     /* As for IOsize, make sure SIOsize is at least as large as AM.MaxTer. */
00620     if ( AM.SIOsize < AM.MaxTer ) { AM.SIOsize = AM.MaxTer; sp->value = AM.SIOsize; }
00621     sp = GetSetupPar((UBYTE *)"spectatorsize");
00622     AM.SpectatorSize = sp->value;
00623 /*
00624     The next code is just for the moment (26-jan-1997) because we have
00625     the new parts combined with the old. Once the old parts are gone
00626     from the program, we can eliminate this code too.
00627 */
00628     sp = GetSetupPar((UBYTE *)"functionlevels");
00629     AM.maxFlevels = sp->value + 1;
00630 #ifdef WITHPTHREADS
00631 #else
00632     AT.Nest = (NESTING)Malloc1((LONG)sizeof(struct NeStInG)*AM.maxFlevels,"functionlevels");
00633     AT.NestStop = AT.Nest + AM.maxFlevels;
```

```
00634     AT.NestPoin = AT.Nest;
00635 #endif
00636
00637     sp = GetSetupPar((UBYTE *)"maxwildcards");
00638     AM.MaxWildcards = sp->value;
00639 #ifdef WITHPTHREADS
00640 #else
00641     AT.WildMask = (WORD *)Malloc1((LONG)AM.MaxWildcards*sizeof(WORD),"maxwildcards");
00642 #endif
00643
00644     sp = GetSetupPar((UBYTE *)"compresssize");
00645     if ( sp->value < 2*AM.MaxTer ) sp->value = 2*AM.MaxTer;
00646     AM.CompressSize = sp->value;
00647 #ifndef WITHPTHREADS
00648     AR.CompressBuffer = (WORD *)Malloc1((AM.CompressSize+10)*sizeof(WORD),"compresssize");
00649     AR.CompressPointer = AR.CompressBuffer;
00650     AR.ComprTop = AR.CompressBuffer + AM.CompressSize;
00651 #endif
00652     sp = GetSetupPar((UBYTE *)"bracketindexsize");
00653     if ( sp->value < 20*AM.MaxTer ) sp->value = 20*AM.MaxTer;
00654     AM.MaxBracketBufferSize = sp->value/sizeof(WORD);
00655
00656     sp = GetSetupPar((UBYTE *)"dotchar");
00657     AO.FortDotChar = ((UBYTE *)(sp->value))[0];
00658     sp = GetSetupPar((UBYTE *)"commentchar");
00659     AP.cComChar = AP.ComChar = ((UBYTE *)(sp->value))[0];
00660     sp = GetSetupPar((UBYTE *)"procedureextension");
00661 /*
00662     Check validity first.
00663 */
00664     s = (UBYTE *)(sp->value);
00665     if ( FG.cTable[*s] != 0 ) {
00666         MesPrint("  Illegal string for procedure extension %s",(UBYTE *)sp->value);
00667         error = -2;
00668     }
00669     else {
00670         s++;
00671         while ( *s ) {
00672             if ( *s == ' ' || *s == '\t' || *s == '\n' ) {
00673                 MesPrint("  Illegal string for procedure extension %s",(UBYTE *)sp->value);
00674                 error = -2;
00675                 break;
00676             }
00677             s++;
00678         }
00679     }
00680     AP.cprocedureExtension = strDup1((UBYTE *)(sp->value),"procedureExtension");
00681     AP.procedureExtension = strDup1(AP.cprocedureExtension,"procedureExtension");
00682
00683     sp = GetSetupPar((UBYTE *)"totalsize");
00684     if ( sp->value != 2 ) AM.PrintTotalSize = sp->value;
00685
00686     sp = GetSetupPar((UBYTE *)"continuationlines");
00687     AM.FortranCont = sp->value;
00688     sp = GetSetupPar((UBYTE *)"oldorder");
00689     AM.OldOrderFlag = sp->value;
00690     sp = GetSetupPar((UBYTE *)"resettimeonclear");
00691     AM.resetTimeOnClear = sp->value;
00692     sp = GetSetupPar((UBYTE *)"nospacesinnumbers");
00693     AO.NoSpacesInNumbers = AM.gNoSpacesInNumbers = AM.ggNoSpacesInNumbers = sp->value;
00694     sp = GetSetupPar((UBYTE *)"indentspace");
00695     AO.IndentSpace = AM.gIndentSpace = AM.ggIndentSpace = sp->value;
00696     sp = GetSetupPar((UBYTE *)"jumpratio");
00697     AM.jumpratio = sp->value;
00698     sp = GetSetupPar((UBYTE *)"nwritestatistics");
00699     AC.StatsFlag = AM.gStatsFlag = AM.ggStatsFlag = 1-sp->value;
00700     sp = GetSetupPar((UBYTE *)"nwritefinalstatistics");
00701     AC.FinalStats = AM.gFinalStats = AM.ggFinalStats = 1-sp->value;
00702     sp = GetSetupPar((UBYTE *)"nwritethreadstatistics");
00703     AC.ThreadStats = AM.gThreadStats = AM.ggThreadStats = 1-sp->value;
00704     sp = GetSetupPar((UBYTE *)"nwriteprocessstatistics");
00705     AC.ProcessStats = AM.gProcessStats = AM.ggProcessStats = 1-sp->value;
00706     sp = GetSetupPar((UBYTE *)"oldparallelstatistics");
00707     AC.OldParallelStats = AM.gOldParallelStats = AM.ggOldParallelStats = sp->value;
00708     sp = GetSetupPar((UBYTE *)"oldfactarg");
00709     AC.OldFactArgFlag = AM.gOldFactArgFlag = AM.ggOldFactArgFlag = sp->value;
00710     sp = GetSetupPar((UBYTE *)"oldgcd");
00711     AC.OldGCDflag = AM.gOldGCDflag = AM.ggOldGCDflag = sp->value;
00712     sp = GetSetupPar((UBYTE *)"oldprfsign");
00713     AC.OldPRFSignFlag = sp->value;
00714     sp = GetSetupPar((UBYTE *)"wtimestats");
00715     if ( sp->value == 2 ) sp->value = AM.ggWTimeStatsFlag;
00716     AC.WTimeStatsFlag = AM.gWTimeStatsFlag = AM.ggWTimeStatsFlag = sp->value;
00717 #ifdef WITHFLOAT
00718     sp = GetSetupPar((UBYTE *)"maxweight");
00719     AC.MaxWeight = AM.gMaxWeight = AM.ggMaxWeight = sp->value;
00720     sp = GetSetupPar((UBYTE *)"defaultprecision");
```

```
00721     AC.DefaultPrecision = AM.gDefaultPrecision = AM.ggDefaultPrecision = sp->value;
00722 #endif
00723     sp = GetSetupPar((UBYTE *)"sorttype");
00724     if ( StrICmp((UBYTE *)"lowfirst",(UBYTE *)sp->value) == 0 ) {
00725         AC.lSortType = SORTLOWFIRST;
00726     }
00727     else if ( StrICmp((UBYTE *)"highfirst",(UBYTE *)sp->value) == 0 ) {
00728         AC.lSortType = SORTHIGHFIRST;
00729     }
00730     else {
00731         MesPrint("  Illegal SortType specification: %s",(UBYTE *)sp->value);
00732         error = -2;
00733     }
00734
00735     sp = GetSetupPar((UBYTE *)"processbucketsize");
00736     AM.hProcessBucketSize = AM.gProcessBucketSize =
00737     AC.ProcessBucketSize = AC.mProcessBucketSize = sp->value;
00738 /*
00739     The store caches (code installed 15-aug-2006 JV)
00740 */
00741     sp = GetSetupPar((UBYTE *)"numstorecaches");
00742     AM.NumStoreCaches = sp->value;
00743     sp = GetSetupPar((UBYTE *)"sizestorecache");
00744     AM.SizeStoreCache = sp->value;
00745     /* Make sure this is a multiple of sizeof(WORD). */
00746     AM.SizeStoreCache = ((AM.SizeStoreCache+sizeof(WORD)-1)/sizeof(WORD))*sizeof(WORD);
00747 #ifndef WITHPTHREADS
00748 /*
00749     Install the store caches (15-aug-2006 JV)
00750     Note that in the case of PTHREADS this is done in InitializeOneThread
00751 */
00752     AT.StoreCache = AT.StoreCacheAlloc = 0;
00753     if ( AM.NumStoreCaches > 0 ) {
00754         STORECACHE sa, sb;
00755         size = sizeof(struct StOrEcAcHe)+AM.SizeStoreCache;
00756         size = ((size-1)/sizeof(size_t)+1)*sizeof(size_t);
00757         AT.StoreCacheAlloc = (STORECACHE)Malloc1(size*AM.NumStoreCaches,"StoreCaches");
00758         AT.StoreCache = AT.StoreCacheAlloc;
00759         sa = AT.StoreCache;
00760         for ( j = 0; j < AM.NumStoreCaches; j++ ) {
00761             sb = (STORECACHE)(void *)((UBYTE *)sa+size);
00762             if ( j == AM.NumStoreCaches-1 ) {
00763                 sa->next = 0;
00764             }
00765             else {
00766                 sa->next = sb;
00767             }
00768             SETBASEPOSITION(sa->position,-1);
00769             SETBASEPOSITION(sa->toppos,-1);
00770             sa = sb;
00771         }
00772     }
00773 #endif
00774
00775 /*
00776     And now some order sensitive things
00777 */
00778     if ( AM.Path == NULL ) {
00779         sp = GetSetupPar((UBYTE *)"path");
00780         AM.Path = strDup1((UBYTE *)(sp->value),"path");
00781     }
00782     if ( AM.IncDir == NULL ) {
00783         sp = GetSetupPar((UBYTE *)"incdir");
00784         AM.IncDir = strDup1((UBYTE *)(sp->value),"incdir");
00785     }
00786 /*
00787     if ( AM.TempDir == NULL ) {
00788         sp = GetSetupPar((UBYTE *)"tempdir");
00789         AM.TempDir = strDup1((UBYTE *)(sp->value),"tempdir");
00790     }
00791 */
00792     return(error);
00793 }
00794
00795 /*
00796         #] AllocSetups :
00797         #[ WriteSetup :
00798
00799     The routine writes the values of the setup parameters.
00800     We should do this better. (JV, 21-may-2008)
00801     The way it should be done is:
00802         a: write the raw values.
00803         b: give readjusted values.
00804         c: give derived values.
00805     Because this is a difficult subject, it would be nice to have a LaTeX
00806     document that explains this all exactly. There should then be a
00807     mechanism to poke the values of the setup into the LaTeX document.
```

```
00808     probably the easiest way is to make a file with lots of \def definitions
00809     and have that included into the LaTeX file.
00810 */
00811
00812 void WriteSetup(void)
00813 {
00814     int n = sizeof(setupparameters)/sizeof(SETUPPARAMETERS);
00815     SETUPPARAMETERS *sp;
00816     MesPrint(" The setup parameters are:");
00817     for ( sp = setupparameters; n > 0; n--, sp++ ) {
00818         switch(sp->type){
00819             case NUMERICALVALUE:
00820                 MesPrint("   %s: %l",sp->parameter,sp->value);
00821                 break;
00822             case PATHVALUE:
00823                 if ( StrICmp(sp->parameter,(UBYTE *)"path") == 0 && AM.Path ) {
00824                     MesPrint("   %s: '%s'",sp->parameter,(UBYTE *)(AM.Path));
00825                     break;
00826                 }
00827                 if ( StrICmp(sp->parameter,(UBYTE *)"incdir") == 0 && AM.IncDir ) {
00828                     MesPrint("   %s: '%s'",sp->parameter,(UBYTE *)(AM.IncDir));
00829                     break;
00830                 }
00831                 /* fall through */
00832             case STRINGVALUE:
00833                 if ( StrICmp(sp->parameter,(UBYTE *)"tempdir") == 0 && AM.TempDir ) {
00834                     MesPrint("   %s: '%s'",sp->parameter,(UBYTE *)(AM.TempDir));
00835                 }
00836                 else if ( StrICmp(sp->parameter,(UBYTE *)"tempsortdir") == 0 && AM.TempSortDir ) {
00837                     MesPrint("   %s: '%s'",sp->parameter,(UBYTE *)(AM.TempSortDir));
00838                 }
00839                 else {
00840                     MesPrint("   %s: '%s'",sp->parameter,(UBYTE *)(sp->value));
00841                 }
00842                 break;
00843             case ONOFFVALUE:
00844                 if ( sp->value == 0 )
00845                     MesPrint("   %s: OFF",sp->parameter);
00846                 else if ( sp->value == 1 )
00847                     MesPrint("   %s: ON",sp->parameter);
00848                 break;
00849             case DEFINEVALUE:
00850 /*
00851                 MesPrint("   %s: '%s'",sp->parameter,(UBYTE *)(sp->value));
00852 */
00853                 break;
00854         }
00855     }
00856     AC.SetupFlag = 0;
00857 }
00858
00859 /*
00860         #] WriteSetup :
00861         #[ AllocSort :
00862
00863         Routine allocates a complete struct for sorting.
00864         To be used for the main allocation of the sort buffers, and
00865         in a later stage for the function and subroutine sort buffers.
00866         The level arg denotes a main buffer allocation (0) or a sub-buffer
00867         allocation (1), used only for printing warning messages for buffer
00868         size adjustments in DEBUGGING mode.
00869 */
00870 SORTING *AllocSort(LONG inLargeSize, LONG inSmallSize, LONG inSmallEsize, LONG inTermsInSmall,
00871                    int inMaxPatches, int inMaxFpatches, LONG inIOsize, int level)
00872 {
00873     DUMMYUSE(level); /* This is only used in DEBUGGING mode */
00874     LONG LargeSize = inLargeSize;
00875     LONG SmallSize = inSmallSize;
00876     LONG SmallEsize = inSmallEsize;
00877     LONG TermsInSmall = inTermsInSmall;
00878     int MaxPatches = inMaxPatches;
00879     int MaxFpatches = inMaxFpatches;
00880     LONG IOsize = inIOsize;
00881
00882     LONG longer,terms2insmall,sortsize,longerp;
00883     LONG IObuffersize = IOsize;
00884     LONG IOtry;
00885     SORTING *sort;
00886     int fname2Size = 0, j = 0;
00887     char *s;
00888     if ( AM.S0 != 0 ) {
00889         s = FG.fname2; fname2Size = 0;
00890         while ( *s ) { s++; fname2Size++; }
00891         fname2Size += 16;
00892     }
00893     if ( MaxFpatches < 4 ) MaxFpatches = 4;
00894     longer = MaxPatches > MaxFpatches ? MaxPatches : MaxFpatches;
```

```
00895     longerp = longer;
00896     while ( (1 « j) < longerp ) j++;
00897     longerp = (1 « j) + 1;
00898     longerp += sizeof(WORD*) - (longerp%sizeof(WORD *));
00899     longer++;
00900     longer += sizeof(WORD*) - (longer%sizeof(WORD *));
00901     if ( SmallSize < 16*AM.MaxTer ) SmallSize = 16*AM.MaxTer+16;
00902     TermsInSmall = (TermsInSmall+15) & (-16L);
00903     terms2insmall = 2*TermsInSmall;  /* Used to be just + 100 rather than *2 */
00904     if ( SmallEsize < (SmallSize*3)/2 ) SmallEsize = (SmallSize*3)/2;
00905     if ( LargeSize > 0 && LargeSize < 2*SmallSize ) LargeSize = 2*SmallSize;
00906     SmallEsize = (SmallEsize+15) & (-16L);
00907     if ( LargeSize < 0 ) LargeSize = 0;
00908     sortsize = sizeof(SORTING);
00909     sortsize = (sortsize+15)&(-16L);
00910     IObuffersize = (IObuffersize+sizeof(WORD)-1)/sizeof(WORD);
00911 /*
00912     The next statement fixes a bug. In the rare case that we have a
00913     problem here, we expand the size of the large buffer or the
00914     small extension
00915 */
00916     if ( (ULONG)( LargeSize+SmallEsize ) < MaxFpatches*((IObuffersize
00917         +COMPINC)*sizeof(WORD)+2*AM.MaxTer) ) {
00918         if ( LargeSize == 0 )
00919             SmallEsize = MaxFpatches*((IObuffersize+COMPINC)*sizeof(WORD)+2*AM.MaxTer);
00920         else
00921             LargeSize  = MaxFpatches*((IObuffersize+COMPINC)*sizeof(WORD)+2*AM.MaxTer)
00922                 - SmallEsize;
00923     }
00924
00925     IOtry = ((LargeSize+SmallEsize)/MaxFpatches-2*AM.MaxTer)/sizeof(WORD)-COMPINC;
00926
00927     /* Here both IObuffersize and IOtry are in units of sizeof(WORD). */
00928     if ( IObuffersize < IOtry ) {
00929         IObuffersize = IOtry;
00930     }
00931
00932 #if DEBUGGING
00933     char *prefix;
00934     if ( level == 0 ) { prefix = ""; }
00935     else { prefix = "Sub"; }
00936     if ( LargeSize != inLargeSize ) { MesPrint("Warning: %sLargeSize adjusted: %l -> %l", prefix,
      inLargeSize, LargeSize); }
00937     if ( SmallSize != inSmallSize ) { MesPrint("Warning: %sSmallSize adjusted: %l -> %l", prefix,
      inSmallSize, SmallSize); }
00938     if ( SmallEsize != inSmallEsize ) {MesPrint("Warning: %sSmallEsize adjusted: %l -> %l", prefix,
      inSmallEsize, SmallEsize); }
00939     if ( TermsInSmall != inTermsInSmall ) { MesPrint("Warning: %sTermsInSmall adjusted: %l -> %l",
      prefix, inTermsInSmall, TermsInSmall); }
00940     if ( MaxPatches != inMaxPatches ) { MesPrint("Warning: MaxPatches adjusted: %d -> %d",
      inMaxPatches, MaxPatches); }
00941     if ( MaxFpatches != inMaxFpatches ) {MesPrint("Warning: MaxFPatches adjusted: %d -> %d",
      inMaxFpatches, MaxFpatches); }
00942     /* This one is always changed if the LargeSize has not been... */
00943     /* if ( IObuffersize != inIOsize/(LONG)sizeof(WORD) ) { MesPrint("Warning: IOsize adjusted: %l ->
      %l", inIOsize/sizeof(WORD), IObuffersize); } */
00944 #endif
00945
00946     /* Allocate separate buffers for most struct members, for better testing and debugging with
      valgrind. */
00947     sort = Malloc1(sizeof(*sort), "AllocSort: sorting struct");
00948
00949     sort->LargeSize = LargeSize/sizeof(WORD);
00950     sort->SmallSize = SmallSize/sizeof(WORD);
00951     sort->SmallEsize = SmallEsize/sizeof(WORD);
00952     sort->MaxPatches = MaxPatches;
00953     sort->MaxFpatches = MaxFpatches;
00954     sort->TermsInSmall = TermsInSmall;
00955     sort->Terms2InSmall = terms2insmall;
00956
00957     sort->sPointer     = Malloc1(sizeof(*(sort->sPointer    ))*terms2insmall, "AllocSort: sPointer");
00958     sort->Patches      = Malloc1(sizeof(*(sort->Patches     ))*longer, "AllocSort: Patches");
00959     sort->pStop        = Malloc1(sizeof(*(sort->pStop       ))*longer, "AllocSort: pStop");
00960     sort->poina        = Malloc1(sizeof(*(sort->poina       ))*longerp, "AllocSort: poina");
00961     sort->poin2a       = Malloc1(sizeof(*(sort->poin2a      ))*longerp, "AllocSort: poin2a");
00962
00963     sort->fPatches     = Malloc1(sizeof(*(sort->fPatches    ))*longer, "AllocSort: fPatches");
00964     sort->fPatchesStop = Malloc1(sizeof(*(sort->fPatchesStop))*longer, "AllocSort: fPatchesStop");
00965     sort->inPatches    = Malloc1(sizeof(*(sort->inPatches   ))*longer, "AllocSort: inPatches");
00966     sort->tree         = Malloc1(sizeof(*(sort->tree        ))*longerp, "AllocSort: tree");
00967     sort->used         = Malloc1(sizeof(*(sort->used        ))*longerp, "AllocSort: used");
00968
00969 #ifdef WITHZLIB
00970     sort->fpcompressed   = Malloc1(sizeof(*(sort->fpcompressed  ))*(longerp+2), "AllocSort:
      fpcompressed");
00971     sort->fpincompressed = Malloc1(sizeof(*(sort->fpincompressed))*(longerp+2), "AllocSort:
      fpincompressed");
```

```
00972     sort->zsparray = 0;
00973 #endif
00974
00975     sort->ktoi         = Malloc1(sizeof(*(sort->ktoi))*(longerp+2), "AllocSort: ktoi");
00976
00977     // The combined Large buffer and Small buffer (+ extension) are used.
00978     // They must be allocated together.
00979     sort->lBuffer      = Malloc1(sizeof(*(sort->lBuffer))*(sort->LargeSize+sort->SmallEsize),
      "AllocSort: lBuffer+sBuffer");
00980     sort->lTop = sort->lBuffer+sort->LargeSize;
00981
00982     sort->sBuffer = sort->lTop;
00983     if ( sort->LargeSize == 0 ) { sort->lBuffer = 0; sort->lTop = 0; }
00984     sort->sTop = sort->sBuffer + sort->SmallSize;
00985     sort->sTop2 = sort->sBuffer + sort->SmallEsize;
00986     sort->sHalf = sort->sBuffer + (LONG)((sort->SmallSize+sort->SmallEsize)»1);
00987
00988     sort->file.POobuffer = Malloc1(IObuffersize*sizeof(*(sort->file.POobuffer))+fname2Size+16,
      "AllocSort: POobuffer");
00989     sort->file.POstop = sort->file.POobuffer+IObuffersize;
00990     sort->file.POsize = IObuffersize * sizeof(WORD);
00991     sort->file.POfill = sort->file.POfull = sort->file.POobuffer;
00992     sort->file.active = 0;
00993     sort->file.handle = -1;
00994     PUTZERO(sort->file.POposition);
00995 #ifdef WITHPTHREADS
00996     sort->file.pthreadslock = dummylock;
00997 #endif
00998 #ifdef WITHZLIB
00999     sort->file.ziosize = IObuffersize*sizeof(WORD);
01000     sort->file.ziobuffer = 0;
01001 #endif
01002     if ( AM.S0 != 0 ) {
01003         sort->file.name = (char *)(sort->file.POobuffer + IObuffersize);
01004         AllocSortFileName(sort);
01005     }
01006     else sort->file.name = 0;
01007     sort->cBuffer = 0;
01008     sort->cBufferSize = 0;
01009     sort->f = 0;
01010     sort->PolyWise = 0;
01011
01012     return(sort);
01013 }
01014
01015 /*
01016         #] AllocSort :
01017         #[ AllocSortFileName :
01018 */
01019
01020 void AllocSortFileName(SORTING *sort)
01021 {
01022     GETIDENTITY
01023     char *s, *t;
01024 /*
01025         This is not the allocation before the tempfiles are determined.
01026         Hence we can use the name in FG.fname2 and modify the tail
01027 */
01028     s = FG.fname2; t = sort->file.name;
01029     while ( *s ) *t++ = *s++;
01030 #ifdef WITHPTHREADS
01031     t[-2] = 'F';
01032     snprintf(t-1,FG.fname2size-(t-1-sort->file.name),"%d.%d",identity,AN.filenum);
01033 #else
01034     t[-2] = 'f';
01035     snprintf(t-1,FG.fname2size-(t-1-sort->file.name),"%d",AN.filenum);
01036 #endif
01037     AN.filenum++;
01038 }
01039
01040 /*
01041         #] AllocSortFileName :
01042         #[ AllocFileHandle :
01043 */
01044
01045 FILEHANDLE *AllocFileHandle(WORD par,char *name)
01046 {
01047     GETIDENTITY
01048     LONG allocation, Ssize;
01049     FILEHANDLE *fh;
01050     int i = 0;
01051     char *s, *t;
01052
01053     s = FG.fname2; i = 0;
01054     while ( *s ) { s++; i++; }
01055     if ( par == 0 ) { i += 16; Ssize = AM.SIOsize; }
01056     else { s = name; while ( *s ) { i++; s++; } i+= 2; Ssize = AM.SpectatorSize; }
```

```
01057
01058     allocation = sizeof(FILEHANDLE) + (Ssize+1)*sizeof(WORD) + i*sizeof(char)
01059               +FG.fname2size+20;
01060     fh = (FILEHANDLE *)Malloc1(allocation,"FileHandle");
01061
01062     fh->PObuffer = (WORD *)(fh+1);
01063     fh->POstop = fh->PObuffer+Ssize;
01064     fh->POsize = Ssize * sizeof(WORD);
01065     fh->active = 0;
01066     fh->handle = -1;
01067     PUTZERO(fh->POposition);
01068 #ifdef WITHPTHREADS
01069     fh->pthreadslock = dummylock;
01070 #endif
01071     if ( par == 0 ) {   /* sort file */
01072       if ( AM.S0 != 0 ) {
01073         fh->name = (char *)(fh->POstop + 1);
01074         s = FG.fname2; t = fh->name;
01075         while ( *s ) *t++ = *s++;
01076 #ifdef WITHPTHREADS
01077         t[-2] = 'F';
01078         snprintf(t-1,FG.fname2size-(t-1-fh->name),"%d-%d",identity,AN.filenum);
01079 #else
01080         t[-2] = 'f';
01081         snprintf(t-1,FG.fname2size-(t-1-fh->name),"%d",AN.filenum);
01082 #endif
01083         AN.filenum++;
01084       }
01085       else fh->name = 0;
01086     }
01087     else {  /* Spectator file */
01088         fh->name = (char *)(fh->POstop + 1);
01089         s = FG.fname; t = fh->name;
01090         for ( i  = 0; i < FG.fnamebase; i++ ) *t++ = *s++;
01091         s = name;
01092         while ( *s ) *t++ = *s++;
01093         *t = 0;
01094     }
01095     fh->POfill = fh->POfull = fh->PObuffer;
01096     return(fh);
01097 }
01098
01099 /*
01100         #] AllocFileHandle :
01101         #[ DeAllocFileHandle :
01102
01103         Made to repair deallocation of AN.filenum. 21-sep-2000
01104 */
01105
01106 void DeAllocFileHandle(FILEHANDLE *fh)
01107 {
01108     GETIDENTITY
01109     if ( fh->handle >= 0 ) {
01110         CloseFile(fh->handle);
01111         fh->handle = -1;
01112         remove(fh->name);
01113     }
01114     AN.filenum--; /* free namespace. was forgotten in first reading */
01115     M_free(fh,"Temporary FileHandle");
01116 }
01117
01118 /*
01119         #] DeAllocFileHandle :
01120         #[ MakeSetupAllocs :
01121 */
01122
01123 int MakeSetupAllocs(void)
01124 {
01125     if ( RecalcSetups() || AllocSetups() ) return(1);
01126     else return(0);
01127 }
01128
01129 /*
01130         #] MakeSetupAllocs :
01131         #[ AllocNormData :
01132
01133     Allocate the arrays within the NORMDATA struct. These are dynamic,
01134     such that valgrind can detect buffer overruns in these arrays.
01135 */
01136
01137 NORMDATA* AllocNormData(void)
01138 {
01139     NORMDATA* tmp = Malloc1(sizeof(NORMDATA), "NormData struct");
01140
01141     tmp->psym = Malloc1(7*NORMSIZE*sizeof(*(tmp->psym)), "Normalize struct psym");
01142     tmp->pvec = Malloc1(1*NORMSIZE*sizeof(*(tmp->pvec)), "Normalize struct pvec");
01143     tmp->pdot = Malloc1(3*NORMSIZE*sizeof(*(tmp->pdot)), "Normalize struct pdot");
```

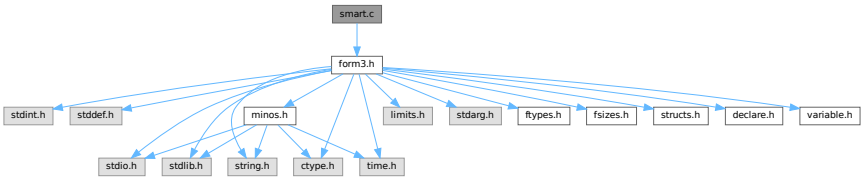
```
01144     tmp->pdel = Malloc1(2*NORMSIZE*sizeof(*(tmp->pdel)), "Normalize struct pdel");
01145     tmp->pind = Malloc1(1*NORMSIZE*sizeof(*(tmp->pind)), "Normalize struct pind");
01146     tmp->peps = Malloc1(NORMSIZE/3*sizeof(*(tmp->peps)), "Normalize struct peps");
01147     tmp->pden = Malloc1(NORMSIZE/3*sizeof(*(tmp->pden)), "Normalize struct pden");
01148     tmp->pcom = Malloc1(1*NORMSIZE*sizeof(*(tmp->pcom)), "Normalize struct pcom");
01149     tmp->pnco = Malloc1(1*NORMSIZE*sizeof(*(tmp->pnco)), "Normalize struct pnco");
01150     tmp->pcon = Malloc1(2*NORMSIZE*sizeof(*(tmp->pcon)), "Normalize struct pcon");
01151
01152     return tmp;
01153 }
01154
01155 /*
01156         #] AllocNormData :
01157         #[ TryFileSetups :
01158
01159         Routine looks in the input file for a start of the type
01160         [#-]
01161         #: setupparameter value
01162         It keeps looking until the first line that does not start with
01163         #-, #+ or #:
01164         Then it rewinds the input.
01165 */
01166
01167 #define SETBUFSIZE 257
01168
01169 int TryFileSetups(void)
01170 {
01171     LONG oldstreamposition;
01172     int oldstream;
01173     int error = 0, eqnum;
01174     int oldNoShowInput = AC.NoShowInput;
01175     UBYTE buff[SETBUFSIZE+1], *s, *t, *u, *settop, c;
01176     LONG linenum, prevline;
01177     AP.FoundFileSetupCount = 0;
01178
01179     if ( AC.CurrentStream == 0 ) return(error);
01180     oldstream = AC.CurrentStream - AC.Streams;
01181     oldstreamposition = GetStreamPosition(AC.CurrentStream);
01182     linenum = AC.CurrentStream->linenumber;
01183     prevline = AC.CurrentStream->prevline;
01184     eqnum = AC.CurrentStream->eqnum;
01185     AC.NoShowInput = 1;
01186     settop = buff + SETBUFSIZE;
01187     if ( AC.CurrentStream->type == INPUTSTREAM && oldstreamposition == 0 )
      AC.CurrentStream->fileposition = 0;
01188     for(;;) {
01189         c = GetInput();
01190         if ( c == '*' || c == '\n' ) {
01191             while ( c != '\n' && c != ENDOFINPUT ) c = GetInput();
01192             if ( c == ENDOFINPUT ) goto eoi;
01193             continue;
01194         }
01195         if ( c == ENDOFINPUT ) goto eoi;
01196         if ( c != '#' ) break;
01197         c = GetInput();
01198         if ( c == ENDOFINPUT ) goto eoi;
01199         if ( c != '-' && c != '+' && c != ':' ) break;
01200         if ( c != ':' ) {
01201             while ( c != '\n' && c != ENDOFINPUT ) c = GetInput();
01202             continue;
01203         }
01204         // Count the number of file setup parameters found
01205         AP.FoundFileSetupCount++;
01206         s = buff;
01207         while ( ( c = GetInput() ) == ' ' || c == '\t' || c == '\r' ) {}
01208         if ( c == ENDOFINPUT ) break;
01209         if ( c == LINEFEED ) continue;
01210         if ( c == 0 || c == ENDOFINPUT ) break;
01211         while ( c != LINEFEED && c != ENDOFINPUT ) {
01212             *s++ = c;
01213             c = GetInput();
01214             if ( c != LINEFEED && c != '\r' ) continue;
01215             if ( s >= settop ) {
01216                 while ( c != '\n' && c != ENDOFINPUT ) c = GetInput();
01217                 MesPrint("Setups in .frm file: Line too long. setup ignored");
01218                 error++; goto nextline;
01219             }
01220         }
01221         *s++ = '\n';
01222         t = s = buff; /* name of the option */
01223         while ( tolower(*s) >= 'a' && tolower(*s) <= 'z' ) s++;
01224         if ( *s != '\n' ) {
01225             *s++ = 0;
01226             while ( *s == ' ' || *s == '\t' ) s++;
01227             u = s; /* 'value' of the option */
01228             while ( *s && *s != '\n' && *s != '\r' ) s++;
01229             if ( *s ) *s++ = 0;
```

```
01230         }
01231         else {
01232             /* The value is empty. */
01233             u = s;
01234             *s++ = 0;
01235         }
01236         error += ProcessOption(t,u,1);
01237 nextline:;
01238     }
01239     AC.NoShowInput = oldNoShowInput;
01240     AC.CurrentStream = AC.Streams + oldstream;
01241     PositionStream(AC.CurrentStream,oldstreamposition);
01242     AC.CurrentStream->linenumber = linenum;
01243     AC.CurrentStream->prevline = prevline;
01244     AC.CurrentStream->eqnum = eqnum;
01245     ClearPushback();
01246     if ( AC.CurrentStream->type == INPUTSTREAM ) AC.CurrentStream->fileposition = -1;
01247     return(error);
01248 eoi:
01249     MesPrint("Input file without a program.");
01250     return(-1);
01251 }
01252
01253 /*
01254         #] TryFileSetups :
01255         #[ TryEnvironment :
01256 */
01257
01258 int TryEnvironment(void)
01259 {
01260     char *s, *t, *u, varname[100];
01261     int i,imax = sizeof(setupparameters)/sizeof(SETUPPARAMETERS);
01262     int error = 0;
01263     varname[0] = 'F'; varname[1] = 'O'; varname[2] = 'R'; varname[3] = 'M';
01264     varname[4] = '_'; varname[5] = 0;
01265     for ( i = 0; i < imax; i++ ) {
01266         t = s = (char *)(setupparameters[i].parameter);
01267         u = varname+5;
01268         while ( *s ) { *u++ = (char)(toupper((unsigned char)*s)); s++; }
01269         *u = 0;
01270         s = (char *)(getenv(varname));
01271         if ( s ) {
01272             error += ProcessOption((UBYTE *)t,(UBYTE *)s,2);
01273         }
01274     }
01275     return(error);
01276 }
01277
01278 /*
01279         #] TryEnvironment :
01280     #] Setups :
01281 */
```

4.129 smart.c File Reference

`#include "form3.h"`

Include dependency graph for smart.c:



Functions

- int StudyPattern (WORD *lhs)
- int MatchIsPossible (WORD *pattern, WORD *term)

4.129.1 Detailed Description

The functions for smart pattern searches in combinations of functions. When many wildcards are involved and the functions are (anti)symmetric an exhaustive search for all possibilities may take very much time (like factorial in the number of wildcards) while a human can often see immediately that there cannot be a match. The routines here try to make FORM a bit smarter in this respect.

This is just the beginning. It still needs lots of work!

Definition in file smart.c.

4.129.2 Function Documentation

4.129.2.1 StudyPattern()

```
int StudyPattern (
            WORD * lhs )
```

Definition at line 62 of file smart.c.

4.129.2.2 MatchIsPossible()

```
int MatchIsPossible (
            WORD * pattern,
            WORD * term )
```

Definition at line 299 of file smart.c.

4.130 smart.c

Go to the documentation of this file.

```
00001
00012 /* #[ License : */
00013 /*
00014  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00015  *   When using this file you are requested to refer to the publication
00016  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00017  *   This is considered a matter of courtesy as the development was paid
00018  *   for by FOM the Dutch physics granting agency and we would like to
00019  *   be able to track its scientific use to convince FOM of its value
00020  *   for the community.
00021  *
00022  *   This file is part of FORM.
00023  *
00024  *   FORM is free software: you can redistribute it and/or modify it under the
00025  *   terms of the GNU General Public License as published by the Free Software
00026  *   Foundation, either version 3 of the License, or (at your option) any later
00027  *   version.
00028  *
00029  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00030  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00031  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00032  *   details.
00033  *
00034  *   You should have received a copy of the GNU General Public License along
00035  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00036  */
00037 /* #] License : */
00038 /*
00039     #[ Includes : function.c
00040 */
```

```
00041
00042 #include "form3.h"
00043
00044 /*
00045     #] Includes :
00046     #[ StudyPattern :
00047
00048     Argument is a complete lhs of an id-statement
00049     Its last word is a zero (new convention(18-may-1997)) to indicate
00050     that no extra information is following. We can add there information
00051     about the pattern that will help during the actual pattern matching.
00052     Currently the WorkPointer points to just after this lhs.
00053
00054     Task of this routine: To study the functions, their symmetry properties
00055     and their wildcards to determine in which order the functions can
00056     be matched best. If the order should be different we can change it here.
00057
00058     Problem encountered 22-dec-2008 (JV): we don't take noncommuting
00059     functions into account.
00060 */
00061
00062 int StudyPattern(WORD *lhs)
00063 {
00064     GETIDENTITY
00065     WORD *fullproto, *pat, *p, *p1, *p2, *pstop, *info, f, nn;
00066     int numfun = 0, numsym = 0, allwilds = 0, i, j, k, nc;
00067     FUN_INFO *finf, *fmin, *f1, *f2, funscratch;
00068
00069     fullproto = lhs + IDHEAD;
00070 /*  if ( *lhs == TYPEIF ) fullproto--; */
00071     pat = fullproto + fullproto[1];
00072     info = pat + *pat;
00073
00074     p = pat + 1;
00075     while ( p < info ) {
00076         if ( *p >= FUNCTION ) {
00077             numfun++;
00078             nn = *p - FUNCTION;
00079             if ( nn >= WILDOFFSET ) nn -= WILDOFFSET;
00080 /*
00081             We check here for cases that are not allowed like ?a inside
00082             symmetric functions or tensors.
00083 */
00084             if ( ( functions[nn].symmetric == SYMMETRIC ) ||
00085                  ( functions[nn].symmetric == ANTISYMMETRIC ) ) {
00086               p2 = p+p[1]; p1 = p+FUNHEAD;
00087               if ( functions[nn].spec > 0 ) {
00088                 while ( p1 < p2 ) {
00089                     if ( *p1 == FUNNYWILD ) {
00090                         MesPrint("&Argument field wildcards are not allowed inside (anti)symmetric
      functions or tensors");
00091                         return(1);
00092                     }
00093                     p1++;
00094                 }
00095               }
00096               else {
00097                 while ( p1 < p2 ) {
00098                     if ( *p1 == -ARGWILD ) {
00099                         MesPrint("&Argument field wildcards are not allowed inside (anti)symmetric
      functions or tensors");
00100                         return(1);
00101                     }
00102                     NEXTARG(p1);
00103                 }
00104               }
00105             }
00106         }
00107         p += p[1];
00108     }
00109     if ( numfun == 0 ) return(0);
00110     if ( ( lhs[2] & SUBMASK ) == SUBALL ) {
00111         p = pat + 1;
00112         while ( p < info ) {
00113             if ( *p == SYMBOL || *p == VECTOR || *p == DOTPRODUCT || *p == INDEX ) {
00114                 MesPrint("&id,all can have only functions and/or tensors in the lhs.");
00115                 return(1);
00116             }
00117             p += p[1];
00118         }
00119     }
00120 /*
00121     We need now some room for the information about the functions
00122 */
00123     if ( numfun > AN.numfuninfo ) {
00124         if ( AN.FunInfo ) M_free(AN.FunInfo,"funinfo");
00125         AN.numfuninfo = numfun + 10;
```

```
00126         AN.FunInfo = (FUN_INFO *)Malloc1(AN.numfuninfo*sizeof(FUN_INFO),"funinfo");
00127     }
00128 /*
00129     Now collect the information. First the locations.
00130 */
00131     p = pat + 1; i = 0;
00132     while ( p < info ) {
00133         if ( *p >= FUNCTION ) AN.FunInfo[i++].location = p;
00134         p += p[1];
00135     }
00136     for ( i = 0, finf = AN.FunInfo; i < numfun; i++, finf++ ) {
00137         p = finf->location;
00138         pstop = p + p[1];
00139         f = *p;
00140         if ( f > FUNCTION+WILDOFFSET ) f -= WILDOFFSET;
00141         finf->numargs = finf->numfunnies = finf->numwildcards = 0;
00142         finf->symmet = functions[f-FUNCTION].symmetric;
00143         finf->tensor = functions[f-FUNCTION].spec;
00144         finf->commute = functions[f-FUNCTION].commute;
00145         if ( finf->tensor >= TENSORFUNCTION ) {
00146             p += FUNHEAD;
00147             while ( p < pstop ) {
00148                 if ( *p == FUNNYWILD ) {
00149                     finf->numfunnies++; p+= 2; continue;
00150                 }
00151                 else if ( *p < 0 ) {
00152                     if ( *p >= AM.OffsetVector + WILDOFFSET && *p < MINSPEC ) {
00153                         finf->numwildcards++;
00154                     }
00155                 }
00156                 else {
00157                     if ( *p >= AM.OffsetIndex + WILDOFFSET &&
00158                     *p <= AM.OffsetIndex + 2*WILDOFFSET ) finf->numwildcards++;
00159                 }
00160                 finf->numargs++;
00161                 p++;
00162             }
00163         }
00164         else {
00165             p += FUNHEAD;
00166             while ( p < pstop ) {
00167                 if ( *p > 0 ) { finf->numargs++; p += *p; continue; }
00168                 if ( *p <= -FUNCTION ) {
00169                     if ( *p <= -FUNCTION - WILDOFFSET ) finf->numwildcards++;
00170                     p++;
00171                 }
00172                 else if ( *p == -SYMBOL ) {
00173                     if ( p[1] >= 2*MAXPOWER ) finf->numwildcards++;
00174                     p += 2;
00175                 }
00176                 else if ( *p == -INDEX ) {
00177                     if ( p[1] >= AM.OffsetIndex + WILDOFFSET &&
00178                     p[1] <= AM.OffsetIndex + 2*WILDOFFSET ) finf->numwildcards++;
00179                     p += 2;
00180                 }
00181                 else if ( *p == -VECTOR || *p == -MINVECTOR ) {
00182                     if ( p[1] >= AM.OffsetVector + WILDOFFSET && p[1] < MINSPEC ) {
00183                         finf->numwildcards++;
00184                     }
00185                     p += 2;
00186                 }
00187                 else if ( *p == -ARGWILD ) {
00188                     finf->numfunnies++;
00189                     p += 2;
00190                 }
00191                 else { p += 2; }
00192                 finf->numargs++;
00193             }
00194         }
00195         if ( finf->symmet ) {
00196             numsym++;
00197             allwilds += finf->numwildcards + finf->numfunnies;
00198         }
00199     }
00200     if ( numsym == 0 ) return(0);
00201     if ( allwilds == 0 ) return(0);
00202 /*
00203     We have the information in the array AN.FunInfo.
00204     We sort things and then write the sorted pattern.
00205     Of course we may not play with the order of the noncommuting functions.
00206     Of course we have to become even smarter in the future and look during
00207     the sorting which wildcards are assigned when.
00208     But for now this should do.
00209 */
00210     for ( nc = numfun-1; nc >= 0; nc-- ) { if ( AN.FunInfo[nc].commute ) break; }
00211
00212     finf = AN.FunInfo;
```

```
00213     for ( i = nc+2; i < numfun; i++ ) {
00214         fmin = finf; finf++;
00215         if ( ( finf->symmet < fmin->symmet ) || (
00216         ( finf->symmet == fmin->symmet ) &&
00217         ( ( finf->numwildcards+finf->numfunnies < fmin->numwildcards+fmin->numfunnies )
00218         || ( ( finf->numwildcards+finf->numfunnies == fmin->numwildcards+fmin->numfunnies )
00219         && ( finf->numwildcards < fmin->numfunnies ) ) ) ) ) {
00220             funscratch = AN.FunInfo[i];
00221             AN.FunInfo[i] = AN.FunInfo[i-1];
00222             AN.FunInfo[i-1] = funscratch;
00223             for ( j = i-1; j > nc && j > 0; j-- ) {
00224                 f1 = AN.FunInfo+j;
00225                 f2 = f1-1;
00226                 if ( ( f1->symmet < f2->symmet ) || (
00227                 ( f1->symmet == f2->symmet ) &&
00228                 ( ( f1->numwildcards+f1->numfunnies < f2->numwildcards+f2->numfunnies )
00229                 || ( ( f1->numwildcards+f1->numfunnies == f2->numwildcards+f2->numfunnies )
00230                 && ( f1->numwildcards < f2->numfunnies ) ) ) ) ) {
00231                     funscratch = AN.FunInfo[j];
00232                     AN.FunInfo[j] = AN.FunInfo[j-1];
00233                     AN.FunInfo[j-1] = funscratch;
00234                 }
00235                 else break;
00236             }
00237         }
00238     }
00239 /*
00240     Now we rewrite the pattern. First into the space after it and then we
00241     copy it back. Be careful with the non-commutative functions. There the
00242     worst one should decide.
00243 */
00244     p = pat + 1;
00245     p2 = info;
00246     for ( i = 0; i < numfun; i++ ) {
00247         if ( i == nc ) {
00248             for ( k = 0; k <= nc; k++ ) {
00249                 if ( AN.FunInfo[k].commute ) {
00250                     p1 = AN.FunInfo[k].location; j = p1[1];
00251                     NCOPY(p2,p1,j)
00252                 }
00253             }
00254         }
00255         else if ( AN.FunInfo[i].commute == 0 ) {
00256             p1 = AN.FunInfo[i].location; j = p1[1];
00257             NCOPY(p2,p1,j)
00258         }
00259     }
00260     p = pat + 1; p1 = info;
00261     while ( p1 < p2 ) *p++ = *p1++;
00262 /*
00263     And now we place the relevant information in the info part
00264 */
00265     p2 = info+1;
00266     for ( i = 0; i < numfun; i++ ) {
00267         if ( i == nc ) {
00268             for ( k = 0; k <= nc; k++ ) {
00269                 if ( AN.FunInfo[k].commute ) {
00270                     finf = AN.FunInfo + k;
00271                     *p2++ = finf->numargs;
00272                     *p2++ = finf->numwildcards;
00273                     *p2++ = finf->numfunnies;
00274                     *p2++ = finf->symmet;
00275                 }
00276             }
00277         }
00278         else if ( AN.FunInfo[i].commute == 0 ) {
00279             finf = AN.FunInfo + i;
00280             *p2++ = finf->numargs;
00281             *p2++ = finf->numwildcards;
00282             *p2++ = finf->numfunnies;
00283             *p2++ = finf->symmet;
00284         }
00285     }
00286     *info = p2-info;
00287     lhs[1] = p2-lhs;
00288     return(0);
00289 }
00290
00291 /*
00292     #] StudyPattern :
00293     #[ MatchIsPossible :
00294
00295     We come here when there are functions and there is nontrivial
00296     symmetry related wildcarding.
00297 */
00298
00299 int MatchIsPossible(WORD *pattern, WORD *term)
```

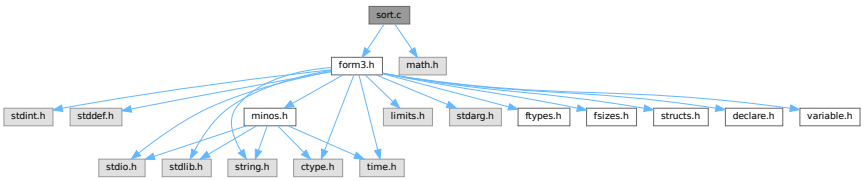
```
00300 {
00301     GETIDENTITY
00302     WORD *info = pattern + *pattern;
00303     WORD *t, *tstop, *tt, *inf, *p;
00304     int numfun = 0, inpat, i, j, numargs;
00305     FUN_INFO *finf;
00306 /*
00307     We need a list of functions and their number of arguments
00308 */
00309     GETSTOP(term,tstop);
00310     t = term + 1;
00311     while ( t < tstop ) {
00312         if ( *t >= FUNCTION ) numfun++;
00313         t += t[1];
00314     }
00315     if ( numfun == 0 ) goto NotPossible;
00316     if ( *info == SETSET ) info += info[1];
00317     inpat = (*info-1)/4;
00318     if ( inpat > numfun ) goto NotPossible;
00319 /*
00320     We need now some room for the information about the functions
00321 */
00322     if ( numfun > AN.numfuninfo ) {
00323         if ( AN.FunInfo ) M_free(AN.FunInfo,"funinfo");
00324         AN.numfuninfo = numfun + 10;
00325         AN.FunInfo = (FUN_INFO *)Malloc1(AN.numfuninfo*sizeof(FUN_INFO),"funinfo");
00326     }
00327     t = term + 1; finf = AN.FunInfo;
00328     while ( t < tstop ) {
00329         if ( *t >= FUNCTION ) {
00330             finf->location = t;
00331             if ( functions[*t-FUNCTION].spec >= TENSORFUNCTION ) {
00332                 numargs = t[1]-FUNHEAD;
00333                 t += t[1];
00334             }
00335             else {
00336                 numargs = 0;
00337                 tt = t + t[1];
00338                 t += FUNHEAD;
00339                 while ( t < tt ) {
00340                     numargs++;
00341                     NEXTARG(t)
00342                 }
00343             }
00344             finf->numargs = numargs;
00345             finf++;
00346         }
00347         else t += t[1];
00348     }
00349 /*
00350     Now we first find a partner for each function in the pattern
00351     with a fixed number of arguments
00352 */
00353     for ( i = 0, inf = info+1, p = pattern+1; i < inpat; i++, inf += 4, p+=p[1] ) {
00354         if ( inf[2] ) continue;
00355         if ( *p >= (FUNCTION+WILDOFFSET) ) continue;
00356         for ( j = 0, finf = AN.FunInfo; j < numfun; j++, finf++ ) {
00357             if ( *p == *(finf->location) && *inf == finf->numargs ) {
00358                 finf->numargs = -finf->numargs-1;
00359                 break;
00360             }
00361         }
00362         if ( j >= numfun ) goto NotPossible;
00363     }
00364     for ( i = 0, inf = info+1, p = pattern+1; i < inpat; i++, inf += 4, p+=p[1] ) {
00365         if ( inf[2] ) continue;
00366         if ( *p < (FUNCTION+WILDOFFSET) ) continue;
00367         for ( j = 0, finf = AN.FunInfo; j < numfun; j++, finf++ ) {
00368             if ( *inf == finf->numargs ) {
00369                 finf->numargs = -finf->numargs-1;
00370                 break;
00371             }
00372         }
00373         if ( j >= numfun ) goto NotPossible;
00374     }
00375     for ( i = 0, inf = info+1, p = pattern+1; i < inpat; i++, inf += 4, p+=p[1] ) {
00376         if ( inf[2] == 0 ) continue;
00377         if ( *p >= (FUNCTION+WILDOFFSET) ) continue;
00378         for ( j = 0, finf = AN.FunInfo; j < numfun; j++, finf++ ) {
00379             if ( *p == *(finf->location) && (*inf-inf[2]) <= finf->numargs ) {
00380                 finf->numargs = -finf->numargs-1;
00381                 break;
00382             }
00383         }
00384         if ( j >= numfun ) goto NotPossible;
00385     }
00386     for ( i = 0, inf = info+1, p = pattern+1; i < inpat; i++, inf += 4, p+=p[1] ) {
```

```
00387         if ( inf[2] == 0 ) continue;
00388         if ( *p < (FUNCTION+WILDOFFSET) ) continue;
00389         for ( j = 0, finf = AN.FunInfo; j < numfun; j++, finf++ ) {
00390             if ( (*inf-inf[2]) <= finf->numargs ) {
00391                 finf->numargs = -finf->numargs-1;
00392                 break;
00393             }
00394         }
00395         if ( j >= numfun ) goto NotPossible;
00396     }
00397 /*
00398     Thus far we have determined that for each function in the pattern
00399     there is a potential partner with enough arguments.
00400     For now the rest is up to the real pattern matcher.
00401
00402     To undo the disabling of the number of arguments we need this code
00403     for ( i = 0, finf = AN.FunInfo; i < numfun; i++, finf++ ) {
00404         if ( finf->numargs < 0 ) finf->numargs = -finf->numargs-1;
00405     }
00406 */
00407     return(1);
00408 NotPossible:
00409 /*
00410     PrintTerm(pattern,"p");
00411     PrintTerm(term,"t");
00412 */
00413     return(0);
00414 }
00415
00416 /*
00417     #] MatchIsPossible :
00418 */
```

4.131 sort.c File Reference

```
#include "form3.h"
#include <math.h>
```

Include dependency graph for sort.c:



Macros

- #define NEWSPLITMERGE
- #define NEWSPLITMERGETIMSORT
- #define HUMANSTRLEN 12
- #define HUMANSUFFLEN 4
- #define HUMANSUFFSTRLEN 4
- #define COL_EXP 16
- #define COL_SPA 8
- #define COL_EQU 42
- #define COL_VAL 53
- #define INSLENGTH(x) w[1] = FUNHEAD+ARGHEAD+x; w[FUNHEAD] = ARGHEAD+x;

Functions

- void HumanString (char *string, float input, const float scale, const char suffix[HUMANSUFFLEN][HUMANSUFFSTRLEN])
- WORD DigitsIn (LONG x)
- void WriteStats (POSITION *plspace, WORD par, WORD checkLogType)
- int NewSort (PHEAD0)
- LONG EndSort (PHEAD WORD *buffer, int par)
- LONG PutIn (FILEHANDLE *file, POSITION *position, WORD *buffer, WORD **take, int npat)
- int Sflush (FILEHANDLE *fi)
- WORD PutOut (PHEAD WORD *term, POSITION *position, FILEHANDLE *fi, WORD ncomp)
- int FlushOut (POSITION *position, FILEHANDLE *fi, int compr)
- int AddCoef (PHEAD WORD **ps1, WORD **ps2)
- int AddPoly (PHEAD WORD **ps1, WORD **ps2)
- void AddArgs (PHEAD WORD *s1, WORD *s2, WORD *m)
- WORD Compare1 (PHEAD WORD *term1, WORD *term2, WORD level)
- WORD CompareSymbols (PHEAD WORD *term1, WORD *term2, WORD par)
- WORD CompareHSymbols (PHEAD WORD *term1, WORD *term2, WORD par)
- LONG ComPress (WORD **ss, LONG *n)
- LONG SplitMerge (PHEAD WORD **Pointer, LONG number)
- void GarbHand (void)
- int MergePatches (WORD par)
- int StoreTerm (PHEAD WORD *term)
- void StageSort (FILEHANDLE *fout)
- int SortWild (WORD *w, WORD nw)
- void CleanUpSort (int num)
- void LowerSortLevel (void)
- WORD * PolyRatFunSpecial (PHEAD WORD *t1, WORD *t2)
- void SimpleSplitMergeRec (WORD *array, WORD num, WORD *auxarray)
- void SimpleSplitMerge (WORD *array, WORD num)
- WORD BinarySearch (WORD *array, WORD num, WORD x)

Variables

- char * toterms [ ] = { " ", " >>", "-->" }
- const char humanTermsSuffix [HUMANSUFFLEN][HUMANSUFFSTRLEN] = {"K ","M ","B ","T "}
- const char humanBytesSuffix [HUMANSUFFLEN][HUMANSUFFSTRLEN] = {"KiB","MiB","GiB","TiB"}

4.131.1 Detailed Description

Contains the sort routines. We distinguish levels of sorting. The ground level is the sorting of terms in an expression. When a term has functions, the arguments can contain terms that need sorting, which this then done by raising the level. This can happen recursively. NewSort and EndSort automatically raise and lower the level. Because the ground level does some special things, sometimes we have to raise immediately to the second level skipping the ground level.

Special routines for the parallel sorting are in the file threads.c Also the sorting of terms in polynomials is special but most of that is controlled by changing the address of the compare routine. Other routines relevant for adding rational polynomials are in the file polynito.c

Definition in file sort.c.

4.131.2 Macro Definition Documentation

4.131.2.1 NEWSPLITMERGE

```
#define NEWSPLITMERGE
```

Definition at line 53 of file sort.c.

4.131.2.2 NEWSPLITMERGETIMSORT

```
#define NEWSPLITMERGETIMSORT
```

Definition at line 55 of file sort.c.

4.131.2.3 HUMANSTRLEN

```
#define HUMANSTRLEN 12
```

Definition at line 84 of file sort.c.

4.131.2.4 HUMANSUFFLEN

```
#define HUMANSUFFLEN 4
```

Definition at line 85 of file sort.c.

4.131.2.5 HUMANSUFFSTRLEN

```
#define HUMANSUFFSTRLEN 4
```

Definition at line 86 of file sort.c.

4.131.2.6 COL_EXP

```
#define COL_EXP 16
```

Definition at line 107 of file sort.c.

4.131.2.7 COL_SPA

```
#define COL_SPA 8
```

Definition at line 108 of file sort.c.

4.131.2.8 COL_EQU

```
#define COL_EQU 42
```

Definition at line 109 of file sort.c.

4.131.2.9 COL_VAL

```
#define COL_VAL 53
```

Definition at line 110 of file sort.c.

4.131.2.10 INSLENGTH

```
#define INSLENGTH(
              x ) w[1] = FUNHEAD+ARGHEAD+x; w[FUNHEAD] = ARGHEAD+x;
```

Definition at line 2089 of file sort.c.

4.131.3 Function Documentation

4.131.3.1 HumanString()

```
void HumanString (
             char * string,
             float input,
             const float scale,
             const char suffix[HUMANSUFFLEN][HUMANSUFFSTRLEN] )
```

Definition at line 89 of file sort.c.

4.131.3.2 DigitsIn()

```
WORD DigitsIn (
             LONG x )
```

Definition at line 112 of file sort.c.

4.131.3.3 WriteStats()

```
void WriteStats (
             POSITION * plspace,
             WORD par,
             WORD checkLogType )
```

Writes the statistics.

Parameters

plspace	The size in bytes currently occupied
par	par = 0 = STATSSPLITMERGE after a splitmerge. par = 1 = STATSMERGETOFILE after merge to sortfile. par = 2 = STATSPOSTSORT after the sort
checkLogType	== CHECKLOGTYPE: The output should not go to the log file if AM.LogType is 1.

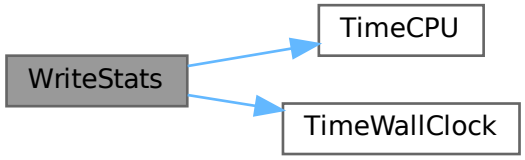
current expression is to be found in AR.CurExpr. terms are in S->TermsLeft. S->GenTerms.

Definition at line 138 of file sort.c.

References TimeCPU(), and TimeWallClock().

Referenced by EndSort(), PF_Processor(), and StoreTerm().

Here is the call graph for this function:



4.131.3.4 NewSort()

```
int NewSort (
            PHEAD0  )
```

Starts a new sort. At the lowest level this is a 'main sort' with the struct according to the parameters in S0. At higher levels this is a sort for functions, subroutines or dollars. We prepare the arrays and structs.

Returns

Regular convention (OK -> 0)

Definition at line 397 of file sort.c.

Referenced by generate_expression(), get_expression(), InFunction(), MakeDollarInteger(), MakeDollarMod(), PF_CollectModifiedDollars(), PF_Processor(), poly_factorize_expression(), poly_sort(), poly_unfactorize_expression(), PolyFunMul(), Processor(), TakeArgContent(), and TestMatch().

4.131.3.5 EndSort()

```
LONG EndSort (
              PHEAD WORD * buffer,
              int par )
```

Finishes a sort. At AR.sLevel == 0 the output is to the regular output stream. When AR.sLevel $>$ 0, the parameter par determines the actual output. The AR.sLevel will be popped. All ongoing stages are finished and if the sortfile is open it is closed. The statistics are printed when AR.sLevel == 0 par == 0 Output to the buffer. par == 1 Sort for function arguments. The output will be copied into the buffer. It is assumed that this is in the WorkSpace. par == 2 Sort for $-variable. We return the address of the buffer that contains the output in buffer (treated like WORD $**$). We first catch the output in a file (unless we can intercept things after the small buffer has been sorted) Then we read from the file into a buffer. Only when par == 0 data compression can be attempted at AT.SS==AT.S0.

Parameters

buffer	buffer for output when needed
par	See above

Returns

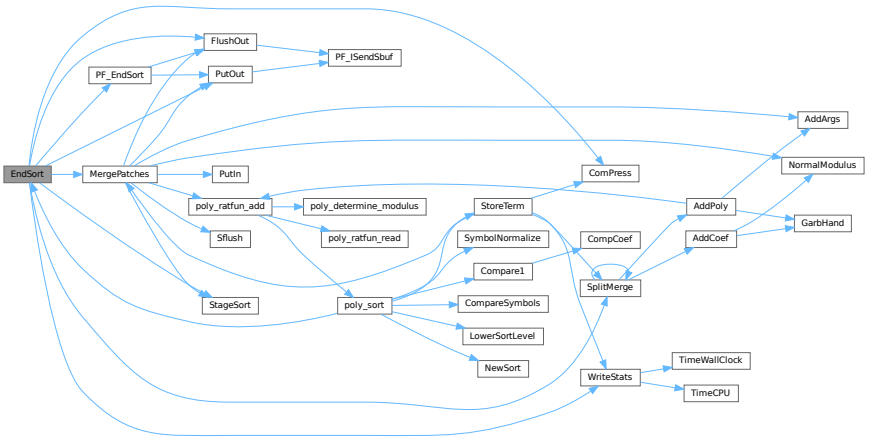
If negative: error. If positive: number of words in output.

Definition at line 488 of file sort.c.

References ComPress(), FlushOut(), MergePatches(), PF_EndSort(), PutOut(), SplitMerge(), StageSort(), and WriteStats().

Referenced by generate_expression(), get_expression(), InFunction(), MakeDollarInteger(), MakeDollarMod(), PF_CollectModifiedDollars(), PF_Processor(), poly_factorize_expression(), poly_sort(), poly_unfactorize_expression(), PolyFunMul(), Processor(), TakeArgContent(), and TestMatch().

Here is the call graph for this function:



4.131.3.6 PutIn()

```
LONG PutIn (
            FILEHANDLE * file,
            POSITION * position,
            WORD * buffer,
            WORD ** take,
            int npat )
```

Reads a new patch from position in file handle. It is put at buffer, anything after take is moved forward. This would be part of a term that hasn't been used yet. Because of this there should be some space before the start of the buffer

Parameters

file	The file system from which to read
position	The position from which to read
buffer	The buffer into which to read
take	The unused tail should be moved before the buffer
npat	The number of the patch. Is needed if the information was compressed with gzip, because each patch has its own independent gzip encoding.

Definition at line 1065 of file sort.c.

References FiLe::handle.

Referenced by MergePatches().

4.131.3.7 Sflush()

```
int Sflush (
            FILEHANDLE * fi )
```

Puts the contents of a buffer to output Only to be used when there is a single patch in the large buffer.

Parameters

fi	The filesystem (or its cache) to which the patch should be written

Definition at line 1131 of file sort.c.

References FiLe::handle.

Referenced by MergePatches().

4.131.3.8 PutOut()

```
WORD PutOut (
             PHEAD WORD * term,
             POSITION * position,
             FILEHANDLE * fi,
             WORD ncomp )
```

Routine writes one term to file handle at position. It returns the new value of the position.

NOTE: For 'final output' we may have to index the brackets. See the struct BRACKETINDEX. We should maintain: 1: a list with brackets array with the brackets 2: a list of objects of type BRACKETINDEX. It contains array with either pointers or offsets to the list of brackets. starting positions in the file. The index may be tied to a maximum size. In that case we may have to prune the list occasionally.

Parameters

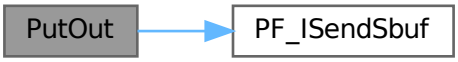
term	The term to be written
position	The position in the file. Afterwards it is updated
fi	The file (or its cache) to which should be written
ncomp	Information about what type of compression should be used

Definition at line 1217 of file sort.c.

References FiLe::handle, and PF_ISendSbuf().

Referenced by EndSort(), generate_expression(), MergePatches(), PF_EndSort(), PF_Processor(), and Processor().

Here is the call graph for this function:



4.131.3.9 FlushOut()

```
int FlushOut (
              POSITION * position,
              FILEHANDLE * fi,
              int compr )
```

Completes output to an output file and writes the trailing zero.

Parameters

position	The position in the file after writing
fi	The file (or its cache)
compr	Indicates whether there should be compression with gzip.

Returns

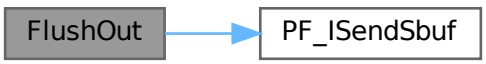
Regular conventions (OK -> 0).

Definition at line 1581 of file sort.c.

References FiLe::handle, BrAcKeTiNfO::indexbuffer, and PF_ISendSbuf().

Referenced by EndSort(), MergePatches(), PF_EndSort(), and Processor().

Here is the call graph for this function:



4.131.3.10 AddCoef()

```
int AddCoef (
              PHEAD WORD ** ps1,
              WORD ** ps2 )
```

Adds the coefficients of the terms ∗ps1 and ∗ps2. The problem comes when there is not enough space for a new longer coefficient. First a local solution is tried. If this is not successful we need to move terms around. The possibility of a garbage collection should not be ignored, as avoiding this costs very much extra space which is nearly wasted otherwise.

If the return value is zero the terms cancelled.

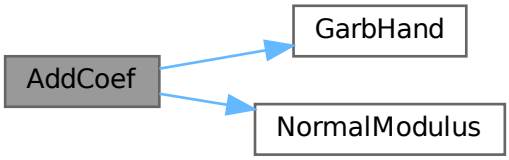
The resulting term is left in ∗ps1.

Definition at line 1795 of file sort.c.

References GarbHand(), and NormalModulus().

Referenced by SplitMerge().

Here is the call graph for this function:



4.131.3.11 AddPoly()

```
int AddPoly (
              PHEAD WORD ** ps1,
              WORD ** ps2 )
```

Routine should be called when S->PolyWise != 0. It points then to the position of AR.PolyFun in both terms.

We add the contents of the arguments of the two polynomials. Special attention has to be given to special arguments. We have to reserve a space equal to the size of one term + the size of the argument of the other. The addition has to be done in this routine because not all objects are reentrant.

Newer addition (12-nov-2007). The PolyFun can have two arguments. In that case S->PolyFlag is 2 and we have to call the routine for adding rational polynomials. We have to be rather careful what happens with: The location of the output The order of the terms in the arguments At first we allow only univariate polynomials in the PolyFun. This restriction will be lifted a.s.a.p.

Parameters

ps1	A pointer to the position of the first term
ps2	A pointer to the position of the second term

Returns

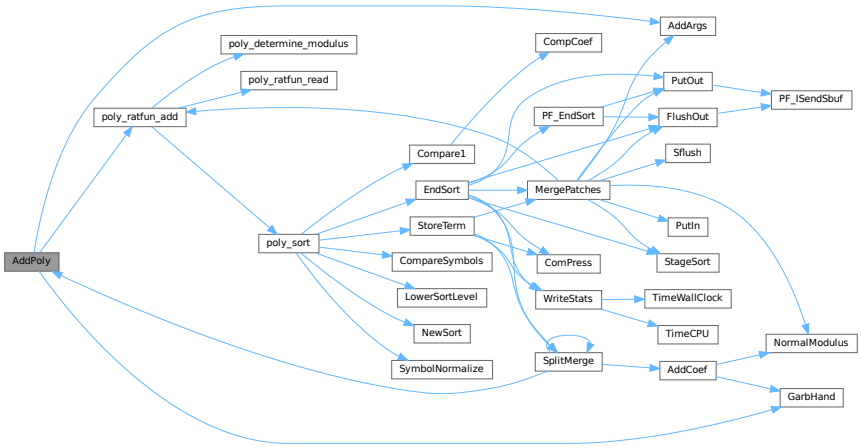
If zero the terms cancel. Otherwise the new term is in ∗ps1.

Definition at line 1925 of file sort.c.

References AddArgs(), GarbHand(), and poly_ratfun_add().

Referenced by SplitMerge().

Here is the call graph for this function:



4.131.3.12 AddArgs()

```
void AddArgs (
            PHEAD WORD * s1,
            WORD * s2,
            WORD * m )
```

Adds the arguments of two occurrences of the PolyFun.

Parameters

s1	Pointer to the first occurrence.
s2	Pointer to the second occurrence.
m	Pointer to where the answer should be.

Definition at line 2098 of file sort.c.

Referenced by AddPoly(), and MergePatches().

4.131.3.13 Compare1()

```
WORD Compare1 (
            PHEAD WORD * term1,
            WORD * term2,
            WORD level )
```

Compares two terms. The answer is: 0 equal (with exception of the coefficient if level == 0.) >0 term1 comes first. <0 term2 comes first. Some special precautions may be needed to keep the CompCoef routine from generating overflows, although this is very unlikely in subterms. This routine should not return an error condition.

Originally this routine was called Compare. With the treatment of special polynomials with terms that contain only symbols and the need for extreme speed for the polynomial routines we made a special compare routine and now we store the address of the current compare routine in AR.CompareRoutine and have a macro Compare which makes all existing code work properly and we can just replace the routine on a thread by thread basis (each thread has its own AR struct).

Parameters

term1	First input term
term2	Second input term
level	The sorting level (may influence on the result)

Returns

0 equal (with exception of the coefficient if level == 0.) >0 term1 comes first. <0 term2 comes first.

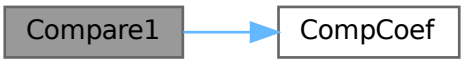
When there are floating point numbers active (float_ = FLOATFUN) the presence of one or more float_ functions is returned in AT.SortFloatMode: 0: no float_ 1: float_ in term1 only 2: float_ in term2 only 3: float_ in both terms

Definition at line 2393 of file sort.c.

References CompCoef().

Referenced by InFunction(), poly_sort(), and StartVariables().

Here is the call graph for this function:



4.131.3.14 CompareSymbols()

```
WORD CompareSymbols (
             PHEAD WORD * term1,
             WORD * term2,
             WORD par )
```

Compares the terms, based on the value of AN.polysortflag. If term1 < term2 the return value is -1 If term1 > term2 the return value is 1 If term1 = term2 the return value is 0 The coefficients may differ. The terms contain only a single subterm of type SYMBOL. If AN.polysortflag = 0 it is a 'regular' compare. If AN.polysortflag = 1 the sum of the powers is more important par is a dummy parameter to make the parameter field identical to that of Compare1 which is the regular compare routine in sort.c

Definition at line 2856 of file sort.c.

Referenced by InFunction(), and poly_sort().

4.131.3.15 CompareHSymbols()

```
WORD CompareHSymbols (
             PHEAD WORD * term1,
             WORD * term2,
             WORD par )
```

Compares terms that can have only SYMBOL and HAAKJE subterms. If term1 < term2 the return value is -1 If term1 > term2 the return value is 1 If term1 = term2 the return value is 0 par is a dummy parameter to make the parameter field identical to that of Compare1 which is the regular compare routine in sort.c

Definition at line 2906 of file sort.c.

4.131.3.16 ComPress()

```
LONG ComPress (
             WORD ** ss,
             LONG * n )
```

Gets a list of pointers to terms and compresses the terms. In n it collects the number of terms and the return value of the function is the space that is occupied.

We have to pay some special attention to the compression of terms with a PolyFun. This PolyFun should occur only straight before the coefficient, so we can use the same trick as for the coefficient to sabotage compression of this object (Replace in the history the function pointer by zero. This is safe, because terms that would be identical otherwise would have been added).

Parameters

ss	Array of pointers to terms to be compressed.
n	Number of pointers in ss.

Returns

Total number of words needed for the compressed result.

Definition at line 2959 of file sort.c.

Referenced by EndSort(), and StoreTerm().

4.131.3.17 SplitMerge()

```
LONG SplitMerge (
            PHEAD WORD ** Pointer,
            LONG number )
```

Algorithm by J.A.M.Vermaseren (31-7-1988)

Note that AN.SplitScratch and AN.InScratch are used also in GarbHand

Merge sort in memory. The input is an array of pointers. Sorting is done recursively by dividing the array in two equal parts and calling SplitMerge for each. When the parts are small enough we can do the compare and take the appropriate action. An addition is that we look for 'runs'. Sequences that are already ordered. This happens a lot when there is very little action in a module. This made FORM faster by a few percent.

Parameters

Pointer	The array of pointers to the terms to be sorted.
number	The number of pointers in Pointer.

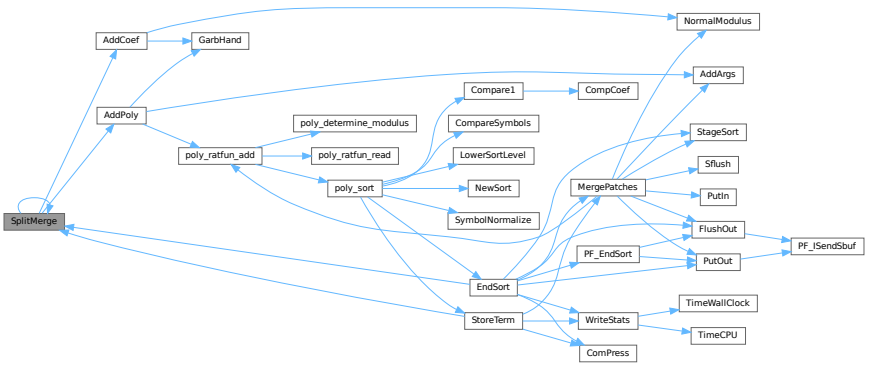
The terms are supposed to be sitting in the small buffer and there is supposed to be an extension to this buffer for when there are two terms that should be added and the result takes more space than each of the original terms. The notation guarantees that the result never needs more space than the sum of the spaces of the original terms.

Definition at line 3125 of file sort.c.

References AddCoef(), AddPoly(), and SplitMerge().

Referenced by EndSort(), SplitMerge(), and StoreTerm().

Here is the call graph for this function:



4.131.3.18 GarbHand()

```
void GarbHand (
            void  )
```

Garbage collection that takes place when the small extension is full and we need to place more terms there. When this is the case there are many holes in the small buffer and the whole can be compactified. The major complication is the buffer for SplitMerge. There are to options for temporary memory: 1: find some buffer that has enough space (maybe in the large buffer). 2: allocate a buffer. Give it back afterwards of course. If the small extension is properly dimensioned this routine should be called very rarely. Most of the time it will be called when the polyfun or polyratfun is active.

Definition at line 3405 of file sort.c.

Referenced by AddCoef(), and AddPoly().

4.131.3.19 MergePatches()

```
int MergePatches (
            WORD par )
```

The general merge routine. Can be used for the large buffer and the file merging. The array S->Patches tells where the patches start S->pStop tells where they end (has to be computed first). The end of a 'line to be merged' is indicated by a zero. If the end is reached without running into a zero or a term runs over the boundary of a patch it is a file merging operation and a new piece from the file is read in.

Parameters

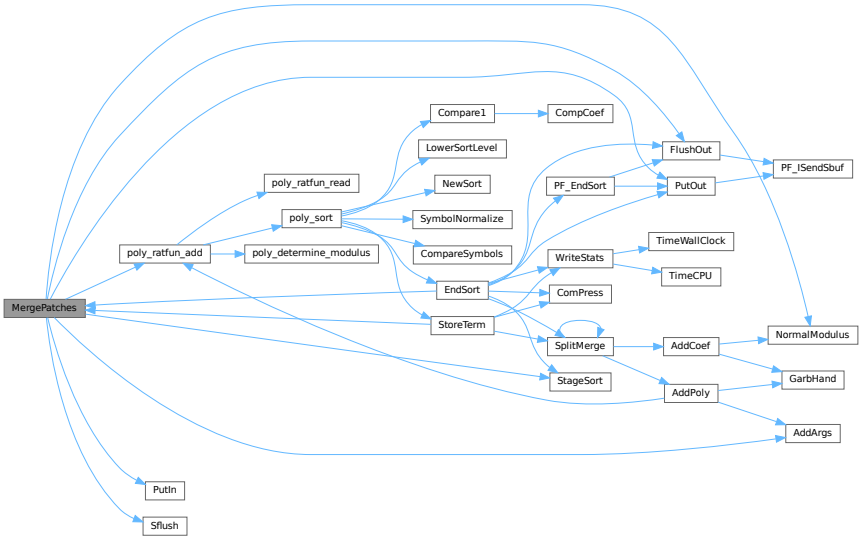
par	If par == 0 the sort is for file -> outputfile. If par == 1 the sort is for large buffer -> sortfile. If par == 2 the sort is for large buffer -> outputfile.

Definition at line 3520 of file sort.c.

References AddArgs(), FlushOut(), FiLe::handle, NormalModulus(), poly_ratfun_add(), PutIn(), PutOut(), Sflush(), and StageSort().

Referenced by EndSort(), and StoreTerm().

Here is the call graph for this function:



4.131.3.20 StoreTerm()

```
int StoreTerm (
            PHEAD WORD * term )
```

The central routine to accept terms, store them and keep things at least partially sorted. A call to EndSort will then complete storing and sorting.

Parameters

term	The term to be stored

Returns

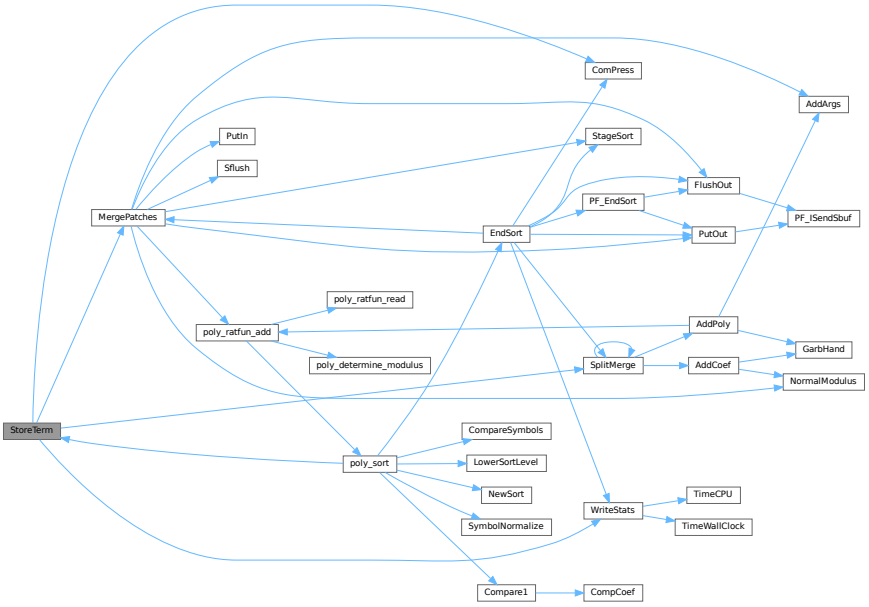
Regular return conventions (OK -> 0)

Definition at line 4311 of file sort.c.

References ComPress(), MergePatches(), SplitMerge(), and WriteStats().

Referenced by Generator(), get_expression(), InFunction(), PF_Processor(), poly_sort(), PolyFunMul(), Processor(), and TakeArgContent().

Here is the call graph for this function:



4.131.3.21 StageSort()

```
void StageSort (
            FILEHANDLE * fout )
```

Prepares a stage 4 or higher sort. Stage 4 sorts occur when the sort file contains more patches than can be merged in one pass.

Definition at line 4437 of file sort.c.

References FiLe::handle.

Referenced by EndSort(), and MergePatches().

4.131.3.22 SortWild()

```
int SortWild (
            WORD * w,
            WORD nw )
```

Sorts the wildcard entries in the parameter w. Double entries are removed. Full space taken is nw words. Routine serves for the reading of wildcards in the compiler. The entries come in the format: (type,4,number,0) in which the zero is reserved for the future replacement of 'number'.

Parameters

w	buffer with wildcard entries.
nw	number of wildcard entries.

Returns

Normal conventions (OK -> 0)

Definition at line 4536 of file sort.c.

4.131.3.23 CleanUpSort()

```
void CleanUpSort (
            int num )
```

Partially or completely frees function sort buffers.

Definition at line 4631 of file sort.c.

4.131.3.24 LowerSortLevel()

```
void LowerSortLevel (
            void  )
```

Lowers the level in the sort system.

Definition at line 4731 of file sort.c.

Referenced by generate_expression(), get_expression(), InFunction(), PF_CollectModifiedDollars(), PF_Processor(), poly_factorize_expression(), poly_sort(), poly_unfactorize_expression(), PolyFunMul(), Processor(), and TestMatch().

4.131.3.25 PolyRatFunSpecial()

```
WORD * PolyRatFunSpecial (
            PHEAD WORD * t1,
            WORD * t2 )
```

Definition at line 4748 of file sort.c.

4.131.3.26 SimpleSplitMergeRec()

```
void SimpleSplitMergeRec (
            WORD * array,
            WORD num,
            WORD * auxarray )
```

Definition at line 4850 of file sort.c.

4.131.3.27 SimpleSplitMerge()

```
void SimpleSplitMerge (
            WORD * array,
            WORD num )
```

Definition at line 4878 of file sort.c.

4.131.3.28 BinarySearch()

```
WORD BinarySearch (
             WORD * array,
             WORD num,
             WORD x )
```

Definition at line 4896 of file sort.c.

4.131.4 Variable Documentation

4.131.4.1 toterms

```
char* toterms[] = { " ", " >>", "-->" }
```

Definition at line 82 of file sort.c.

4.131.4.2 humanTermsSuffix

```
const char humanTermsSuffix[HUMANSUFFLEN][HUMANSUFFSTRLEN] = {"K ","M ","B ","T "}
```

Definition at line 87 of file sort.c.

4.131.4.3 humanBytesSuffix

```
const char humanBytesSuffix[HUMANSUFFLEN][HUMANSUFFSTRLEN] = {"KiB","MiB","GiB","TiB"}
```

Definition at line 88 of file sort.c.

4.132 sort.c

Go to the documentation of this file.

```
00001
00017 /* #[ License : */
00018 /*
00019  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00020  *   When using this file you are requested to refer to the publication
00021  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00022  *   This is considered a matter of courtesy as the development was paid
00023  *   for by FOM the Dutch physics granting agency and we would like to
00024  *   be able to track its scientific use to convince FOM of its value
00025  *   for the community.
00026  *
00027  *   This file is part of FORM.
00028  *
00029  *   FORM is free software: you can redistribute it and/or modify it under the
00030  *   terms of the GNU General Public License as published by the Free Software
00031  *   Foundation, either version 3 of the License, or (at your option) any later
00032  *   version.
00033  *
00034  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00035  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00036  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00037  *   details.
00038  *
00039  *   You should have received a copy of the GNU General Public License along
00040  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00041  */
```

```
00042 /* #] License : */
00043 /*
00044     #[ Includes : sort.c
00045
00046     Sort routines according to new conventions (25-jun-1997).
00047     This is more object oriented.
00048     The active sort is indicated by AT.SS which should agree with
00049     AN.FunSorts[AR.sLevel];
00050
00051 #define GZIPDEBUG
00052 */
00053 #define NEWSPLITMERGE
00054 /* Comment to turn off Timsort in SplitMerge for debugging: */
00055 #define NEWSPLITMERGETIMSORT
00056 /* During SplitMerge, print pointer array state on entry. Very spammy, for debugging. */
00057 /* #define SPLITMERGEDEBUG */
00058 /* Debug printing for GarbHand */
00059 /* #define TESTGARB */
00060
00061 #include "form3.h"
00062 #include <math.h>
00063
00064 #ifdef WITHPTHREADS
00065 UBYTE THRbuf[100];
00066 #endif
00067
00068 #ifdef WITHSTATS
00069 extern LONG numwrites;
00070 extern LONG numreads;
00071 extern LONG numseeks;
00072 extern LONG nummallocs;
00073 extern LONG numfrees;
00074 #endif
00075
00076 /*
00077     #] Includes :
00078     #[ SortUtilities :
00079         #[ WriteStats :             void WriteStats(lspace,par,checkLogType)
00080 */
00081
00082 char *toterms[] = { "    ", " »", "-->" };
00083
00084 #define HUMANSTRLEN 12
00085 #define HUMANSUFFLEN 4
00086 #define HUMANSUFFSTRLEN 4
00087 const char humanTermsSuffix[HUMANSUFFLEN][HUMANSUFFSTRLEN] = {"K  ","M  ","B  ","T  "};
00088 const char humanBytesSuffix[HUMANSUFFLEN][HUMANSUFFSTRLEN] = {"KiB","MiB","GiB","TiB"};
00089 void HumanString(char* string, float input, const float scale,
00090     const char suffix[HUMANSUFFLEN][HUMANSUFFSTRLEN]) {
00091
00092     int ind = -1;
00093     while (ind < 0 || (input >= scale && ind+1 < HUMANSUFFLEN) ) {
00094         input /= scale;
00095         ind++;
00096     }
00097     if ( input <= 0.5f ) {
00098         snprintf(string, HUMANSTRLEN,
00099             "  ( <1 %s)", suffix[ind]);
00100     }
00101     else {
00102         snprintf(string, HUMANSTRLEN,
00103             "  (%3ld %s)", lroundf(input), suffix[ind]);
00104     }
00105 }
00106
00107 #define COL_EXP 16
00108 #define COL_SPA 8
00109 #define COL_EQU 42
00110 #define COL_VAL 53
00111
00112 WORD DigitsIn(LONG x) {
00113     if ( x < 0 ) x = -x;
00114     WORD dig = 1;
00115     while ( x > 9 ) {
00116         dig++;
00117         x /= 10;
00118     }
00119     return dig;
00120 }
00121
00138 void WriteStats(POSITION *plspace, WORD par, WORD checkLogType)
00139 {
00140     GETIDENTITY
00141     char buf[120];
00142     LONG millitime;
00143     UWORD timepart;
00144     SORTING *S;
```

```
00145     int use_wtime;
00146
00147     if ( AT.SS == AT.S0 && AC.StatsFlag ) {
00148 #ifdef WITHPTHREADS
00149         if ( AC.ThreadStats == 0 && identity > 0 ) return;
00150 #elif defined(WITHMPI)
00151         if ( AC.OldParallelStats ) return;
00152         if ( ! AC.ProcessStats && PF.me != MASTER ) return;
00153 #endif
00154         if ( Expressions == 0 ) return;
00155
00156         if ( par == STATSSPLITMERGE ) {
00157             if ( AC.ShortStatsMax == 0 ) return;
00158             AR.ShortSortCount++;
00159             if ( AR.ShortSortCount < AC.ShortStatsMax ) return;
00160         }
00161         AR.ShortSortCount = 0;
00162
00163         S = AT.SS;
00164
00165         char humanGenTermsText[HUMANSTRLEN] = "";
00166         char humanTermsLeftText[HUMANSTRLEN] = "";
00167         char humanBytesText[HUMANSTRLEN] = "";
00168         char humanUnsortedBytesText[HUMANSTRLEN] = "";
00169         char humanComparisonsText[HUMANSTRLEN] = "";
00170         char humanMaxTermSizeText[HUMANSTRLEN] = "";
00171         if ( AC.HumanStatsFlag ) {
00172             HumanString(humanGenTermsText, (float)(S->GenTerms), 1000.0f, humanTermsSuffix);
00173             HumanString(humanTermsLeftText, (float)(S->TermsLeft), 1000.0f, humanTermsSuffix);
00174             HumanString(humanBytesText, (float)(BASEPOSITION(*plspace)), 1024.0f, humanBytesSuffix);
00175             HumanString(humanUnsortedBytesText, (float)(S->verbUnsortedSize), 1024.0f,
    humanBytesSuffix);
00176             HumanString(humanComparisonsText, (float)(S->verbComparisons), 1000.0f, humanTermsSuffix);
00177             HumanString(humanMaxTermSizeText, (float)(S->verbMaxTermSize), 1000.0f, humanTermsSuffix);
00178         }
00179
00180         MLOCK(ErrorMessageLock);
00181
00182         /* If the statistics should not go to the log file, temporarily hide the
00183          * LogHandle. This must be done within the ErrorMessageLock region to
00184          * avoid a data race between threads. */
00185         const WORD oldLogHandle = AC.LogHandle;
00186         if ( checkLogType && AM.LogType ) {
00187             AC.LogHandle = -1;
00188         }
00189
00190         if ( AC.ShortStats ) {}
00191         else {
00192 #ifdef WITHPTHREADS
00193             if ( identity > 0 ) {
00194                 MesPrint("             Thread %d reporting",identity);
00195             }
00196             else {
00197                 MesPrint("");
00198             }
00199 #elif defined(WITHMPI)
00200             if ( PF.me != MASTER ) {
00201                 MesPrint("             Process %d reporting",PF.me);
00202             }
00203             else {
00204                 MesPrint("");
00205             }
00206 #else
00207             MesPrint("");
00208 #endif
00209         }
00210         /*
00211          * We define WTimeStatsFlag as a flag to print the wall-clock time on
00212          * the *master*, not in workers. This can be confusing in thread
00213          * statistics when short statistics is used. Technically,
00214          * TimeWallClock() is not thread-safe in TFORM.
00215          */
00216         use_wtime = AC.WTimeStatsFlag;
00217 #if defined(WITHPTHREADS)
00218         if ( use_wtime && identity > 0 ) use_wtime = 0;
00219 #elif defined(WITHMPI)
00220         if ( use_wtime && PF.me != MASTER ) use_wtime = 0;
00221 #endif
00222         char *wpref = use_wtime ? "W" : "";
00223         char *wspac = use_wtime ? "" : " ";
00224         millitime = use_wtime ? TimeWallClock(1) * 10 : TimeCPU(1);
00225         timepart = (UWORD)(millitime%1000);
00226         millitime /= 1000;
00227         timepart /= 10;
00228
00229         if ( AC.ShortStats ) {
00230 #if defined(WITHPTHREADS) || defined(WITHMPI)
```

```
00231 #ifdef WITHPTHREADS
00232             if ( identity > 0 ) {
00233 #else
00234             if ( PF.me != MASTER ) {
00235                 const int identity = PF.me;
00236 #endif
00237                 if ( par == STATSSPLITMERGE || par == STATSPOSTSORT ) {
00238                     snprintf(buf, sizeof(buf),
00239                         "%d: %7ld.%02us %8ld>%10ld%3s%10ld:%10ld %s %s",identity,
00240                         millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
00241                         S->TermsLeft,BASEPOSITION(*plspace),EXPRNAME(AR.CurExpr),
00242                         AC.Commercial);
00243                     MesPrint("%s", buf);
00244                 }
00245                 else if ( par == STATSMERGETOFILE ) {
00246                     snprintf(buf, sizeof(buf),
00247                         "%d: %7ld.%02us %10ld:%10ld",identity,millitime,timepart,
00248                         S->TermsLeft,BASEPOSITION(*plspace));
00249                     MesPrint("%s", buf);
00250                 }
00251             }
00252             else
00253 #endif
00254             {
00255                 if ( par == STATSSPLITMERGE || par == STATSPOSTSORT ) {
00256                     snprintf(buf, sizeof(buf),
00257                         "%7ld.%02us %8ld>%10ld%3s%10ld:%10ld %s %s",
00258                         millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
00259                         S->TermsLeft,BASEPOSITION(*plspace),EXPRNAME(AR.CurExpr),
00260                         AC.Commercial);
00261                     MesPrint("%s", buf);
00262                 }
00263                 else if ( par == STATSMERGETOFILE ) {
00264                     snprintf(buf, sizeof(buf),
00265                         "%7ld.%02us %10ld:%10ld",millitime,timepart,
00266                         S->TermsLeft,BASEPOSITION(*plspace));
00267                     MesPrint("%s", buf);
00268                 }
00269             }
00270         }
00271         else {
00272             if ( par == STATSMERGETOFILE ) {
00273                 snprintf(buf, sizeof(buf),
00274                     "%sTime = %7ld.%02u sec",wpref,millitime,timepart);
00275                 MesPrint("%s", buf);
00276             }
00277             else {
00278                 snprintf(buf, sizeof(buf),
00279                     "%sTime = %7ld.%02u sec   %sGenerated terms =%11ld%s",
00280                     wpref, millitime, timepart, wspac, S->GenTerms, humanGenTermsText);
00281                 MesPrint("%s", buf);
00282             }
00283
00284             const int exprlen = strlen((char*)EXPRNAME(AR.CurExpr));
00285             const int overflow = MaX(0, exprlen - COL_EXP);
00286             if ( par == STATSSPLITMERGE ) {
00287                 int width = snprintf(buf, sizeof(buf),
00288                     "%*s %*ld Terms %s",
00289                     COL_EXP, EXPRNAME(AR.CurExpr),
00290                     MaX(0,COL_SPA-1-overflow), AN.ninterms, FG.swmes[par]);
00291                 width += snprintf(buf+width, sizeof(buf)-width,
00292                     "%*s=",
00293                     MaX(0,COL_EQU-1-width), "");
00294                 snprintf(buf+width, sizeof(buf)-width,
00295                     "%*ld%s",
00296                     MaX(0,COL_VAL-width), S->TermsLeft, humanTermsLeftText);
00297                 MesPrint("%s", buf);
00298             }
00299             else {
00300 #ifdef WITHPTHREADS
00301                 if ( identity > 0 && par == STATSPOSTSORT ) {
00302                     int width = snprintf(buf, sizeof(buf),
00303                         "%*s%*s Terms in thread",
00304                         COL_EXP, EXPRNAME(AR.CurExpr), MaX(0,COL_SPA-overflow), "");
00305                     width += snprintf(buf+width, sizeof(buf)-width,
00306                         "%*s=",
00307                         MaX(0,COL_EQU-1-width), "");
00308                     snprintf(buf+width, sizeof(buf)-width,
00309                         "%*ld%s",
00310                         MaX(0,COL_VAL-width), S->TermsLeft, humanTermsLeftText);
00311                     MesPrint("%s", buf);
00312                 }
00313                 else
00314 #elif defined(WITHMPI)
00315                 if ( PF.me != MASTER && par == STATSPOSTSORT ) {
00316                     int width = snprintf(buf, sizeof(buf),
00317                         "%*s%*s Terms in process=",
```

```
00318                       COL_EXP, EXPRNAME(AR.CurExpr), MaX(0,COL_SPA-overflow), "");
00319                   snprintf(buf+width, sizeof(buf)-width,
00320                       "%*ld%s",
00321                       MaX(0,COL_VAL-width), S->TermsLeft, humanTermsLeftText);
00322                   MesPrint("%s", buf);
00323               }
00324               else
00325 #endif
00326               {
00327                   int width = snprintf(buf, sizeof(buf),
00328                       "%*s%*s Terms %s",
00329                       COL_EXP, EXPRNAME(AR.CurExpr), MaX(0,COL_SPA-overflow), "",
00330                       FG.swmes[par]);
00331                   width += snprintf(buf+width, sizeof(buf)-width,
00332                       "%*s=",
00333                       MaX(0,COL_EQU-1-width), "");
00334                   snprintf(buf+width, sizeof(buf)-width,
00335                       "%*ld%s",
00336                       MaX(0,COL_VAL-width), S->TermsLeft, humanTermsLeftText);
00337                   MesPrint("%s", buf);
00338               }
00339           }
00340
00341           const WORD dig = DigitsIn(BASEPOSITION(*plspace));
00342           snprintf(buf, sizeof(buf),
00343               "%*s Bytes used%*s=%11ld%s",
00344               COL_EXP+COL_SPA, AC.Commercial, MiN(6,17-dig), "",
00345               BASEPOSITION(*plspace), humanBytesText);
00346           MesPrint("%s", buf);
00347       }
00348
00349       if ( par == STATSPOSTSORT ) {
00350           if ( AC.SortVerbose ) {
00351               snprintf(buf, sizeof(buf), "%*s Unsorted bytes  =%11ld%s",
00352                   COL_EXP+COL_SPA, "", S->verbUnsortedSize, humanUnsortedBytesText);
00353               MesPrint("%s", buf);
00354               snprintf(buf, sizeof(buf), "%*s Small Buffer    =%5ld,%5ld",
00355                   COL_EXP+COL_SPA, "", S->verbSBsortTerms, S->verbSBsortCap);
00356               MesPrint("%s", buf);
00357               snprintf(buf, sizeof(buf), "%*s Large Buffer    =%5ld,%5ld",
00358                   COL_EXP+COL_SPA, "", S->verbLBsortPatches, S->verbLBsortCap);
00359               MesPrint("%s", buf);
00360               snprintf(buf, sizeof(buf), "%*s Comparisons     =%11ld%s",
00361                   COL_EXP+COL_SPA, "", S->verbComparisons, humanComparisonsText);
00362               MesPrint("%s", buf);
00363               snprintf(buf, sizeof(buf), "%24s Largest Term    =%11ld%s",
00364                   "",S->verbMaxTermSize,humanMaxTermSizeText);
00365               MesPrint("%s", buf);
00366           }
00367       }
00368
00369 #ifdef WITHSTATS
00370       MesPrint("Total number of writes: %l, reads: %l, seeks, %l"
00371           ,numwrites,numreads,numseeks);
00372       MesPrint("Total number of mallocs: %l, frees: %l"
00373           ,nummallocs,numfrees);
00374 #endif
00375       /* Put back the original LogHandle, it was changed if the statistics were
00376        * not printed in the log file. */
00377       AC.LogHandle = oldLogHandle;
00378
00379       MUNLOCK(ErrorMessageLock);
00380   }
00381 }
00382
00383 /*
00384       #] WriteStats :
00385       #[ NewSort :               WORD NewSort()
00386 */
00397 int NewSort(PHEAD0)
00398 {
00399     GETBIDENTITY
00400     SORTING *S, **newFS;
00401     int i, newsize;
00402     if ( AN.SoScratC == 0 )
00403         AN.SoScratC = (UWORD *)Malloc1(2*(AM.MaxTal+2)*sizeof(UWORD),"NewSort");
00404     AR.sLevel++;
00405     if ( AR.sLevel >= AN.NumFunSorts ) {
00406         if ( AN.NumFunSorts == 0 ) newsize = 100;
00407         else newsize = 2*AN.NumFunSorts;
00408         newFS = (SORTING **)Malloc1((newsize+1)*sizeof(SORTING *),"FunSort pointers");
00409         for ( i = 0; i < AN.NumFunSorts; i++ ) newFS[i] = AN.FunSorts[i];
00410         for ( ; i <= newsize; i++ ) newFS[i] = 0;
00411         if ( AN.FunSorts ) M_free(AN.FunSorts,"FunSort pointers");
00412         AN.FunSorts = newFS; AN.NumFunSorts = newsize;
00413     }
00414     if ( AR.sLevel == 0 ) {
```

```
00415
00416         AN.FunSorts[0] = AT.S0;
00417         if ( AR.PolyFun == 0 ) { AT.S0->PolyFlag = 0; }
00418         else if ( AR.PolyFunType == 1 ) { AT.S0->PolyFlag = 1; }
00419         else if ( AR.PolyFunType == 2 ) {
00420             if ( AR.PolyFunExp == 2
00421               || AR.PolyFunExp == 3 ) AT.S0->PolyFlag = 1;
00422             else                      AT.S0->PolyFlag = 2;
00423         }
00424         AR.ShortSortCount = 0;
00425     }
00426     else {
00427         if ( AN.FunSorts[AR.sLevel] == 0 ) {
00428             AN.FunSorts[AR.sLevel] = AllocSort(
00429                 AM.SLargeSize,AM.SSmallSize,AM.SSmallEsize,AM.STermsInSmall
00430                     ,AM.SMaxPatches,AM.SMaxFpatches,AM.SIOsize,1);
00431         }
00432         AN.FunSorts[AR.sLevel]->PolyFlag = 0;
00433     }
00434     AT.SS = S = AN.FunSorts[AR.sLevel];
00435     S->sFill = S->sBuffer;
00436     S->lFill = S->lBuffer;
00437     S->lPatch = 0;
00438     S->fPatchN = 0;
00439     S->GenTerms = S->TermsLeft = S->GenSpace = S->SpaceLeft = 0;
00440     S->PoinFill = S->sPointer;
00441     *S->PoinFill = S->sFill;
00442     if ( AR.sLevel > 0 ) { S->PolyWise = 0; }
00443     PUTZERO(S->SizeInFile[0]); PUTZERO(S->SizeInFile[1]); PUTZERO(S->SizeInFile[2]);
00444     S->sTerms = 0;
00445     PUTZERO(S->file.POposition);
00446     S->stage4 = 0;
00447     if ( AR.sLevel > AN.MaxFunSorts ) AN.MaxFunSorts = AR.sLevel;
00448
00449     // Zero the SortVerbose counters:
00450     S->verbComparisons = 0;
00451     S->verbMaxTermSize = 0;
00452     S->verbSBsortTerms = 0;
00453     S->verbSBsortCap = 0;
00454     S->verbLBsortPatches = 0;
00455     S->verbLBsortCap = 0;
00456     S->verbUnsortedSize = 0;
00457     return(0);
00458 }
00459
00460 /*
00461         #] NewSort :
00462         #[ EndSort :                WORD EndSort(PHEAD buffer,par)
00463 */
00488 LONG EndSort(PHEAD WORD *buffer, int par)
00489 {
00490   GETBIDENTITY
00491   SORTING *S = AT.SS;
00492   WORD j, **ss, *to, *t;
00493   LONG sSpace, over, tover, spare, retval = 0;
00494   POSITION position, pp;
00495   off_t lSpace;
00496   FILEHANDLE *fout = 0, *oldoutfile = 0, *newout = 0;
00497
00498   if ( AM.exitflag && AR.sLevel == 0 ) return(0);
00499 #ifdef WITHMPI
00500   if( (retval = PF_EndSort()) > 0){
00501     oldoutfile = AR.outfile;
00502     retval = 0;
00503     goto RetRetval;
00504   }
00505   else if(retval < 0){
00506     retval = -1;
00507     goto RetRetval;
00508   }
00509     /* PF_EndSort returned 0: for S != AM.S0 and slaves still do the regular sort */
00510 #endif /* WITHMPI */
00511     oldoutfile = AR.outfile;
00512 /*      PolyFlag repair action
00513     if ( S == AT.S0 ) {
00514         if ( AR.PolyFun == 0 ) { S->PolyFlag = 0; }
00515         else if ( AR.PolyFunType == 1 ) { S->PolyFlag = 1; }
00516         else if ( AR.PolyFunType == 2 ) {
00517             if ( AR.PolyFunExp == 2
00518               || AR.PolyFunExp == 3 ) S->PolyFlag = 1;
00519             else                      S->PolyFlag = 2;
00520         }
00521         S->PolyWise = 0;
00522     }
00523     else {
00524         S->PolyFlag = S->PolyWise = 0;
00525     }
```

```
00526 */
00527     S->PolyWise = 0;
00528     *(S->PoinFill) = 0;
00529 #ifdef SPLITTIME
00530         PrintTime((UBYTE *)"EndSort, before SplitMerge");
00531 #endif
00532     S->sPointer[SplitMerge(BHEAD S->sPointer,S->sTerms)] = 0;
00533 #ifdef SPLITTIME
00534         PrintTime((UBYTE *)"Endsort,  after SplitMerge");
00535 #endif
00536     sSpace = 0;
00537     tover = over = S->sTerms;
00538     ss = S->sPointer;
00539     if ( over >= 0 ) {
00540         if ( S->lPatch > 0 || S->file.handle >= 0 ) {
00541             ss[over] = 0;
00542             sSpace = ComPress(ss,&spare);
00543             S->TermsLeft -= over - spare;
00544             if ( par == 1 ) { AR.outfile = newout = AllocFileHandle(0,(char *)0); }
00545         }
00546         else if ( S != AT.S0 ) {
00547             ss[over] = 0;
00548             if ( par == 2 ) {
00549                 sSpace = 3;
00550                 while ( ( t = *ss++ ) != 0 ) { sSpace += *t; }
00551                 if ( AN.tryterm > 0 && ( (sSpace+1)*sizeof(WORD) < (size_t)(AM.MaxTer) ) ) {
00552                     to = TermMalloc("$-sort space");
00553                 }
00554                 else {
00555                     LONG allocsp = sSpace+1;
00556                     if ( allocsp < MINALLOC ) allocsp = MINALLOC;
00557                     allocsp = ((allocsp+7)/8)*8;
00558                     to = (WORD *)Malloc1(allocsp*sizeof(WORD),"$-sort space");
00559                     if ( AN.tryterm > 0 ) AN.tryterm = 0;
00560                 }
00561                 *((WORD **)buffer) = to;
00562                 ss = S->sPointer;
00563                 while ( ( t = *ss++ ) != 0 ) {
00564                     j = *t; while ( --j >= 0 ) *to++ = *t++;
00565                 }
00566                 *to = 0;
00567                 retval = sSpace + 1;
00568             }
00569             else {
00570                 to = buffer;
00571                 sSpace = 0;
00572                 while ( ( t = *ss++ ) != 0 ) {
00573                     j = *t;
00574                     if ( ( sSpace += j ) > AM.MaxTer/((LONG)sizeof(WORD)) ) {
00575                         /* Too big! Get the total size for useful error message */
00576                         while ( ( t = *ss++ ) != 0 ) {
00577                             sSpace += *t;
00578                         }
00579                         MLOCK(ErrorMessageLock);
00580                         MesPrint("Sorted function argument too long (%d words). Increase MaxTermSize
      (%l words).", sSpace, AM.MaxTer/((LONG)sizeof(WORD)));
00581                         MUNLOCK(ErrorMessageLock);
00582                         retval = -1; goto RetRetval;
00583                     }
00584                     while ( --j >= 0 ) *to++ = *t++;
00585                 }
00586                 *to = 0;
00587                 retval = to - buffer;
00588             }
00589             goto RetRetval;
00590         }
00591         else {
00592             POSITION oldpos;
00593             if ( S == AT.S0 ) {
00594                 fout = AR.outfile;
00595                 *AR.CompressPointer = 0;
00596                 SeekScratch(AR.outfile,&position);
00597             }
00598             else {
00599                 fout = &(S->file);
00600                 PUTZERO(position);
00601             }
00602             oldpos = position;
00603             S->TermsLeft = 0;
00604 /*
00605             Here we can go directly to the output.
00606 */
00607 #ifdef WITHZLIB
00608             { int oldgzipCompress = AR.gzipCompress;
00609                 AR.gzipCompress = 0;
00610 #endif
00611             if ( tover > 0 ) {
```

```
00612                 ss = S->sPointer;
00613                 while ( ( t = *ss++ ) != 0 ) {
00614                     if ( *t ) S->TermsLeft++;
00615 #ifdef WITHPTHREADS
00616                     if ( AS.MasterSort && ( fout == AR.outfile ) ) { PutToMaster(BHEAD t); }
00617                     else
00618 #endif
00619                     if ( PutOut(BHEAD t,&position,fout,1) < 0 ) {
00620                         retval = -1; goto RetRetval;
00621                     }
00622                 }
00623             }
00624 #ifdef WITHPTHREADS
00625             if ( AS.MasterSort && ( fout == AR.outfile ) ) { PutToMaster(BHEAD 0); }
00626             else
00627 #endif
00628             if ( FlushOut(&position,fout,1) ) {
00629                 retval = -1; goto RetRetval;
00630             }
00631 #ifdef WITHZLIB
00632                 AR.gzipCompress = oldgzipCompress;
00633             }
00634 #endif
00635 #ifdef WITHPTHREADS
00636             if ( AS.MasterSort && ( fout == AR.outfile ) ) goto RetRetval;
00637 #endif
00638 #ifdef WITHMPI
00639             if ( PF.me != MASTER && PF.exprtodo < 0 ) goto RetRetval;
00640 #endif
00641             DIFPOS(oldpos,position,oldpos);
00642             S->SpaceLeft = BASEPOSITION(oldpos);
00643             WriteStats(&oldpos,STATSPOSTSORT,NOCHECKLOGTYPE);
00644             pp = oldpos;
00645             goto RetRetval;
00646         }
00647     }
00648     else if ( par == 1 && newout == 0 ) { AR.outfile = newout = AllocFileHandle(0,(char *)0); }
00649     sSpace++;
00650     lSpace = sSpace + (S->lFill - S->lBuffer) - (LONG)S->lPatch*(AM.MaxTer/sizeof(WORD));
00651 /*         Note wrt MaxTer and lPatch: each patch starts with space for decompression */
00652 /*         Not needed if only large buffer, but needed when using files (?) */
00653     SETBASEPOSITION(pp,lSpace);
00654     MULPOS(pp,sizeof(WORD));
00655     if ( S->file.handle >= 0 ) {
00656         ADD2POS(pp,S->fPatches[S->fPatchN]);
00657     }
00658     if ( S == AT.S0 ) {
00659         if ( S->lPatch > 0 || S->file.handle >= 0 ) {
00660             WriteStats(&pp,STATSSPLITMERGE,CHECKLOGTYPE);
00661         }
00662     }
00663     if ( par == 2 ) { AR.outfile = newout = AllocFileHandle(0,(char *)0); }
00664     if ( S->lPatch > 0 ) {
00665         if ( ( S->lPatch >= S->MaxPatches ) ||
00666             ( ( (WORD *)(((UBYTE *)(S->lFill + sSpace)) + 2*AM.MaxTer) ) >= S->lTop ) ) {
00667 /*
00668             The large buffer is too full. Merge and write it
00669 */
00670             // Update SortVerbose counters
00671             if ( S->lPatch >= S->MaxPatches ) S->verbLBsortPatches++;
00672             else S->verbLBsortCap++;
00673
00674 #ifdef GZIPDEBUG
00675             MLOCK(ErrorMessageLock);
00676             MesPrint("%w EndSort: lPatch = %d, MaxPatches = %d,lFill = %x, sSpace = %ld, MaxTer = %d,
    lTop = %x"
00677                     ,S->lPatch,S->MaxPatches,S->lFill,sSpace,AM.MaxTer/sizeof(WORD),S->lTop);
00678             MUNLOCK(ErrorMessageLock);
00679 #endif
00680
00681             if ( MergePatches(1) ) {
00682                 MLOCK(ErrorMessageLock);
00683                 MesCall("EndSort");
00684                 MUNLOCK(ErrorMessageLock);
00685                 retval = -1; goto RetRetval;
00686             }
00687             S->lPatch = 0;
00688             pp = S->SizeInFile[1];
00689             MULPOS(pp,sizeof(WORD));
00690 #ifndef WITHPTHREADS
00691             if ( S == AT.S0 )
00692 #endif
00693             {
00694                 POSITION pppp;
00695                 SETBASEPOSITION(pppp,0);
00696                 SeekFile(S->file.handle,&pppp,SEEK_CUR);
00697                 SeekFile(S->file.handle,&pp,SEEK_END);
```

```
00698                 SeekFile(S->file.handle,&pppp,SEEK_SET);
00699                 WriteStats(&pp,STATSMERGETOFILE,CHECKLOGTYPE);
00700                 UpdateMaxSize();
00701             }
00702         }
00703         else {
00704             S->Patches[S->lPatch++] = S->lFill;
00705             to = (WORD *)(((UBYTE *)(S->lFill)) + AM.MaxTer);
00706             if ( tover > 0 ) {
00707                 ss = S->sPointer;
00708                 while ( ( t = *ss++ ) != 0 ) {
00709                     j = *t;
00710                     if ( j < 0 ) j = t[1] + 2;
00711                     while ( --j >= 0 ) *to++ = *t++;
00712                 }
00713             }
00714             *to++ = 0;
00715             S->lFill = to;
00716             if ( S->file.handle < 0 ) {
00717                 if ( MergePatches(2) ) {
00718                     MLOCK(ErrorMessageLock);
00719                     MesCall("EndSort");
00720                     MUNLOCK(ErrorMessageLock);
00721                     retval = -1; goto RetRetval;
00722                 }
00723                 if ( S == AT.S0 ) {
00724                     pp = S->SizeInFile[2];
00725                     MULPOS(pp,sizeof(WORD));
00726 #ifdef WITHPTHREADS
00727                     if ( AS.MasterSort && ( fout == AR.outfile ) ) goto RetRetval;
00728 #endif
00729                     WriteStats(&pp,STATSPOSTSORT,NOCHECKLOGTYPE);
00730                     UpdateMaxSize();
00731                 }
00732                 else {
00733                     if ( par == 2 && newout->handle >= 0 ) {
00734                         POSITION zeropos;
00735                         PUTZERO(zeropos);
00736 #ifdef ALLLOCK
00737                         LOCK(newout->pthreadslock);
00738 #endif
00739                         SeekFile(newout->handle,&zeropos,SEEK_SET);
00740                         to = (WORD *)Malloc1(BASEPOSITION(newout->filesize)+sizeof(WORD)*2
00741                                 ,"$-buffer reading");
00742                         if ( AN.tryterm > 0 ) AN.tryterm = 0;
00743                         if ( ( retval = ReadFile(newout->handle,(UBYTE
    *)to,BASEPOSITION(newout->filesize)) ) !=
00744                                 BASEPOSITION(newout->filesize) ) {
00745 /* INTERNAL_ERROR_EXCL_START */
00746                             MLOCK(ErrorMessageLock);
00747                             MesPrint("!>Error reading information for $ variable");
00748                             MUNLOCK(ErrorMessageLock);
00749                             M_free(to,"$-buffer reading");
00750                             retval = -1;
00751 /* INTERNAL_ERROR_EXCL_STOP */
00752                         }
00753                         else {
00754                             *((WORD **)buffer) = to;
00755                             retval /= sizeof(WORD);
00756                         }
00757 #ifdef ALLLOCK
00758                         UNLOCK(newout->pthreadslock);
00759 #endif
00760                     }
00761                     else if ( newout->handle >= 0 ) {
00762 /*
00763                         We land here if par == 1 (function arg sort) and PutOut has created a file.
00764                         This means that the term is larger than SIOsize, which we ensure is at least
00765                         as large as MaxTermSize in setfile.c. Thus we know already that the term won't
    fit.
00766 */
00767 TooLarge:
00768                         MLOCK(ErrorMessageLock);
00769                         MesPrint("(1)Output should fit inside a single term. Increase MaxTermSize?");
00770                         MesCall("EndSort");
00771                         MUNLOCK(ErrorMessageLock);
00772                         retval = -1; goto RetRetval;
00773                     }
00774                     else {
00775                         t = newout->PObuffer;
00776                         // We deal with the par == 2 case after RetRetval.
00777                         if ( par != 2 ) {
00778                             j = newout->POfill - t;
00779                             to = buffer;
00780                             if ( to >= AT.WorkSpace && to < AT.WorkTop && to+j > AT.WorkTop )
00781                                 goto WorkSpaceError;
00782                             if ( j > AM.MaxTer ) {
```

```
00783 /* INTERNAL_ERROR_EXCL_START */
00784                                 MLOCK(ErrorMessageLock);
00785                                 MesPrint("!>Encountered term of size: %d words.", j/(LONG)sizeof(WORD)
      );
00786                                 MUNLOCK(ErrorMessageLock);
00787                                 goto TooLarge;
00788 /* INTERNAL_ERROR_EXCL_STOP */
00789                             }
00790                             NCOPY(to,t,j);
00791                             retval = to - buffer - 1;
00792                         }
00793                     }
00794                 }
00795                 goto RetRetval;
00796             }
00797             if ( MergePatches(1) ) { /* --> SortFile */
00798                 MLOCK(ErrorMessageLock);
00799                 MesCall("EndSort");
00800                 MUNLOCK(ErrorMessageLock);
00801                 retval = -1; goto RetRetval;
00802             }
00803             UpdateMaxSize();
00804             pp = S->SizeInFile[1];
00805             MULPOS(pp,sizeof(WORD));
00806 #ifndef WITHPTHREADS
00807             if ( S == AT.S0 )
00808 #endif
00809             {
00810                 POSITION pppp;
00811                 SETBASEPOSITION(pppp,0);
00812                 SeekFile(S->file.handle,&pppp,SEEK_CUR);
00813                 SeekFile(S->file.handle,&pp,SEEK_END);
00814                 SeekFile(S->file.handle,&pppp,SEEK_SET);
00815                 WriteStats(&pp,STATSMERGETOFILE,CHECKLOGTYPE);
00816             }
00817 #ifdef WITHERRORXXX
00818             if ( S != AT.S0 ) {
00819 /*
00820                 This is wrong! We have sorted to the sort file.
00821                 Things are not sitting in the output yet.
00822 */
00823                 if ( newout->handle >= 0 ) goto TooLarge;
00824                 t = newout->POobuffer;
00825                 j = newout->POfill - t;
00826                 to = buffer;
00827                 if ( to >= AT.WorkSpace && to < AT.WorkTop && to+j > AT.WorkTop )
00828                     goto WorkSpaceError;
00829                 if ( j > AM.MaxTer ) goto TooLarge;
00830                 NCOPY(to,t,j);
00831                 goto RetRetval;
00832             }
00833 #endif
00834         }
00835     }
00836     if ( S->file.handle >= 0 ) {
00837 #ifdef GZIPDEBUG
00838         MLOCK(ErrorMessageLock);
00839         MesPrint("%w EndSort: fPatchN = %d, lPatch = %d, position = %12p"
00840             ,S->fPatchN,S->lPatch,&(S->fPatches[S->fPatchN]));
00841         MUNLOCK(ErrorMessageLock);
00842 #endif
00843         if ( S->lPatch <= 0 ) {
00844             StageSort(&(S->file));
00845             position = S->fPatches[S->fPatchN];
00846             ss = S->sPointer;
00847             if ( *ss ) {
00848                 *AR.CompressPointer = 0;
00849 #ifdef WITHZLIB
00850                 if ( S == AT.S0 && AR.NoCompress == 0 && AR.gzipCompress > 0 )
00851                     S->fpcompressed[S->fPatchN] = 1;
00852                 else
00853                     S->fpcompressed[S->fPatchN] = 0;
00854                 SetupOutputGZIP(&(S->file));
00855 #endif
00856                 while ( ( t = *ss++ ) != 0 ) {
00857                     if ( PutOut(BHEAD t,&position,&(S->file),1) < 0 ) {
00858                         retval = -1; goto RetRetval;
00859                     }
00860                 }
00861                 if ( FlushOut(&position,&(S->file),1) ) {
00862                     retval = -1; goto RetRetval;
00863                 }
00864                 ++(S->fPatchN);
00865                 S->fPatches[S->fPatchN] = position;
00866                 UpdateMaxSize();
00867 #ifdef GZIPDEBUG
00868                 MLOCK(ErrorMessageLock);
```

```
00869                 MesPrint("%w EndSort+: fPatchN = %d, lPatch = %d, position = %12p"
00870                     ,S->fPatchN,S->lPatch,&(S->fPatches[S->fPatchN]));
00871                 MUNLOCK(ErrorMessageLock);
00872 #endif
00873             }
00874         }
00875         AR.Stage4Name = 0;
00876 #ifdef WITHPTHREADS
00877         if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
00878             if ( S->file.handle >= 0 ) {
00879                 SynchFile(S->file.handle);
00880             }
00881         }
00882 #endif
00883         UpdateMaxSize();
00884         if ( MergePatches(0) ) {
00885             MLOCK(ErrorMessageLock);
00886             MesCall("EndSort");
00887             MUNLOCK(ErrorMessageLock);
00888             retval = -1; goto RetRetval;
00889         }
00890         S->stage4 = 0;
00891 #ifdef WITHPTHREADS
00892         if ( AS.MasterSort && ( fout == AR.outfile ) ) goto RetRetval;
00893 #endif
00894         pp = S->SizeInFile[0];
00895         MULPOS(pp,sizeof(WORD));
00896         WriteStats(&pp,STATSPOSTSORT,NOCHECKLOGTYPE);
00897         UpdateMaxSize();
00898     }
00899 RetRetval:
00900
00901 #ifdef WITHMPI
00902     /* NOTE: PF_EndSort has been changed such that it sets S->TermsLeft. (TU 30 Jun 2011) */
00903     if ( AR.sLevel == 0 && (PF.me == MASTER || PF.exprtodo >= 0) ) {
00904         Expressions[AR.CurExpr].counter = S->TermsLeft;
00905         Expressions[AR.CurExpr].size = pp;
00906     }
00907 #else
00908     if ( AR.sLevel == 0 ) {
00909         Expressions[AR.CurExpr].counter = S->TermsLeft;
00910         Expressions[AR.CurExpr].size = pp;
00911     }/*if ( AR.sLevel == 0 )*/
00912 #endif
00913 /*:[25nov2003 mt]*/
00914     if ( S->file.handle >= 0 && ( par != 1 ) && ( par != 2 ) ) {
00915                 /* sortfile is still open */
00916         UpdateMaxSize();
00917 #ifdef WITHZLIB
00918         ClearSortGZIP(&(S->file));
00919 #endif
00920         CloseFile(S->file.handle);
00921         S->file.handle = -1;
00922         remove(S->file.name);
00923 #ifdef GZIPDEBUG
00924         MLOCK(ErrorMessageLock);
00925         MesPrint("%wEndSort: sortfile %s removed",S->file.name);
00926         MUNLOCK(ErrorMessageLock);
00927 #endif
00928     }
00929     AR.outfile = oldoutfile;
00930     AR.sLevel--;
00931     if ( AR.sLevel >= 0 ) AT.SS = AN.FunSorts[AR.sLevel];
00932     if ( par == 1 ) {
00933         if ( retval < 0 ) {
00934             UpdateMaxSize();
00935             if ( newout ) {
00936                 DeAllocFileHandle(newout);
00937                 newout = 0;
00938             }
00939         }
00940         else if ( newout ) {
00941           if ( newout->handle >= 0 ) {
00942             MLOCK(ErrorMessageLock);
00943             MesPrint("(2)Output should fit inside a single term. Increase MaxTermSize?");
00944             MesCall("EndSort");
00945             MUNLOCK(ErrorMessageLock);
00946             Terminate(-1);
00947           }
00948           else if ( newout->POfill > newout->PObuffer ) {
00949 /*
00950             Here we have to copy the contents of the 'file' into
00951             the buffer. We assume that this buffer lies in the WorkSpace.
00952             Hence
00953 */
00954             j = newout->POfill-newout->PObuffer;
00955             if ( buffer >= AT.WorkSpace && buffer < AT.WorkTop && buffer+j > AT.WorkTop )
```

```
00956                 goto WorkSpaceError;
00957             else {
00958                 to = buffer; t = newout->PObuffer;
00959                 while ( j-- > 0 ) *to++ = *t++;
00960             }
00961             UpdateMaxSize();
00962           }
00963           DeAllocFileHandle(newout);
00964           newout = 0;
00965         }
00966     }
00967     else if ( par == 2 ) {
00968         if ( newout ) {
00969             if ( retval == 0 ) {
00970               if ( newout->handle >= 0 ) {
00971 /*
00972                 output resides at the moment in a file
00973                 Find the size, make a buffer, copy into the buffer and clean up.
00974 */
00975                 POSITION zeropos;
00976                 PUTZERO(position);
00977 #ifdef ALLLOCK
00978                 LOCK(newout->pthreadslock);
00979 #endif
00980                 SeekFile(newout->handle,&position,SEEK_END);
00981                 PUTZERO(zeropos);
00982                 SeekFile(newout->handle,&zeropos,SEEK_SET);
00983                 to = (WORD *)Malloc1(BASEPOSITION(position)+sizeof(WORD)*3
00984                         ,"$-buffer reading");
00985                 if ( AN.tryterm > 0 ) AN.tryterm = 0;
00986                 if ( ( retval = ReadFile(newout->handle,(UBYTE *)to,BASEPOSITION(position)) ) !=
00987                 BASEPOSITION(position) ) {
00988 /* INTERNAL_ERROR_EXCL_START */
00989                     MLOCK(ErrorMessageLock);
00990                     MesPrint("!>Error reading information for $ variable");
00991                     MUNLOCK(ErrorMessageLock);
00992                     M_free(to,"$-buffer reading");
00993                     retval = -1;
00994 /* INTERNAL_ERROR_EXCL_STOP */
00995                 }
00996                 else {
00997                     *((WORD **)buffer) = to;
00998                     retval /= sizeof(WORD);
00999                 }
01000 #ifdef ALLLOCK
01001                 UNLOCK(newout->pthreadslock);
01002 #endif
01003               }
01004               else {
01005 /*
01006                 output resides in the cache buffer and the file was never opened
01007 */
01008                 LONG wsiz = newout->POfill - newout->PObuffer;
01009                 if ( AN.tryterm > 0 && ( (wsiz+2)*sizeof(WORD) < (size_t)(AM.MaxTer) ) ) {
01010                     to = TermMalloc("$-sort space");
01011                 }
01012                 else {
01013                     LONG allocsp = wsiz+2;
01014                     if ( allocsp < MINALLOC ) allocsp = MINALLOC;
01015                     allocsp = ((allocsp+7)/8)*8;
01016                     to = (WORD *)Malloc1(allocsp*sizeof(WORD),"$-buffer reading");
01017                     if ( AN.tryterm > 0 ) AN.tryterm = 0;
01018                 }
01019                 *((WORD **)buffer) = to; t = newout->PObuffer;
01020                 retval = wsiz;
01021                 NCOPY(to,t,wsiz);
01022               }
01023             }
01024             UpdateMaxSize();
01025             DeAllocFileHandle(newout);
01026             newout = 0;
01027         }
01028     }
01029     else {
01030         if ( newout ) {
01031             DeAllocFileHandle(newout);
01032             newout = 0;
01033         }
01034     }
01035
01036     return(retval);
01037 WorkSpaceError:
01038     MLOCK(ErrorMessageLock);
01039     MesWork();
01040     MesCall("EndSort");
01041     MUNLOCK(ErrorMessageLock);
01042     Terminate(-1);
```

```
01043     return(-1);
01044 }
01045
01046 /*
01047         #] EndSort :
01048         #[ PutIn :                  LONG PutIn(handle,position,buffer,take,npat)
01049 */
01065 LONG PutIn(FILEHANDLE *file, POSITION *position, WORD *buffer, WORD **take, int npat)
01066 {
01067     LONG i, RetCode;
01068     WORD *from, *to;
01069 #ifndef WITHZLIB
01070     DUMMYUSE(npat);
01071 #endif
01072     from = buffer + ( file->POsize * sizeof(UBYTE) )/sizeof(WORD);
01073     i = from - *take;
01074     if ( i*((LONG)(sizeof(WORD))) > AM.MaxTer ) {
01075 /* INTERNAL_ERROR_EXCL_START */
01076         MLOCK(ErrorMessageLock);
01077         MesPrint("!>Problems in PutIn");
01078         MUNLOCK(ErrorMessageLock);
01079         Terminate(-1);
01080 /* INTERNAL_ERROR_EXCL_STOP */
01081     }
01082     to = buffer;
01083     while ( --i >= 0 ) *--to = *--from;
01084     *take = to;
01085 #ifdef WITHZLIB
01086     if ( ( RetCode = FillInputGZIP(file,position,(UBYTE *)buffer
01087                                     ,file->POsize,npat) ) < 0 ) {
01088 /* INTERNAL_ERROR_EXCL_START */
01089         MLOCK(ErrorMessageLock);
01090         MesPrint("!>PutIn: We have RetCode = %x while reading %x bytes",
01091             RetCode,file->POsize);
01092         MUNLOCK(ErrorMessageLock);
01093         Terminate(-1);
01094 /* INTERNAL_ERROR_EXCL_STOP */
01095     }
01096 #else
01097 #ifdef ALLLOCK
01098     LOCK(file->pthreadslock);
01099 #endif
01100     SeekFile(file->handle,position,SEEK_SET);
01101     if ( ( RetCode = ReadFile(file->handle,(UBYTE *)buffer,file->POsize) ) < 0 ) {
01102 #ifdef ALLLOCK
01103         UNLOCK(file->pthreadslock);
01104 #endif
01105 /* INTERNAL_ERROR_EXCL_START */
01106         MLOCK(ErrorMessageLock);
01107         MesPrint("!>PutIn: We have RetCode = %x while reading %x bytes",
01108             RetCode,file->POsize);
01109         MUNLOCK(ErrorMessageLock);
01110         Terminate(-1);
01111 /* INTERNAL_ERROR_EXCL_STOP */
01112     }
01113 #ifdef ALLLOCK
01114     UNLOCK(file->pthreadslock);
01115 #endif
01116 #endif
01117     return(RetCode);
01118 }
01119
01120 /*
01121         #] PutIn :
01122         #[ Sflush :                 WORD Sflush(file)
01123 */
01131 int Sflush(FILEHANDLE *fi)
01132 {
01133     LONG size, RetCode;
01134 #ifdef WITHZLIB
01135     GETIDENTITY
01136     int dobracketindex = 0;
01137     if ( AR.sLevel <= 0 && Expressions[AR.CurExpr].newbracketinfo
01138         && ( fi == AR.outfile || fi == AR.hidefile ) ) dobracketindex = 1;
01139 #endif
01140     if ( fi->handle < 0 ) {
01141         if ( ( RetCode = CreateFile(fi->name) ) >= 0 ) {
01142 #ifdef GZIPDEBUG
01143             MLOCK(ErrorMessageLock);
01144             MesPrint("%w Sflush created scratch file %s",fi->name);
01145             MUNLOCK(ErrorMessageLock);
01146 #endif
01147             fi->handle = (WORD)RetCode;
01148             PUTZERO(fi->filesize);
01149             PUTZERO(fi->POposition);
01150         }
01151         else {
```

```
01152             MLOCK(ErrorMessageLock);
01153             MesPrint("Cannot create scratch file %s",fi->name);
01154             MUNLOCK(ErrorMessageLock);
01155             return(-1);
01156         }
01157     }
01158 #ifdef WITHZLIB
01159     if ( AT.SS == AT.S0 && !AR.NoCompress && AR.gzipCompress > 0
01160     && dobracketindex == 0 ) {
01161         if ( FlushOutputGZIP(fi) ) return(-1);
01162         fi->POfill = fi->PObuffer;
01163     }
01164     else
01165 #endif
01166     {
01167 #ifdef ALLLOCK
01168       LOCK(fi->pthreadslock);
01169 #endif
01170       size = (fi->POfill-fi->PObuffer)*sizeof(WORD);
01171       SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
01172       if ( WriteFile(fi->handle,(UBYTE *)(fi->PObuffer),size) != size ) {
01173 #ifdef ALLLOCK
01174         UNLOCK(fi->pthreadslock);
01175 #endif
01176         MLOCK(ErrorMessageLock);
01177         MesPrint("Write error while finishing sort. Disk full?");
01178         MUNLOCK(ErrorMessageLock);
01179         return(-1);
01180       }
01181       ADDPOS(fi->filesize,size);
01182       ADDPOS(fi->POposition,size);
01183       fi->POfill = fi->PObuffer;
01184 #ifdef ALLLOCK
01185       UNLOCK(fi->pthreadslock);
01186 #endif
01187     }
01188     return(0);
01189 }
01190
01191 /*
01192         #] Sflush :
01193         #[ PutOut :                 WORD PutOut(term,position,file,ncomp)
01194 */
01217 WORD PutOut(PHEAD WORD *term, POSITION *position, FILEHANDLE *fi, WORD ncomp)
01218 {
01219     GETBIDENTITY
01220     WORD i, *p, ret, *r, *rr, j, k, first;
01221     int dobracketindex = 0;
01222     LONG RetCode;
01223
01224     if ( AT.SS != AT.S0 ) {
01225 /*
01226         For this case no compression should be used
01227 */
01228         if ( ( i = *term ) <= 0 ) return(0);
01229         ret = i;
01230         ADDPOS(*position,i*sizeof(WORD));
01231         p = fi->POfill;
01232         do {
01233             if ( p >= fi->POstop ) {
01234                 if ( fi->handle < 0 ) {
01235                     if ( ( RetCode = CreateFile(fi->name) ) >= 0 ) {
01236 #ifdef GZIPDEBUG
01237                         MLOCK(ErrorMessageLock);
01238                         MesPrint("%w PutOut created sortfile %s",fi->name);
01239                         MUNLOCK(ErrorMessageLock);
01240 #endif
01241                         fi->handle = (WORD)RetCode;
01242                         PUTZERO(fi->filesize);
01243                         PUTZERO(fi->POposition);
01244 /*
01245                         Should not be here anymore?
01246 #ifdef WITHZLIB
01247                         fi->ziobuffer = 0;
01248 #endif
01249 */
01250                     }
01251                     else {
01252                         MLOCK(ErrorMessageLock);
01253                         MesPrint("Cannot create scratch file %s",fi->name);
01254                         MUNLOCK(ErrorMessageLock);
01255                         return(-1);
01256                     }
01257                 }
01258 #ifdef ALLLOCK
01259                 LOCK(fi->pthreadslock);
01260 #endif
```

```
01261                 if ( fi == AR.hidefile ) {
01262                     LOCK(AS.inputslock);
01263                 }
01264                 SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
01265                 if ( ( RetCode = WriteFile(fi->handle,(UBYTE *)(fi->PObuffer),fi->POsize) ) !=
      fi->POsize ) {
01266                     if ( fi == AR.hidefile ) {
01267                         UNLOCK(AS.inputslock);
01268                     }
01269 #ifdef ALLLOCK
01270                     UNLOCK(fi->pthreadslock);
01271 #endif
01272                     MLOCK(ErrorMessageLock);
01273                     MesPrint("Write error during sort. Disk full?");
01274                     MesPrint("Attempt to write %l bytes on file %d at position %15p",
01275                                 fi->POsize,fi->handle,&(fi->POposition));
01276                     MesPrint("RetCode = %l, Buffer address = %l",RetCode,(LONG)(fi->PObuffer));
01277                     MUNLOCK(ErrorMessageLock);
01278                     return(-1);
01279                 }
01280                 ADDPOS(fi->filesize,fi->POsize);
01281                 p = fi->PObuffer;
01282                 ADDPOS(fi->POposition,fi->POsize);
01283                 if ( fi == AR.hidefile ) {
01284                     UNLOCK(AS.inputslock);
01285                 }
01286 #ifdef ALLLOCK
01287                 UNLOCK(fi->pthreadslock);
01288 #endif
01289 #ifdef WITHPTHREADS
01290                 if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
01291                     if ( fi->handle >= 0 ) SynchFile(fi->handle);
01292                 }
01293 #endif
01294             }
01295             *p++ = *term++;
01296         } while ( --i > 0 );
01297         fi->POfull = fi->POfill = p;
01298         return(ret);
01299     }
01300     if ( ( AP.PreDebug & DUMPOUTTERMS ) == DUMPOUTTERMS ) {
01301             MLOCK(ErrorMessageLock);
01302 #ifdef WITHPTHREADS
01303             snprintf((char *)(THRbuf),100,"PutOut(%d)",AT.identity);
01304             PrintTerm(term,(char *)(THRbuf));
01305 #else
01306             PrintTerm(term,"PutOut");
01307 #endif
01308             MesPrint("ncomp = %d, AR.NoCompress = %d, AR.sLevel = %d",ncomp,AR.NoCompress,AR.sLevel);
01309             MesPrint("File %s, position %p",fi->name,position);
01310             MUNLOCK(ErrorMessageLock);
01311     }
01312
01313     if ( AR.sLevel <= 0 && Expressions[AR.CurExpr].newbracketinfo
01314         && ( fi == AR.outfile || fi == AR.hidefile ) ) dobracketindex = 1;
01315     r = rr = AR.CompressPointer;
01316     first = j = k = ret = 0;
01317     if ( ( i = *term ) != 0 ) {
01318         if ( i < 0 ) {          /* Compressed term */
01319             i = term[1] + 2;
01320             if ( fi == AR.outfile || fi == AR.hidefile ) {
01321 /* INTERNAL_ERROR_EXCL_START */
01322                 MLOCK(ErrorMessageLock);
01323                 MesPrint("!>Ran into precompressed term");
01324                 MUNLOCK(ErrorMessageLock);
01325                 Terminate(-1);
01326                 return(-1);
01327 /* INTERNAL_ERROR_EXCL_STOP */
01328             }
01329         }
01330         else if ( !AR.NoCompress && ( ncomp > 0 ) && AR.sLevel <= 0 ) { /* Must compress */
01331             if ( dobracketindex ) {
01332                 PutBracketInIndex(BHEAD term,position);
01333             }
01334             j = *r++ - 1;
01335             p = term + 1;
01336             i--;
01337             if ( AR.PolyFun ) {
01338                 WORD *polystop, *sa;
01339                 sa = p + i;
01340                 sa -= ABS(sa[-1]);
01341                 polystop = p;
01342                 while ( polystop < sa && *polystop != AR.PolyFun ) {
01343                     polystop += polystop[1];
01344                 }
01345                 if ( polystop < sa ) {
01346                     if ( AR.PolyFunType == 2 ) polystop[2] &= ~MUSTCLEANPRF;
```

```
01347                     while ( i > 0 && j > 0 && *p == *r && p < polystop ) {
01348                         i--; j--; k--; p++; r++;
01349                     }
01350                 }
01351                 else {
01352                     while ( i > 0 && j > 0 && *p == *r && p < sa ) { i--; j--; k--; p++; r++; }
01353                 }
01354             }
01355 #ifdef WITHFLOAT
01356             else if ( AT.aux_ != 0 ) {
01357                 WORD *floatstop, *sa;
01358                 sa = p + i;
01359                 sa -= ABS(sa[-1]);
01360                 floatstop = p;
01361                 while ( floatstop < sa && *floatstop != FLOATFUN ) {
01362                     floatstop += floatstop[1];
01363                 }
01364                 if ( floatstop < sa ) {
01365                     while ( i > 0 && j > 0 && *p == *r && p < floatstop ) {
01366                         i--; j--; k--; p++; r++;
01367                     }
01368                 }
01369                 else {
01370                     while ( i > 0 && j > 0 && *p == *r && p < sa ) { i--; j--; k--; p++; r++; }
01371                 }
01372             }
01373 #endif
01374             else {
01375                 WORD *sa;
01376                 sa = p + i;
01377                 sa -= ABS(sa[-1]);
01378                 while ( i > 0 && j > 0 && *p == *r && p < sa ) { i--; j--; k--; p++; r++; }
01379             }
01380             if ( k > -2 ) {
01381 nocompress:
01382                 j = i = *term;
01383                 k = 0;
01384                 p = term;
01385                 r = rr;
01386                 NCOPY(r,p,j);
01387             }
01388             else {
01389                 *rr = *term;
01390                 term = p;
01391                 j = i;
01392                 NCOPY(r,p,j);
01393                 j = i;
01394                 i += 2;
01395                 first = 2;
01396             }
01397 /*                  Sabotage getting into the coefficient next time */
01398             r[-(ABS(r[-1]))] = 0;
01399             if ( r >= AR.ComprTop ) {
01400                 MLOCK(ErrorMessageLock);
01401                 MesPrint("CompressSize of %10l is insufficient",AM.CompressSize);
01402                 MUNLOCK(ErrorMessageLock);
01403                 Terminate(-1);
01404                 return(-1);
01405             }
01406         }
01407         else if ( !AR.NoCompress && ( ncomp < 0 ) && AR.sLevel <= 0 ) {
01408             /* No compress but put in compress buffer anyway */
01409             if ( dobracketindex ) {
01410                 PutBracketInIndex(BHEAD term,position);
01411             }
01412             j = *r++ - 1;
01413             p = term + 1;
01414             i--;
01415             if ( AR.PolyFun ) {
01416                 WORD *polystop, *sa;
01417                 sa = p + i;
01418                 sa -= ABS(sa[-1]);
01419                 polystop = p;
01420                 while ( polystop < sa && *polystop != AR.PolyFun ) {
01421                     polystop += polystop[1];
01422                 }
01423                 if ( polystop < sa ) {
01424                     if ( AR.PolyFunType == 2 ) polystop[2] &= ~MUSTCLEANPRF;
01425                     while ( i > 0 && j > 0 && *p == *r && p < polystop ) {
01426                         i--; j--; k--; p++; r++;
01427                     }
01428                 }
01429                 else {
01430                     while ( i > 0 && j > 0 && *p == *r ) { i--; j--; k--; p++; r++; }
01431                 }
01432             }
01433             else {
```

```
01434                 while ( i > 0 && j > 0 && *p == *r ) { i--; j--; k--; p++; r++; }
01435             }
01436             goto nocompress;
01437         }
01438         else {
01439             if ( AR.PolyFunType == 2 ) {
01440                 WORD *t, *tstop;
01441                 tstop = term + *term;
01442                 tstop -= ABS(tstop[-1]);
01443                 t = term+1;
01444                 while ( t < tstop ) {
01445                     if ( *t == AR.PolyFun ) {
01446                         t[2] &= ~MUSTCLEANPRF;
01447                     }
01448                     t += t[1];
01449                 }
01450             }
01451             if ( dobracketindex ) {
01452                 PutBracketInIndex(BHEAD term,position);
01453             }
01454         }
01455         ret = i;
01456         ADDPOS(*position,i*sizeof(WORD));
01457         p = fi->POfill;
01458         do {
01459             if ( p >= fi->POstop ) {
01460 #ifdef WITHMPI /* [16mar1998 ar] */
01461               if ( PF.me != MASTER && AR.sLevel <= 0 && (fi == AR.outfile || fi == AR.hidefile) &&
      PF.parallel && PF.exprtodo < 0 ) {
01462                 PF_BUFFER *sbuf = PF.sbuf;
01463                 sbuf->fill[sbuf->active] = fi->POstop;
01464                 PF_ISendSbuf(MASTER,PF_BUFFER_MSGTAG);
01465                 p = fi->PObuffer = fi->POfill = fi->POfull =
01466                   sbuf->buff[sbuf->active];
01467                 fi->POstop = sbuf->stop[sbuf->active];
01468               }
01469               else
01470 #endif /* WITHMPI [16mar1998 ar] */
01471               {
01472                 if ( fi->handle < 0 ) {
01473                     if ( ( RetCode = CreateFile(fi->name) ) >= 0 ) {
01474 #ifdef GZIPDEBUG
01475                         MLOCK(ErrorMessageLock);
01476                         MesPrint("%w PutOut created sortfile %s",fi->name);
01477                         MUNLOCK(ErrorMessageLock);
01478 #endif
01479                         fi->handle = (WORD)RetCode;
01480                         PUTZERO(fi->filesize);
01481                         PUTZERO(fi->POposition);
01482 /*
01483                         Should not be here?
01484 #ifdef WITHZLIB
01485                         fi->ziobuffer = 0;
01486 #endif
01487 */
01488                     }
01489                     else {
01490                         MLOCK(ErrorMessageLock);
01491                         MesPrint("Cannot create scratch file %s",fi->name);
01492                         MUNLOCK(ErrorMessageLock);
01493                         return(-1);
01494                     }
01495                 }
01496 #ifdef WITHZLIB
01497                 if ( !AR.NoCompress && ncomp > 0 && AR.gzipCompress > 0
01498                     && dobracketindex == 0 && fi->zsp != 0 ) {
01499                     fi->POfill = p;
01500                     if ( PutOutputGZIP(fi) ) return(-1);
01501                     p = fi->PObuffer;
01502                 }
01503                 else
01504 #endif
01505                 {
01506 #ifdef ALLLOCK
01507                   LOCK(fi->pthreadslock);
01508 #endif
01509                   if ( fi == AR.hidefile ) {
01510                     LOCK(AS.inputslock);
01511                   }
01512                   SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
01513                   if ( ( RetCode = WriteFile(fi->handle,(UBYTE *)(fi->PObuffer),fi->POsize) ) !=
      fi->POsize ) {
01514                     if ( fi == AR.hidefile ) {
01515                         UNLOCK(AS.inputslock);
01516                     }
01517 #ifdef ALLLOCK
01518                     UNLOCK(fi->pthreadslock);
```

```
01519 #endif
01520                     MLOCK(ErrorMessageLock);
01521                     MesPrint("Write error during sort. Disk full?");
01522                     MesPrint("Attempt to write %l bytes on file %d at position %15p",
01523                                 fi->POsize,fi->handle,&(fi->POposition));
01524                     MesPrint("RetCode = %l, Buffer address = %l",RetCode,(LONG)(fi->PObuffer));
01525                     MUNLOCK(ErrorMessageLock);
01526                     return(-1);
01527                   }
01528                   ADDPOS(fi->filesize,fi->POsize);
01529                   p = fi->PObuffer;
01530                   ADDPOS(fi->POposition,fi->POsize);
01531                   if ( fi == AR.hidefile ) {
01532                     UNLOCK(AS.inputslock);
01533                   }
01534 #ifdef ALLLOCK
01535                   UNLOCK(fi->pthreadslock);
01536 #endif
01537 #ifdef WITHPTHREADS
01538                   if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
01539                     if ( fi->handle >= 0 ) SynchFile(fi->handle);
01540                   }
01541 #endif
01542                 }
01543               }
01544             }
01545             if ( first ) {
01546                 if ( first == 2 ) *p++ = k;
01547                 else *p++ = j;
01548                 first--;
01549             }
01550             else *p++ = *term++;
01551 /*
01552             if ( AP.DebugFlag ) {
01553                 TalToLine((UWORD)(p[-1])); TokenToLine((UBYTE *)"  ");
01554             }
01555 */
01556         } while ( --i > 0 );
01557         fi->POfull = fi->POfill = p;
01558     }
01559 /*
01560     if ( AP.DebugFlag ) {
01561         AO.OutSkip = 0;
01562         FiniLine();
01563     }
01564 */
01565     return(ret);
01566 }
01567
01568 /*
01569         #] PutOut :
01570         #[ FlushOut :               WORD FlushOut(position,file,compr)
01571 */
01581 int FlushOut(POSITION *position, FILEHANDLE *fi, int compr)
01582 {
01583     GETIDENTITY
01584     LONG size, RetCode;
01585     int dobracketindex = 0;
01586 #ifndef WITHZLIB
01587     DUMMYUSE(compr);
01588 #endif
01589     if ( AR.sLevel <= 0 && Expressions[AR.CurExpr].newbracketinfo
01590         && ( fi == AR.outfile || fi == AR.hidefile ) ) dobracketindex = 1;
01591 #ifdef WITHMPI /* [16mar1998 ar] */
01592     if ( PF.me != MASTER && AR.sLevel <= 0 && (fi == AR.outfile || fi == AR.hidefile) && PF.parallel
      && PF.exprtodo < 0 ) {
01593         PF_BUFFER *sbuf = PF.sbuf;
01594         if ( fi->POfill >= fi->POstop ){
01595           sbuf->fill[sbuf->active] = fi->POstop;
01596           PF_ISendSbuf(MASTER,PF_BUFFER_MSGTAG);
01597           fi->POfull = fi->POfill = fi->PObuffer = sbuf->buff[sbuf->active];
01598           fi->POstop = sbuf->stop[sbuf->active];
01599         }
01600         *(fi->POfill)++ = 0;
01601         sbuf->fill[sbuf->active] = fi->POfill;
01602         PF_ISendSbuf(MASTER,PF_ENDBUFFER_MSGTAG);
01603         fi->PObuffer = fi->POfill = fi->POfull = sbuf->buff[sbuf->active];
01604         fi->POstop = sbuf->stop[sbuf->active];
01605         return(0);
01606     }
01607 #endif /* WITHMPI [16mar1998 ar] */
01608     if ( fi->POfill >= fi->POstop ) {
01609         if ( fi->handle < 0 ) {
01610             if ( ( RetCode = CreateFile(fi->name) ) >= 0 ) {
01611 #ifdef GZIPDEBUG
01612                 MLOCK(ErrorMessageLock);
01613                 MesPrint("%w FlushOut created scratch file %s",fi->name);
```

```
01614                   MUNLOCK(ErrorMessageLock);
01615 #endif
01616                   PUTZERO(fi->filesize);
01617                   PUTZERO(fi->POposition);
01618                   fi->handle = (WORD)RetCode;
01619 /*
01620                   Should not be here?
01621 #ifdef WITHZLIB
01622                   fi->ziobuffer = 0;
01623 #endif
01624 */
01625               }
01626               else {
01627                   MLOCK(ErrorMessageLock);
01628                   MesPrint("Cannot create scratch file %s",fi->name);
01629                   MUNLOCK(ErrorMessageLock);
01630                   return(-1);
01631               }
01632           }
01633 #ifdef WITHZLIB
01634           if ( AT.SS == AT.S0 && !AR.NoCompress && AR.gzipCompress > 0
01635           && dobracketindex == 0 && ( compr > 0 ) && fi->zsp != 0 ) {
01636               if ( PutOutputGZIP(fi) ) return(-1);
01637               fi->POfill = fi->POobuffer;
01638           }
01639           else
01640 #endif
01641           {
01642 #ifdef ALLLOCK
01643             LOCK(fi->pthreadslock);
01644 #endif
01645             if ( fi == AR.hidefile ) {
01646               LOCK(AS.inputslock);
01647             }
01648             SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
01649             if ( ( RetCode = WriteFile(fi->handle,(UBYTE *)(fi->POobuffer),fi->POsize) ) != fi->POsize )
      {
01650 #ifdef ALLLOCK
01651               UNLOCK(fi->pthreadslock);
01652 #endif
01653               if ( fi == AR.hidefile ) {
01654                   UNLOCK(AS.inputslock);
01655               }
01656               MLOCK(ErrorMessageLock);
01657               MesPrint("Write error while sorting. Disk full?");
01658               MesPrint("Attempt to write %l bytes on file %d at position %15p",
01659                           fi->POsize,fi->handle,&(fi->POposition));
01660               MesPrint("RetCode = %l, Buffer address = %l",RetCode,(LONG)(fi->POobuffer));
01661               MUNLOCK(ErrorMessageLock);
01662               return(-1);
01663             }
01664             ADDPOS(fi->filesize,fi->POsize);
01665             fi->POfill = fi->POobuffer;
01666             ADDPOS(fi->POposition,fi->POsize);
01667             if ( fi == AR.hidefile ) {
01668               UNLOCK(AS.inputslock);
01669             }
01670 #ifdef ALLLOCK
01671             UNLOCK(fi->pthreadslock);
01672 #endif
01673 #ifdef WITHPTHREADS
01674             if ( AS.MasterSort && AC.ThreadSortFileSynch && fi != AR.hidefile ) {
01675               if ( fi->handle >= 0 ) SynchFile(fi->handle);
01676             }
01677 #endif
01678           }
01679       }
01680       *(fi->POfill)++ = 0;
01681       fi->POfull = fi->POfill;
01682 /*
01683       {
01684           UBYTE OutBuf[140];
01685           if ( AP.DebugFlag ) {
01686               AO.OutFill = AO.OutputLine = OutBuf;
01687               AO.OutSkip = 3;
01688               FiniLine();
01689               TokenToLine((UBYTE *)"End of expression written");
01690               FiniLine();
01691           }
01692       }
01693 */
01694       size = (fi->POfill-fi->POobuffer)*sizeof(WORD);
01695       if ( fi->handle >= 0 ) {
01696 #ifdef WITHZLIB
01697           if ( AT.SS == AT.S0 && !AR.NoCompress && AR.gzipCompress > 0
01698           && dobracketindex == 0 && ( compr > 0 ) && fi->zsp != 0 ) {
01699               if ( FlushOutputGZIP(fi) ) return(-1);
```

```
01700             fi->POfill = fi->PObuffer;
01701         }
01702         else
01703 #endif
01704         {
01705 #ifdef ALLLOCK
01706           LOCK(fi->pthreadslock);
01707 #endif
01708           if ( fi == AR.hidefile ) {
01709             LOCK(AS.inputslock);
01710           }
01711           SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
01712 /*
01713           MesPrint("FlushOut: writing %l bytes to position %12p",size,&(fi->POposition));
01714 */
01715           if ( ( RetCode = WriteFile(fi->handle,(UBYTE *)(fi->PObuffer),size) ) != size ) {
01716 #ifdef ALLLOCK
01717             UNLOCK(fi->pthreadslock);
01718 #endif
01719             if ( fi == AR.hidefile ) {
01720                 UNLOCK(AS.inputslock);
01721             }
01722             MLOCK(ErrorMessageLock);
01723             MesPrint("Write error while finishing sorting. Disk full?");
01724             MesPrint("Attempt to write %l bytes on file %d at position %15p",
01725                       size,fi->handle,&(fi->POposition));
01726             MesPrint("RetCode = %l, Buffer address = %l",RetCode,(LONG)(fi->PObuffer));
01727             MUNLOCK(ErrorMessageLock);
01728             return(-1);
01729           }
01730           ADDPOS(fi->filesize,size);
01731           ADDPOS(fi->POposition,size);
01732           fi->POfill = fi->PObuffer;
01733           if ( fi == AR.hidefile ) {
01734             UNLOCK(AS.inputslock);
01735           }
01736 #ifdef ALLLOCK
01737           UNLOCK(fi->pthreadslock);
01738 #endif
01739 #ifdef WITHPTHREADS
01740           if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
01741             if ( fi->handle >= 0 ) SynchFile(fi->handle);
01742           }
01743 #endif
01744         }
01745     }
01746     if ( dobracketindex ) {
01747         BRACKETINFO *b = Expressions[AR.CurExpr].newbracketinfo;
01748         if ( b->indexfill > 0 ) {
01749             DIFPOS(b->indexbuffer[b->indexfill-1].next,*position,Expressions[AR.CurExpr].onfile);
01750         }
01751     }
01752 #ifdef WITHZLIB
01753     if ( AT.SS == AT.S0 && !AR.NoCompress && AR.gzipCompress > 0
01754         && dobracketindex == 0 && ( compr > 0 ) && fi->zsp != 0 ) {
01755         PUTZERO(*position);
01756         if ( fi->handle >= 0 ) {
01757 #ifdef ALLLOCK
01758             LOCK(fi->pthreadslock);
01759 #endif
01760             SeekFile(fi->handle,position,SEEK_END);
01761 #ifdef ALLLOCK
01762             UNLOCK(fi->pthreadslock);
01763 #endif
01764         }
01765         else {
01766             ADDPOS(*position,((UBYTE *)fi->POfill-(UBYTE *)fi->PObuffer));
01767         }
01768     }
01769     else
01770 #endif
01771     {
01772         ADDPOS(*position,sizeof(WORD));
01773     }
01774     return(0);
01775 }
01776
01777 /*
01778         #] FlushOut :
01779         #[ AddCoef :                WORD AddCoef(pterm1,pterm2)
01780 */
01795 int AddCoef(PHEAD WORD **ps1, WORD **ps2)
01796 {
01797     GETBIDENTITY
01798     SORTING *S = AT.SS;
01799     WORD *s1, *s2;
01800     WORD l1, l2, i;
```

```
01801     WORD OutLen, *t, j;
01802     UWORD *OutCoef;
01803 #ifdef WITHFLOAT
01804     if ( AT.SortFloatMode ) return(AddWithFloat(BHEAD ps1,ps2));
01805 #endif
01806     OutCoef = AN.SoScratC;
01807     s1 = *ps1; s2 = *ps2;
01808     GETCOEF(s1,l1);
01809     GETCOEF(s2,l2);
01810     if ( AddRat(BHEAD (UWORD *)s1,l1,(UWORD *)s2,l2,OutCoef,&OutLen) ) {
01811         MLOCK(ErrorMessageLock);
01812         MesCall("AddCoef");
01813         MUNLOCK(ErrorMessageLock);
01814         Terminate(-1);
01815     }
01816     if ( AN.ncmod != 0 ) {
01817         if ( ( AC.modmode & POSNEG ) != 0 ) {
01818             NormalModulus(OutCoef,&OutLen);
01819 /*
01820             We had forgotten that this can also become smaller but the
01821             denominator isn't there. Correct in the other case
01822             17-may-2009 [JV]
01823 */
01824             j = ABS(OutLen); OutCoef[j] = 1;
01825             for ( i = 1; i < j; i++ ) OutCoef[j+i] = 0;
01826         }
01827         else if ( BigLong(OutCoef,OutLen,(UWORD *)AC.cmod,ABS(AN.ncmod)) >= 0 ) {
01828             SubPLon(OutCoef,OutLen,(UWORD *)AC.cmod,ABS(AN.ncmod),OutCoef,&OutLen);
01829             OutCoef[OutLen] = 1;
01830             for ( i = 1; i < OutLen; i++ ) OutCoef[OutLen+i] = 0;
01831         }
01832     }
01833     if ( !OutLen ) { *ps1 = *ps2 = 0; return(0); }
01834     OutLen *= 2;
01835     if ( OutLen < 0 ) i = - ( --OutLen );
01836     else              i = ++OutLen;
01837     if ( l1 < 0 ) l1 = -l1;
01838     l1 *= 2; l1++;
01839     if ( i <= l1 ) {    /* Fits in 1 */
01840         l1 -= i;
01841         **ps1 -= l1;
01842         s2 = (WORD *)OutCoef;
01843         while ( --i > 0 ) *s1++ = *s2++;
01844         *s1++ = OutLen;
01845         while ( --l1 >= 0 ) *s1++ = 0;
01846         goto RegEnd;
01847     }
01848     if ( l2 < 0 ) l2 = -l2;
01849     l2 *= 2; l2++;
01850     if ( i <= l2 ) {    /* Fits in 2 */
01851         l2 -= i;
01852         **ps2 -= l2;
01853         s1 = (WORD *)OutCoef;
01854         while ( --i > 0 ) *s2++ = *s1++;
01855         *s2++ = OutLen;
01856         while ( --l2 >= 0 ) *s2++ = 0;
01857         *ps1 = *ps2;
01858         goto RegEnd;
01859     }
01860
01861     /* Doesn't fit. Make a new term. */
01862
01863     t = s1;
01864     s1 = *ps1;
01865     j = *s1++ + i - l1;     /* Space needed */
01866     if ( (S->sFill + j) >= S->sTop2 ) {
01867         GarbHand();
01868         s1 = *ps1;
01869         t = s1 + *s1 - 1;
01870         j = *s1++ + i - l1;     /* Space needed */
01871         l1 = *t;
01872         if ( l1 < 0 ) l1  = - l1;
01873         t -= l1-1;
01874     }
01875     s2 = S->sFill;
01876     *s2++ = j;
01877     while ( s1 < t ) *s2++ = *s1++;
01878     s1 = (WORD *)OutCoef;
01879     while ( --i > 0 ) *s2++ = *s1++;
01880     *s2++ = OutLen;
01881     *ps1 = S->sFill;
01882     S->sFill = s2;
01883 RegEnd:
01884     *ps2 = 0;
01885     if ( **ps1 > AM.MaxTer/((LONG)(sizeof(WORD))) ) {
01886         MLOCK(ErrorMessageLock);
01887         MesPrint("Term too complex after addition in sort. MaxTermSize = %10l",
```

```
01888         AM.MaxTer/sizeof(WORD));
01889         MUNLOCK(ErrorMessageLock);
01890         Terminate(-1);
01891     }
01892     if ( **ps1 > S->verbMaxTermSize ) S->verbMaxTermSize = **ps1;
01893     return(1);
01894 }
01895
01896 /*
01897         #] AddCoef :
01898         #[ AddPoly :                WORD AddPoly(pterm1,pterm2)
01899 */
01925 int AddPoly(PHEAD WORD **ps1, WORD **ps2)
01926 {
01927     GETBIDENTITY
01928     SORTING *S = AT.SS;
01929     WORD i;
01930     WORD *s1, *s2, *m, *w, *t, oldpw = S->PolyWise;
01931     s1 = *ps1 + S->PolyWise;
01932     s2 = *ps2 + S->PolyWise;
01933     w = AT.WorkPointer;
01934 /*
01935     Add here the two arguments. Is a straight merge.
01936 */
01937     if ( S->PolyFlag == 2 && AR.PolyFunExp != 2 && AR.PolyFunExp != 3 ) {
01938         WORD **oldSplitScratch = AN.SplitScratch;
01939         LONG oldSplitScratchSize = AN.SplitScratchSize;
01940         LONG oldInScratch = AN.InScratch;
01941         WORD oldtype = AR.SortType;
01942         if ( (WORD *)((UBYTE *)w + AM.MaxTer) >= AT.WorkTop ) {
01943             MLOCK(ErrorMessageLock);
01944             MesPrint("Program was adding polyratfun arguments");
01945             MesWork();
01946             MUNLOCK(ErrorMessageLock);
01947         }
01948         AR.SortType = SORTHIGHFIRST;
01949         S->PolyWise = 0;
01950         AN.SplitScratch = AN.SplitScratch1;
01951         AN.SplitScratchSize = AN.SplitScratchSize1;
01952         AN.InScratch = AN.InScratch1;
01953         poly_ratfun_add(BHEAD s1,s2);
01954         S->PolyWise = oldpw;
01955         AN.SplitScratch1 = AN.SplitScratch;
01956         AN.SplitScratchSize1 = AN.SplitScratchSize;
01957         AN.InScratch1 = AN.InScratch;
01958         AN.SplitScratch = oldSplitScratch;
01959         AN.SplitScratchSize = oldSplitScratchSize;
01960         AN.InScratch = oldInScratch;
01961         AT.WorkPointer = w;
01962         AR.SortType = oldtype;
01963         if ( w[1] <= FUNHEAD ||
01964             ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 ) ) {
01965             *ps1 = *ps2 = 0; return(0);
01966         }
01967     }
01968     else {
01969         if ( w + s1[1] + s2[1] + 12 + ARGHEAD >= AT.WorkTop ) {
01970             MLOCK(ErrorMessageLock);
01971             MesPrint("Program was adding polyfun arguments");
01972             MesWork();
01973             MUNLOCK(ErrorMessageLock);
01974         }
01975         AddArgs(BHEAD s1,s2,w);
01976     }
01977 /*
01978     Now we need to store the result in a convenient place.
01979 */
01980     if ( w[1] <= FUNHEAD ) { *ps1 = *ps2 = 0; return(0); }
01981     if ( w[1] <= s1[1] || w[1] <= s2[1] ) {   /* Fits in place. */
01982         if ( w[1] > s1[1] ) {
01983             *ps1 = *ps2;
01984             s1 = s2;
01985         }
01986         t = s1 + s1[1];
01987         m = *ps1 + **ps1;
01988         i = w[1];
01989         NCOPY(s1,w,i);
01990         if ( s1 != t ) {
01991             while ( t < m ) *s1++ = *t++;
01992             **ps1 = WORDDIF(s1,(*ps1));
01993         }
01994         *ps2 = 0;
01995     }
01996     else {      /* Make new term */
01997 #ifdef TESTGARB
01998         s2 = *ps2;
01999 #endif
```

```
02000         *ps2 = 0;
02001         if ( (S->sFill + (**ps1 + w[1] - s1[1])) >= S->sTop2 ) {
02002 #ifdef TESTGARB
02003             MesPrint("------Garbage collection-------");
02004 #endif
02005             AT.WorkPointer += w[1];
02006             GarbHand();
02007             AT.WorkPointer = w;
02008             s1 = *ps1;
02009             if ( (S->sFill + (**ps1 + w[1] - s1[1])) >= S->sTop2 ) {
02010 #ifdef TESTGARB
02011                 UBYTE OutBuf[140];
02012                 MLOCK(ErrorMessageLock);
02013                 AO.OutFill = AO.OutputLine = OutBuf;
02014                 AO.OutSkip = 3;
02015                 FiniLine();
02016                 i = *s2;
02017                 while ( --i >= 0 ) {
02018                     TalToLine((UWORD)(*s2++)); TokenToLine((UBYTE *)"  ");
02019                 }
02020                 FiniLine();
02021                 AO.OutFill = AO.OutputLine = OutBuf;
02022                 AO.OutSkip = 3;
02023                 FiniLine();
02024                 s2 = *ps1;
02025                 i = *s2;
02026                 while ( --i >= 0 ) {
02027                     TalToLine((UWORD)(*s2++)); TokenToLine((UBYTE *)"  ");
02028                 }
02029                 FiniLine();
02030                 AO.OutFill = AO.OutputLine = OutBuf;
02031                 AO.OutSkip = 3;
02032                 FiniLine();
02033                 s2 = w;
02034                 i = w[1];
02035                 while ( --i >= 0 ) {
02036                     TalToLine((UWORD)(*s2++)); TokenToLine((UBYTE *)"  ");
02037                 }
02038                 FiniLine();
02039                 if ( AR.sLevel > 0 ) {
02040                     MesPrint("Please increase SubSmallExtension setup parameter.");
02041                 }
02042                 else {
02043                     MesPrint("Please increase SmallExtension setup parameter.");
02044                 }
02045                 MUNLOCK(ErrorMessageLock);
02046 #else
02047                 MLOCK(ErrorMessageLock);
02048                 if ( AR.sLevel > 0 ) {
02049                     MesPrint("Please increase SubSmallExtension setup parameter.");
02050                 }
02051                 else {
02052                     MesPrint("Please increase SmallExtension setup parameter.");
02053                 }
02054                 MUNLOCK(ErrorMessageLock);
02055 #endif
02056                 Terminate(-1);
02057             }
02058         }
02059         t = *ps1;
02060         s2 = S->sFill;
02061         m = s2;
02062         i = S->PolyWise;
02063         NCOPY(s2,t,i);
02064         i = w[1];
02065         NCOPY(s2,w,i);
02066         t = t + t[1];
02067         w = *ps1 + **ps1;
02068         while ( t < w ) *s2++ = *t++;
02069         *m = WORDDIF(s2,m);
02070         *ps1 = m;
02071         S->sFill = s2;
02072         if ( *m > AM.MaxTer/((LONG)sizeof(WORD)) ) {
02073             MLOCK(ErrorMessageLock);
02074             MesPrint("Term too complex after polynomial addition. MaxTermSize = %10l",
02075             AM.MaxTer/sizeof(WORD));
02076             MUNLOCK(ErrorMessageLock);
02077             Terminate(-1);
02078         }
02079         if ( *m > S->verbMaxTermSize ) S->verbMaxTermSize = *m;
02080     }
02081     return(1);
02082 }
02083
02084 /*
02085         #] AddPoly :
02086         #[ AddArgs :                void AddArgs(arg1,arg2,to)
```

```
02087 */
02088
02089 #define INSLENGTH(x)  w[1] = FUNHEAD+ARGHEAD+x; w[FUNHEAD] = ARGHEAD+x;
02090
02098 void AddArgs(PHEAD WORD *s1, WORD *s2, WORD *m)
02099 {
02100     GETBIDENTITY
02101     WORD i1, i2;
02102     WORD *w = m, *mm, *t, *t1, *t2, *tstop1, *tstop2;
02103     WORD tempterm[8+FUNHEAD];
02104
02105     *m++ = AR.PolyFun; *m++ = 0; FILLFUN(m)
02106     *m++ = 0; *m++ = 0; FILLARG(m)
02107     if ( s1[FUNHEAD] < 0 || s2[FUNHEAD] < 0 ) {
02108         if ( s1[FUNHEAD] < 0 ) {
02109             if ( s2[FUNHEAD] < 0 ) {    /* Both are special */
02110                 if ( s1[FUNHEAD] <= -FUNCTION ) {
02111                     if ( s2[FUNHEAD] == s1[FUNHEAD] ) {
02112                         *m++ = 4+FUNHEAD; *m++ = -s1[FUNHEAD]; *m++ = FUNHEAD;
02113                         FILLFUN(m)
02114                         *m++ = 2; *m++ = 1; *m++ = 3;
02115                         INSLENGTH(4+FUNHEAD)
02116                     }
02117                     else if ( s2[FUNHEAD] <= -FUNCTION ) {
02118                         i1 = functions[-FUNCTION-s1[FUNHEAD]].commute != 0;
02119                         i2 = functions[-FUNCTION-s2[FUNHEAD]].commute != 0;
02120                         if ( ( !i1 && i2 ) || ( i1 == i2 && i1 > i2 ) ) {
02121                             i1 = s2[FUNHEAD];
02122                             s2[FUNHEAD] = s1[FUNHEAD];
02123                             s1[FUNHEAD] = i1;
02124                         }
02125                         *m++ = 4+FUNHEAD; *m++ = -s1[FUNHEAD]; *m++ = FUNHEAD;
02126                         FILLFUN(m)
02127                         *m++ = 1; *m++ = 1; *m++ = 3;
02128                         *m++ = 4+FUNHEAD; *m++ = -s2[FUNHEAD]; *m++ = FUNHEAD;
02129                         FILLFUN(m)
02130                         *m++ = 1; *m++ = 1; *m++ = 3;
02131                         INSLENGTH(8+2*FUNHEAD)
02132                     }
02133                     else if ( s2[FUNHEAD] == -SYMBOL ) {
02134                         *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = s2[FUNHEAD+1]; *m++ = 1;
02135                         *m++ = 1; *m++ = 1; *m++ = 3;
02136                         *m++ = 4+FUNHEAD; *m++ = -s1[FUNHEAD]; *m++ = FUNHEAD;
02137                         FILLFUN(m)
02138                         *m++ = 1; *m++ = 1; *m++ = 3;
02139                         INSLENGTH(12+FUNHEAD)
02140                     }
02141                     else {      /* number */
02142                         *m++ = 4;
02143                         *m++ = ABS(s2[FUNHEAD+1]); *m++ = 1; *m++ = s2[FUNHEAD+1] < 0 ? -3: 3;
02144                         *m++ = 4+FUNHEAD; *m++ = -s1[FUNHEAD]; *m++ = FUNHEAD;
02145                         FILLFUN(m)
02146                         *m++ = 1; *m++ = 1; *m++ = 3;
02147                         INSLENGTH(8+FUNHEAD)
02148                     }
02149                 }
02150                 else if ( s1[FUNHEAD] == -SYMBOL ) {
02151                     if ( s2[FUNHEAD] == s1[FUNHEAD] ) {
02152                         if ( s1[FUNHEAD+1] == s2[FUNHEAD+1] ) {
02153                             *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = s1[FUNHEAD+1];
02154                             *m++ = 1; *m++ = 2; *m++ = 1; *m++ = 3;
02155                             INSLENGTH(8)
02156                         }
02157                         else {
02158                             if ( s1[FUNHEAD+1] > s2[FUNHEAD+1] )
02159                                 { i1 = s2[FUNHEAD+1]; i2 = s1[FUNHEAD+1]; }
02160                             else { i1 = s1[FUNHEAD+1]; i2 = s2[FUNHEAD+1]; }
02161                             *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = i1;
02162                             *m++ = 1; *m++ = 1; *m++ = 1; *m++ = 3;
02163                             *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = i2;
02164                             *m++ = 1; *m++ = 1; *m++ = 1; *m++ = 3;
02165                             INSLENGTH(16)
02166                         }
02167                     }
02168                     else if ( s2[FUNHEAD] <= -FUNCTION ) {
02169                         *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = s1[FUNHEAD+1]; *m++ = 1;
02170                         *m++ = 1; *m++ = 1; *m++ = 3;
02171                         *m++ = 4+FUNHEAD; *m++ = -s2[FUNHEAD]; *m++ = FUNHEAD;
02172                         FILLFUN(m)
02173                         *m++ = 1; *m++ = 1; *m++ = 3;
02174                         INSLENGTH(12+FUNHEAD)
02175                     }
02176                     else {
02177                         *m++ = 4;
02178                         *m++ = ABS(s2[FUNHEAD+1]); *m++ = 1; *m++ = s2[FUNHEAD+1] < 0 ? -3: 3;
02179                         *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = s1[FUNHEAD+1]; *m++ = 1;
02180                         *m++ = 1; *m++ = 1; *m++ = 3;
```

```
02181                       INSLENGTH(12)
02182                   }
02183               }
02184               else {  /* Must be -SNUMBER! */
02185                   if ( s2[FUNHEAD] <= -FUNCTION ) {
02186                       *m++ = 4;
02187                       *m++ = ABS(s1[FUNHEAD+1]); *m++ = 1; *m++ = s1[FUNHEAD+1] < 0 ? -3: 3;
02188                       *m++ = 4+FUNHEAD; *m++ = -s2[FUNHEAD]; *m++ = FUNHEAD;
02189                       FILLFUN(m)
02190                       *m++ = 1; *m++ = 1; *m++ = 3;
02191                       INSLENGTH(8+FUNHEAD)
02192                   }
02193                   else if ( s2[FUNHEAD] == -SYMBOL ) {
02194                       *m++ = 4;
02195                       *m++ = ABS(s1[FUNHEAD+1]); *m++ = 1; *m++ = s1[FUNHEAD+1] < 0 ? -3: 3;
02196                       *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = s2[FUNHEAD+1]; *m++ = 1;
02197                       *m++ = 1; *m++ = 1; *m++ = 3;
02198                       INSLENGTH(12)
02199                   }
02200                   else {      /* Both are numbers. add. */
02201                       LONG x1;
02202                       x1 = (LONG)s1[FUNHEAD+1] + (LONG)s2[FUNHEAD+1];
02203                       if ( x1 < 0 ) { i1 = (WORD)(-x1); i2 = -3; }
02204                       else { i1 = (WORD)x1; i2 = 3; }
02205                       if ( x1 && AN.ncmod != 0 ) {
02206                           m[0] = 4;
02207                           m[1] = i1;
02208                           m[2] = 1;
02209                           m[3] = i2;
02210                           if ( Modulus(m) ) Terminate(-1);
02211                           if ( *m == 0 ) w[1] = 0;
02212                           else {
02213                               if ( *m == 4 && ( m[1] & MAXPOSITIVE ) == m[1]
02214                               && m[3] == 3 ) {
02215                                   i1 = m[1];
02216                                   m -= ARGHEAD;
02217                                   *m++ = -SNUMBER;
02218                                   *m++ = i1;
02219                                   INSLENGTH(4)
02220                               }
02221                               else {
02222                                   INSLENGTH(*m)
02223                                   m += *m;
02224                               }
02225                           }
02226                       }
02227                       else {
02228                           if ( x1 == 0 ) {
02229                               w[1] = FUNHEAD;
02230                           }
02231                           else if ( ( i1 & MAXPOSITIVE ) == i1 ) {
02232                               m -= ARGHEAD;
02233                               *m++ = -SNUMBER;
02234                               *m++ = (WORD)x1;
02235                               w[1] = FUNHEAD+2;
02236                           }
02237                           else {
02238                               *m++ = 4; *m++ = i1; *m++ = 1; *m++ = i2;
02239                               INSLENGTH(4)
02240                           }
02241                       }
02242                   }
02243               }
02244           }
02245           else {  /* Only s1 is special */
02246 s1only:
02247 /*
02248           Compose a term in `tempterm'
02249 */
02250               t = tempterm;
02251               if ( s1[FUNHEAD] <= -FUNCTION ) {
02252                   *t++ = 4+FUNHEAD; *t++ = -s1[FUNHEAD]; *t++ = FUNHEAD;
02253                   FILLFUN(t)
02254                   *t++ = 1; *t++ = 1; *t++ = 3;
02255               }
02256               else if ( s1[FUNHEAD] == -SYMBOL ) {
02257                   *t++ = 8; *t++ = SYMBOL; *t++ = 4;
02258                   *t++ = s1[FUNHEAD+1]; *t++ = 1;
02259                   *t++ = 1; *t++ = 1; *t++ = 3;
02260               }
02261               else {
02262                   *t++ = 4;   *t++ = ABS(s1[FUNHEAD+1]);
02263                   *t++ = 1;   *t++ = s1[FUNHEAD+1] < 0 ? -3: 3;
02264               }
02265               tstop1 = t;
02266               s1 = tempterm;
02267               goto twogen;
```

```
02268             }
02269         }
02270         else {      /* Only s2 is special */
02271             t = s1;
02272             s1 = s2;
02273             s2 = t;
02274             goto s1only;
02275         }
02276     }
02277     else {
02278         int oldPolyFlag;
02279         tstop1 = s1 + s1[1];
02280         s1 += FUNHEAD+ARGHEAD;
02281 twogen:
02282         tstop2 = s2 + s2[1];
02283         s2 += FUNHEAD+ARGHEAD;
02284 /*
02285         Now we should merge the expressions in s1 and s2 into m.
02286 */
02287         oldPolyFlag = AT.SS->PolyFlag;
02288         AT.SS->PolyFlag = 0;
02289         while ( s1 < tstop1 && s2 < tstop2 ) {
02290             i1 = CompareTerms(BHEAD s1,s2,(WORD)(-1));
02291             if ( i1 > 0 ) {
02292                 i2 = *s1;
02293                 NCOPY(m,s1,i2);
02294             }
02295             else if ( i1 < 0 ) {
02296                 i2 = *s2;
02297                 NCOPY(m,s2,i2);
02298             }
02299             else {  /* Coefficients should be added. */
02300                 WORD i;
02301                 t = s1+*s1;
02302                 i1 = t[-1];
02303                 i2 = *s1 - ABS(i1);
02304                 t2 = s2 + i2;
02305                 s2 += *s2;
02306                 mm = m;
02307                 NCOPY(m,s1,i2);
02308                 t1 = s1;
02309                 s1 = t;
02310                 i2 = s2[-1];
02311 /*
02312                 t1,i1 is the first coefficient
02313                 t2,i2 is the second coefficient
02314                 It should be placed at m,i1
02315 */
02316                 i1 = REDLENG(i1);
02317                 i2 = REDLENG(i2);
02318                 if ( AddRat(BHEAD (UWORD *)t1,i1,(UWORD *)t2,i2,(UWORD *)m,&i) ) {
02319 /* INTERNAL_ERROR_EXCL_START */
02320                     MLOCK(ErrorMessageLock);
02321                     MesPrint("!>Addition of coefficients of PolyFun");
02322                     MUNLOCK(ErrorMessageLock);
02323                     Terminate(-1);
02324 /* INTERNAL_ERROR_EXCL_STOP */
02325                 }
02326                 if ( i == 0 ) {
02327                     m = mm;
02328                 }
02329                 else {
02330                     i1 = INCLENG(i);
02331                     m += ABS(i1);
02332                     m[-1] = i1;
02333                     *mm = WORDDIF(m,mm);
02334                     if ( AN.ncmod != 0 ) {
02335                         if ( Modulus(mm) ) Terminate(-1);
02336                         if ( !*mm ) m = mm;
02337                         else m = mm + *mm;
02338                     }
02339                 }
02340             }
02341         }
02342         while ( s1 < tstop1 ) *m++ = *s1++;
02343         while ( s2 < tstop2 ) *m++ = *s2++;
02344         w[1] = WORDDIF(m,w);
02345         w[FUNHEAD] = w[1] - FUNHEAD;
02346         if ( ToFast(w+FUNHEAD,w+FUNHEAD) ) {
02347             if ( w[FUNHEAD] <= -FUNCTION ) w[1] = FUNHEAD+1;
02348             else w[1] = FUNHEAD+2;
02349             if ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 ) w[1] = FUNHEAD;
02350         }
02351 /*      AT.SS->PolyFlag = AR.PolyFunType;*/
02352         AT.SS->PolyFlag = oldPolyFlag;
02353     }
02354 }
```

```
02355
02356 /*
02357         #] AddArgs :
02358         #[ Compare1 :               WORD Compare1(term1,term2,level)
02359 */
02393 WORD Compare1(PHEAD WORD *term1, WORD *term2, WORD level)
02394 {
02395     SORTING *S = AT.SS;
02396     WORD *stopper1, *stopper2, *t2;
02397     WORD *s1, *s2, *t1;
02398     WORD *stopex1, *stopex2;
02399     WORD c1, c2;
02400     WORD prevorder;
02401     WORD count = -1, localPoly, polyhit = -1;
02402
02403     // Update SortVerbose counter
02404     S->verbComparisons++;
02405
02406     if ( S->PolyFlag ) {
02407 /*
02408         if ( S->PolyWise != 0 ) {
02409             MLOCK(ErrorMessageLock);
02410             MesPrint("S->PolyWise is not zero!!!!!");
02411             MUNLOCK(ErrorMessageLock);
02412         }
02413 */
02414         count = 0; localPoly = 1; S->PolyWise = polyhit = 0;
02415         S->PolyFlag = AR.PolyFunType;
02416         if ( AR.PolyFunType == 2 &&
02417              ( AR.PolyFunExp == 2 || AR.PolyFunExp == 3 ) ) S->PolyFlag = 1;
02418     }
02419     else { localPoly = 0; }
02420 #ifdef WITHFLOAT
02421     AT.SortFloatMode = 0;
02422 #endif
02423     prevorder = 0;
02424     GETSTOP(term1,s1);
02425     stopper1 = s1;
02426     GETSTOP(term2,stopper2);
02427     t1 = term1 + 1;
02428     t2 = term2 + 1;
02429     while ( t1 < stopper1 && t2 < stopper2 ) {
02430         if ( *t1 != *t2 ) {
02431             if ( *t1 == HAAKJE ) return(PREV(-1));
02432             if ( *t2 == HAAKJE ) return(PREV(1));
02433             if ( *t1 >= (FUNCTION-1) ) {
02434                 if ( *t2 < (FUNCTION-1) ) return(PREV(-1));
02435                 if ( *t1 < FUNCTION && *t2 < FUNCTION ) return(PREV(*t2-*t1));
02436                 if ( *t1 < FUNCTION ) return(PREV(1));
02437                 if ( *t2 < FUNCTION ) return(PREV(-1));
02438                 c1 = functions[*t1-FUNCTION].commute;
02439                 c2 = functions[*t2-FUNCTION].commute;
02440                 if ( !c1 ) {
02441                     if ( c2 ) return(PREV(1));
02442                     else return(PREV(*t2-*t1));
02443                 }
02444                 else {
02445                     if ( !c2 ) return(PREV(-1));
02446                     else return(PREV(*t2-*t1));
02447                 }
02448             }
02449             else return(PREV(*t2-*t1));
02450         }
02451         s1 = t1 + 2;
02452         s2 = t2 + 2;
02453         c1 = *t1;
02454         t1 += t1[1];
02455         t2 += t2[1];
02456         if ( localPoly && c1 < FUNCTION ) {
02457             polyhit = 1;
02458         }
02459         if ( c1 <= (FUNCTION-1)
02460         || ( c1 >= FUNCTION && functions[c1-FUNCTION].spec > 0 ) ) {
02461             if ( c1 == SYMBOL ) {
02462                 if ( *s1 == FACTORSYMBOL && *s2 == FACTORSYMBOL
02463                  && s1[-1] == 4 && s2[-1] == 4
02464                  && ( ( t1 < stopper1 && *t1 == HAAKJE )
02465                  || ( t1 == stopper1 && AT.fromindex ) ) ) {
02466 /*
02467                     We have to be very careful with the criteria here, because
02468                     Compare1 is called both in the regular sorting and by the
02469                     routine that makes the bracket index. In the last case
02470                     there is no HAAKJE subterm.
02471 */
02472                     if ( s1[1] != s2[1] ) return(s2[1]-s1[1]);
02473                     s1 += 2; s2 += 2;
02474                 }
```

```
02475                 else if ( AR.SortType >= SORTPOWERFIRST ) {
02476                     WORD i1 = 0, *r1;
02477                     r1 = s1;
02478                     while ( s1 < t1 ) { i1 += s1[1]; s1 += 2; }
02479                     s1 = r1; r1 = s2;
02480                     while ( s2 < t2 ) { i1 -= s2[1]; s2 += 2; }
02481                     s2 = r1;
02482                     if ( i1 ) {
02483                         if ( AR.SortType >= SORTANTIPOWER ) i1 = -i1;
02484                         return(PREV(i1));
02485                     }
02486                 }
02487                 while ( s1 < t1 ) {
02488                     if ( s2 >= t2 ) {
02489 /*                      return(PREV(1));  */
02490                         if ( AR.SortType==SORTLOWFIRST ) {
02491                             return(PREV((s1[1]>0?-1:1)));
02492                         }
02493                         else {
02494                             return(PREV((s1[1]<0?-1:1)));
02495                         }
02496                     }
02497                     if ( *s1 != *s2 ) {
02498 /*                      return(PREV(*s2-*s1)); */
02499                         if ( AR.SortType==SORTLOWFIRST ) {
02500                             if ( *s1 < *s2 ) {
02501                                 return(PREV((s1[1]<0?1:-1)));
02502                             }
02503                             else {
02504                                 return(PREV((s2[1]<0?-1:1)));
02505                             }
02506                         }
02507                         else {
02508                             if ( *s1 < *s2 ) {
02509                                 return(PREV((s1[1]<0?-1:1)));
02510                             }
02511                             else {
02512                                 return(PREV((s2[1]<0?1:-1)));
02513                             }
02514                         }
02515                     }
02516                     s1++; s2++;
02517                     if ( *s1 != *s2 ) return(
02518                         PREV((AR.SortType==SORTLOWFIRST?*s2-*s1:*s1-*s2)));
02519                     s1++; s2++;
02520                 }
02521                 if ( s2 < t2 ) {
02522 /*                  return(PREV(-1));  */
02523                     if ( AR.SortType==SORTLOWFIRST ) {
02524                         return(PREV((s2[1]<0?-1:1)));
02525                     }
02526                     else {
02527                         return(PREV((s2[1]<0?1:-1)));
02528                     }
02529                 }
02530             }
02531             else if ( c1 == DOTPRODUCT ) {
02532                 if ( AR.SortType >= SORTPOWERFIRST ) {
02533                     WORD i1 = 0, *r1;
02534                     r1 = s1;
02535                     while ( s1 < t1 ) { i1 += s1[2]; s1 += 3; }
02536                     s1 = r1; r1 = s2;
02537                     while ( s2 < t2 ) { i1 -= s2[2]; s2 += 3; }
02538                     s2 = r1;
02539                     if ( i1 ) {
02540                         if ( AR.SortType >= SORTANTIPOWER ) i1 = -i1;
02541                         return(PREV(i1));
02542                     }
02543                 }
02544                 while ( s1 < t1 ) {
02545                     if ( s2 >= t2 ) return(PREV(1));
02546                     if ( *s1 != *s2 ) return(PREV(*s2-*s1));
02547                     s1++; s2++;
02548                     if ( *s1 != *s2 ) return(PREV(*s2-*s1));
02549                     s1++; s2++;
02550                     if ( *s1 != *s2 ) return(
02551                         PREV((AR.SortType==SORTLOWFIRST?*s2-*s1:*s1-*s2)));
02552                     s1++; s2++;
02553                 }
02554                 if ( s2 < t2 ) return(PREV(-1));
02555             }
02556             else {
02557                 while ( s1 < t1 ) {
02558                     if ( s2 >= t2 ) return(PREV(1));
02559                     if ( *s1 != *s2 ) return(PREV(*s2-*s1));
02560                     s1++; s2++;
02561                 }
```

```
02562                 if ( s2 < t2 ) return(PREV(-1));
02563             }
02564         }
02565         else {
02566 #if FUNHEAD != 2
02567             s1 += FUNHEAD-2;
02568             s2 += FUNHEAD-2;
02569 #endif
02570             if ( localPoly && c1 == AR.PolyFun ) {
02571                 if ( count == 0 ) {
02572                   if ( S->PolyFlag == 1 ) {
02573                     WORD i1, i2;
02574                     if ( *s1 > 0 ) i1 = *s1;
02575                     else if ( *s1 <= -FUNCTION ) i1 = 1;
02576                     else i1 = 2;
02577                     if ( *s2 > 0 ) i2 = *s2;
02578                     else if ( *s2 <= -FUNCTION ) i2 = 1;
02579                     else i2 = 2;
02580                     if ( s1+i1 == t1 && s2+i2 == t2 ) { /* This is the stuff */
02581 /*
02582                         Test for scalar nature
02583 */
02584                         if ( !polyhit ) {
02585                             WORD *u1, *u2, *ustop;
02586                             if ( *s1 < 0 ) {
02587                                 if ( *s1 != -SNUMBER && *s1 != -SYMBOL && *s1 > -FUNCTION )
02588                                     goto NoPoly;
02589                             }
02590                             else {
02591                                 u1 = s1 + ARGHEAD;
02592                                 while ( u1 < t1 ) {
02593                                     u2 = u1 + *u1;
02594                                     ustop = u2 - ABS(u2[-1]);
02595                                     u1++;
02596                                     while ( u1 < ustop ) {
02597                                         if ( *u1 == INDEX ) goto NoPoly;
02598                                         u1 += u1[1];
02599                                     }
02600                                     u1 = u2;
02601                                 }
02602                             }
02603                             if ( *s2 < 0 ) {
02604                                 if ( *s2 != -SNUMBER && *s2 != -SYMBOL && *s2 > -FUNCTION )
02605                                     goto NoPoly;
02606                             }
02607                             else {
02608                                 u1 = s2 + ARGHEAD;
02609                                 while ( u1 < t2 ) {
02610                                     u2 = u1 + *u1;
02611                                     ustop = u2 - ABS(u2[-1]);
02612                                     u1++;
02613                                     while ( u1 < ustop ) {
02614                                         if ( *u1 == INDEX ) goto NoPoly;
02615                                         u1 += u1[1];
02616                                     }
02617                                     u1 = u2;
02618                                 }
02619                             }
02620                         }
02621                         S->PolyWise = WORDDIF(s1,term1);
02622                         S->PolyWise -= FUNHEAD;
02623                         count = 1;
02624                         continue;
02625                     }
02626                     else {
02627 NoPoly:
02628                         S->PolyWise = localPoly = 0;
02629                     }
02630                   }
02631                   else if ( AR.PolyFunType == 2 ) {
02632                     WORD i1, i2, i1a, i2a;
02633                     if ( *s1 > 0 ) i1 = *s1;
02634                     else if ( *s1 <= -FUNCTION ) i1 = 1;
02635                     else i1 = 2;
02636                     if ( *s2 > 0 ) i2 = *s2;
02637                     else if ( *s2 <= -FUNCTION ) i2 = 1;
02638                     else i2 = 2;
02639                     if ( s1[i1] > 0 ) i1a = s1[i1];
02640                     else if ( s1[i1] <= -FUNCTION ) i1a = 1;
02641                     else i1a = 2;
02642                     if ( s2[i2] > 0 ) i2a = s2[i2];
02643                     else if ( s2[i2] <= -FUNCTION ) i2a = 1;
02644                     else i2a = 2;
02645                     if ( s1+i1+i1a == t1 && s2+i2+i2a == t2 ) { /* This is the stuff */
02646 /*
02647                         Test for scalar nature
02648 */
```

```
02649                       if ( !polyhit ) {
02650                           WORD *u1, *u2, *ustop;
02651                           if ( *s1 < 0 ) {
02652                               if ( *s1 != -SNUMBER && *s1 != -SYMBOL && *s1 > -FUNCTION )
02653                                   goto NoPoly;
02654                           }
02655                           else {
02656                               u1 = s1 + ARGHEAD;
02657                               while ( u1 < s1+i1 ) {
02658                                   u2 = u1 + *u1;
02659                                   ustop = u2 - ABS(u2[-1]);
02660                                   u1++;
02661                                   while ( u1 < ustop ) {
02662                                       if ( *u1 == INDEX ) goto NoPoly;
02663                                       u1 += u1[1];
02664                                   }
02665                                   u1 = u2;
02666                               }
02667                           }
02668                           if ( s1[i1] < 0 ) {
02669                               if ( s1[i1] != -SNUMBER && s1[i1] != -SYMBOL && s1[i1] > -FUNCTION )
02670                                   goto NoPoly;
02671                           }
02672                           else {
02673                               u1 = s1 +i1 + ARGHEAD;
02674                               while ( u1 < t1 ) {
02675                                   u2 = u1 + *u1;
02676                                   ustop = u2 - ABS(u2[-1]);
02677                                   u1++;
02678                                   while ( u1 < ustop ) {
02679                                       if ( *u1 == INDEX ) goto NoPoly;
02680                                       u1 += u1[1];
02681                                   }
02682                                   u1 = u2;
02683                               }
02684                           }
02685                           if ( *s2 < 0 ) {
02686                               if ( *s2 != -SNUMBER && *s2 != -SYMBOL && *s2 > -FUNCTION )
02687                                   goto NoPoly;
02688                           }
02689                           else {
02690                               u1 = s2 + ARGHEAD;
02691                               while ( u1 < s2+i2 ) {
02692                                   u2 = u1 + *u1;
02693                                   ustop = u2 - ABS(u2[-1]);
02694                                   u1++;
02695                                   while ( u1 < ustop ) {
02696                                       if ( *u1 == INDEX ) goto NoPoly;
02697                                       u1 += u1[1];
02698                                   }
02699                                   u1 = u2;
02700                               }
02701                           }
02702                           if ( s2[i2] < 0 ) {
02703                               if ( s2[i2] != -SNUMBER && s2[i2] != -SYMBOL && s2[i2] > -FUNCTION )
02704                                   goto NoPoly;
02705                           }
02706                           else {
02707                               u1 = s2 + i2 + ARGHEAD;
02708                               while ( u1 < t2 ) {
02709                                   u2 = u1 + *u1;
02710                                   ustop = u2 - ABS(u2[-1]);
02711                                   u1++;
02712                                   while ( u1 < ustop ) {
02713                                       if ( *u1 == INDEX ) goto NoPoly;
02714                                       u1 += u1[1];
02715                                   }
02716                                   u1 = u2;
02717                               }
02718                           }
02719                       }
02720                       S->PolyWise = WORDDIF(s1,term1);
02721                       S->PolyWise -= FUNHEAD;
02722                       count = 1;
02723                       continue;
02724                   }
02725                   else {
02726                       S->PolyWise = localPoly = 0;
02727                   }
02728               }
02729               else {
02730                   S->PolyWise = localPoly = 0;
02731               }
02732           }
02733           else {
02734               t1 = term1 + S->PolyWise;
02735               t2 = term2 + S->PolyWise;
```

```
02736                       S->PolyWise = 0;
02737                       localPoly = 0;
02738                       continue;
02739                   }
02740               }
02741 #ifdef WITHFLOAT
02742               if ( level == 0 && c1 == FLOATFUN && t1 == stopper1 && t2 == stopper2 && AT.aux_ != 0 ) {
02743 /*
02744                   We have two FLOATFUN's. Test whether they are 'legal'
02745 */
02746                   if ( TestFloat(s1-FUNHEAD) ) {
02747                       if ( TestFloat(s2-FUNHEAD) ) { AT.SortFloatMode = 3; return(0); }
02748                       else { return(1); }
02749                   }
02750                   else if ( TestFloat(s2-FUNHEAD) ) { return(-1); }
02751               }
02752 #endif
02753               while ( s1 < t1 ) {
02754 /*
02755                   The next statement was added 9-nov-2001. It repaired a bad error
02756 */
02757                   if ( s2 >= t2 ) return(PREV(-1));
02758 /*
02759                   There is a little problem here with fast arguments
02760                   We don't want to sacrifice speed, but we like to
02761                   keep a rational ordering. This last one suffers in
02762                   the solution that has been chosen here.
02763 */
02764                   if ( AC.properorderflag ) {
02765                       WORD oldpolyflag;
02766                       oldpolyflag = S->PolyFlag;
02767                       S->PolyFlag = 0;
02768                       if ( ( c2 = -CompArg(s1,s2) ) != 0 ) {
02769                           S->PolyFlag = oldpolyflag; return(PREV(c2));
02770                       }
02771                       S->PolyFlag = oldpolyflag;
02772                       NEXTARG(s1)
02773                       NEXTARG(s2)
02774                   }
02775                   else {
02776                       if ( *s1 > 0 ) {
02777                           if ( *s2 > 0 ) {
02778                               WORD oldpolyflag;
02779                               stopex1 = s1 + *s1;
02780                               if ( s2 >= t2 ) return(PREV(-1));
02781                               stopex2 = s2 + *s2;
02782                               s1 += ARGHEAD; s2 += ARGHEAD;
02783                               oldpolyflag = S->PolyFlag;
02784                               S->PolyFlag = 0;
02785                               while ( s1 < stopex1 ) {
02786                                   if ( s2 >= stopex2 ) {
02787                                       S->PolyFlag = oldpolyflag; return(PREV(-1));
02788                                   }
02789                                   if ( ( c2 = CompareTerms(BHEAD s1,s2,(WORD)1) ) != 0 ) {
02790                                       S->PolyFlag = oldpolyflag; return(PREV(c2));
02791                                   }
02792                                   s1 += *s1;
02793                                   s2 += *s2;
02794                               }
02795                               S->PolyFlag = oldpolyflag;
02796                               if ( s2 < stopex2 ) return(PREV(1));
02797                           }
02798                           else return(PREV(1));
02799                       }
02800                       else {
02801                           if ( *s2 > 0 ) return(PREV(-1));
02802                           if ( *s1 != *s2 ) { return(PREV(*s1-*s2)); }
02803                           if ( *s1 > -FUNCTION ) {
02804                               if ( *++s1 != *++s2 ) { return(PREV(*s2-*s1)); }
02805                           }
02806                           s1++; s2++;
02807                       }
02808                   }
02809               }
02810               if ( s2 < t2 ) return(PREV(1));
02811           }
02812       }
02813 #ifdef WITHFLOAT
02814       if ( level == 0 && t1 < stopper1 && *t1 == FLOATFUN && t1+t1[1] == stopper1
02815               && TestFloat(t1) && AT.aux_ != 0 ) {
02816           AT.SortFloatMode = 1; return(0);
02817       }
02818       else if ( level == 0 && t2 < stopper2 && *t2 == FLOATFUN && t2+t2[1] == stopper2
02819               && TestFloat(t2) && AT.aux_ != 0 ) {
02820           AT.SortFloatMode = 2; return(0);
02821       }
02822 #endif
```

```
02823     {
02824         if ( AR.SortType != SORTLOWFIRST ) {
02825             if ( t1 < stopper1 ) return(PREV(1));
02826             if ( t2 < stopper2 ) return(PREV(-1));
02827         }
02828         else {
02829             if ( t1 < stopper1 ) return(PREV(-1));
02830             if ( t2 < stopper2 ) return(PREV(1));
02831         }
02832     }
02833     if ( level == 3 ) return(CompCoef(term1,term2));
02834     if ( level >= 1 )
02835         return(CompCoef(term2,term1));
02836     return(0);
02837 }
02838
02839 /*
02840         #] Compare1 :
02841         #[ CompareSymbols :         WORD CompareSymbols(term1,term2,par)
02842 */
02856 WORD CompareSymbols(PHEAD WORD *term1, WORD *term2, WORD par)
02857 {
02858     WORD *t1, *t2, *tt1, *tt2;
02859     DUMMYUSE(par);
02860     const int low = AR.SortType == SORTLOWFIRST ? 1 : -1;
02861     const int high = - low;
02862
02863     t1 = term1 + 3; tt1 = term1+*term1; tt1 -= ABS(tt1[-1]);
02864     t2 = term2 + 3; tt2 = term2+*term2; tt2 -= ABS(tt2[-1]);
02865
02866 //  Currently, FORM never sets polysortflag != 0, so disable this code.
02867 //  if ( AN.polysortflag > 0 ) {
02868 //      WORD sum1, sum2;
02869 //      sum1 = 0; sum2 = 0;
02870 //      while ( t1 < tt1 ) { sum1 += t1[1]; t1 += 2; }
02871 //      while ( t2 < tt2 ) { sum2 += t2[1]; t2 += 2; }
02872 //      if ( sum1 < sum2 ) return(low);
02873 //      if ( sum1 > sum2 ) return(high);
02874 //      t1 = term1+3; t2 = term2 + 3;
02875 //  }
02876
02877     while ( t1 < tt1 && t2 < tt2 ) {
02878         const WORD s1 = *t1;
02879         const WORD s2 = *t2;
02880         if ( s1 != s2 ) { return ( s1 > s2 ) ? low : high ; }
02881         const WORD p1 = t1[1];
02882         const WORD p2 = t2[1];
02883         if ( p1 != p2 )  { return ( p1 < p2 ) ? low : high ; }
02884         t1 += 2; t2 += 2;
02885     }
02886
02887     if ( t1 < tt1 ) return(high);
02888     if ( t2 < tt2 ) return(low);
02889
02890     return(0);
02891 }
02892
02893 /*
02894         #] CompareSymbols :
02895         #[ CompareHSymbols :        WORD CompareHSymbols(term1,term2,par)
02896 */
02906 WORD CompareHSymbols(PHEAD WORD *term1, WORD *term2, WORD par)
02907 {
02908     WORD *t1, *t2, *tt1, *tt2, *ttt1, *ttt2;
02909     DUMMYUSE(par);
02910     DUMMYUSE(AT.WorkPointer);
02911     t1 = term1 + 1; tt1 = term1+*term1; tt1 -= ABS(tt1[-1]); t1 += 2;
02912     t2 = term2 + 1; tt2 = term2+*term2; tt2 -= ABS(tt2[-1]); t2 += 2;
02913     while ( t1 < tt1 && t2 < tt2 ) {
02914         if ( *t1 != *t2 ) {
02915             if ( t1[0] < t2[0] ) return(-1);
02916             return(1);
02917         }
02918         else if ( *t1 == HAAKJE ) {
02919             t1 += 3; t2 += 3; continue;
02920         }
02921         ttt1 = t1+t1[1]; ttt2 = t2+t2[1];
02922         while ( t1 < ttt1 && t2 < ttt2 ) {
02923             if ( *t1 > *t2 ) return(-1);
02924             if ( *t1 < *t2 ) return(1);
02925             if ( t1[1] < t2[1] ) return(-1);
02926             if ( t1[1] > t2[1] ) return(1);
02927             t1 += 2; t2 += 2;
02928         }
02929         if ( t1 < ttt1 ) return(1);
02930         if ( t2 < ttt2 ) return(-1);
02931     }
```

```
02932     if ( t1 < tt1 ) return(1);
02933     if ( t2 < tt2 ) return(-1);
02934     return(0);
02935 }
02936
02937 /*
02938         #] CompareHSymbols :
02939         #[ ComPress :               LONG ComPress(ss,n)
02940 */
02959 LONG ComPress(WORD **ss, LONG *n)
02960 {
02961     GETIDENTITY
02962     WORD *t, *s, j, k;
02963     LONG size = 0;
02964     int newsize, i;
02965 /*
02966             #[ debug :
02967
02968     WORD **sss = ss;
02969
02970     if ( AP.DebugFlag ) {
02971         UBYTE OutBuf[140];
02972         MLOCK(ErrorMessageLock);
02973         MesPrint("ComPress:");
02974         AO.OutFill = AO.OutputLine = OutBuf;
02975         AO.OutSkip = 3;
02976         FiniLine();
02977         ss = sss;
02978         while ( *ss ) {
02979             s = *ss++;
02980             j = *s;
02981             if ( j < 0 ) {
02982                 j = s[1] + 2;
02983             }
02984             while ( --j >= 0 ) {
02985                 TalToLine((UWORD)(*s++)); TokenToLine((UBYTE *)"  ");
02986             }
02987             FiniLine();
02988         }
02989         AO.OutSkip = 0;
02990         FiniLine();
02991         MUNLOCK(ErrorMessageLock);
02992         ss = sss;
02993     }
02994
02995             #] debug :
02996 */
02997     *n = 0;
02998     if ( AT.SS == AT.S0 && !AR.NoCompress ) {
02999         if ( AN.compressSize == 0 ) {
03000             if ( *ss ) { AN.compressSize = **ss + 64; }
03001             else       { AN.compressSize = AM.MaxTer/sizeof(WORD) + 2; }
03002             AN.compressSpace = (WORD *)Malloc1(AN.compressSize*sizeof(WORD),"Compression");
03003         }
03004         AN.compressSpace[0] = 0;
03005         while ( *ss ) {
03006             k = 0;
03007             s = *ss;
03008             j = *s++;
03009             if ( j > AN.compressSize ) {
03010                 newsize = j + 64;
03011                 t = (WORD *)Malloc1(newsize*sizeof(WORD),"Compression");
03012                 t[0] = 0;
03013                 if ( AN.compressSpace ) {
03014                     for ( i = 0; i < *AN.compressSpace; i++ ) t[i] = AN.compressSpace[i];
03015                     M_free(AN.compressSpace,"Compression");
03016                 }
03017                 AN.compressSpace = t;
03018                 AN.compressSize = newsize;
03019             }
03020             t = AN.compressSpace;
03021             i = *t - 1;
03022             *t++ = j; j--;
03023             if ( AR.PolyFun ) {
03024                 WORD *polystop, *sa;
03025                 sa = s + j;
03026                 sa -= ABS(sa[-1]);
03027                 polystop = s;
03028                 while ( polystop < sa && *polystop != AR.PolyFun ) {
03029                     polystop += polystop[1];
03030                 }
03031                 while ( i > 0 && j > 0 && *s == *t && s < polystop ) {
03032                     i--; j--; s++; t++; k--;
03033                 }
03034             }
03035             else {
03036                 WORD *sa;
```

```
03037                 sa = s + j;
03038                 sa -= ABS(sa[-1]);
03039                 while ( i > 0 && j > 0 && *s == *t && s < sa ) { i--; j--; s++; t++; k--; }
03040             }
03041             if ( k < -1 ) {
03042                 s[-1] = j;
03043                 s[-2] = k;
03044                 *ss = s-2;
03045                 size += j + 2;
03046             }
03047             else {
03048                 size += *AN.compressSpace;
03049                 if ( k == -1 ) { t--; s--; j++; }
03050             }
03051             while ( --j >= 0 ) *t++ = *s++;
03052 /*                  Sabotage getting into the coefficient next time */
03053             t = AN.compressSpace + *AN.compressSpace;
03054             t[-(ABS(t[-1]))] = 0;
03055             ss++;
03056             (*n)++;
03057         }
03058     }
03059     else {
03060         while ( *ss ) {
03061             size += *(*ss++);
03062             (*n)++;
03063         }
03064     }
03065 /*
03066             #[ debug :
03067
03068     if ( AP.DebugFlag ) {
03069         UBYTE OutBuf[140];
03070         AO.OutFill = AO.OutputLine = OutBuf;
03071         AO.OutSkip = 3;
03072         FiniLine();
03073         ss = sss;
03074         while ( *ss ) {
03075             s = *ss++;
03076             j = *s;
03077             if ( j < 0 ) {
03078                 j = s[1] + 2;
03079             }
03080             while ( --j >= 0 ) {
03081                 TalToLine((UWORD)(*s++)); TokenToLine((UBYTE *)"  ");
03082             }
03083             FiniLine();
03084         }
03085         AO.OutSkip = 0;
03086         FiniLine();
03087     }
03088
03089             #] debug :
03090 */
03091     return(size);
03092 }
03093
03094 /*
03095         #] ComPress :
03096         #[ SplitMerge :             void SplitMerge(Point,number)
03097 */
03123 #ifdef NEWSPLITMERGE
03124
03125 LONG SplitMerge(PHEAD WORD **Pointer, LONG number)
03126 {
03127     GETBIDENTITY
03128     SORTING *S = AT.SS;
03129     WORD **pp3, **pp1, **pp2, **pptop;
03130     LONG i, newleft, newright, split;
03131
03132 #ifdef SPLITMERGEDEBUG
03133     /* Print current array state on entry. */
03134     printf("%4ld: ", number);
03135     for (int ii = 0; ii < S->sTerms; ii++) {
03136         if ( (S->sPointer)[ii] ) {
03137             printf("%4d ", (unsigned)(S->sPointer[ii]-S->sBuffer));
03138         }
03139         else {
03140             printf(".... ");
03141         }
03142     }
03143     printf("\n");
03144     fflush(stdout);
03145 #endif
03146
03147     if ( number < 2 ) return(number);
03148     if ( number == 2 ) {
```

```
03149         pp1 = Pointer; pp2 = pp1 + 1;
03150         if ( ( i = CompareTerms(BHEAD *pp1,*pp2,(WORD)0) ) < 0 ) {
03151             pp3 = (WORD **)(*pp1); *pp1 = *pp2; *pp2 = (WORD *)pp3;
03152         }
03153         else if ( i == 0 ) {
03154             number--;
03155             if ( S->PolyWise ) { if ( AddPoly(BHEAD pp1,pp2) == 0 ) number = 0; }
03156             else               { if ( AddCoef(BHEAD pp1,pp2) == 0 ) number = 0; }
03157         }
03158         return(number);
03159     }
03160     pptop = Pointer + number;
03161     split = number/2;
03162     newleft  = SplitMerge(BHEAD Pointer,split);
03163     newright = SplitMerge(BHEAD Pointer+split,number-split);
03164     if ( newright == 0 ) return(newleft);
03165 /*
03166     We compare the last of the left with the first of the right
03167     If they are already in order, we will be done quickly.
03168     We may have to compactify the buffer because the recursion may
03169     have created holes. Also this compare may result in equal terms.
03170     Addition of 23-jul-1999. It makes things a bit faster.
03171 */
03172     if ( newleft > 0 && newright > 0 &&
03173     ( i = CompareTerms(BHEAD Pointer[newleft-1],Pointer[split],(WORD)0) ) >= 0 ) {
03174         pp2 = Pointer+split; pp1 = Pointer+newleft-1;
03175         if ( i == 0 ) {
03176             if ( S->PolyWise ) {
03177                 if ( AddPoly(BHEAD pp1,pp2) > 0 ) pp1++;
03178                 else newleft--;
03179             }
03180             else {
03181                 if ( AddCoef(BHEAD pp1,pp2) > 0 ) pp1++;
03182                 else newleft--;
03183             }
03184             *pp2++ = 0; newright--;
03185         }
03186         else pp1++;
03187         newleft += newright;
03188         if ( pp1 < pp2 ) {
03189             while ( --newright >= 0 ) *pp1++ = *pp2++;
03190             while ( pp1 < pptop ) *pp1++ = 0;
03191         }
03192         return(newleft);
03193     }
03194
03195     if ( split >= AN.SplitScratchSize ) {
03196         AN.SplitScratchSize = (split*3)/2+100;
03197         if ( AN.SplitScratchSize > S->Terms2InSmall/2 )
03198             AN.SplitScratchSize = S->Terms2InSmall/2;
03199         if ( AN.SplitScratch ) M_free(AN.SplitScratch,"AN.SplitScratch");
03200         AN.SplitScratch = (WORD **)Malloc1(AN.SplitScratchSize*sizeof(WORD *),"AN.SplitScratch");
03201     }
03202
03203     pp3 = AN.SplitScratch; pp1 = Pointer;
03204     /* Move rather than copy, so GarbHand can't double-count. */
03205     for ( i = 0; i < newleft; i++ ) { *pp3++ = *pp1; *pp1++ = 0; }
03206     AN.InScratch = newleft;
03207     pp1 = AN.SplitScratch; pp2 = Pointer + split; pp3 = Pointer;
03208
03209 #ifdef NEWSPLITMERGETIMSORT
03210 /*
03211         An improvement in the style of Timsort
03212 */
03213     while ( newleft > 8 ) {
03214         /* Check the middle of the LHS terms */
03215         LONG nnleft = newleft/2;
03216         if ( ( i = CompareTerms(BHEAD pp1[nnleft],*pp2,(WORD)0) ) < 0 ) {
03217             /* The terms are not in order. Break out and continue as normal. */
03218             break;
03219         }
03220         /* The terms merge or are in order. Copy pointers up to this point. */
03221         /* In the copy, zero the skipped pointers so GarbHand can't double-count. */
03222         for (int iii = 0; iii < nnleft; iii++) {
03223             *pp3++ = *pp1;
03224             *pp1++ = 0;
03225         }
03226         newleft -= nnleft;
03227         if ( i == 0 ) {
03228             if ( S->PolyWise ) { i = AddPoly(BHEAD pp1,pp2); }
03229             else               { i = AddCoef(BHEAD pp1,pp2); }
03230             if ( i == 0 ) {
03231                 /* The terms cancelled. The next term goes in *pp3. Don't move. */
03232             }
03233             else {
03234                 /* The terms added. Advance pp3. */
03235                 *pp3++ = *pp1;
```

```
03236             }
03237             /* We have taken a LHS (copy) and RHS term. */
03238             *pp2++ = 0;
03239             newright--;
03240             *pp1++ = 0;
03241             newleft--;
03242             break;
03243         }
03244     }
03245 #endif
03246
03247     while ( newleft > 0 && newright > 0 ) {
03248         if ( ( i = CompareTerms(BHEAD *pp1,*pp2,(WORD)0) ) < 0 ) {
03249             *pp3++ = *pp2;
03250             *pp2++ = 0;
03251             newright--;
03252         }
03253         else if ( i > 0 ) {
03254             *pp3++ = *pp1;
03255             *pp1++ = 0;
03256             newleft--;
03257         }
03258         else {
03259             if ( S->PolyWise ) { if ( AddPoly(BHEAD pp1,pp2) > 0 ) *pp3++ = *pp1; }
03260             else {               if ( AddCoef(BHEAD pp1,pp2) > 0 ) *pp3++ = *pp1; }
03261             *pp1++ = 0; *pp2++ = 0; newleft--; newright--;
03262         }
03263     }
03264     for ( i = 0; i < newleft; i++ ) { *pp3++ = *pp1; *pp1++ = 0; }
03265     if ( pp3 == pp2 ) {
03266         pp3 += newright;
03267     } else {
03268         for ( i = 0; i < newright; i++ ) { *pp3++ = *pp2++; }
03269     }
03270     newleft = pp3 - Pointer;
03271     while ( pp3 < pptop ) *pp3++ = 0;
03272     AN.InScratch = 0;
03273     return(newleft);
03274 }
03275
03276 #else
03277
03278 LONG SplitMerge(PHEAD WORD **Pointer, LONG number)
03279 {
03280     GETBIDENTITY
03281     SORTING *S = AT.SS;
03282     WORD **pp3, **pp1, **pp2;
03283     LONG nleft, nright, i, newleft, newright;
03284     WORD **pptop;
03285
03286 #ifdef SPLITMERGEDEBUG
03287     /* Print current array state on entry. */
03288     printf("%4ld: ", number);
03289     for (int ii = 0; ii < S->sTerms; ii++) {
03290         if ( (S->sPointer)[ii] ) {
03291             printf("%4d ", (unsigned)(S->sPointer[ii]-S->sBuffer));
03292         }
03293         else {
03294             printf(".... ");
03295         }
03296     }
03297     printf("\n");
03298     fflush(stdout);
03299 #endif
03300
03301     if ( number < 2 ) return(number);
03302     if ( number == 2 ) {
03303         pp1 = Pointer; pp2 = pp1 + 1;
03304         if ( ( i = CompareTerms(BHEAD *pp1,*pp2,(WORD)0) ) < 0 ) {
03305             pp3 = (WORD **)(*pp1); *pp1 = *pp2; *pp2 = (WORD *)pp3;
03306         }
03307         else if ( i == 0 ) {
03308           number--;
03309           if ( S->PolyWise ) { if ( AddPoly(BHEAD pp1,pp2) == 0 ) { number = 0; } }
03310           else {               if ( AddCoef(BHEAD pp1,pp2) == 0 ) { number = 0; } }
03311         }
03312         return(number);
03313     }
03314     pptop = Pointer + number;
03315     nleft = number » 1; nright = number - nleft;
03316     newleft  = SplitMerge(BHEAD Pointer,nleft);
03317     newright = SplitMerge(BHEAD Pointer+nleft,nright);
03318 /*
03319     We compare the last of the left with the first of the right
03320     If they are already in order, we will be done quickly.
03321     We may have to compactify the buffer because the recursion may
03322     have created holes. Also this compare may result in equal terms.
```

```
03323     Addition of 23-jul-1999. It makes things a bit faster.
03324 */
03325     if ( newleft > 0 && newright > 0 &&
03326     ( i = CompareTerms(BHEAD Pointer[newleft-1],Pointer[nleft],(WORD)0) ) >= 0 ) {
03327         pp2 = Pointer+nleft; pp1 = Pointer+newleft-1;
03328         if ( i == 0 ) {
03329           if ( S->PolyWise ) {
03330             if ( AddPoly(BHEAD pp1,pp2) > 0 ) pp1++;
03331             else newleft--;
03332           }
03333           else {
03334             if ( AddCoef(BHEAD pp1,pp2) > 0 ) pp1++;
03335             else newleft--;
03336           }
03337           *pp2++ = 0; newright--;
03338         }
03339         else pp1++;
03340         newleft += newright;
03341         if ( pp1 < pp2 ) {
03342             while ( --newright >= 0 ) *pp1++ = *pp2++;
03343             while ( pp1 < pptop ) *pp1++ = 0;
03344         }
03345         return(newleft);
03346     }
03347     if ( nleft > AN.SplitScratchSize ) {
03348         AN.SplitScratchSize = (nleft*3)/2+100;
03349         if ( AN.SplitScratchSize > S->Terms2InSmall/2 )
03350             AN.SplitScratchSize = S->Terms2InSmall/2;
03351         if ( AN.SplitScratch ) M_free(AN.SplitScratch,"AN.SplitScratch");
03352         AN.SplitScratch = (WORD **)Malloc1(AN.SplitScratchSize*sizeof(WORD *),"AN.SplitScratch");
03353     }
03354     pp3 = AN.SplitScratch; pp1 = Pointer; i = nleft;
03355     do { *pp3++ = *pp1; *pp1++ = 0; } while ( *pp1 && --i > 0 );
03356     if ( i > 0 ) { *pp3 = 0; i--; }
03357     AN.InScratch = nleft - i;
03358     pp1 = AN.SplitScratch; pp2 = Pointer + nleft; pp3 = Pointer;
03359     while ( nleft > 0 && nright > 0 && *pp1 && *pp2 ) {
03360         if ( ( i = CompareTerms(BHEAD *pp1,*pp2,(WORD)0) ) < 0 ) {
03361             *pp3++ = *pp2;
03362             *pp2++ = 0;
03363             nright--;
03364         }
03365         else if ( i > 0 ) {
03366             *pp3++ = *pp1;
03367             *pp1++ = 0;
03368             nleft--;
03369         }
03370         else {
03371           if ( S->PolyWise ) { if ( AddPoly(BHEAD pp1,pp2) > 0 ) *pp3++ = *pp1; }
03372           else {               if ( AddCoef(BHEAD pp1,pp2) > 0 ) *pp3++ = *pp1; }
03373           *pp1++ = 0; *pp2++ = 0; nleft--; nright--;
03374         }
03375     }
03376     while ( --nleft  >= 0 && *pp1 ) { *pp3++ = *pp1; *pp1++ = 0; }
03377     while ( --nright >= 0 && *pp2 ) { *pp3++ = *pp2++; }
03378     nleft = pp3 - Pointer;
03379     while ( pp3 < pptop ) *pp3++ = 0;
03380     AN.InScratch = 0;
03381     return(nleft);
03382 }
03383
03384 #endif
03385
03386 /*
03387         #] SplitMerge :
03388         #[ GarbHand :               void GarbHand()
03389 */
03405 void GarbHand(void)
03406 {
03407     GETIDENTITY
03408     SORTING *S = AT.SS;
03409     WORD **Point, *s2, *t, *garbuf, i;
03410     LONG k, total = 0;
03411     int tobereturned = 0;
03412 /*
03413     Compute the size needed. Put it in total.
03414 */
03415 #ifdef TESTGARB
03416     MLOCK(ErrorMessageLock);
03417     MesPrint("in:  S->sFill = %l, S->sTop2 = %l",S->sFill-S->sBuffer,S->sTop2-S->sBuffer);
03418 #endif
03419     Point = S->sPointer;
03420     k = S->sTerms;
03421     while ( --k >= 0 ) {
03422         if ( ( s2 = *Point++ ) != 0 ) { total += *s2; }
03423     }
03424     Point = AN.SplitScratch;
```

```
03425     k = AN.InScratch;
03426     while ( --k >= 0 ) {
03427         if ( ( s2 = *Point++ ) != 0 ) { total += *s2; }
03428     }
03429 #ifdef TESTGARB
03430     MesPrint("total = %l, nterms = %l",total,AN.InScratch);
03431     MUNLOCK(ErrorMessageLock);
03432 #endif
03433 /*
03434     Test now whether it fits. If so deal with the problem inside
03435     the memory at the tail of the large buffer.
03436 */
03437     if ( S->lBuffer != 0 && S->lFill + total <= S->lTop ) {
03438         garbuf = S->lFill;
03439     }
03440     else {
03441         garbuf = (WORD *)Malloc1(total*sizeof(WORD),"Garbage buffer");
03442         tobereturned = 1;
03443     }
03444     t = garbuf;
03445     Point = S->sPointer;
03446     k = S->sTerms;
03447     while ( --k >= 0 ) {
03448         if ( *Point ) {
03449             s2 = *Point++;
03450             i = *s2;
03451             NCOPY(t,s2,i);
03452         }
03453         else { Point++; }
03454     }
03455     Point = AN.SplitScratch;
03456     k = AN.InScratch;
03457     while ( --k >= 0 ) {
03458         if ( *Point ) {
03459             s2 = *Point++;
03460             i = *s2;
03461             NCOPY(t,s2,i);
03462         }
03463         else Point++;
03464     }
03465     s2 = S->sBuffer;
03466     t = garbuf;
03467     Point = S->sPointer;
03468     k = S->sTerms;
03469     while ( --k >= 0 ) {
03470         if ( *Point ) {
03471             *Point++ = s2;
03472             i = *t;
03473             NCOPY(s2,t,i);
03474         }
03475         else { Point++; }
03476     }
03477     Point = AN.SplitScratch;
03478     k = AN.InScratch;
03479     while ( --k >= 0 ) {
03480         if ( *Point ) {
03481             *Point++ = s2;
03482             i = *t;
03483             NCOPY(s2,t,i);
03484         }
03485         else Point++;
03486     }
03487     S->sFill = s2;
03488 #ifdef TESTGARB
03489     MLOCK(ErrorMessageLock);
03490     MesPrint("out: S->sFill = %l, S->sTop2 = %l",S->sFill-S->sBuffer,S->sTop2-S->sBuffer);
03491     if ( S->sFill >= S->sTop2 ) {
03492         MesPrint("We are in deep trouble");
03493     }
03494     MUNLOCK(ErrorMessageLock);
03495 #endif
03496     if ( tobereturned ) M_free(garbuf,"Garbage buffer");
03497     return;
03498 }
03499
03500 /*
03501         #] GarbHand :
03502         #[ MergePatches :           WORD MergePatches(par)
03503 */
03520 int MergePatches(WORD par)
03521 {
03522     GETIDENTITY
03523     SORTING *S = AT.SS;
03524     WORD **poin, **poin2, ul, k, i, im, *m1;
03525     WORD *p, lpat, mpat, level, l1, l2, r1, r2, r3, c;
03526     WORD *m2, *m3, r31, r33, ki, *rr;
03527     UWORD *coef;
```

```
03528     POSITION position;
03529     FILEHANDLE *fin, *fout;
03530     int fhandle;
03531 /*
03532     UBYTE *s;
03533 */
03534 #ifdef WITHZLIB
03535     POSITION position2;
03536     int oldgzipCompress = AR.gzipCompress;
03537     if ( par == 2 ) {
03538         AR.gzipCompress = 0;
03539     }
03540 #endif
03541     fin = &S->file;
03542     fout = &(AR.FoStage4[0]);
03543 NewMerge:
03544     coef = AN.SoScratC;
03545     poin = S->poina; poin2 = S->poin2a;
03546     rr = AR.CompressPointer;
03547     *rr = 0;
03548 /*
03549         #[ Setup :
03550 */
03551     if ( par == 1 ) {
03552         fout = &(S->file);
03553         if ( fout->handle < 0 ) {
03554 FileMake:
03555             PUTZERO(S->OldPosOut);
03556             if ( ( fhandle = CreateFile(fout->name) ) < 0 ) {
03557                 MLOCK(ErrorMessageLock);
03558                 MesPrint("Cannot create file %s",fout->name);
03559                 MUNLOCK(ErrorMessageLock);
03560                 goto ReturnError;
03561             }
03562 #ifdef GZIPDEBUG
03563             MLOCK(ErrorMessageLock);
03564             MesPrint("%w MergePatches created output file %s",fout->name);
03565             MUNLOCK(ErrorMessageLock);
03566 #endif
03567             fout->handle = fhandle;
03568             PUTZERO(fout->filesize);
03569             PUTZERO(fout->POposition);
03570 /*
03571             Should not be here?
03572 #ifdef WITHZLIB
03573             fout->ziobuffer = 0;
03574 #endif
03575 */
03576 #ifdef ALLLOCK
03577             LOCK(fout->pthreadslock);
03578 #endif
03579             SeekFile(fout->handle,&(fout->filesize),SEEK_SET);
03580 #ifdef ALLLOCK
03581             UNLOCK(fout->pthreadslock);
03582 #endif
03583             S->fPatchN = 0;
03584             PUTZERO(S->fPatches[0]);
03585             fout->POfill = fout->POobuffer;
03586             PUTZERO(fout->POposition);
03587         }
03588 ConMer:
03589         StageSort(fout);
03590 #ifdef WITHZLIB
03591         if ( S == AT.S0 && AR.NoCompress == 0 && AR.gzipCompress > 0 )
03592             S->fpcompressed[S->fPatchN] = 1;
03593         else
03594             S->fpcompressed[S->fPatchN] = 0;
03595         SetupOutputGZIP(fout);
03596 #endif
03597     }
03598     else if ( par == 0 && S->stage4 > 0 ) {
03599 /*
03600         We will have to do our job more than once.
03601         Input is from S->file and output will go to AR.FoStage4.
03602         The file corresponding to this last one must be made now.
03603 */
03604         AR.Stage4Name ^= 1;
03605 /*
03606         s = (UBYTE *)(fout->name); while ( *s ) s++;
03607         if ( AR.Stage4Name ) s[-1] += 1;
03608         else                 s[-1] -= 1;
03609 */
03610         S->iPatches = S->fPatches;
03611         S->fPatches = S->inPatches;
03612         S->inPatches = S->iPatches;
03613         (S->inNum) = S->fPatchN;
03614         S->OldPosIn = S->OldPosOut;
```

```
03615 #ifdef WITHZLIB
03616         m1 = S->fpincompressed;
03617         S->fpincompressed = S->fpcompressed;
03618         S->fpcompressed = m1;
03619         for ( i = 0; i < S->inNum; i++ ) {
03620             S->fPatchesStop[i] = S->iPatches[i+1];
03621 #ifdef GZIPDEBUG
03622             MLOCK(ErrorMessageLock);
03623             MesPrint("%w fPatchesStop[%d] = %10p",i,&(S->fPatchesStop[i]));
03624             MUNLOCK(ErrorMessageLock);
03625 #endif
03626         }
03627 #endif
03628         S->stage4 = 0;
03629         goto FileMake;
03630     }
03631     else {
03632 #ifdef WITHZLIB
03633 /*
03634         The next statement is just for now
03635 */
03636         AR.gzipCompress = 0;
03637 #endif
03638         if ( par == 0 ) {
03639             S->iPatches = S->fPatches;
03640             S->inNum = S->fPatchN;
03641 #ifdef WITHZLIB
03642             m1 = S->fpincompressed;
03643             S->fpincompressed = S->fpcompressed;
03644             S->fpcompressed = m1;
03645             for ( i = 0; i < S->inNum; i++ ) {
03646                 S->fPatchesStop[i] = S->fPatches[i+1];
03647 #ifdef GZIPDEBUG
03648                 MLOCK(ErrorMessageLock);
03649                 MesPrint("%w fPatchesStop[%d] = %10p",i,&(S->fPatchesStop[i]));
03650                 MUNLOCK(ErrorMessageLock);
03651 #endif
03652             }
03653 #endif
03654         }
03655         fout = AR.outfile;
03656     }
03657     if ( par ) {                /* Mark end of patches */
03658         S->Patches[S->lPatch] = S->lFill;
03659         for ( i = 0; i < S->lPatch; i++ ) {
03660             S->pStop[i] = S->Patches[i+1]-1;
03661             S->Patches[i] = (WORD *)(((UBYTE *)(S->Patches[i])) + AM.MaxTer);
03662         }
03663     }
03664     else {  /* Load the patches */
03665         S->lPatch = (S->inNum);
03666 #ifdef WITHMPI
03667         if ( S->lPatch > 1 || ( (PF.exprtodo <0) && (fout == AR.outfile || fout == AR.hidefile ) ) ) {
03668 #else
03669         if ( S->lPatch > 1 ) {
03670 #endif
03671 #ifdef WITHZLIB
03672             SetupAllInputGZIP(S);
03673 #endif
03674             p = S->lBuffer;
03675             for ( i = 0; i < S->lPatch; i++ ) {
03676                 p = (WORD *)(((UBYTE *)p)+2*AM.MaxTer+COMPINC*sizeof(WORD));
03677                 S->Patches[i] = p;
03678                 p = (WORD *)(((UBYTE *)p) + fin->POsize);
03679                 S->pStop[i] = m2 = p;
03680 #ifdef WITHZLIB
03681                 PutIn(fin,&(S->iPatches[i]),S->Patches[i],&m2,i);
03682 #else
03683                 ADDPOS(S->iPatches[i],PutIn(fin,&(S->iPatches[i]),S->Patches[i],&m2,i));
03684 #endif
03685             }
03686         }
03687     }
03688     if ( fout->handle >= 0 ) {
03689         PUTZERO(position);
03690 #ifdef ALLLOCK
03691         LOCK(fout->pthreadslock);
03692 #endif
03693         SeekFile(fout->handle,&position,SEEK_END);
03694         ADDPOS(position,((fout->POfill-fout->PObuffer)*sizeof(WORD)));
03695 #ifdef ALLLOCK
03696         UNLOCK(fout->pthreadslock);
03697 #endif
03698     }
03699     else {
03700         SETBASEPOSITION(position,(fout->POfill-fout->PObuffer)*sizeof(WORD));
03701     }
```

```
03702 /*
03703         #] Setup :
03704
03705     The old code had to be replaced because all output needs to go
03706     through PutOut. For this we have to go term by term and keep
03707     track of the compression.
03708 */
03709     if ( S->lPatch == 1 ) { /* Single patch --> direct copy. Very rare. */
03710         LONG length;
03711
03712         if ( fout->handle < 0 ) if ( Sflush(fout) ) goto PatCall;
03713         if ( par ) {        /* Memory to file */
03714 #ifdef WITHZLIB
03715 /*
03716             We fix here the problem that the thing needs to go through PutOut
03717 */
03718             m2 = m1 = *S->Patches; /* The m2 is to keep the compiler from complaining */
03719             while ( *m1 ) {
03720                 if ( *m1 < 0 ) { /* Need to uncompress */
03721                     i = -(*m1++); m2 += i; im = *m1+i+1;
03722                     while ( i > 0 ) { *m1-- = *m2--; i--; }
03723                     *m1 = im;
03724                 }
03725 #ifdef WITHPTHREADS
03726                 /* Check here (and in the following) that we are at ground level
03727                    (so S == AT.S0) to use PutToMaster. Control can reach here,
03728                    with AS.MasterSort, but with S != AT.S0, when the SortBots are
03729                    adding PolyRatFun and the sorting of their arguments requires
03730                    large buffer patches to be merged. In this case, terms escape
03731                    the PolyRatFun argument and end up at ground level, if we use
03732                    PutToMaster. */
03733                 if ( AS.MasterSort && ( fout == AR.outfile ) && S == AT.S0 ) {
03734                     im = PutToMaster(BHEAD m1);
03735                 }
03736                 else
03737 #endif
03738                 if ( ( im = PutOut(BHEAD m1,&position,fout,1) ) < 0 ) goto ReturnError;
03739                 ADDPOS(S->SizeInFile[par],im);
03740                 m2 = m1;
03741                 m1 += *m1;
03742             }
03743 #ifdef WITHPTHREADS
03744             if ( AS.MasterSort && ( fout == AR.outfile ) && S == AT.S0 ) {
03745                 PutToMaster(BHEAD 0);
03746             }
03747             else
03748 #endif
03749             if ( FlushOut(&position,fout,1) ) goto ReturnError;
03750             ADDPOS(S->SizeInFile[par],1);
03751 #else
03752 /* old code */
03753             length = (LONG)(*S->pStop)-(LONG)(*S->Patches)+sizeof(WORD);
03754             if ( WriteFile(fout->handle,(UBYTE *)(*S->Patches),length) != length )
03755                 goto PatwCall;
03756             ADDPOS(position,length);
03757             ADDPOS(fout->POposition,length);
03758             ADDPOS(fout->filesize,length);
03759             ADDPOS(S->SizeInFile[par],length/sizeof(WORD));
03760 #endif
03761         }
03762         else {              /* File to file */
03763 #ifdef WITHZLIB
03764 /*
03765             Note: if we change FRONTSIZE we need to make the minimum value
03766             of SmallEsize in AllocSort correspondingly larger or smaller.
03767             Theoretically we could get close to 2*AM.MaxTer!
03768 */
03769             #define FRONTSIZE (2*AM.MaxTer)
03770             WORD *copybuf = (WORD *)(((UBYTE *)(S->sBuffer)) + FRONTSIZE);
03771             WORD *copytop;
03772             SetupAllInputGZIP(S);
03773             m1 = m2 = copybuf;
03774             position2 = S->iPatches[0];
03775             while ( ( length = FillInputGZIP(fin,&position2,
03776                     (UBYTE *)copybuf,
03777                     (S->SmallEsize*sizeof(WORD)-FRONTSIZE),0) ) > 0 ) {
03778                 copytop = (WORD *)(((UBYTE *)copybuf)+length);
03779                 while ( *m1 && ( ( *m1 > 0 && m1+*m1 < copytop ) ||
03780                 ( *m1 < 0 && ( m1+1 < copytop ) && ( m1+m1[1]+1 < copytop ) ) ) )
03781 /*
03782     22-jun-2013 JV  Extremely nasty bug that has been around for a while.
03783                     What if the end is in the remaining part? We will loose terms!
03784                 while ( *m1 && ( (WORD *)(((UBYTE *)(m1)) + AM.MaxTer ) < S->sTop2 ) )
03785 */
03786                 {
03787                     if ( *m1 < 0 ) { /* Need to uncompress */
03788                         i = -(*m1++); m2 += i; im = *m1+i+1;
```

```
03789                       while ( i > 0 ) { *m1-- = *m2--; i--; }
03790                       *m1 = im;
03791                   }
03792 #ifdef WITHPTHREADS
03793                   if ( AS.MasterSort && ( fout == AR.outfile ) && S == AT.S0 ) {
03794                       im = PutToMaster(BHEAD m1);
03795                   }
03796                   else
03797 #endif
03798                   if ( ( im = PutOut(BHEAD m1,&position,fout,1) ) < 0 ) goto ReturnError;
03799                   ADDPOS(S->SizeInFile[par],im);
03800                   m2 = m1;
03801                   m1 += *m1;
03802               }
03803               if ( m1 < copytop && *m1 == 0 ) break;
03804 /*
03805               Now move the remaining part 'back'
03806 */
03807               m3 = copybuf;
03808               m1 = copytop;
03809               while ( m1 > m2 ) *--m3 = *--m1;
03810               m2 = m3;
03811               m1 = m2 + *m2;
03812           }
03813           if ( length < 0 ) {
03814 /* INTERNAL_ERROR_EXCL_START */
03815               MLOCK(ErrorMessageLock);
03816               MesPrint("!>Readerror");
03817               goto PatCall2;
03818 /* INTERNAL_ERROR_EXCL_STOP */
03819           }
03820 #ifdef WITHPTHREADS
03821           if ( AS.MasterSort && ( fout == AR.outfile ) && S == AT.S0 ) {
03822               PutToMaster(BHEAD 0);
03823           }
03824           else
03825 #endif
03826           if ( FlushOut(&position,fout,1) ) goto ReturnError;
03827           ADDPOS(S->SizeInFile[par],1);
03828 #else
03829 /* old code */
03830           SeekFile(fin->handle,&(S->iPatches[0]),SEEK_SET); /* needed for stage4 */
03831           while ( ( length = ReadFile(fin->handle,
03832                   (UBYTE *)(S->sBuffer),S->SmallEsize*sizeof(WORD)) ) > 0 ) {
03833               if ( WriteFile(fout->handle,(UBYTE *)(S->sBuffer),length) != length )
03834                   goto PatwCall;
03835               ADDPOS(position,length);
03836               ADDPOS(fout->POposition,length);
03837               ADDPOS(fout->filesize,length);
03838               ADDPOS(S->SizeInFile[par],length/sizeof(WORD));
03839           }
03840           if ( length < 0 ) {
03841 /* INTERNAL_ERROR_EXCL_START */
03842               MLOCK(ErrorMessageLock);
03843               MesPrint("!>Readerror");
03844               goto PatCall2;
03845 /* INTERNAL_ERROR_EXCL_STOP */
03846           }
03847 #endif
03848       }
03849       goto EndOfAll;
03850   }
03851   else if ( S->lPatch > 0 ) {
03852
03853       /* More than one patch. Construct the tree. */
03854
03855       lpat = 1;
03856       do { lpat *= 2; } while ( lpat < S->lPatch );
03857       mpat = ( lpat >> 1 ) - 1;
03858       k = lpat - S->lPatch;
03859
03860       /* k is the number of empty places in the tree. they will
03861          be at the even positions from 2 to 2*k */
03862
03863       for ( i = 1; i < lpat; i++ ) {
03864           S->tree[i] = -1;
03865       }
03866       for ( i = 1; i <= k; i++ ) {
03867           im = ( i * 2 ) - 1;
03868           poin[im] = S->Patches[i-1];
03869           poin2[im] = poin[im] + *(poin[im]);
03870           S->used[i] = im;
03871           S->ktoi[im] = i-1;
03872           S->tree[mpat+i] = 0;
03873           poin[im-1] = poin2[im-1] = 0;
03874       }
03875       for ( i = (k*2)+1; i <= lpat; i++ ) {
```

```
03876             S->used[i-k] = i;
03877             S->ktoi[i] = i-k-1;
03878             poin[i] = S->Patches[i-k-1];
03879             poin2[i] = poin[i] + *(poin[i]);
03880         }
03881 /*
03882         the array poin tells the position of the i-th element of the S->tree
03883         'S->used' is a stack with the S->tree elements that need to be entered
03884         into the S->tree. at the beginning this is S->lPatch. during the
03885         sort there will be only very few elements.
03886         poin2 is the next value of poin. it has to be determined
03887         before the comparisons as the position or the size of the
03888         term indicated by poin may change.
03889         S->ktoi translates a S->tree element back to its stream number.
03890
03891         start the sort
03892 */
03893         level = S->lPatch;
03894
03895         /* introduce one term */
03896 OneTerm:
03897         k = S->used[level];
03898         i = k + lpat - 1;
03899         if ( !*(poin[k]) ) {
03900             do { if ( !( i >= 1 ) ) goto EndOfMerge; } while ( !S->tree[i] );
03901             if ( S->tree[i] == -1 ) {
03902                 S->tree[i] = 0;
03903                 level--;
03904                 goto OneTerm;
03905             }
03906             k = S->tree[i];
03907             S->used[level] = k;
03908             S->tree[i] = 0;
03909         }
03910 /*
03911         move terms down the tree
03912 */
03913         while ( i >= 1 ) {
03914             if ( S->tree[i] > 0 ) {
03915                 if ( ( c = CompareTerms(BHEAD poin[S->tree[i]],poin[k],(WORD)0) ) > 0 ) {
03916 /*
03917                     S->tree[i] is the smaller. Exchange and go on.
03918 */
03919                     S->used[level] = S->tree[i];
03920                     S->tree[i] = k;
03921                     k = S->used[level];
03922                 }
03923                 else if ( !c ) {    /* Terms are equal */
03924                     S->TermsLeft--;
03925 /*
03926                         Here the terms are equal and their coefficients
03927                         have to be added.
03928 */
03929                     l1 = *( m1 = poin[S->tree[i]] );
03930                     l2 = *( m2 = poin[k] );
03931                     if ( S->PolyWise ) {  /* Here we work with PolyFun */
03932                         WORD *tt1, *w;
03933                         tt1 = m1;
03934                         m1 += S->PolyWise;
03935                         m2 += S->PolyWise;
03936                         if ( S->PolyFlag == 2 ) {
03937                             w = poly_ratfun_add(BHEAD m1,m2);
03938                             if ( *tt1 + w[1] - m1[1] > AM.MaxTer/((LONG)sizeof(WORD)) ) {
03939                                 MLOCK(ErrorMessageLock);
03940                                 MesPrint("Term too complex in PolyRatFun addition. MaxTermSize of %10l
      is too small",AM.MaxTer);
03941                                 MUNLOCK(ErrorMessageLock);
03942                                 Terminate(-1);
03943                             }
03944                             AT.WorkPointer = w;
03945                         }
03946                         else {
03947                             w = AT.WorkPointer;
03948                             if ( w + m1[1] + m2[1] > AT.WorkTop ) {
03949                                 MLOCK(ErrorMessageLock);
03950                                 MesPrint("A WorkSpace of %10l is too small",AM.WorkSize);
03951                                 MUNLOCK(ErrorMessageLock);
03952                                 Terminate(-1);
03953                             }
03954                             AddArgs(BHEAD m1,m2,w);
03955                         }
03956                         r1 = w[1];
03957                         if ( r1 <= FUNHEAD
03958                             || ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 ) )
03959                                 { goto cancelled; }
03960                         if ( r1 == m1[1] ) {
03961                             NCOPY(m1,w,r1);
```

```
03962                           }
03963                           else if ( r1 < m1[1] ) {
03964                               r2 = m1[1] - r1;
03965                               m2 = w + r1;
03966                               m1 += m1[1];
03967                               while ( --r1 >= 0 ) *--m1 = *--m2;
03968                               m2 = m1 - r2;
03969                               r1 = S->PolyWise;
03970                               while ( --r1 >= 0 ) *--m1 = *--m2;
03971                               *m1 -= r2;
03972                               poin[S->tree[i]] = m1;
03973                           }
03974                           else {
03975                               r2 = r1 - m1[1];
03976                               m2 = tt1 - r2;
03977                               r1 = S->PolyWise;
03978                               m1 = tt1;
03979                               *m1 += r2;
03980                               poin[S->tree[i]] = m2;
03981                               NCOPY(m2,m1,r1);
03982                               r1 = w[1];
03983                               NCOPY(m2,w,r1);
03984                           }
03985                       }
03986 #ifdef WITHFLOAT
03987                       else if ( AT.SortFloatMode ) {
03988                           WORD *term1, *term2;
03989                           term1 = poin[S->tree[i]];
03990                           term2 = poin[k];
03991                           if ( MergeWithFloat(BHEAD &term1,&term2) == 0 )
03992                               goto cancelled;
03993                           poin[S->tree[i]] = term1;
03994                       }
03995 #endif
03996                       else {
03997                         r1 = *( m1 += l1 - 1 );
03998                         m1 -= ABS(r1) - 1;
03999                         r1 = ( ( r1 > 0 ) ? (r1-1) : (r1+1) ) >> 1;
04000                         r2 = *( m2 += l2 - 1 );
04001                         m2 -= ABS(r2) - 1;
04002                         r2 = ( ( r2 > 0 ) ? (r2-1) : (r2+1) ) >> 1;
04003
04004                         if ( AddRat(BHEAD (UWORD *)m1,r1,(UWORD *)m2,r2,coef,&r3) ) {
04005                           MLOCK(ErrorMessageLock);
04006                           MesCall("MergePatches");
04007                           MUNLOCK(ErrorMessageLock);
04008                           SETERROR(-1)
04009                         }
04010
04011                         if ( AN.ncmod != 0 ) {
04012                           if ( ( AC.modmode & POSNEG ) != 0 ) {
04013                               NormalModulus(coef,&r3);
04014                           }
04015                           else if ( BigLong(coef,r3,(UWORD *)AC.cmod,ABS(AN.ncmod)) >= 0 ) {
04016                               WORD ii;
04017                               SubPLon(coef,r3,(UWORD *)AC.cmod,ABS(AN.ncmod),coef,&r3);
04018                               coef[r3] = 1;
04019                               for ( ii = 1; ii < r3; ii++ ) coef[r3+ii] = 0;
04020                           }
04021                         }
04022                         r3 *= 2;
04023                         r33 = ( r3 > 0 ) ? ( r3 + 1 ) : ( r3 - 1 );
04024                         if ( r3 < 0 ) r3 = -r3;
04025                         if ( r1 < 0 ) r1 = -r1;
04026                         r1 *= 2;
04027                         r31 = r3 - r1;
04028                         if ( !r3 ) {      /* Terms cancel */
04029 cancelled:
04030                           ul = S->used[level] = S->tree[i];
04031                           S->tree[i] = -1;
04032 /*
04033                           We skip to the next term in stream ul
04034 */
04035                           im = *poin2[ul];
04036                           if ( im < 0 ) {
04037                               r1 = poin2[ul][1] - im + 1;
04038                               m1 = poin2[ul] + 2;
04039                               m2 = poin[ul] - im + 1;
04040                               while ( ++im <= 0 ) *--m1 = *--m2;
04041                               *--m1 = r1;
04042                               poin2[ul] = m1;
04043                               im = r1;
04044                           }
04045                           poin[ul] = poin2[ul];
04046                           ki = S->ktoi[ul];
04047                           if ( !par && (poin[ul] + im + COMPINC) >= S->pStop[ki]
04048                           && im > 0 ) {
```

```
04049 #ifdef WITHZLIB
04050                           PutIn(fin,&(S->iPatches[ki]),S->Patches[ki],&(poin[ul]),ki);
04051 #else
04052                           ADDPOS(S->iPatches[ki],PutIn(fin,&(S->iPatches[ki]),
04053                           S->Patches[ki],&(poin[ul]),ki));
04054 #endif
04055                           poin2[ul] = poin[ul] + im;
04056                         }
04057                         else {
04058                           poin2[ul] += im;
04059                         }
04060                         S->used[++level] = k;
04061                         S->TermsLeft--;
04062                       }
04063                       else if ( !r31 ) {        /* copy coef into term1 */
04064                         goto CopCof2;
04065                       }
04066                       else if ( r31 < 0 ) {     /* copy coef into term1
04067                                                    and adjust the length of term1 */
04068                         goto CopCoef;
04069                       }
04070                       else {
04071 /*
04072                           this is the dreaded calamity.
04073                           is there enough space?
04074 */
04075                         if( (poin[S->tree[i]]+l1+r31) >= poin2[S->tree[i]] ) {
04076 /*
04077                             no space! now the special trick for which
04078                             we left 2*maxlng spaces open at the beginning
04079                             of each patch.
04080 */
04081                           if ( (l1 + r31) > AM.MaxTer/((LONG)sizeof(WORD)) ) {
04082                             MLOCK(ErrorMessageLock);
04083                             MesPrint("Coefficient overflow during sort");
04084                             MUNLOCK(ErrorMessageLock);
04085                             goto ReturnError;
04086                           }
04087                           m2 = poin[S->tree[i]];
04088                           m3 = ( poin[S->tree[i]] -= r31 );
04089                           do { *m3++ = *m2++; } while ( m2 < m1 );
04090                           m1 = m3;
04091                         }
04092 CopCoef:
04093                         *(poin[S->tree[i]]) += r31;
04094 CopCof2:
04095                         m2 = (WORD *)coef; im = r3;
04096                         NCOPY(m1,m2,im);
04097                         *m1 = r33;
04098                       }
04099                     }
04100 /*
04101                     Now skip to the next term in stream k.
04102 */
04103 NextTerm:
04104                     im = poin2[k][0];
04105                     if ( im < 0 ) {
04106                         r1 = poin2[k][1] - im + 1;
04107                         m1 = poin2[k] + 2;
04108                         m2 = poin[k] - im + 1;
04109                         while ( ++im <= 0 ) *--m1 = *--m2;
04110                         *--m1 = r1;
04111                         poin2[k] = m1;
04112                         im = r1;
04113                     }
04114                     poin[k] = poin2[k];
04115                     ki = S->ktoi[k];
04116                     if ( !par && ( (poin[k] + im + COMPINC) >= S->pStop[ki] )
04117                     && im > 0 ) {
04118 #ifdef WITHZLIB
04119                         PutIn(fin,&(S->iPatches[ki]),S->Patches[ki],&(poin[k]),ki);
04120 #else
04121                         ADDPOS(S->iPatches[ki],PutIn(fin,&(S->iPatches[ki]),
04122                         S->Patches[ki],&(poin[k]),ki));
04123 #endif
04124                         poin2[k] = poin[k] + im;
04125                     }
04126                     else {
04127                         poin2[k] += im;
04128                     }
04129                     goto OneTerm;
04130                 }
04131             }
04132             else if ( S->tree[i] < 0 ) {
04133                 S->tree[i] = k;
04134                 level--;
04135                 goto OneTerm;
```

```
04136             }
04137         }
04138 /*
04139             found the smallest in the set. indicated by k.
04140             write to its destination.
04141 */
04142 #ifdef WITHPTHREADS
04143         if ( AS.MasterSort && ( fout == AR.outfile ) && S == AT.S0 ) {
04144             im = PutToMaster(BHEAD poin[k]);
04145         }
04146         else
04147 #endif
04148         if ( ( im = PutOut(BHEAD poin[k],&position,fout,1) ) < 0 ) {
04149 /* INTERNAL_ERROR_EXCL_START */
04150             MLOCK(ErrorMessageLock);
04151             MesPrint("!>Called from MergePatches with k = %d (stream %d)",k,S->ktoi[k]);
04152             MUNLOCK(ErrorMessageLock);
04153             goto ReturnError;
04154 /* INTERNAL_ERROR_EXCL_STOP */
04155         }
04156         ADDPOS(S->SizeInFile[par],im);
04157         goto NextTerm;
04158     }
04159     else {
04160         goto NormalReturn;
04161     }
04162 EndOfMerge:
04163 #ifdef WITHPTHREADS
04164         if ( AS.MasterSort && ( fout == AR.outfile ) && S == AT.S0 ) {
04165             PutToMaster(BHEAD 0);
04166         }
04167         else
04168 #endif
04169     if ( FlushOut(&position,fout,1) ) goto ReturnError;
04170     ADDPOS(S->SizeInFile[par],1);
04171 EndOfAll:
04172     if ( par == 1 ) {   /* Set the fpatch pointers */
04173 #ifdef WITHZLIB
04174         SeekFile(fout->handle,&position,SEEK_CUR);
04175 #endif
04176         (S->fPatchN)++;
04177         S->fPatches[S->fPatchN] = position;
04178     }
04179     if ( par == 0 && fout != AR.outfile ) {
04180 /*
04181             Output went to sortfile. We have two possibilities:
04182             1:  We are not finished with the current in-out cycle
04183                 In that case we should pop to the next set of patches
04184             2:  We finished a cycle and should clean up the in file
04185                 Then we restart the sort.
04186 */
04187         (S->fPatchN)++;
04188         S->fPatches[S->fPatchN] = position;
04189         if ( ISNOTZEROPOS(S->OldPosIn) ) {      /* We are not done */
04190
04191             SeekFile(fin->handle,&(S->OldPosIn),SEEK_SET);
04192 /*
04193             We don't need extra provisions for the zlib compression here.
04194             If part of an expression has been sorted, the whole has been so.
04195             This means that S->fpincompressed[] will remain the same
04196 */
04197             if ( (ULONG)ReadFile(fin->handle,(UBYTE *)(&(S->inNum)),(LONG)sizeof(WORD)) !=
04198                 sizeof(WORD)
04199               || (ULONG)ReadFile(fin->handle,(UBYTE *)(&(S->OldPosIn)),(LONG)sizeof(POSITION)) !=
04200                 sizeof(POSITION)
04201               || (ULONG)ReadFile(fin->handle,(UBYTE *)S->iPatches,(LONG)((S->inNum)+1)
04202                     *sizeof(POSITION)) != ((S->inNum)+1)*sizeof(POSITION) ) {
04203 /* INTERNAL_ERROR_EXCL_START */
04204                 MLOCK(ErrorMessageLock);
04205                 MesPrint("!>Read error fourth stage sorting");
04206                 MUNLOCK(ErrorMessageLock);
04207                 goto ReturnError;
04208 /* INTERNAL_ERROR_EXCL_STOP */
04209             }
04210             *rr = 0;
04211 #ifdef WITHZLIB
04212             for ( i = 0; i < S->inNum; i++ ) {
04213                 S->fPatchesStop[i] = S->iPatches[i+1];
04214 #ifdef GZIPDEBUG
04215                 MLOCK(ErrorMessageLock);
04216                 MesPrint("%w fPatchesStop[%d] = %10p",i,&(S->fPatchesStop[i]));
04217                 MUNLOCK(ErrorMessageLock);
04218 #endif
04219             }
04220 #endif
04221             goto ConMer;
04222         }
```

```
04223         else {
04224 /*
04225             if ( fin == &(AR.FoStage4[0]) ) {
04226                 s = (UBYTE *)(fin->name); while ( *s ) s++;
04227                 if ( AR.Stage4Name == 1 ) s[-1] -= 1;
04228                 else                      s[-1] += 1;
04229             }
04230 */
04231 /*          TruncateFile(fin->handle); */
04232             UpdateMaxSize();
04233 #ifdef WITHZLIB
04234             ClearSortGZIP(fin);
04235 #endif
04236             CloseFile(fin->handle);
04237             remove(fin->name);      /* Gives diskspace free again. */
04238 #ifdef GZIPDEBUG
04239             MLOCK(ErrorMessageLock);
04240             MesPrint("%w MergePatches removed in file %s",fin->name);
04241             MUNLOCK(ErrorMessageLock);
04242 #endif
04243 /*
04244             if ( fin == &(AR.FoStage4[0]) ) {
04245                 s = (UBYTE *)(fin->name); while ( *s ) s++;
04246                 if ( AR.Stage4Name == 1 ) s[-1] += 1;
04247                 else                      s[-1] -= 1;
04248             }
04249 */
04250             fin->handle = -1;
04251             { FILEHANDLE *ff = fin; fin = fout; fout = ff; }
04252             PUTZERO(S->SizeInFile[0]);
04253             goto NewMerge;
04254         }
04255     }
04256     if ( par == 0 ) {
04257 /*      TruncateFile(fin->handle); */
04258         UpdateMaxSize();
04259 #ifdef WITHZLIB
04260         ClearSortGZIP(fin);
04261 #endif
04262         CloseFile(fin->handle);
04263         remove(fin->name);
04264         fin->handle = -1;
04265 #ifdef GZIPDEBUG
04266         MLOCK(ErrorMessageLock);
04267         MesPrint("%w MergePatches removed in file %s",fin->name);
04268         MUNLOCK(ErrorMessageLock);
04269 #endif
04270     }
04271 NormalReturn:
04272 #ifdef WITHZLIB
04273     AR.gzipCompress = oldgzipCompress;
04274 #endif
04275     return(0);
04276 ReturnError:
04277 #ifdef WITHZLIB
04278     AR.gzipCompress = oldgzipCompress;
04279 #endif
04280     return(-1);
04281 #ifndef WITHZLIB
04282 PatwCall:
04283     MLOCK(ErrorMessageLock);
04284     MesPrint("Error while writing to file.");
04285     goto PatCall2;
04286 #endif
04287 PatCall:;
04288     MLOCK(ErrorMessageLock);
04289 PatCall2:;
04290     MesCall("MergePatches");
04291     MUNLOCK(ErrorMessageLock);
04292 #ifdef WITHZLIB
04293     AR.gzipCompress = oldgzipCompress;
04294 #endif
04295     SETERROR(-1)
04296 }
04297
04298 /*
04299         #] MergePatches :
04300         #[ StoreTerm :              WORD StoreTerm(term)
04301 */
04311 int StoreTerm(PHEAD WORD *term)
04312 {
04313     GETBIDENTITY
04314     SORTING *S = AT.SS;
04315     WORD **ss, *lfill, j, *t;
04316     POSITION pp;
04317     LONG lSpace, sSpace, RetCode, over, tover;
04318
```

```
04319     // Update SortVerbose counters
04320     S->verbUnsortedSize += *term * sizeof(*term);
04321     if ( S->verbMaxTermSize < *term ) S->verbMaxTermSize = *term;
04322
04323     if ( ( ( AP.PreDebug & DUMPTOSORT ) == DUMPTOSORT ) && AR.sLevel == 0 ) {
04324 #ifdef WITHPTHREADS
04325         snprintf((char *)(THRbuf),100,"StoreTerm(%d)",AT.identity);
04326         PrintTerm(term,(char *)(THRbuf));
04327 #else
04328         PrintTerm(term,"StoreTerm");
04329 #endif
04330     }
04331     if ( AM.exitflag && AR.sLevel == 0 ) return(0);
04332     S->sFill = *(S->PoinFill);
04333     if ( S->sTerms >= S->TermsInSmall || ( S->sFill + *term ) >= S->sTop ) {
04334 /*
04335     The small buffer is full. It has to be sorted and written.
04336 */
04337         // Update SortVerbose counters
04338         if ( S->sTerms >= S->TermsInSmall ) S->verbSBsortTerms++;
04339         else S->verbSBsortCap++;
04340
04341         tover = over = S->sTerms;
04342         ss = S->sPointer;
04343         ss[over] = 0;
04344 #ifdef SPLITTIME
04345         PrintTime((UBYTE *)"Before SplitMerge");
04346 #endif
04347         ss[SplitMerge(BHEAD ss,over)] = 0;
04348 #ifdef SPLITTIME
04349         PrintTime((UBYTE *)"After SplitMerge");
04350 #endif
04351         sSpace = 0;
04352         if ( over > 0 ) {
04353             sSpace = ComPress(ss,&RetCode);
04354             S->TermsLeft -= over - RetCode;
04355         }
04356         sSpace++;
04357
04358         lSpace = sSpace + (S->lFill - S->lBuffer)
04359                  - (AM.MaxTer/sizeof(WORD))*((LONG)S->lPatch);
04360         SETBASEPOSITION(pp,lSpace);
04361         MULPOS(pp,sizeof(WORD));
04362         if ( S->file.handle >= 0 ) {
04363             ADD2POS(pp,S->fPatches[S->fPatchN]);
04364         }
04365         if ( S == AT.S0 ) { /* Only statistics at ground level */
04366             WriteStats(&pp,STATSSPLITMERGE,CHECKLOGTYPE);
04367         }
04368         if ( ( S->lPatch >= S->MaxPatches ) ||
04369              ( ( (WORD *)(((UBYTE *)(S->lFill + sSpace)) + 2*AM.MaxTer ) ) >= S->lTop ) ) {
04370 /*
04371             The large buffer is too full. Merge and write it
04372 */
04373             // Update SortVerbose counters
04374             if ( S->lPatch >= S->MaxPatches ) S->verbLBsortPatches++;
04375             else S->verbLBsortCap++;
04376
04377             if ( MergePatches(1) ) goto StoreCall;
04378 /*
04379             pp = S->SizeInFile[1];
04380             ADDPOS(pp,sSpace);
04381             MULPOS(pp,sizeof(WORD));
04382 */
04383             SETBASEPOSITION(pp,sSpace);
04384             MULPOS(pp,sizeof(WORD));
04385             ADD2POS(pp,S->fPatches[S->fPatchN]);
04386
04387             if ( S == AT.S0 ) { /* Only statistics at ground level */
04388                 WriteStats(&pp,STATSMERGETOFILE,CHECKLOGTYPE);
04389             }
04390             S->lPatch = 0;
04391             S->lFill = S->lBuffer;
04392         }
04393         S->Patches[S->lPatch++] = S->lFill;
04394         lfill = (WORD *)(((UBYTE *)(S->lFill)) + AM.MaxTer);
04395         if ( tover > 0 ) {
04396             ss = S->sPointer;
04397             while ( ( t = *ss++ ) != 0 ) {
04398                 j = *t;
04399                 if ( j < 0 ) j = t[1] + 2;
04400                 while ( --j >= 0 ){
04401                   *lfill++ = *t++;
04402                 }
04403             }
04404         }
04405         *lfill++ = 0;
```

```
04406         S->lFill = lfill;
04407         S->sTerms = 0;
04408         S->PoinFill = S->sPointer;
04409         *(S->PoinFill) = S->sFill = S->sBuffer;
04410     }
04411     j = *term;
04412     while ( --j >= 0 ) *S->sFill++ = *term++;
04413     S->sTerms++;
04414     S->GenTerms++;
04415     S->TermsLeft++;
04416     *++S->PoinFill = S->sFill;
04417
04418     return(0);
04419
04420 StoreCall:
04421     MLOCK(ErrorMessageLock);
04422     MesCall("StoreTerm");
04423     MUNLOCK(ErrorMessageLock);
04424     SETERROR(-1)
04425 }
04426
04427 /*
04428         #] StoreTerm :
04429         #[ StageSort :              void StageSort(FILEHANDLE *fout)
04430 */
04437 void StageSort(FILEHANDLE *fout)
04438 {
04439     GETIDENTITY
04440     SORTING *S = AT.SS;
04441     if ( S->fPatchN >= S->MaxFpatches ) {
04442         POSITION position;
04443         if ( S != AT.S0 ) {
04444 /*
04445             There are no proper provisions for stage 4 or higher sorts
04446             for function arguments and $ variables. The reason:
04447             The current code maps out the patches, based on the size of
04448             the buffers in the FoStage4 structs, while they are used
04449             inside the S->file struct that may have far smaller buffers.
04450             By itself that might still be repairable, but it goes completely
04451             wrong when during the sort polyRatFuns have to be added and they
04452             would go into stage4 (very rare but possible).
04453             The only really correct solution would be to put FoStage4 structs
04454             in all sort levels. Messy. (JV 8-oct-2018).
04455 */
04456             MLOCK(ErrorMessageLock);
04457             MesPrint("Currently Stage 4 sorts are not allowed for function arguments or $
      variables.");
04458             MesPrint("Please increase correspondingsorting parameters (sub-) in the setup.");
04459             MUNLOCK(ErrorMessageLock);
04460             Terminate(-1);
04461         }
04462         PUTZERO(position);
04463         MLOCK(ErrorMessageLock);
04464 #ifdef WITHPTHREADS
04465         MesPrint("StageSort in thread %d",identity);
04466 #elif defined(WITHMPI)
04467         MesPrint("StageSort in process %d",PF.me);
04468 #else
04469         MesPrint("StageSort");
04470 #endif
04471         MUNLOCK(ErrorMessageLock);
04472         SeekFile(fout->handle,&position,SEEK_END);
04473 /*
04474         No extra compression data has to be written.
04475         S->fpincompressed should remain valid.
04476 */
04477         if ( (ULONG)WriteFile(fout->handle,(UBYTE *)(&(S->fPatchN)),(LONG)sizeof(WORD)) !=
04478             sizeof(WORD)
04479           || (ULONG)WriteFile(fout->handle,(UBYTE *)(&(S->OldPosOut)),(LONG)sizeof(POSITION)) !=
04480             sizeof(POSITION)
04481           || (ULONG)WriteFile(fout->handle,(UBYTE *)(S->fPatches),(LONG)(S->fPatchN+1)
04482                   *sizeof(POSITION)) != (S->fPatchN+1)*sizeof(POSITION) ) {
04483             MLOCK(ErrorMessageLock);
04484             MesPrint("Write error while staging sort. Disk full?");
04485             MUNLOCK(ErrorMessageLock);
04486             Terminate(-1);
04487         }
04488         S->OldPosOut = position;
04489         fout->filesize = position;
04490         ADDPOS(fout->filesize,(S->fPatchN+2)*sizeof(POSITION) + sizeof(WORD));
04491         fout->POposition = fout->filesize;
04492         S->fPatches[0] = fout->filesize;
04493         S->fPatchN = 0;
04494
04495         if ( AR.FoStage4[0].PObuffer == 0 ) {
04496             AR.FoStage4[0].PObuffer = (WORD *)Malloc1(AR.FoStage4[0].POsize*sizeof(WORD)
04497                                             ,"Stage 4 buffer");
```

```
04498             AR.FoStage4[0].POfill   = AR.FoStage4[0].PObuffer;
04499             AR.FoStage4[0].POstop   = AR.FoStage4[0].PObuffer
04500                         + AR.FoStage4[0].POsize/sizeof(WORD);
04501 #ifdef WITHPTHREADS
04502             AR.FoStage4[0].pthreadslock = dummylock;
04503 #endif
04504         }
04505         if ( AR.FoStage4[1].PObuffer == 0 ) {
04506             AR.FoStage4[1].PObuffer = (WORD *)Malloc1(AR.FoStage4[1].POsize*sizeof(WORD)
04507                                                 ,"Stage 4 buffer");
04508             AR.FoStage4[1].POfill   = AR.FoStage4[1].PObuffer;
04509             AR.FoStage4[1].POstop   = AR.FoStage4[1].PObuffer
04510                         + AR.FoStage4[1].POsize/sizeof(WORD);
04511 #ifdef WITHPTHREADS
04512             AR.FoStage4[1].pthreadslock = dummylock;
04513 #endif
04514         }
04515         S->stage4 = 1;
04516     }
04517 }
04518
04519 /*
04520         #] StageSort :
04521         #[ SortWild :               WORD SortWild(w,nw)
04522 */
04536 int SortWild(WORD *w, WORD nw)
04537 {
04538     GETIDENTITY
04539     WORD *v, *s, *m, k, i;
04540     WORD *pScrat, *stop, *sv;
04541     int error = 0;
04542     pScrat = AT.WorkPointer;
04543     if ( ( AT.WorkPointer + 8 * AM.MaxWildcards ) >= AT.WorkTop ) {
04544         MLOCK(ErrorMessageLock);
04545         MesWork();
04546         MUNLOCK(ErrorMessageLock);
04547         return(-1);
04548     }
04549     stop = w + nw;
04550     i = 0;
04551     while ( i < nw ) {
04552         m = w + i;
04553         v = m + m[1];
04554         while ( v < stop && (
04555              *v == FROMSET || *v == SETTONUM || *v == LOADDOLLAR ) ) v += v[1];
04556         while ( v < stop ) {
04557             if ( *v >= 0 ) {
04558                 if ( AM.Ordering[*v] < AM.Ordering[*m] ) {
04559                     m = v;
04560                 }
04561                 else if ( *v == *m ) {
04562                     if ( v[2] < m[2] ) {
04563                         m = v;
04564                     }
04565                     else if ( v[2] == m[2] ) {
04566                         s = m + m[1];
04567                         sv = v + v[1];
04568                         if ( s < stop && ( *s == FROMSET
04569                          || *s == SETTONUM || *s == LOADDOLLAR ) ) {
04570                             if ( sv < stop && ( *sv == FROMSET
04571                             || *sv == SETTONUM || *sv == LOADDOLLAR ) ) {
04572                                 if ( s[2] != sv[2] ) {
04573 /* INTERNAL_ERROR_EXCL_START */
04574                                     error = -1;
04575                                     MLOCK(ErrorMessageLock);
04576                                     MesPrint("!>Wildcard set conflict");
04577                                     MUNLOCK(ErrorMessageLock);
04578 /* INTERNAL_ERROR_EXCL_STOP */
04579                                 }
04580                             }
04581                             *v = -1;
04582                         }
04583                         else {
04584                             if ( sv < stop && ( *sv == FROMSET
04585                             || *sv == SETTONUM || *sv == LOADDOLLAR ) ) {
04586                                 *m = -1;
04587                                 m = v;
04588                             }
04589                             else {
04590                                 *v = -1;
04591                             }
04592                         }
04593                     }
04594                 }
04595             }
04596             v += v[1];
04597             while ( v < stop && ( *v == FROMSET
```

```
04598               || *v == SETTONUM || *v == LOADDOLLAR ) ) v += v[1];
04599         }
04600         s = pScrat;
04601         v = m;
04602         k = m[1];
04603         NCOPY(s,m,k);
04604         while ( m < stop && ( *m == FROMSET
04605          || *m == SETTONUM || *m == LOADDOLLAR ) ) {
04606             k = m[1];
04607             NCOPY(s,m,k);
04608         }
04609         *v = -1;
04610         pScrat = s;
04611         i = 0;
04612         while ( i < nw && ( w[i] < 0 || w[i] == FROMSET
04613         || w[i] == SETTONUM || w[i] == LOADDOLLAR ) ) i += w[i+1];
04614     }
04615     AC.NwildC = k = WORDDIF(pScrat,AT.WorkPointer);
04616     s = AT.WorkPointer;
04617     m = w;
04618     NCOPY(m,s,k);
04619     AC.WildC = m;
04620     return(error);
04621 }
04622
04623 /*
04624         #] SortWild :
04625         #[ CleanUpSort :            void CleanUpSort(num)
04626 */
04631 void CleanUpSort(int num)
04632 {
04633     GETIDENTITY
04634     SORTING *S;
04635     int minnum = num, i;
04636     if ( AN.FunSorts ) {
04637         if ( num == -1 ) {
04638             if ( AN.MaxFunSorts > 3 ) {
04639                 minnum = (AN.MaxFunSorts+4)/2;
04640             }
04641             else minnum = 4;
04642         }
04643         else if ( minnum == 0 ) minnum = 1;
04644         for ( i = minnum; i < AN.NumFunSorts; i++ ) {
04645             S = AN.FunSorts[i];
04646             if ( S ) {
04647                 if ( S->file.handle >= 0 ) {
04648 /*                  TruncateFile(S->file.handle); */
04649                     UpdateMaxSize();
04650 #ifdef WITHZLIB
04651                     ClearSortGZIP(&(S->file));
04652 #endif
04653                     CloseFile(S->file.handle);
04654                     S->file.handle = -1;
04655                     remove(S->file.name);
04656 #ifdef GZIPDEBUG
04657                     MLOCK(ErrorMessageLock);
04658                     MesPrint("%w CleanUpSort removed file %s",S->file.name);
04659                     MUNLOCK(ErrorMessageLock);
04660 #endif
04661                 }
04662                 M_free(S->sPointer, "CleanUpSort: sPointer");
04663                 M_free(S->Patches, "CleanUpSort: Patches");
04664                 M_free(S->pStop, "CleanUpSort: pStop");
04665                 M_free(S->poina, "CleanUpSort: poina");
04666                 M_free(S->poin2a, "CleanUpSort: poin2a");
04667                 M_free(S->fPatches, "CleanUpSort: fPatches");
04668                 M_free(S->fPatchesStop, "CleanUpSort: fPatchesStop");
04669                 M_free(S->inPatches, "CleanUpSort: inPatches");
04670                 M_free(S->tree, "CleanUpSort: tree");
04671                 M_free(S->used, "CleanUpSort: used");
04672 #ifdef WITHZLIB
04673                 M_free(S->fpcompressed, "CleanUpSort: fpcompressed");
04674                 M_free(S->fpincompressed, "CleanUpSort: fpincompressed");
04675 #endif
04676                 M_free(S->ktoi, "CleanUpSort: ktoi");
04677                 M_free(S->lBuffer, "CleanUpSort: lBuffer+sBuffer");
04678                 M_free(S->file.PObuffer, "CleanUpSort: PObuffer");
04679                 M_free(S, "CleanUpSort: sorting struct");
04680             }
04681             AN.FunSorts[i] = 0;
04682         }
04683         AN.MaxFunSorts = minnum;
04684         if ( num == 0 ) {
04685             S = AN.FunSorts[0];
04686             if ( S ) {
04687                 if ( S->file.handle >= 0 ) {
04688 /*                  TruncateFile(S->file.handle); */
```

```
04689                     UpdateMaxSize();
04690 #ifdef WITHZLIB
04691                     ClearSortGZIP(&(S->file));
04692 #endif
04693                     CloseFile(S->file.handle);
04694                     S->file.handle = -1;
04695                     remove(S->file.name);
04696 #ifdef GZIPDEBUG
04697                     MLOCK(ErrorMessageLock);
04698                     MesPrint("%w CleanUpSort removed file %s",S->file.name);
04699                     MUNLOCK(ErrorMessageLock);
04700 #endif
04701                 }
04702             }
04703         }
04704     }
04705     for ( i = 0; i < 2; i++ ) {
04706         if ( AR.FoStage4[i].handle >= 0 ) {
04707             UpdateMaxSize();
04708 #ifdef WITHZLIB
04709             ClearSortGZIP(&(AR.FoStage4[i]));
04710 #endif
04711             CloseFile(AR.FoStage4[i].handle);
04712             remove(AR.FoStage4[i].name);
04713             AR.FoStage4[i].handle = -1;
04714 #ifdef GZIPDEBUG
04715             MLOCK(ErrorMessageLock);
04716             MesPrint("%w CleanUpSort removed stage4 file %s",AR.FoStage4[i].name);
04717             MUNLOCK(ErrorMessageLock);
04718 #endif
04719         }
04720     }
04721 }
04722
04723 /*
04724         #] CleanUpSort :
04725         #[ LowerSortLevel :         void LowerSortLevel()
04726 */
04731 void LowerSortLevel(void)
04732 {
04733     GETIDENTITY
04734     if ( AR.sLevel >= 0 ) {
04735         AR.sLevel--;
04736         if ( AR.sLevel >= 0 ) AT.SS = AN.FunSorts[AR.sLevel];
04737     }
04738 }
04739
04740 /*
04741         #] LowerSortLevel :
04742         #[ PolyRatFunSpecial :
04743
04744         Keeps only the most divergent term in AR.PolyFunVar
04745         We assume that the terms are already in that notation.
04746 */
04747
04748 WORD *PolyRatFunSpecial(PHEAD WORD *t1, WORD *t2)
04749 {
04750     WORD *oldworkpointer = AT.WorkPointer, *t, *r;
04751     WORD exp1, exp2;
04752     int i;
04753     t = t1+FUNHEAD;
04754     if ( *t == -SYMBOL ) {
04755         if ( t[1] != AR.PolyFunVar ) goto Illegal;
04756         exp1 = 1;
04757         if ( t[2] != -SNUMBER ) goto Illegal;
04758         t[3] = 1;
04759     }
04760     else if ( *t == -SNUMBER ) {
04761         t[1] = 1;
04762         t += 2;
04763         if ( *t == -SYMBOL ) {
04764             if ( t[1] != AR.PolyFunVar ) goto Illegal;
04765             exp1 = -1;
04766         }
04767         else if ( *t == -SNUMBER ) {
04768             t[1] = 1;
04769             exp1 = 0;
04770         }
04771         else if ( *t == ARGHEAD+8 && t[ARGHEAD] == 8 && t[ARGHEAD+1] == SYMBOL
04772             && t[ARGHEAD+3] == AR.PolyFunVar ) {
04773             t[ARGHEAD+5] = 1;
04774             t[ARGHEAD+6] = 1;
04775             t[ARGHEAD+7] = 3;
04776             exp1 = -t[ARGHEAD+4];
04777         }
04778         else goto Illegal;
04779     }
```

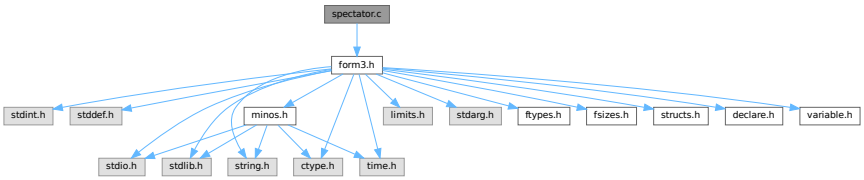
```
04780     else if ( *t == ARGHEAD+8 && t[ARGHEAD] == 8 && t[ARGHEAD+1] == SYMBOL
04781         && t[ARGHEAD+3] == AR.PolyFunVar ) {
04782         t[ARGHEAD+5] = 1;
04783         t[ARGHEAD+6] = 1;
04784         t[ARGHEAD+7] = 3;
04785         exp1 = t[ARGHEAD+4];
04786         t += *t;
04787         if ( *t != -SNUMBER ) goto Illegal;
04788         t[1] = 1;
04789     }
04790     else goto Illegal;
04791
04792     t = t2+FUNHEAD;
04793     if ( *t == -SYMBOL ) {
04794         if ( t[1] != AR.PolyFunVar ) goto Illegal;
04795         exp2 = 1;
04796         if ( t[2] != -SNUMBER ) goto Illegal;
04797         t[3] = 1;
04798     }
04799     else if ( *t == -SNUMBER ) {
04800         t[1] = 1;
04801         t += 2;
04802         if ( *t == -SYMBOL ) {
04803             if ( t[1] != AR.PolyFunVar ) goto Illegal;
04804             exp2 = -1;
04805         }
04806         else if ( *t == -SNUMBER ) {
04807             t[1] = 1;
04808             exp2 = 0;
04809         }
04810         else if ( *t == ARGHEAD+8 && t[ARGHEAD] == 8 && t[ARGHEAD+1] == SYMBOL
04811             && t[ARGHEAD+3] == AR.PolyFunVar ) {
04812             t[ARGHEAD+5] = 1;
04813             t[ARGHEAD+6] = 1;
04814             t[ARGHEAD+7] = 3;
04815             exp2 = -t[ARGHEAD+4];
04816         }
04817         else goto Illegal;
04818     }
04819     else if ( *t == ARGHEAD+8 && t[ARGHEAD] == 8 && t[ARGHEAD+1] == SYMBOL
04820         && t[ARGHEAD+3] == AR.PolyFunVar ) {
04821         t[ARGHEAD+5] = 1;
04822         t[ARGHEAD+6] = 1;
04823         t[ARGHEAD+7] = 3;
04824         exp2 = t[ARGHEAD+4];
04825         t += *t;
04826         if ( *t != -SNUMBER ) goto Illegal;
04827         t[1] = 1;
04828     }
04829     else goto Illegal;
04830
04831     if ( exp1 <= exp2 ) { i = t1[1]; r = t1; }
04832     else                { i = t2[1]; r = t2; }
04833     t = oldworkpointer;
04834     NCOPY(t,r,i)
04835
04836     return(oldworkpointer);
04837 Illegal:
04838     MesPrint("Illegal occurrence of PolyRatFun with divergent option");
04839     Terminate(-1);
04840     return(0);
04841 }
04842
04843 /*
04844         #] PolyRatFunSpecial :
04845         #[ SimpleSplitMerge :
04846
04847         Sorts an array of WORDs. No adding of equal objects.
04848 */
04849
04850 void SimpleSplitMergeRec(WORD *array,WORD num,WORD *auxarray)
04851 {
04852     WORD n1,n2,i,j,k,*t1,*t2;
04853     if ( num < 2 ) return;
04854     if ( num == 2 ) {
04855         if ( array[0] > array[1] ) {
04856             EXCH(array[0],array[1])
04857         }
04858         return;
04859     }
04860     n1 = num/2;
04861     n2 = num - n1;
04862     SimpleSplitMergeRec(array,n1,auxarray);
04863     SimpleSplitMergeRec(array+n1,n2,auxarray);
04864     if ( array[n1-1] <= array[n1] ) return;
04865
04866     t1 = array; t2 = auxarray; i = n1; NCOPY(t2,t1,i);
```

```
04867     i = 0; j = n1; k = 0;
04868     while ( i < n1 && j < num ) {
04869         if ( auxarray[i] <= array[j] ) { array[k++] = auxarray[i++]; }
04870         else { array[k++] = array[j++]; }
04871     }
04872     while ( i < n1 ) array[k++] = auxarray[i++];
04873 /*
04874     Remember: remnants of j are still in place!
04875 */
04876 }
04877
04878 void SimpleSplitMerge(WORD *array,WORD num)
04879 {
04880     WORD *auxarray = Malloc1(sizeof(WORD)*num/2,"SimpleSplitMerge");
04881     SimpleSplitMergeRec(array,num,auxarray);
04882     M_free(auxarray,"SimpleSplitMerge");
04883 }
04884
04885 /*
04886         #] SimpleSplitMerge :
04887         #[ BinarySearch :
04888
04889         Searches in the sorted array with length num for the object x.
04890         If x is in the list, it returns the number of the array element
04891         that matched. If it is not in the list, it returns -1.
04892         If there are identical objects in the list, which one will
04893         match is quasi random.
04894 */
04895
04896 WORD BinarySearch(WORD *array,WORD num,WORD x)
04897 {
04898     WORD i, bot, top, med;
04899     if ( num < 8 ) {
04900         for ( i = 0; i < num; i++ ) if ( array[i] == x ) return(i);
04901         return(-1);
04902     }
04903     if ( array[0] > x || array[num-1] < x ) return(-1);
04904     bot = 0; top = num-1; med = (top+bot)/2;
04905     do {
04906         if ( array[med] == x ) return(med);
04907         if ( array[med] < x ) { bot = med+1; }
04908         else { top = med-1; }
04909         med = (top+bot)/2;
04910     } while ( med >= bot && med <= top );
04911     return(-1);
04912 }
04913
04914 /*
04915         #] BinarySearch :
04916     #] SortUtilities :
04917 */
```

4.133 spectator.c File Reference

`#include "form3.h"`

Include dependency graph for spectator.c:



Functions

- int CoCreateSpectator (UBYTE *inp)
- int CoToSpectator (UBYTE *inp)

- int CoRemoveSpectator (UBYTE *inp)
- int CoEmptySpectator (UBYTE *inp)
- int PutInSpectator (WORD *term, WORD specnum)
- void FlushSpectators (void)
- int CoCopySpectator (UBYTE *inp)
- WORD GetFromSpectator (WORD *term, WORD specnum)
- void ClearSpectators (WORD par)

4.133.1 Detailed Description

File contains the code for the spectator files and their control.

Definition in file spectator.c.

4.133.2 Function Documentation

4.133.2.1 CoCreateSpectator()

```
int CoCreateSpectator (
            UBYTE * inp )
```

Definition at line 89 of file spectator.c.

4.133.2.2 CoToSpectator()

```
int CoToSpectator (
            UBYTE * inp )
```

Definition at line 181 of file spectator.c.

4.133.2.3 CoRemoveSpectator()

```
int CoRemoveSpectator (
            UBYTE * inp )
```

Definition at line 233 of file spectator.c.

4.133.2.4 CoEmptySpectator()

```
int CoEmptySpectator (
            UBYTE * inp )
```

Definition at line 293 of file spectator.c.

4.133.2.5 PutInSpectator()

```
int PutInSpectator (
            WORD * term,
            WORD specnum )
```

Definition at line 345 of file spectator.c.

4.133.2.6 FlushSpectators()

```
void FlushSpectators (
            void  )
```

Definition at line 404 of file spectator.c.

4.133.2.7 CoCopySpectator()

```
int CoCopySpectator (
            UBYTE * inp )
```

Definition at line 463 of file spectator.c.

4.133.2.8 GetFromSpectator()

```
WORD GetFromSpectator (
            WORD * term,
            WORD specnum )
```

Definition at line 605 of file spectator.c.

4.133.2.9 ClearSpectators()

```
void ClearSpectators (
            WORD par )
```

Definition at line 685 of file spectator.c.

4.134 spectator.c

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032 /*
00033     #[ includes :
00034 */
00035
00036 #include "form3.h"
00037
00038 /*
00039     #] includes :
00040     #[ Commentary :
00041
00042     We use an array of SPECTATOR structs in AM.SpectatorFiles.
00043     When a spectator is removed this leaves a hole. This means that
00044     we cannot use AM.NumSpectatorFiles but always have to scan up to
00045     AM.SizeForSpectatorFiles which is the size of the array.
00046     An element is in use when it has a name. This is the name of the
00047     expression that is associated with it. There is also the number of
00048     the expression, but because the expressions are frequently renumbered
00049     at the end of a module, we always search for the spectators by name.
00050     The expression number is only valid in the current module.
00051     During execution we use the number of the spectator.
00052
00053     The FILEHANDLE struct is borrowed from the structs for the scratch
00054     files, but we do not keep copies for all workers as with the scratch
00055     files. This brings some limitations (but saves much space). Basically
00056     the reading can only be done by one master or worker. And because
00057     we use the buffer both for writing and for reading we cannot read and
00058     write in the same module.
00059
00060     Processor can see that an expression is to be filled with a spectator
00061     because we replace the compiler buffer number in the prototype by
00062     -specnum-1. Of course, after this filling has taken place we should
00063     make sure that in the next module there is a nonnegative number there.
00064     The input is then obtained from GetFromSpectator instead from GetTerm.
00065     This needed an extra argument in ThreadsProcessor. InParallelProcessor
00066     can figure it out by itself. ParFORM still needs to be treated for this.
00067
00068     The writing is to a single buffer. Hence it needs a lock. It is possible
00069     to give all workers their own buffers (at great memory cost) and merge
00070     the results when needed. That would be friendlier on ParFORM. We ALWAYS
00071     assume that the order of the terms in the spectator file is random.
00072
00073     In the first version there is no compression in the file. This could
00074     change in later versions because both the writing and the reading are
00075     purely sequential. Brackets are not possible.
00076
00077     Currently, ParFORM allows use of spectators only in the sequential
00078     mode. The parallelization is switched off in modules containing
00079     ToSpectator or CopySpectator. Workers never create or access to
00080     spectator files. Their file handles are always -1. We leave the
00081     parallelization of modules with spectators for future work.
00082
00083     #] Commentary :
00084     #[ CoCreateSpectator :
00085
00086     Syntax: CreateSpectator name_of_expr "filename";
```

```
00087 */
00088
00089 int CoCreateSpectator(UBYTE *inp)
00090 {
00091     UBYTE *p, *q, *filename, c, cc;
00092     WORD c1, c2, numexpr = 0, specnum, HadOne = 0;
00093     FILEHANDLE *fh;
00094     while ( *inp == ',' ) inp++;
00095     if ( ( q = SkipAName(inp) ) == 0 || q[-1] == '_' ) {
00096         MesPrint("&Illegal name for expression");
00097         return(1);
00098     }
00099     c = *q; *q = 0;
00100     if ( GetVar(inp,&c1,&c2,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
00101         if ( c2 == CEXPRESSION &&
00102                 Expressions[c1].status == DROPSPECTATOREXPRESSION ) {
00103             numexpr = c1;
00104             Expressions[numexpr].status = SPECTATOREXPRESSION;
00105             HadOne = 1;
00106         }
00107         else {
00108             MesPrint("&The name %s has been used already.",inp);
00109             *q = c;
00110             return(1);
00111         }
00112     }
00113     p = q+1;
00114     while ( *p == ',' ) p++;
00115     if ( *p != '"' ) goto Syntax;
00116     p++; filename = p;
00117     while ( *p && *p != '"' ) {
00118         if ( *p == '\\' ) p++;
00119         p++;
00120     }
00121     if ( *p != '"' ) goto Syntax;
00122     q = p+1;
00123     while ( *q && ( *q == ',' || *q == ' ' || *q == '\t' ) ) q++;
00124     if ( *q ) goto Syntax;
00125     cc = *p; *p = 0;
00126 /*
00127     Now we need to: create a struct for the spectator file.
00128 */
00129     if ( HadOne == 0 )
00130         numexpr = EntVar(CEXPRESSION,inp,SPECTATOREXPRESSION,0,0,0);
00131     fh = AllocFileHandle(1,(char *)filename);
00132 /*
00133     Make sure there is space in the AM.spectatorfiles array
00134 */
00135     if ( AM.NumSpectatorFiles >= AM.SizeForSpectatorFiles || AM.SpectatorFiles == 0 ) {
00136         int newsize, i;
00137         SPECTATOR *newspectators;
00138         if ( AM.SizeForSpectatorFiles == 0 ) {
00139             newsize = 10;
00140             AM.NumSpectatorFiles = AM.SizeForSpectatorFiles = 0;
00141         }
00142         else newsize = AM.SizeForSpectatorFiles*2;
00143         newspectators = (SPECTATOR *)Malloc1(newsize*sizeof(SPECTATOR),"Spectators");
00144         for ( i = 0; i < AM.NumSpectatorFiles; i++ )
00145             newspectators[i] = AM.SpectatorFiles[i];
00146         for ( ; i < newsize; i++ ) {
00147             newspectators[i].fh = 0;
00148             newspectators[i].name = 0;
00149             newspectators[i].exprnumber = -1;
00150             newspectators[i].flags = 0;
00151             PUTZERO(newspectators[i].position);
00152             PUTZERO(newspectators[i].readpos);
00153         }
00154         AM.SizeForSpectatorFiles = newsize;
00155         if ( AM.SpectatorFiles != 0 ) M_free(AM.SpectatorFiles,"Spectators");
00156         AM.SpectatorFiles = newspectators;
00157         specnum = AM.NumSpectatorFiles++;
00158     }
00159     else {
00160         for ( specnum = 0; specnum < AM.SizeForSpectatorFiles; specnum++ ) {
00161             if ( AM.SpectatorFiles[specnum].name == 0 ) break;
00162         }
00163         AM.NumSpectatorFiles++;
00164     }
00165     PUTZERO(AM.SpectatorFiles[specnum].position);
00166     AM.SpectatorFiles[specnum].name = (char *)(strDup1(inp,"Spectator expression name"));
00167     AM.SpectatorFiles[specnum].fh = fh;
00168     AM.SpectatorFiles[specnum].exprnumber = numexpr;
00169     *p = cc;
00170     return(0);
00171 Syntax:
00172     MesPrint("&Proper syntax is: CreateSpectator,exprname,\"filename\";");
00173     return(-1);
```

```
00174 }
00175
00176 /*
00177     #] CoCreateSpectator :
00178     #[ CoToSpectator :
00179 */
00180
00181 int CoToSpectator(UBYTE *inp)
00182 {
00183     UBYTE *q;
00184     WORD c1, numexpr;
00185     int i;
00186     while ( *inp == ',' ) inp++;
00187     if ( ( q = SkipAName(inp) ) == 0 || q[-1] == '_' ) {
00188         MesPrint("&Illegal name for expression");
00189         return(1);
00190     }
00191     if ( *q != 0 ) goto Syntax;
00192     if ( GetVar(inp,&c1,&numexpr,ALLVARIABLES,NOAUTO) == NAMENOTFOUND ||
00193             c1 != CEXPRESSION ) {
00194         MesPrint("&%s is not a valid expression.",inp);
00195         return(1);
00196     }
00197     if ( Expressions[numexpr].status != SPECTATOREXPRESSION ) {
00198         MesPrint("&%s is not an active spectator.",inp);
00199         return(1);
00200     }
00201     for ( i = 0; i < AM.SizeForSpectatorFiles; i++ ) {
00202       if ( AM.SpectatorFiles[i].name != 0 ) {
00203         if ( StrCmp((UBYTE *)(AM.SpectatorFiles[i].name),(UBYTE *)(inp)) == 0 ) break;
00204       }
00205     }
00206     if ( i >= AM.SizeForSpectatorFiles ) {
00207         MesPrint("&Spectator %s not found.",inp);
00208         return(1);
00209     }
00210     if ( ( AM.SpectatorFiles[i].flags & READSPECTATORFLAG ) != 0 ) {
00211         MesPrint("&Spectator %s: It is not permitted to read from and write to the same spectator in
      one module.",inp);
00212         return(1);
00213     }
00214     AM.SpectatorFiles[i].exprnumber = numexpr;
00215     Add3Com(TYPETOSPECTATOR,i);
00216 #ifdef WITHMPI
00217     /*
00218      * In ParFORM, ToSpectator has to be executed on the master.
00219      */
00220     AC.mparallelflag |= NOPARALLEL_SPECTATOR;
00221 #endif
00222     return(0);
00223 Syntax:
00224     MesPrint("&Proper syntax is: ToSpectator,exprname;");
00225     return(-1);
00226 }
00227
00228 /*
00229     #] CoToSpectator :
00230     #[ CoRemoveSpectator :
00231 */
00232
00233 int CoRemoveSpectator(UBYTE *inp)
00234 {
00235     UBYTE *q;
00236     WORD c1, numexpr;
00237     int i;
00238     SPECTATOR *sp;
00239     while ( *inp == ',' ) inp++;
00240     if ( ( q = SkipAName(inp) ) == 0 || q[-1] == '_' ) {
00241         MesPrint("&Illegal name for expression");
00242         return(1);
00243     }
00244     if ( *q != 0 ) goto Syntax;
00245     if ( GetVar(inp,&c1,&numexpr,ALLVARIABLES,NOAUTO) == NAMENOTFOUND ||
00246             c1 != CEXPRESSION ) {
00247         MesPrint("&%s is not a valid expression.",inp);
00248         return(1);
00249     }
00250     if ( Expressions[numexpr].status != SPECTATOREXPRESSION ) {
00251         MesPrint("&%s is not a spectator.",inp);
00252         return(1);
00253     }
00254     for ( i = 0; i < AM.SizeForSpectatorFiles; i++ ) {
00255         if ( AM.SpectatorFiles[i].name != 0 ) {
00256             if ( StrCmp((UBYTE *)(AM.SpectatorFiles[i].name),(UBYTE *)(inp)) == 0 ) {
00257                 break;
00258             }
00259         }
```

```
00260     }
00261     if ( i >= AM.SizeForSpectatorFiles ) {
00262         MesPrint("&Spectator %s not found.",inp);
00263         return(1);
00264     }
00265     sp = AM.SpectatorFiles+i;
00266     Expressions[numexpr].status = DROPSPECTATOREXPRESSION;
00267     if ( sp->fh->handle != -1 ) {
00268         CloseFile(sp->fh->handle);
00269         sp->fh->handle = -1;
00270         remove(sp->fh->name);
00271     }
00272     M_free(sp->fh,"Temporary FileHandle");
00273     M_free(sp->name,"Spectator expression name");
00274
00275     PUTZERO(sp->position);
00276     PUTZERO(sp->readpos);
00277     sp->fh = 0;
00278     sp->name = 0;
00279     sp->exprnumber = -1;
00280     sp->flags = 0;
00281     AM.NumSpectatorFiles--;
00282     return(0);
00283 Syntax:
00284     MesPrint("&Proper syntax is: RemoveSpectator,exprname;");
00285     return(-1);
00286 }
00287
00288 /*
00289     #] CoRemoveSpectator :
00290     #[ CoEmptySpectator :
00291 */
00292
00293 int CoEmptySpectator(UBYTE *inp)
00294 {
00295     UBYTE *q;
00296     WORD c1, numexpr;
00297     int i;
00298     SPECTATOR *sp;
00299     while ( *inp == ',' ) inp++;
00300     if ( ( q = SkipAName(inp) ) == 0 || q[-1] == '_' ) {
00301         MesPrint("&Illegal name for expression");
00302         return(1);
00303     }
00304     if ( *q != 0 ) goto Syntax;
00305     if ( GetVar(inp,&c1,&numexpr,ALLVARIABLES,NOAUTO) == NAMENOTFOUND ||
00306             c1 != CEXPRESSION ) {
00307         MesPrint("&%s is not a valid expression.",inp);
00308         return(1);
00309     }
00310     if ( Expressions[numexpr].status != SPECTATOREXPRESSION ) {
00311         MesPrint("&%s is not a spectator.",inp);
00312         return(1);
00313     }
00314     for ( i = 0; i < AM.SizeForSpectatorFiles; i++ ) {
00315         if ( StrCmp((UBYTE *)(AM.SpectatorFiles[i].name),(UBYTE *)(inp)) == 0 ) break;
00316     }
00317     if ( i >= AM.SizeForSpectatorFiles ) {
00318         MesPrint("&Spectator %s not found.",inp);
00319         return(1);
00320     }
00321     sp = AM.SpectatorFiles+i;
00322     if ( sp->fh->handle != -1 ) {
00323         CloseFile(sp->fh->handle);
00324         sp->fh->handle = -1;
00325         remove(sp->fh->name);
00326     }
00327     sp->fh->POfill = sp->fh->POfull = sp->fh->PObuffer;
00328     PUTZERO(sp->position);
00329     PUTZERO(sp->readpos);
00330     return(0);
00331 Syntax:
00332     MesPrint("&Proper syntax is: EmptySpectator,exprname;");
00333     return(-1);
00334 }
00335
00336 /*
00337     #] CoEmptySpectator :
00338     #[ PutInSpectator :
00339
00340     We need to use locks! There is only one file.
00341     The code was copied (and modified) from PutOut.
00342     Here we use no compression.
00343 */
00344
00345 int PutInSpectator(WORD *term,WORD specnum)
00346 {
```

```
00347     GETBIDENTITY
00348     WORD i, *p, ret;
00349     LONG RetCode;
00350     SPECTATOR *sp = &(AM.SpectatorFiles[specnum]);
00351     FILEHANDLE *fi = sp->fh;
00352
00353     if ( ( i = *term ) <= 0 ) return(0);
00354     LOCK(fi->pthreadslock);
00355     ret = i;
00356     p = fi->POfill;
00357     do {
00358         if ( p >= fi->POstop ) {
00359             if ( fi->handle < 0 ) {
00360                 if ( ( RetCode = CreateFile(fi->name) ) >= 0 ) {
00361                     fi->handle = (WORD)RetCode;
00362                     PUTZERO(fi->filesize);
00363                     PUTZERO(fi->POposition);
00364                 }
00365                 else {
00366 /* INTERNAL_ERROR_EXCL_START */
00367                     MLOCK(ErrorMessageLock);
00368                     MesPrint("!>Cannot create spectator file %s",fi->name);
00369                     MUNLOCK(ErrorMessageLock);
00370                     UNLOCK(fi->pthreadslock);
00371                     return(-1);
00372 /* INTERNAL_ERROR_EXCL_STOP */
00373                 }
00374             }
00375             SeekFile(fi->handle,&(sp->position),SEEK_SET);
00376             if ( ( RetCode = WriteFile(fi->handle,(UBYTE *)(fi->PObuffer),fi->POsize) ) != fi->POsize
      ) {
00377                 MLOCK(ErrorMessageLock);
00378                 MesPrint("Error during spectator write. Disk full?");
00379                 MesPrint("Attempt to write %l bytes on file %d at position %15p",
00380                             fi->POsize,fi->handle,&(fi->POposition));
00381                 MesPrint("RetCode = %l, Buffer address = %l",RetCode,(LONG)(fi->PObuffer));
00382                 MUNLOCK(ErrorMessageLock);
00383                 UNLOCK(fi->pthreadslock);
00384                 return(-1);
00385             }
00386             ADDPOS(fi->filesize,fi->POsize);
00387             p = fi->PObuffer;
00388             ADDPOS(sp->position,fi->POsize);
00389             fi->POposition = sp->position;
00390         }
00391         *p++ = *term++;
00392     } while ( --i > 0 );
00393     fi->POfull = fi->POfill = p;
00394     Expressions[AM.SpectatorFiles[specnum].exprnumber].counter++;
00395     UNLOCK(fi->pthreadslock);
00396     return(ret);
00397 }
00398
00399 /*
00400     #] PutInSpectator :
00401     #[ FlushSpectators :
00402 */
00403
00404 void FlushSpectators(void)
00405 {
00406     SPECTATOR *sp = AM.SpectatorFiles;
00407     FILEHANDLE *fh;
00408     LONG RetCode;
00409     int i;
00410     LONG size;
00411     if ( AM.NumSpectatorFiles <= 0 ) return;
00412     for ( i = 0; i < AM.SizeForSpectatorFiles; i++, sp++ ) {
00413         if ( sp->name == 0 ) continue;
00414         fh = sp->fh;
00415         if ( ( sp->flags & READSPECTATORFLAG ) != 0 ) { /* reset for writing */
00416             sp->flags &= ~READSPECTATORFLAG;
00417             fh->POfill = fh->PObuffer;
00418             if ( fh->handle >= 0 ) {
00419                 SeekFile(fh->handle,&(sp->position),SEEK_SET);
00420                 fh->POposition = sp->position;
00421             }
00422             continue;
00423         }
00424         if ( fh->POfill <= fh->PObuffer ) continue; /* is clean */
00425         if ( fh->handle < 0 ) {    /* File needs to be created */
00426             if ( ( RetCode = CreateFile(fh->name) ) >= 0 ) {
00427                 PUTZERO(fh->filesize);
00428                 PUTZERO(fh->POposition);
00429                 fh->handle = (WORD)RetCode;
00430             }
00431             else {
00432 /* INTERNAL_ERROR_EXCL_START */
```

```
00433                 MLOCK(ErrorMessageLock);
00434                 MesPrint("!>Cannot create spectator file %s",fh->name);
00435                 MUNLOCK(ErrorMessageLock);
00436                 Terminate(-1);
00437 /* INTERNAL_ERROR_EXCL_STOP */
00438             }
00439             PUTZERO(sp->position);
00440         }
00441         SeekFile(fh->handle,&(sp->position),SEEK_SET);
00442         size = (fh->POfill - fh->POobuffer)*sizeof(WORD);
00443         if ( ( RetCode = WriteFile(fh->handle,(UBYTE *)(fh->POobuffer),size) ) != size ) {
00444             MLOCK(ErrorMessageLock);
00445             MesPrint("Write error synching spectator file. Disk full?");
00446             MesPrint("Attempt to write %l bytes on file %s at position %15p",
00447                         size,fh->name,&(sp->position));
00448             MUNLOCK(ErrorMessageLock);
00449             Terminate(-1);
00450         }
00451         fh->POfill = fh->POobuffer;
00452         SeekFile(fh->handle,&(sp->position),SEEK_END);
00453         fh->POposition = sp->position;
00454     }
00455     return;
00456 }
00457
00458 /*
00459     #] FlushSpectators :
00460     #[ CoCopySpectator :
00461 */
00462
00463 int CoCopySpectator(UBYTE *inp)
00464 {
00465     GETIDENTITY
00466     UBYTE *q, c, *exprname, *p;
00467     WORD c1, c2, numexpr;
00468     int specnum, error = 0;
00469     SPECTATOR *sp;
00470     while ( *inp == ',' ) inp++;
00471     if ( ( q = SkipAName(inp) ) == 0 || q[-1] == '_' ) {
00472         MesPrint("&Illegal name for expression");
00473         return(1);
00474     }
00475     if ( *q == 0 ) goto Syntax;
00476     c = *q; *q = 0;
00477     if ( GetVar(inp,&c1,&c2,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
00478         MesPrint("&%s is the name of an existing variable.",inp);
00479         return(1);
00480     }
00481     numexpr = EntVar(CEXPRESSION,inp,LOCALEXPRESSION,0,0,0);
00482     p = q;
00483     exprname = inp;
00484     *q = c;
00485     while ( *q == ' ' || *q == ',' || *q == '\t' ) q++;
00486     if ( *q != '=' ) goto Syntax;
00487     q++;
00488     while ( *q == ' ' || *q == ',' || *q == '\t' ) q++;
00489     inp = q;
00490     if ( ( q = SkipAName(inp) ) == 0 || q[-1] == '_' ) {
00491         MesPrint("&Illegal name for spectator expression");
00492         return(1);
00493     }
00494     if ( *q != 0 ) goto Syntax;
00495     if ( AM.NumSpectatorFiles <= 0 ) {
00496         MesPrint("&CopySpectator: There are no spectator expressions!");
00497         return(1);
00498     }
00499     sp = AM.SpectatorFiles;
00500     for ( specnum = 0; specnum < AM.SizeForSpectatorFiles; specnum++, sp++ ) {
00501       if ( sp->name != 0 ) {
00502         if ( StrCmp((UBYTE *)(sp->name),(UBYTE *)(inp)) == 0 ) break;
00503       }
00504     }
00505     if ( specnum >= AM.SizeForSpectatorFiles ) {
00506         MesPrint("&Spectator %s not found.",inp);
00507         return(1);
00508     }
00509     sp->flags |= READSPECTATORFLAG;
00510     PUTZERO(sp->fh->POposition);
00511     PUTZERO(sp->readpos);
00512     sp->fh->POfill = sp->fh->POobuffer;
00513     if ( sp->fh->handle >= 0 ) {
00514         SeekFile(sp->fh->handle,&(sp->fh->POposition),SEEK_SET);
00515     }
00516 /*
00517     Now we have:
00518     1: The name of the target expression: numexpr
00519     2: The spectator: sp (or specnum).
```

```
00520     Time for some action. We need:
00521     a: Write a prototype to create the expression
00522     b: Signal to Processor that this is a spectator.
00523        We do this by giving a negative compiler buffer number.
00524 */
00525     {
00526         WORD *OldWork, *w;
00527         POSITION pos;
00528         OldWork = w = AT.WorkPointer;
00529         *w++ = TYPEEXPRESSION;
00530         *w++ = 3+SUBEXPSIZE;
00531         *w++ = numexpr;
00532         AC.ProtoType = w;
00533         AR.CurExpr = numexpr;               /* Block expression numexpr */
00534         *w++ = SUBEXPRESSION;
00535         *w++ = SUBEXPSIZE;
00536         *w++ = numexpr;
00537         *w++ = 1;
00538         *w++ = -specnum-1;      /* Indicates "spectator" to Processor */
00539         FILLSUB(w)
00540         *w++ = 1;
00541         *w++ = 1;
00542         *w++ = 3;
00543         *w++ = 0;
00544         SeekScratch(AR.outfile,&pos);
00545         Expressions[numexpr].counter = 1;
00546         Expressions[numexpr].onfile = pos;
00547         Expressions[numexpr].whichbuffer = 0;
00548 #ifdef PARALLELCODE
00549         Expressions[numexpr].partodo = AC.inparallelflag;
00550 #endif
00551         OldWork[2] = w - OldWork - 3;
00552         AT.WorkPointer = w;
00553
00554         if ( PutOut(BHEAD OldWork+2,&pos,AR.outfile,0) < 0 ) {
00555             c = *p; *p = 0;
00556             MesPrint("&Cannot create expression %s",exprname);
00557             *p = c;
00558             error = -1;
00559         }
00560         else {
00561             OldWork[2] = 4+SUBEXPSIZE;
00562             OldWork[4] = SUBEXPSIZE;
00563             OldWork[5] = numexpr;
00564             OldWork[SUBEXPSIZE+3] = 1;
00565             OldWork[SUBEXPSIZE+4] = 1;
00566             OldWork[SUBEXPSIZE+5] = 3;
00567             OldWork[SUBEXPSIZE+6] = 0;
00568             if ( PutOut(BHEAD OldWork+2,&pos,AR.outfile,0) < 0
00569             || FlushOut(&pos,AR.outfile,0) ) {
00570                 c = *p; *p = 0;
00571                 MesPrint("&Cannot create expression %s",exprname);
00572                 *p = c;
00573                 error = -1;
00574             }
00575             AR.outfile->POfull = AR.outfile->POfill;
00576         }
00577         OldWork[2] = numexpr;
00578 /*
00579         Seems unnecessary (13-feb-2018)
00580
00581         AddNtoL(OldWork[1],OldWork);
00582 */
00583         AT.WorkPointer = OldWork;
00584         if ( AC.dumnumflag ) Add2Com(TYPEDETCURDUM)
00585     }
00586 #ifdef WITHMPI
00587     /*
00588      * In ParFORM, substitutions of spectators has to be done on the master.
00589      */
00590     AC.mparallelflag |= NOPARALLEL_SPECTATOR;
00591 #endif
00592     return(error);
00593 Syntax:
00594     MesPrint("&Proper syntax is: CopySpectator,exprname=spectatorname;");
00595     return(-1);
00596 }
00597
00598 /*
00599     #] CoCopySpectator :
00600     #[ GetFromSpectator :
00601
00602     Note that if we did things right, we do not need a lock for the reading.
00603 */
00604
00605 WORD GetFromSpectator(WORD *term,WORD specnum)
00606 {
```

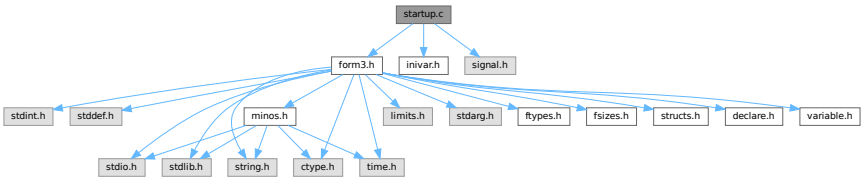
```
00607     SPECTATOR *sp = &(AM.SpectatorFiles[specnum]);
00608     FILEHANDLE *fh = sp->fh;
00609     WORD i, size, *t = term;
00610     LONG InIn;
00611     if ( fh-> handle < 0 ) {
00612         *term = 0;
00613         return(0);
00614     }
00615 /*
00616     sp->position marks the 'end' of the file: the point where writing should
00617     take place. sp->readpos marks from where to read.
00618     fh->POposition marks where the file is currently positioned.
00619     Note that when we read, we need to
00620 */
00621     if ( ISZEROPOS(sp->readpos) ) { /* we start reading. Fill buffer. */
00622 FillBuffer:
00623         SeekFile(fh->handle,&(sp->readpos),SEEK_SET);
00624         InIn = ReadFile(fh->handle,(UBYTE *)(fh->PObuffer),fh->POsize);
00625         if ( InIn < 0 || ( InIn & 1 ) ) {
00626 /* INTERNAL_ERROR_EXCL_START */
00627             MLOCK(ErrorMessageLock);
00628             MesPrint("!>Error reading information for %s spectator",sp->name);
00629             MUNLOCK(ErrorMessageLock);
00630             Terminate(-1);
00631 /* INTERNAL_ERROR_EXCL_STOP */
00632         }
00633         InIn /= sizeof(WORD);
00634         if ( InIn == 0 ) { *term = 0; return(0); }
00635         SeekFile(fh->handle,&(sp->readpos),SEEK_CUR);
00636         fh->POposition = sp->readpos;
00637         fh->POfull = fh->PObuffer+InIn;
00638         fh->POfill = fh->PObuffer;
00639     }
00640     if ( fh->POfill == fh->POfull ) { /* not even the size of the term! */
00641         if ( ISLESSPOS(sp->readpos,sp->position) ) goto FillBuffer;
00642         *term = 0;
00643         return(0);
00644     }
00645     size = *fh->POfill++; *t++ = size;
00646     for ( i = 1; i < size; i++ ) {
00647         if ( fh->POfill >= fh->POfull ) {
00648             SeekFile(fh->handle,&(sp->readpos),SEEK_SET);
00649             InIn = ReadFile(fh->handle,(UBYTE *)(fh->PObuffer),fh->POsize);
00650             if ( InIn < 0 || ( InIn & 1 ) ) {
00651 /* INTERNAL_ERROR_EXCL_START */
00652                 MLOCK(ErrorMessageLock);
00653                 MesPrint("!>Error reading information for %s spectator",sp->name);
00654                 MUNLOCK(ErrorMessageLock);
00655                 Terminate(-1);
00656 /* INTERNAL_ERROR_EXCL_STOP */
00657             }
00658             InIn /= sizeof(WORD);
00659             if ( InIn == 0 ) {
00660 /* INTERNAL_ERROR_EXCL_START */
00661                 MLOCK(ErrorMessageLock);
00662                 MesPrint("!>Reading incomplete information for %s spectator",sp->name);
00663                 MUNLOCK(ErrorMessageLock);
00664                 Terminate(-1);
00665 /* INTERNAL_ERROR_EXCL_STOP */
00666             }
00667             SeekFile(fh->handle,&(sp->readpos),SEEK_CUR);
00668             fh->POposition = sp->readpos;
00669             fh->POfull = fh->PObuffer+InIn;
00670             fh->POfill = fh->PObuffer;
00671         }
00672         *t++ = *fh->POfill++;
00673     }
00674     return(size);
00675 }
00676
00677 /*
00678     #] GetFromSpectator :
00679     #[ ClearSpectators :
00680
00681     Removes all spectators.
00682     In case of .store, the ones that are protected by .global stay.
00683 */
00684
00685 void ClearSpectators(WORD par)
00686 {
00687     SPECTATOR *sp = AM.SpectatorFiles;
00688     WORD numexpr, c1;
00689     int i;
00690     if ( AM.NumSpectatorFiles > 0 ) {
00691         for ( i = 0; i < AM.SizeForSpectatorFiles; i++, sp++ ) {
00692             if ( sp->name == 0 ) continue;
00693             if ( ( sp->flags & GLOBALSPECTATORFLAG ) == 1 && par == STOREMODULE ) continue;
```

```
00694
00695             if ( GetVar((UBYTE *)(sp->name),&c1,&numexpr,ALLVARIABLES,NOAUTO) == NAMENOTFOUND ||
00696                 c1 != CEXPRESSION ) {
00697                 MesPrint("&%s is not a valid expression.",sp->name);
00698                 continue;
00699             }
00700             Expressions[numexpr].status = DROPPEDEXPRESSION;
00701             if ( sp->fh->handle != -1 ) {
00702                 CloseFile(sp->fh->handle);
00703                 sp->fh->handle = -1;
00704                 remove(sp->fh->name);
00705             }
00706             M_free(sp->fh,"Temporary FileHandle");
00707             M_free(sp->name,"Spectator expression name");
00708             PUTZERO(sp->position);
00709             sp->fh = 0;
00710             sp->name = 0;
00711             sp->exprnumber = -1;
00712             sp->flags = 0;
00713             AM.NumSpectatorFiles--;
00714         }
00715     }
00716 }
00717
00718 /*
00719     #] ClearSpectators :
00720 */
```

4.135 startup.c File Reference

```
#include "form3.h"
#include "inivar.h"
#include <signal.h>
```

Include dependency graph for startup.c:



Macros

- #define STRINGIFY(x) STRINGIFY__(x)
- #define STRINGIFY__(x) #x
- #define FORMNAME "FORM"
- #define VERSIONSTR__ STRINGIFY(MAJORVERSION) "." STRINGIFY(MINORVERSION) "." STRINGIFY(PATCHVERSION)
- #define VERSIONSTR FORMNAME " " VERSIONSTR__ " (" PRODUCTIONDATE ")"
- #define TAKEPATH(x) if(s[1]== '='){x=s+2;} else{x=*argv++;argc--;}
- #define DEFAULTFNAMELENGTH 16
- #define **STR2**(x) #x
- #define **STR**(x) STR2(x)

Functions

- int DoTail (int argc, UBYTE **argv)
- int OpenInput (void)
- void ReserveTempFiles (int par)
- void StartVariables (void)
- void StartMore (void)
- void IniVars (void)
- int main (int argc, char **argv)
- void CleanUp (WORD par)
- void TerminateImpl (int errorcode, const char *file, int line, const char *function)
- void PrintDeprecation (const char *feature, const char *issue)
- void PrintRunningTime (void)
- LONG GetRunningTime (void)

Variables

- UBYTE * emptystring = (UBYTE *)"."
- UBYTE * defaulttempfilename = (UBYTE *)"xformxxx.str"

4.135.1 Detailed Description

This file contains the main program. It also deals with the very early stages of the startup of FORM and the final stages when the program attempts some cleanup. Here is the routine that analyses the command tail.

Definition in file startup.c.

4.135.2 Macro Definition Documentation

4.135.2.1 STRINGIFY

```
#define STRINGIFY(
              x ) STRINGIFY__(x)
```

Definition at line 57 of file startup.c.

4.135.2.2 STRINGIFY__

```
#define STRINGIFY__(
              x ) #x
```

Definition at line 58 of file startup.c.

4.135.2.3 FORMNAME

```
#define FORMNAME "FORM"
```

Definition at line 68 of file startup.c.

4.135.2.4 VERSIONSTR__

```
#define VERSIONSTR__ STRINGIFY(MAJORVERSION) "." STRINGIFY(MINORVERSION) "." STRINGIFY(PATCHVERSION)
```

Definition at line 100 of file startup.c.

4.135.2.5 VERSIONSTR

```
#define VERSIONSTR FORMNAME " " VERSIONSTR__ " (" PRODUCTIONDATE ")"
```

Definition at line 102 of file startup.c.

4.135.2.6 TAKEPATH

```
#define TAKEPATH(
              x ) if(s[1]== '=' ){x=s+2;} else{x=*argv++;argc--;}
```

Definition at line 280 of file startup.c.

4.135.2.7 DEFAULTFNAMELENGTH

```
#define DEFAULTFNAMELENGTH 16
```

Definition at line 711 of file startup.c.

4.135.3 Function Documentation

4.135.3.1 DoTail()

```
int DoTail (
             int argc,
             UBYTE ** argv )
```

Definition at line 282 of file startup.c.

4.135.3.2 OpenInput()

```
int OpenInput (
             void  )
```

Definition at line 589 of file startup.c.

4.135.3.3 ReserveTempFiles()

```
void ReserveTempFiles (
             int par )
```

Definition at line 713 of file startup.c.

4.135.3.4 StartVariables()

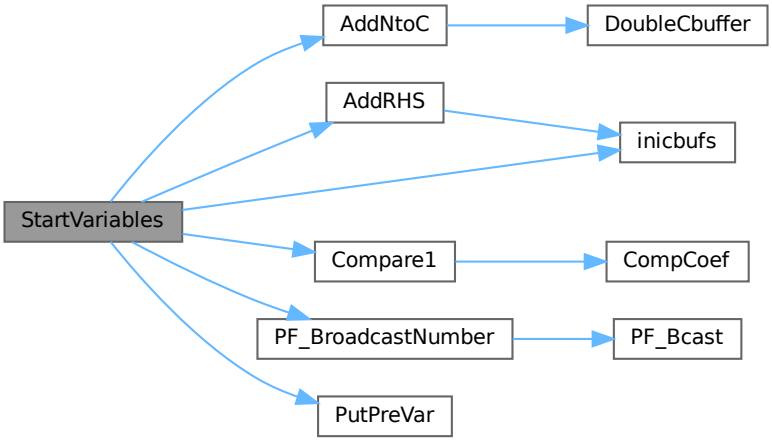
```
void StartVariables (
            void  )
```

All functions (well, nearly all) are declared here.

Definition at line 952 of file startup.c.

References AddNtoC(), AddRHS(), Compare1(), inicbufs(), FuNcTiOn::name, PF_BroadcastNumber(), PutPreVar(), and FuNcTiOn::symmetric.

Here is the call graph for this function:



4.135.3.5 StartMore()

```
void StartMore (
            void  )
```

Definition at line 1387 of file startup.c.

4.135.3.6 IniVars()

```
void IniVars (
            void  )
```

Definition at line 1425 of file startup.c.

4.135.3.7 main()

```
int main (
            int argc,
            char ** argv )
```

Definition at line 1732 of file startup.c.

4.135.3.8 CleanUp()

```
void CleanUp (
             WORD par )
```

Definition at line 1834 of file startup.c.

4.135.3.9 TerminateImpl()

```
void TerminateImpl (
             int errorcode,
             const char * file,
             int line,
             const char * function )
```

Definition at line 1923 of file startup.c.

4.135.3.10 PrintDeprecation()

```
void PrintDeprecation (
             const char * feature,
             const char * issue )
```

Prints a deprecation warning for a given feature.

Parameters

feature	The name of the deprecated feature.
issue	The associated issue, e.g., "issues/700".

Definition at line 2143 of file startup.c.

4.135.3.11 PrintRunningTime()

```
void PrintRunningTime (
             void  )
```

Definition at line 2167 of file startup.c.

4.135.3.12 GetRunningTime()

```
LONG GetRunningTime (
             void  )
```

Definition at line 2208 of file startup.c.

4.135.4 Variable Documentation

4.135.4.1 emptystring

```
UBYTE* emptystring = (UBYTE *)"."
```

Definition at line 707 of file startup.c.

4.135.4.2 defaulttempfilename

```
UBYTE* defaulttempfilename = (UBYTE *)"xformxxx.str"
```

Definition at line 708 of file startup.c.

4.136 startup.c

Go to the documentation of this file.

```
00001
00008 /* #[ License : */
00009 /*
00010  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00011  *   When using this file you are requested to refer to the publication
00012  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013  *   This is considered a matter of courtesy as the development was paid
00014  *   for by FOM the Dutch physics granting agency and we would like to
00015  *   be able to track its scientific use to convince FOM of its value
00016  *   for the community.
00017  *
00018  *   This file is part of FORM.
00019  *
00020  *   FORM is free software: you can redistribute it and/or modify it under the
00021  *   terms of the GNU General Public License as published by the Free Software
00022  *   Foundation, either version 3 of the License, or (at your option) any later
00023  *   version.
00024  *
00025  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00027  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00028  *   details.
00029  *
00030  *   You should have received a copy of the GNU General Public License along
00031  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00032  */
00033 /* #] License : */
00034 /*
00035         #[ includes :
00036 */
00037
00038 #include "form3.h"
00039 #include "inivar.h"
00040
00041 #ifdef TRAPSIGNALS
00042 #include "portsignals.h"
00043 #else
00044 #include <signal.h>
00045 #endif
00046 #ifdef ENABLE_BACKTRACE
00047     #include <execinfo.h>
00048 #ifdef LINUX
00049     #include <stdint.h>
00050     #include <inttypes.h>
00051 #endif
00052 #endif
00053
00054 /*
00055  * A macro for translating the contents of `x' into a string after expanding.
00056  */
00057 #define STRINGIFY(x)  STRINGIFY__(x)
00058 #define STRINGIFY__(x) #x
00059
00060 /*
```

```
00061  * FORMNAME = "FORM" or "TFORM" or "ParFORM".
00062  */
00063 #if defined(WITHPTHREADS)
00064     #define FORMNAME "TFORM"
00065 #elif defined(WITHMPI)
00066     #define FORMNAME "ParFORM"
00067 #else
00068     #define FORMNAME "FORM"
00069 #endif
00070
00071 /*
00072  * VERSIONSTR is the version information printed in the header line.
00073  */
00074 #ifdef HAVE_CONFIG_H
00075     /* We have also version.h. */
00076     #include "version.h"
00077     #ifndef REPO_VERSION
00078         #define REPO_VERSION STRINGIFY(REPO_MAJOR_VERSION) "." STRINGIFY(REPO_MINOR_VERSION) "."
      STRINGIFY(REPO_PATCH_VERSION)
00079     #endif
00080     #ifndef REPO_DATE
00081         /* The build date, instead of the repo date. */
00082         #define REPO_DATE __DATE__
00083     #endif
00084     #ifdef REPO_REVISION
00085         #define VERSIONSTR FORMNAME " " REPO_VERSION " (" REPO_DATE ", " REPO_REVISION ")"
00086     #else
00087         #define VERSIONSTR FORMNAME " " REPO_VERSION " (" REPO_DATE ")"
00088     #endif
00089     #define MAJORVERSION REPO_MAJOR_VERSION
00090     #define MINORVERSION REPO_MINOR_VERSION
00091     #define PATCHVERSION REPO_PATCH_VERSION
00092 #else
00093     /*
00094      * Otherwise, form3.h defines MAJORVERSION, MINORVERSION, PATCHVERSION
00095      * and PRODUCTIONDATE, possibly BETAVERSION.
00096      */
00097     #ifdef BETAVERSION
00098         #define VERSIONSTR__ STRINGIFY(MAJORVERSION) "." STRINGIFY(MINORVERSION) "."
      STRINGIFY(PATCHVERSION) "Beta"
00099     #else
00100         #define VERSIONSTR__ STRINGIFY(MAJORVERSION) "." STRINGIFY(MINORVERSION) "."
      STRINGIFY(PATCHVERSION)
00101     #endif
00102     #define VERSIONSTR FORMNAME " " VERSIONSTR__ " (" PRODUCTIONDATE ")"
00103 #endif
00104
00105 /*
00106         #] includes :
00107         #[ PrintHeader :
00108 */
00109
00116 static void PrintBuildInfo(void) {
00117 #if defined(__INTEL_LLVM_COMPILER)
00118     MesPrint("Compiler: Intel LLVM oneAPI %d",__INTEL_LLVM_COMPILER);
00119 #elif defined(__INTEL_COMPILER)
00120     MesPrint("Compiler: Intel Classic %d",__INTEL_COMPILER);
00121 #elif defined(__clang__) && defined(__apple_build_version__)
00122     MesPrint("Compiler: Apple Clang %d.%d.%d (build
      %d)",__clang_major__,__clang_minor__,__clang_patchlevel__,__apple_build_version__);
00123 #elif defined(__clang__)
00124     MesPrint("Compiler: Clang %d.%d.%d",__clang_major__,__clang_minor__,__clang_patchlevel__);
00125 #elif defined(__GNUC__)
00126     MesPrint("Compiler: GCC %d.%d.%d",__GNUC__,__GNUC_MINOR__,__GNUC_PATCHLEVEL__);
00127 #elif defined(_MSC_VER)
00128     MesPrint("Compiler: MSVC %d",_MSC_VER);
00129 #else
00130     MesPrint("Compiler: Unknown");
00131 #endif
00132
00133 #if defined(__x86_64__) || defined(_M_X64)
00134     MesPrint("Architecture: x86_64");
00135 #elif defined(__i386__) || defined(_M_IX86)
00136     MesPrint("Architecture: x86 (32-bit)");
00137 #elif defined(__aarch64__) || defined(_M_ARM64)
00138     MesPrint("Architecture: arm64");
00139 #elif defined(__arm__) || defined(_M_ARM)
00140     MesPrint("Architecture: arm (32-bit)");
00141 #else
00142     MesPrint("Architecture: Unknown");
00143 #endif
00144 }
00145
00154 static void PrintHeader(int par)
00155 {
00156 #ifdef WITHMPI
00157     if ( PF.me == MASTER && !AM.silent ) {
```

```
00158 #else
00159     if ( !AM.silent ) {
00160 #endif
00161         char buffer1[250], buffer2[80], *s = buffer1, *t = buffer2;
00162         WORD length, n;
00163         for ( n = 0; n < 250; n++ ) buffer1[n] = ' ';
00164         /*
00165          * NOTE: we expect that the compiler optimizes strlen("string literal")
00166          * to just a number.
00167          */
00168         if ( strlen(VERSIONSTR) <= 100 ) {
00169             strcpy(s,VERSIONSTR);
00170             s += strlen(VERSIONSTR);
00171             *s = 0;
00172         }
00173         else {
00174             /*
00175              * Truncate when it is too long.
00176              */
00177             strncpy(s,VERSIONSTR,97);
00178             s[97] = '.';
00179             s[98] = '.';
00180             s[99] = ')';
00181             s[100] = '\0';
00182             s += 100;
00183         }
00184 /*
00185         By now we omit the message about 32/64 bits. It should all be 64.
00186
00187         s += snprintf(s,250-(s-buffer1)," %d-bits",(WORD)(sizeof(WORD)*16));
00188         *s = 0;
00189 */
00190         if ( par == 1 ) {
00191 #if defined(WITHPTHREADS) || defined(WITHMPI)
00192 #if defined(WITHPTHREADS)
00193             int nworkers = AM.totalnumberofthreads-1;
00194 #elif defined(WITHMPI)
00195             int nworkers = PF.numtasks-1;
00196 #endif
00197             s += snprintf(s,250-(s-buffer1)," %d worker",nworkers);
00198             *s = 0;
00199 /*          while ( *s ) s++; */
00200             if ( nworkers != 1 ) {
00201                 *s++ = 's';
00202                 *s = '\0';
00203             }
00204 #endif
00205
00206             snprintf(t,80-(t-buffer2),"Run: %s",MakeDate());
00207             while ( *t ) t++;
00208
00209             /*
00210              * Align the date to the right, if it fits in a line.
00211              */
00212             length = (s-buffer1) + (t-buffer2);
00213             if ( length+2 <= AC.LineLength ) {
00214                 for ( n = AC.LineLength-length; n > 0; n-- ) *s++ = ' ';
00215                 *s = 0;
00216                 strcat(s,buffer2);
00217                 while ( *s ) s++;
00218             }
00219             else {
00220                 *s = 0;
00221                 strcat(s,"  ");
00222                 while ( *s ) s++;
00223                 *s = 0;
00224                 strcat(s,buffer2);
00225                 while ( *s ) s++;
00226             }
00227         }
00228
00229         /*
00230          * If the header information doesn't fit in a line, we need to extend
00231          * the line length temporarily.
00232          */
00233         length = s-buffer1;
00234         if ( length <= AC.LineLength ) {
00235             MesPrint("%s",buffer1);
00236         }
00237         else {
00238             WORD oldLineLength = AC.LineLength;
00239             AC.LineLength = length;
00240             MesPrint("%s",buffer1);
00241             AC.LineLength = oldLineLength;
00242         }
00243
00244         if ( par == 2 ) {
```

```
00245             PrintFeatureList();
00246             PrintBuildInfo();
00247         }
00248     }
00249 #ifdef WINDOWS
00250     PrintDeprecation("the native Windows version", "issues/623");
00251 #endif
00252 #if BITSINWORD == 16
00253     PrintDeprecation("the 32-bit version", "issues/624");
00254 #endif
00255 #ifdef WITHMPI
00256     PrintDeprecation("the MPI version (ParFORM)", "issues/625");
00257 #endif
00258     if ( AC.CheckpointFlag ) {
00259         PrintDeprecation("the checkpoint mechanism", "issues/626");
00260     }
00261 }
00262
00263 /*
00264         #] PrintHeader :
00265         #[ DoTail :
00266
00267         Routine reads the command tail and handles the commandline options.
00268         It sets the flags for later actions and stored pathnames for
00269         the setup file, include/prc/sub directories etc.
00270         Finally the name of the program is passed on.
00271         Note that we do not support interactive use yet. This will come
00272         to pass in the distant future when we can couple STedi to FORM.
00273         Routine made 23-feb-1993 by J.Vermaseren
00274 */
00275
00276 #ifdef WITHINTERACTION
00277 static UBYTE deflogname[] = "formsession.log";
00278 #endif
00279
00280 #define TAKEPATH(x) if(s[1]== '=' ){x=s+2;} else{x=*argv++;argc--;}
00281
00282 int DoTail(int argc, UBYTE **argv)
00283 {
00284     int errorflag = 0, onlyversion = 1;
00285     UBYTE *s, *t, *copy;
00286     int threadnum = 0;
00287     argc--; argv++;
00288     AM.ClearStore = 0;
00289     AM.TimeLimit = 0;
00290     AM.LogType = -1;
00291     AM.HoldFlag = AM.qError = AM.Interact = AM.FileOnlyFlag = 0;
00292     AM.InputFileName = AM.LogFileName = AM.IncDir = AM.TempDir = AM.TempSortDir =
00293     AM.SetupDir = AM.SetupFile = AM.Path = NULL;
00294     AM.FromStdin = 0;
00295     /* Always use MultiRun, "-M" option is now ignored. */
00296     AM.MultiRun = 1;
00297     if ( argc < 1 ) {
00298         onlyversion = 0;
00299         goto printversion;
00300     }
00301     while ( argc >= 1 ) {
00302         s = *argv++; argc--;
00303         if ( *s == '-' || ( *s == '/' && ( argc > 0 || AM.Interact ) ) ) {
00304             s++;
00305             switch (*s) {
00306                 case 'c': /* Error checking only */
00307                           AM.qError = 1;   break;
00308                 case 'C': /* Next arg is filename of log file */
00309                           TAKEPATH(AM.LogFileName)  break;
00310                 case 'D':
00311                 case 'd': /* Next arg is define preprocessor var. */
00312                           t = copy = strDup1(*argv,"Dotail");
00313                           while ( *t && *t != '=' ) t++;
00314                           if ( *t == 0 ) {
00315                               if ( PutPreVar(copy,(UBYTE *)"1",0,0) < 0 ) return(-1);
00316                           }
00317                           else {
00318                               *t++ = 0;
00319                               if ( PutPreVar(copy,t,0,0) < 0 ) return(-1);
00320                               t[-1] = '=';
00321                           }
00322                           M_free(copy,"-d prevar");
00323                           argv++; argc--; break;
00324                 case 'f': /* Output only to regular log file */
00325                           AM.FileOnlyFlag = 1; AM.LogType = 0; break;
00326                 case 'F': /* Output only to log file. Further like L. */
00327                           AM.FileOnlyFlag = 1; AM.LogType = 1; break;
00328                 case 'h': /* For old systems: wait for key before exit */
00329                           AM.HoldFlag = 1; break;
00330                 case 'i':
00331                           if ( StrCmp(s, (UBYTE *)"ignore-deprecation") == 0 ) {
```

```
00332                                 AM.IgnoreDeprecation = 1;
00333                                 break;
00334                             }
00335 #ifdef WITHINTERACTION
00336                             /* Interactive session (not used yet) */
00337                             AM.Interact = 1;
00338                             break;
00339 #else
00340                             goto IllegalOption;
00341 #endif
00342                 case 'I': /* Next arg is dir for inc/prc/sub files */
00343                             TAKEPATH(AM.IncDir)  break;
00344                 case 'l': /* Make regular log file */
00345                             if ( s[1] == 'l' ) AM.LogType = 1; /*compatibility! */
00346                             else               AM.LogType = 0;
00347                             break;
00348                 case 'L': /* Make log file with only final statistics */
00349                             AM.LogType = 1;  break;
00350                 case 'M': /* Multirun. Name of tempfiles will contain PID */
00351                             /* This option is now ignored. We always use MultiRun. */
00352                             /* AM.MultiRun = 1; */
00353                             break;
00354                 case 'm': /* Read number of threads */
00355                 case 'w': /* Read number of workers */
00356                             t = s++;
00357                             threadnum = 0;
00358                             while ( *s >= '0' && *s <= '9' )
00359                                 threadnum = 10*threadnum + *s++ - '0';
00360                             if ( *s ) {
00361 #ifdef WITHMPI
00362                                 if ( PF.me == MASTER )
00363 #endif
00364                                 printf("Illegal value for option m or w: %s\n",t);
00365                                 errorflag++;
00366                             }
00367 /*                          if ( threadnum == 1 ) threadnum = 0; */
00368                             threadnum++;
00369                             break;
00370                 case 'W': /* Print the wall-clock time on the master. */
00371                             AM.ggWTimeStatsFlag = 1;
00372                             break;
00373 /*
00374                 case 'n':
00375                             Reserved for number of slaves without MPI
00376 */
00377                 case 'p':
00378 #ifdef WITHEXTERNALCHANNEL
00379                     /*There are two possibilities: -p|-pipe*/
00380                     if(s[1]=='i'){
00381                         if( (s[2]=='p')&&(s[3]=='e')&&(s[4]=='\0') ){
00382                             argc--;
00383                             /*Initialize pre-set external channels, see
00384                                 the file extcmd.c:*/
00385                             if(initPresetExternalChannels(*argv++,AX.timeout)<1){
00386 #ifdef WITHMPI
00387                                 if ( PF.me == MASTER )
00388 #endif
00389                                 printf("Error initializing preset external channels\n");
00390                                 errorflag++;
00391                             }
00392                             AX.timeout=-1;/*This indicates that preset channels
00393                                             are initialized from cmdline*/
00394                         }else{
00395 #ifdef WITHMPI
00396                             if ( PF.me == MASTER )
00397 #endif
00398                             printf("Illegal option in call of FORM: %s\n",s);
00399                             errorflag++;
00400                         }
00401                     }else
00402 #else
00403                     if ( s[1] ) {
00404                         if ( ( s[1]=='i' ) && ( s[2] == 'p' ) && (s[3] == 'e' )
00405                         && ( s[4] == '\0' ) ){
00406 #ifdef WITHMPI
00407                             if ( PF.me == MASTER )
00408 #endif
00409                             printf("Illegal option: Pipes not supported on this system.\n");
00410                         }
00411                         else {
00412 #ifdef WITHMPI
00413                             if ( PF.me == MASTER )
00414 #endif
00415                             printf("Illegal option: %s\n",s);
00416                         }
00417                         errorflag++;
00418                     }
```

```
00419                     else
00420 #endif
00421                     {
00422                              /* Next arg is a path variable like in environment */
00423                         TAKEPATH(AM.Path)
00424                     }
00425                     break;
00426                 case 'q': /* Quiet option. Only output. Same as -si */
00427                           AM.silent = 1; break;
00428                 case 'R': /* recover from saved snapshot */
00429                           AC.CheckpointFlag = -1;
00430                           break;
00431                 case 's': /* Next arg is dir with form.set to be used */
00432                           if ( ( s[1] == 'o' ) && ( s[2] == 'r' ) && ( s[3] == 't' ) ) {
00433                               if(s[4]== '=' ) {
00434                                   AM.TempSortDir = s+5;
00435                               }
00436                               else {
00437                                   AM.TempSortDir = *argv++;
00438                                   argc--;
00439                               }
00440                           }
00441                           else if ( s[1] == 'i' ) { /* compatibility: silent/quiet */
00442                               AM.silent = 1;
00443                           }
00444                           else {
00445                               TAKEPATH(AM.SetupDir)
00446                           }
00447                           break;
00448                 case 'S': /* Next arg is setup file */
00449                           TAKEPATH(AM.SetupFile) break;
00450                 case 't': /* Next arg is directory for temp files */
00451                           if ( s[1] == 's' ) {
00452                               s++;
00453                               AM.havesortdir = 1;
00454                               TAKEPATH(AM.TempSortDir)
00455                           }
00456                           else {
00457                               TAKEPATH(AM.TempDir)
00458                           }
00459                           break;
00460                 case 'T': /* Print the total size used at end of job */
00461                           AM.PrintTotalSize = 1; break;
00462                 case 'v': /* Print version information */
00463                           AC.FinalStats = 0;
00464                           if ( s[1] == 'v' ) {  /* verbose version information */
00465                               PrintHeader(2);
00466                               return(1);
00467                           }
00468 printversion:;
00469                           PrintHeader(0);
00470                           if ( onlyversion ) return(1);
00471                           goto NoFile;
00472                 case 'y': /* Preprocessor dumps output. No compilation. */
00473                           AP.PreDebug = PREPROONLY;   break;
00474                 case 'z': /* The number following is a time limit in sec. */
00475                           t = s++;
00476                           AM.TimeLimit = 0;
00477                           while ( *s >= '0' && *s <= '9' )
00478                               AM.TimeLimit = 10*AM.TimeLimit + *s++ - '0';
00479                           break;
00480                 case 'Z': /* Removes the .str file on crash, no matter its contents */
00481                           AM.ClearStore = 1;   break;
00482                 case '\0': /* "-" to use STDIN for the input. */
00483 #ifdef WITHMPI
00484                           /* At the moment, ParFORM doesn't implement STDIN broadcasts. */
00485                           if ( PF.me == MASTER )
00486                               printf("Sorry, reading STDIN as input is currently not supported by
      ParFORM\n");
00487                           errorflag++;
00488 #endif
00489                           AM.FromStdin = 1;
00490                           AC.NoShowInput = 1; // disable input echoing by default
00491                           break;
00492                 default:
00493                         if ( FG.cTable[*s] == 1 ) {
00494                             AM.SkipClears = 0; t = s;
00495                             while ( FG.cTable[*t] == 1 )
00496                                 AM.SkipClears = 10*AM.SkipClears + *t++ - '0';
00497                             if ( *t != 0 ) {
00498 #ifdef WITHMPI
00499                                 if ( PF.me == MASTER )
00500 #endif
00501                                 printf("Illegal numerical option in call of FORM: %s\n",s);
00502                                 errorflag++;
00503                             }
00504                         }
```

```
00505                           else {
00506 IllegalOption:
00507 #ifdef WITHMPI
00508                               if ( PF.me == MASTER )
00509 #endif
00510                               printf("Illegal option in call of FORM: %s\n",s);
00511                               errorflag++;
00512                           }
00513                           break;
00514               }
00515           }
00516           else if ( argc == 0 && !AM.Interact ) AM.InputFileName = argv[-1];
00517           else {
00518 #ifdef WITHMPI
00519               if ( PF.me == MASTER )
00520 #endif
00521               printf("Illegal option in call of FORM: %s\n",s);
00522               errorflag++;
00523           }
00524       }
00525       AM.totalnumberofthreads = threadnum;
00526       if ( AM.InputFileName ) {
00527           if ( AM.FromStdin ) {
00528               printf("STDIN and the input filename cannot be specified simultaneously\n");
00529               errorflag++;
00530           }
00531           s = AM.InputFileName;
00532           while ( *s ) s++;
00533           if ( s < AM.InputFileName+4 ||
00534           s[-4] != '.' || s[-3] != 'f' || s[-2] != 'r' || s[-1] != 'm' ) {
00535               t = (UBYTE *)Malloc1((s-AM.InputFileName)+5,"adding .frm");
00536               s = AM.InputFileName;
00537               AM.InputFileName = t;
00538               while ( *s ) *t++ = *s++;
00539               *t++ = '.'; *t++ = 'f'; *t++ = 'r'; *t++ = 'm'; *t = 0;
00540           }
00541           if ( AM.LogType >= 0 && AM.LogFileName == NULL ) {
00542               AM.LogFileName = strDup1(AM.InputFileName,"name of logfile");
00543               s = AM.LogFileName;
00544               while ( *s ) s++;
00545               s[-3] = 'l'; s[-2] = 'o'; s[-1] = 'g';
00546           }
00547       }
00548 #ifdef WITHINTERACTION
00549       else if ( AM.Interact ) {
00550           if ( AM.LogType >= 0 ) {
00551 /*
00552               We may have to do better than just taking a name.
00553               It is not unique! This will be left for later.
00554 */
00555               AM.LogFileName = deflogname;
00556           }
00557       }
00558 #endif
00559       else if ( AM.FromStdin ) {
00560           /* Do nothing. */
00561       }
00562       else {
00563 NoFile:
00564 #ifdef WITHMPI
00565           if ( PF.me == MASTER )
00566 #endif
00567           printf("No filename specified in call of FORM\n");
00568           errorflag++;
00569       }
00570       if ( AM.Path == NULL ) AM.Path = (UBYTE *)getenv("FORMPATH");
00571       if ( AM.Path ) {
00572           /*
00573            * AM.Path is taken from argv or getenv. Reallocate it to avoid invalid
00574            * frees when AM.Path has to be changed.
00575            */
00576           AM.Path = strDup1(AM.Path,"DoTail Path");
00577       }
00578       return(errorflag);
00579 }
00580
00581 /*
00582           #] DoTail :
00583           #[ OpenInput :
00584
00585           Major task here after opening is to skip the proper number of
00586           .clear instructions if so desired without using interpretation
00587 */
00588
00589 int OpenInput(void)
00590 {
00591       int oldNoShowInput = AC.NoShowInput;
```

```
00592     UBYTE c;
00593     if ( !AM.Interact ) {
00594         if ( AM.FromStdin ) {
00595             if ( OpenStream(0,INPUTSTREAM,0,PRENOACTION) == 0 ) {
00596                 Error0("Cannot open STDIN");
00597                 return(-1);
00598             }
00599         }
00600         else {
00601         if ( OpenStream(AM.InputFileName,FILESTREAM,0,PRENOACTION) == 0 ) {
00602             Error1("Cannot open file",AM.InputFileName);
00603             return(-1);
00604         }
00605         if ( AC.CurrentStream->inbuffer <= 0 ) {
00606             Error1("No input in file",AM.InputFileName);
00607             return(-1);
00608         }
00609         }
00610         AC.NoShowInput = 1;
00611         while ( AM.SkipClears > 0 ) {
00612             c = GetInput();
00613             if ( c == ENDOFINPUT ) {
00614                 Error0("Not enough .clear instructions in input file");
00615             }
00616             if ( c == '\\' ) {
00617                 c = GetInput();
00618                 if ( c == ENDOFINPUT )
00619                     Error0("Not enough .clear instructions in input file");
00620                 continue;
00621             }
00622             if ( c == ' ' || c == '\t' ) continue;
00623             if ( c == '.' ) {
00624                 c = GetInput();
00625                 if ( tolower(c) == 'c' ) {
00626                     c = GetInput();
00627                     if ( tolower(c) == 'l' ) {
00628                         c = GetInput();
00629                         if ( tolower(c) == 'e' ) {
00630                             c = GetInput();
00631                             if ( tolower(c) == 'a' ) {
00632                                 c = GetInput();
00633                                 if ( tolower(c) == 'r' ) {
00634                                     c = GetInput();
00635                                     if ( FG.cTable[c] > 2 ) {
00636                                         AM.SkipClears--;
00637                                     }
00638                                 }
00639                             }
00640                         }
00641                     }
00642                 }
00643                 while ( c != '\n' && c != '\r' && c != ENDOFINPUT ) {
00644                     c = GetInput();
00645                     if ( c == '\\' ) c = GetInput();
00646                 }
00647             }
00648             else if ( c == '\n' || c == '\r' ) continue;
00649             else {
00650                 while ( ( c = GetInput() ) != '\n' && c != '\r' ) {
00651                     if ( c == ENDOFINPUT ) {
00652                         Error0("Not enough .clear instructions in input file");
00653                     }
00654                 }
00655             }
00656         }
00657         AC.NoShowInput = oldNoShowInput;
00658     }
00659     if ( AM.LogFileName ) {
00660 #ifdef WITHMPI
00661         if ( PF.me != MASTER ) {
00662             /*
00663              * Only the master writes to the log file. On slaves, we need
00664              * a dummy handle, without opening the file.
00665              */
00666             extern FILES **filelist;  /* in tools.c */
00667             int i = CreateHandle();
00668             RWLOCKW(AM.handlelock);
00669             filelist[i] = (FILES *)123;  /* Must be nonzero to prevent a reuse in CreateHandle. */
00670             UNRWLOCK(AM.handlelock);
00671             AC.LogHandle = i;
00672         }
00673         else
00674 #endif
00675         if ( AC.CheckpointFlag != -1 ) {
00676             if ( ( AC.LogHandle = CreateLogFile((char *)(AM.LogFileName)) ) < 0 ) {
00677                 Error1("Cannot create logfile",AM.LogFileName);
00678                 return(-1);
```

```
00679             }
00680         }
00681         else {
00682             if ( ( AC.LogHandle = OpenAddFile((char *)(AM.LogFileName)) ) < 0 ) {
00683                 Error1("Cannot re-open logfile",AM.LogFileName);
00684                 return(-1);
00685             }
00686         }
00687     }
00688     return(0);
00689 }
00690
00691 /*
00692         #] OpenInput :
00693         #[ ReserveTempFiles :
00694
00695         Order of preference:
00696         a: if there is a path in the commandtail, take that.
00697         b: if none, try in the form.set file.
00698         c: if still none, try in the environment for the variable FORMTMP
00699         d: if still none, try the current directory.
00700
00701         The parameter indicates action in the case of multithreaded running.
00702         par = 0 : We just run on a single processor. Keep everything normal.
00703         par = 1 : Multithreaded running startup phase 1.
00704         par = 2 : Multithreaded running startup phase 2.
00705 */
00706
00707 UBYTE *emptystring = (UBYTE *)".";
00708 UBYTE *defaulttempfilename = (UBYTE *)"xformxxx.str";
00709 /* This is the length of the above default, but with 7 spaces for PID digits
00710    instead of the "xxx". (Previously FORM used 5 digits and this value was 14) */
00711 #define DEFAULTFNAMELENGTH 16
00712
00713 void ReserveTempFiles(int par)
00714 {
00715     GETIDENTITY
00716     SETUPPARAMETERS *sp;
00717     UBYTE *s, *t, *tenddir, *tenddir2, c;
00718     int i = 0;
00719     WORD j;
00720     if ( par == 0 || par == 1 ) {
00721     if ( AM.TempDir == NULL ) {
00722         sp = GetSetupPar((UBYTE *)"tempdir");
00723         if ( ( sp->flags & USEDFLAG ) != USEDFLAG ) {
00724             AM.TempDir = (UBYTE *)getenv("FORMTMP");
00725             if ( AM.TempDir == NULL ) AM.TempDir = emptystring;
00726         }
00727         else AM.TempDir = (UBYTE *)(sp->value);
00728     }
00729     if ( AM.TempSortDir == NULL ) {
00730         if ( AM.havesortdir ) {
00731             sp = GetSetupPar((UBYTE *)"tempsortdir");
00732             AM.TempSortDir = (UBYTE *)(sp->value);
00733         }
00734         else {
00735             AM.TempSortDir = (UBYTE *)getenv("FORMTMPSORT");
00736             if ( AM.TempSortDir == NULL ) AM.TempSortDir = AM.TempDir;
00737         }
00738     }
00739 /*
00740     We have now in principle a path but we will use its first element only.
00741     Later that should become more complicated. Then we will use a path and
00742     when one device is full we can continue on the next one.
00743 */
00744     s = AM.TempDir; i = 200;   /* Some extra for VMS */
00745     while ( *s && *s != PATHSEPARATOR ) { if ( *s == '\\' ) s++; s++; i++; }
00746     FG.fnamesize = sizeof(UBYTE)*(i+DEFAULTFNAMELENGTH);
00747     FG.fname = (char *)Malloc1(FG.fnamesize,"name for temporary files");
00748     s = AM.TempDir; t = (UBYTE *)FG.fname;
00749     while ( *s && *s != PATHSEPARATOR ) { if ( *s == '\\' ) s++; *t++ = *s++; }
00750     if ( (char *)t > FG.fname && t[-1] != SEPARATOR && t[-1] != ALTSEPARATOR )
00751         *t++ = SEPARATOR;
00752     *t = 0;
00753     tenddir = t;
00754     FG.fnamebase = t-(UBYTE *)(FG.fname);
00755
00756     s = AM.TempSortDir; i = 200;   /* Some extra for VMS */
00757     while ( *s && *s != PATHSEPARATOR ) { if ( *s == '\\' ) s++; s++; i++; }
00758
00759     FG.fname2size = sizeof(UBYTE)*(i+DEFAULTFNAMELENGTH);
00760     FG.fname2 = (char *)Malloc1(FG.fname2size,"name for sort files");
00761     s = AM.TempSortDir; t = (UBYTE *)FG.fname2;
00762     while ( *s && *s != PATHSEPARATOR ) { if ( *s == '\\' ) s++; *t++ = *s++; }
00763     if ( (char *)t > FG.fname2 && t[-1] != SEPARATOR && t[-1] != ALTSEPARATOR )
00764         *t++ = SEPARATOR;
00765     *t = 0;
```

```
00766     tenddir2 = t;
00767     FG.fname2base = t-(UBYTE *)(FG.fname2);
00768
00769     t = tenddir;
00770     s = defaulttempfilename;
00771 #ifdef WITHMPI
00772     {
00773         int iii;
00774         iii = snprintf((char*)t,FG.fnamesize-((char*)t-FG.fname),"%d",PF.me);
00775         t+= iii;
00776         s+= iii; /* in case defaulttmpfilename is too short */
00777     }
00778 #endif
00779     while ( *s ) *t++ = *s++;
00780     *t = 0;
00781 /*
00782         There are problems when running many FORM jobs at the same time
00783         from make or minos. If they start up simultaneously, occasionally
00784         they can make the same .str file. We prevent this with first trying
00785         a file that contains the digits of the pid. If this file
00786         has already been taken we fall back on the old scheme.
00787         The whole is controlled with the -M (MultiRun) parameter in the
00788         command tail.
00789 */
00790     if ( AM.MultiRun ) {
00791         int num = ((int)GetPID())%10000000;
00792         t += 4;
00793         *t = 0;
00794         t[-1] = 'r';
00795         t[-2] = 't';
00796         t[-3] = 's';
00797         t[-4] = '.';
00798         t[-5] = (UBYTE)('0' + num%10);
00799         t[-6] = (UBYTE)('0' + (num/10)%10);
00800         t[-7] = (UBYTE)('0' + (num/100)%10);
00801         t[-8] = (UBYTE)('0' + (num/1000)%10);
00802         t[-9] = (UBYTE)('0' + (num/10000)%10);
00803         t[-10] = (UBYTE)('0' + (num/100000)%10);
00804         t[-11] = (UBYTE)('0' + num/1000000);
00805         if ( ( AC.StoreHandle = CreateFile((char *)FG.fname) ) < 0 ) {
00806             t[-5] = 'x'; t[-6] = 'x'; t[-7] = 'x'; t[-8] = 'x'; t[-9] = 'x'; t[-10] = 'x'; t[-11] =
    'x';
00807             goto classic;
00808         }
00809     }
00810     else
00811     {
00812 classic:;
00813       for(;;) {
00814         if ( ( AC.StoreHandle = OpenFile((char *)FG.fname) ) < 0 ) {
00815             if ( ( AC.StoreHandle = CreateFile((char *)FG.fname) ) >= 0 ) break;
00816         }
00817         else CloseFile(AC.StoreHandle);
00818         c = t[-5];
00819         if ( c == 'x' ) t[-5] = '0';
00820         else if ( c == '9' ) {
00821             t[-5] = '0';
00822             c = t[-6];
00823             if ( c == 'x' ) t[-6] = '0';
00824             else if ( c == '9' ) {
00825                 t[-6] = '0';
00826                 c = t[-7];
00827                 if ( c == 'x' ) t[-7] = '0';
00828                 else if ( c == '9' ) {
00829 /*
00830                     Note that we tried 1111 names!
00831 */
00832                     MesPrint("Name space for temp files exhausted");
00833                     t[-7] = 0;
00834                     MesPrint("Please remove files of the type %s or try a different directory"
00835                         ,FG.fname);
00836                     Terminate(-1);
00837                 }
00838                 else t[-7] = (UBYTE)(c+1);
00839             }
00840             else t[-6] = (UBYTE)(c+1);
00841         }
00842         else t[-5] = (UBYTE)(c+1);
00843       }
00844     }
00845 /*
00846     Now we should make sure that the tempsortdir cq tempsortfilename makes it
00847     into a similar construction.
00848 */
00849     s = tenddir; t = tenddir2; while ( *s ) *t++ = *s++;
00850     *t = 0;
00851
```

```
00852 /*
00853     Now we should assign a name to the main sort file and the two stage 4 files.
00854 */
00855     AM.S0->file.name = (char *)Malloc1(sizeof(char)*(i+DEFAULTFNAMELENGTH),"name for temporary
      files");
00856     s = (UBYTE *)AM.S0->file.name;
00857     t = (UBYTE *)FG.fname2;
00858     i = 1;
00859     while ( *t ) { *s++ = *t++; i++; }
00860     s[-2] = 'o'; *s = 0;
00861     }
00862 /*
00863     Try to create the sort file already, so we can Terminate earlier if this fails.
00864 */
00865 #ifdef WITHPTHREADS
00866     if ( par <= 1 ) {
00867 #endif
00868     if ( ( AM.S0->file.handle = CreateFile((char *)AM.S0->file.name) ) < 0 ) {
00869         MesPrint("Could not create sort file: %s", AM.S0->file.name);
00870         Terminate(-1);
00871     };
00872     /* Close and clean up the test file */
00873     CloseFile(AM.S0->file.handle);
00874     AM.S0->file.handle = -1;
00875     remove(AM.S0->file.name);
00876 #ifdef WITHPTHREADS
00877     }
00878 #endif
00879 /*
00880     With the stage4 and scratch file names we have to be a bit more careful.
00881     They are to be allocated after the threads are initialized when there
00882     are threads of course.
00883 */
00884     if ( par == 0 ) {
00885         s = (UBYTE *)((void *)(FG.fname2)); i = 0;
00886         while ( *s ) { s++; i++; }
00887         /* +1 for null terminator */
00888         s = (UBYTE *)Malloc1(sizeof(char)*(i+1),"name for stage4 file a");
00889         AR.FoStage4[1].name = (char *)s;
00890         t = (UBYTE *)FG.fname2;
00891         while ( *t ) *s++ = *t++;
00892         s[-2] = '4'; s[-1] = 'a'; *s = 0;
00893         s = (UBYTE *)((void *)(FG.fname)); i = 0;
00894         while ( *s ) { s++; i++; }
00895         /* +1 for null terminator */
00896         s = (UBYTE *)Malloc1(sizeof(char)*(i+1),"name for stage4 file b");
00897         AR.FoStage4[0].name = (char *)s;
00898         t = (UBYTE *)FG.fname;
00899         while ( *t ) *s++ = *t++;
00900         s[-2] = '4'; s[-1] = 'b'; *s = 0;
00901         for ( j = 0; j < 3; j++ ) {
00902             /* +1 for null terminator */
00903             s = (UBYTE *)Malloc1(sizeof(char)*(i+1),"name for scratch file");
00904             AR.Fscr[j].name = (char *)s;
00905             t = (UBYTE *)FG.fname;
00906             while ( *t ) *s++ = *t++;
00907             s[-2] = 'c'; s[-1] = (UBYTE)('0'+j); *s = 0;
00908         }
00909     }
00910 #ifdef WITHPTHREADS
00911     else if ( par == 2 ) {
00912         size_t tname;
00913         s = (UBYTE *)((void *)(FG.fname2)); i = 0;
00914         while ( *s ) { s++; i++; }
00915         /* +1 for null terminator, +10 for 32bit int, +1 for "." */
00916         tname = sizeof(char)*(i+12);
00917         s = (UBYTE *)Malloc1(tname,"name for stage4 file a");
00918         snprintf((char *)s,tname,"%s.%d",FG.fname2,AT.identity);
00919         s[i-2] = '4'; s[i-1] = 'a';
00920         AR.FoStage4[1].name = (char *)s;
00921         s = (UBYTE *)((void *)(FG.fname)); i = 0;
00922         while ( *s ) { s++; i++; }
00923         /* +1 for null terminator, +10 for 32bit int, +1 for "." */
00924         tname = sizeof(char)*(i+12);
00925         s = (UBYTE *)Malloc1(tname,"name for stage4 file b");
00926         snprintf((char *)s,tname,"%s.%d",FG.fname,AT.identity);
00927         s[i-2] = '4'; s[i-1] = 'b';
00928         AR.FoStage4[0].name = (char *)s;
00929         if ( AT.identity == 0 ) {
00930             for ( j = 0; j < 3; j++ ) {
00931                 /* +1 for null terminator */
00932                 s = (UBYTE *)Malloc1(sizeof(char)*(i+1),"name for scratch file");
00933                 AR.Fscr[j].name = (char *)s;
00934                 t = (UBYTE *)FG.fname;
00935                 while ( *t ) *s++ = *t++;
00936                 s[-2] = 'c'; s[-1] = (UBYTE)('0'+j); *s = 0;
00937             }
```

```
00938         }
00939     }
00940 #endif
00941 }
00942
00943 /*
00944         #] ReserveTempFiles :
00945         #[ StartVariables :
00946 */
00947
00948 #ifdef WITHPTHREADS
00949 ALLPRIVATES *DummyPointer = 0;
00950 #endif
00951
00952 void StartVariables(void)
00953 {
00954     int i, ii;
00955     PUTZERO(AM.zeropos);
00956     StartPrepro();
00957 /*
00958     The module counter:
00959 */
00960     AC.CModule=0;
00961 #ifdef WITHPTHREADS
00962 /*
00963     We need a value in AB because in the startup some routines may call AB[0].
00964 */
00965     AB = (ALLPRIVATES **)&DummyPointer;
00966 #endif
00967 /*
00968     separators used to delimit arguments in #call and #do, by default ',' and '|':
00969     Be sure, it is en empty set:
00970 */
00971     set_sub(AC.separators,AC.separators,AC.separators);
00972     set_set(',',AC.separators);
00973     set_set('|',AC.separators);
00974
00975     AM.BracketFactors[0] = 8;
00976     AM.BracketFactors[1] = SYMBOL;
00977     AM.BracketFactors[2] = 4;
00978     AM.BracketFactors[3] = FACTORSYMBOL;
00979     AM.BracketFactors[4] = 1;
00980     AM.BracketFactors[5] = 1;
00981     AM.BracketFactors[6] = 1;
00982     AM.BracketFactors[7] = 3;
00983
00984     AM.SkipClears = 0;
00985     AC.Cnumpows = 0;
00986     AC.OutputMode = 72;
00987     AC.OutputSpaces = NORMALFORMAT;
00988     AC.LineLength = 79;
00989     AM.gIsFortran90 = AC.IsFortran90 = ISNOTFORTRAN90;
00990     AM.gFortran90Kind = AC.Fortran90Kind = 0;
00991     AM.gCnumpows = 0;
00992     AC.exprfillwarning = 0;
00993     AM.gLineLength = 79;
00994     AM.OutBufSize = 80;
00995     AM.MaxStreamSize = MAXFILESTREAMSIZE;
00996     AP.MaxPreAssignLevel = 4;
00997     AC.iBufferSize = 512;
00998     AP.pSize = 128;
00999     AP.MaxPreIfLevel = 10;
01000     AP.cComChar = AP.ComChar = '*';
01001     AP.firstnamespace = 0;
01002     AP.lastnamespace = 0;
01003     AP.fullnamesize = 127;
01004     AP.fullname = (UBYTE *)Malloc1((AP.fullnamesize+1)*sizeof(UBYTE),"AP.fullname");
01005     AM.OffsetVector = -2*WILDOFFSET+MINSPEC;
01006     AC.cbufList.num = 0;
01007     AM.hparallelflag = AM.gparallelflag =
01008     AC.parallelflag = AC.mparallelflag = PARALLELFLAG;
01009 #ifdef WITHMPI
01010     if ( PF.numtasks < 2 ) AM.hparallelflag |= NOPARALLEL_NPROC;
01011 #endif
01012     AC.tablefilling = 0;
01013     AC.models = 0;
01014     AC.nummodels = 0;
01015     AC.ModelLevel = 0;
01016     AC.modelspace = 0;
01017     AM.resetTimeOnClear = 1;
01018     AM.gnumextrasym = AM.ggnumextrasym = 0;
01019     AM.havesortdir = 0;
01020     AM.SpectatorFiles = 0;
01021     AM.NumSpectatorFiles = 0;
01022     AM.SizeForSpectatorFiles = 0;
01023 /*
01024     Information for the lists of variables. Part of error message and size:
```

```
01025 */
01026     AP.ProcList.message = "procedure";
01027     AP.ProcList.size = sizeof(PROCEDURE);
01028     AP.LoopList.message = "doloop";
01029     AP.LoopList.size = sizeof(DOLOOP);
01030     AP.PreVarList.message = "PreVariable";
01031     AP.PreVarList.size = sizeof(PREVAR);
01032     AC.SymbolList.message = "symbol";
01033     AC.SymbolList.size = sizeof(struct SyMbOl);
01034     AC.IndexList.message = "index";
01035     AC.IndexList.size = sizeof(struct InDeX);
01036     AC.VectorList.message = "vector";
01037     AC.VectorList.size = sizeof(struct VeCtOr);
01038     AC.FunctionList.message = "function";
01039     AC.FunctionList.size = sizeof(struct FuNcTiOn);
01040     AC.SetList.message = "set";
01041     AC.SetList.size = sizeof(struct SeTs);
01042     AC.SetElementList.message = "set element";
01043     AC.SetElementList.size = sizeof(WORD);
01044     AC.ExpressionList.message = "expression";
01045     AC.ExpressionList.size = sizeof(struct ExPrEsSiOn);
01046     AC.cbufList.message = "compiler buffer";
01047     AC.cbufList.size = sizeof(CBUF);
01048     AC.ChannelList.message = "channel buffer";
01049     AC.ChannelList.size = sizeof(CHANNEL);
01050     AP.DollarList.message = "$-variable";
01051     AP.DollarList.size = sizeof(struct DoLlArS);
01052     AC.DubiousList.message = "ambiguous variable";
01053     AC.DubiousList.size = sizeof(struct DuBiOuS);
01054     AC.TableBaseList.message = "list of tablebases";
01055     AC.TableBaseList.size = sizeof(DBASE);
01056     AC.TestValue = 0;
01057     AC.InnerTest = 0;
01058
01059     AC.AutoSymbolList.message = "autosymbol";
01060     AC.AutoSymbolList.size = sizeof(struct SyMbOl);
01061     AC.AutoIndexList.message = "autoindex";
01062     AC.AutoIndexList.size = sizeof(struct InDeX);
01063     AC.AutoVectorList.message = "autovector";
01064     AC.AutoVectorList.size = sizeof(struct VeCtOr);
01065     AC.AutoFunctionList.message = "autofunction";
01066     AC.AutoFunctionList.size = sizeof(struct FuNcTiOn);
01067     AC.PotModDolList.message = "potentially modified dollar";
01068     AC.PotModDolList.size = sizeof(WORD);
01069     AC.ModOptDolList.message = "moduleoptiondollar";
01070     AC.ModOptDolList.size = sizeof(MODOPTDOLLAR);
01071
01072     AO.FortDotChar = '_';
01073     AO.ErrorBlock = 0;
01074     AC.firstconstindex = 1;
01075     AO.Optimize.mctsconstant.fval = 1.0;
01076     AO.Optimize.horner = O_MCTS;
01077     AO.Optimize.hornerdirection = O_FORWARDORBACKWARD;
01078     AO.Optimize.method = O_GREEDY;
01079     AO.Optimize.mctstimelimit = 0;
01080     AO.Optimize.mctsnumexpand = 1000;
01081     AO.Optimize.mctsnumkeep = 10;
01082     AO.Optimize.mctsnumrepeat = 1;
01083     AO.Optimize.greedytimelimit = 0;
01084     AO.Optimize.greedyminnum = 10;
01085     AO.Optimize.greedymaxperc = 5;
01086     AO.Optimize.printstats = 0;
01087     AO.Optimize.debugflags = 0;
01088     AO.OptimizeResult.code = NULL;
01089     AO.inscheme = 0;
01090     AO.schemenum = 0;
01091     AO.wpos = 0;
01092     AO.wpoin = 0;
01093     AO.wlen = 0;
01094     AM.dollarzero = 0;
01095     AC.doloopstack = 0;
01096     AC.doloopstacksize = 0;
01097     AC.dolooplevel = 0;
01098 /*
01099     Set up the main name trees:
01100 */
01101     AC.varnames  = MakeNameTree();
01102     AC.exprnames = MakeNameTree();
01103     AC.dollarnames = MakeNameTree();
01104     AC.autonames = MakeNameTree();
01105     AC.activenames = &(AC.varnames);
01106     AP.preError = 0;
01107 /*
01108     Initialize the compiler:
01109 */
01110     inictable();
01111     AM.rbufnum = inicbufs();        /* Regular compiler buffer */
```

```
01112 #ifndef WITHPTHREADS
01113     AT.ebufnum = inicbufs();        /* Buffer for extras during execution */
01114     AT.fbufnum = inicbufs();        /* Buffer for caching in factorization */
01115     AT.allbufnum = inicbufs();      /* Buffer for id,all */
01116     AT.aebufnum = inicbufs();       /* Buffer for id,all */
01117     AN.tryterm = 0;
01118 #else
01119     AS.MasterSort = 0;
01120 #endif
01121     AM.dbufnum = inicbufs();        /* Buffer for dollar variables */
01122     AM.sbufnum = inicbufs();        /* Subterm buffer for polynomials and optimization */
01123     AC.ffbufnum = inicbufs();       /* Buffer number for user defined factorizations */
01124     AM.zbufnum = inicbufs();        /* For very special values */
01125     {
01126         CBUF *C = cbuf+AM.zbufnum;
01127         WORD one[5] = {4,1,1,3,0};
01128         WORD zero = 0;
01129         AddRHS(AM.zbufnum,1);
01130         AM.zerorhs = C->numrhs;
01131         AddNtoC(AM.zbufnum,1,&zero,17);
01132         AddRHS(AM.zbufnum,1);
01133         AM.onerhs = C->numrhs;
01134         AddNtoC(AM.zbufnum,5,one,17);
01135     }
01136     AP.inside.inscbuf = inicbufs(); /* For the #inside instruction */
01137 /*
01138     Enter the built in objects
01139 */
01140     AC.Symbols = &(AC.SymbolList);
01141     AC.Indices = &(AC.IndexList);
01142     AC.Vectors = &(AC.VectorList);
01143     AC.Functions = &(AC.FunctionList);
01144     AC.vetofilling = 0;
01145
01146     AddDollar((UBYTE *)"$",DOLUNDEFINED,0,0);
01147
01148     cbuf[AM.dbufnum].mnumlhs = cbuf[AM.dbufnum].numlhs;
01149     cbuf[AM.dbufnum].mnumrhs = cbuf[AM.dbufnum].numrhs;
01150
01151     AddSymbol((UBYTE *)"i_",-MAXPOWER,MAXPOWER,VARTYPEIMAGINARY,0);
01152     AddSymbol((UBYTE *)"pi_",-MAXPOWER,MAXPOWER,VARTYPENONE,0);
01153 /*
01154     coeff_ should have the number COEFFSYMBOL and den_ the number DENOMINATOR
01155     and the three should be in this order!
01156 */
01157     AddSymbol((UBYTE *)"coeff_",-MAXPOWER,MAXPOWER,VARTYPENONE,0);
01158     AddSymbol((UBYTE *)"num_",-MAXPOWER,MAXPOWER,VARTYPENONE,0);
01159     AddSymbol((UBYTE *)"den_",-MAXPOWER,MAXPOWER,VARTYPENONE,0);
01160     AddSymbol((UBYTE *)"xarg_",-MAXPOWER,MAXPOWER,VARTYPENONE,0);
01161     AddSymbol((UBYTE *)"dimension_",-MAXPOWER,MAXPOWER,VARTYPENONE,0);
01162     AddSymbol((UBYTE *)"factor_",-MAXPOWER,MAXPOWER,VARTYPENONE,0);
01163     AddSymbol((UBYTE *)"sep_",-MAXPOWER,MAXPOWER,VARTYPENONE,0);
01164     AddSymbol((UBYTE *)"ee_",-MAXPOWER,MAXPOWER,VARTYPENONE,0);
01165     AddSymbol((UBYTE *)"em_",-MAXPOWER,MAXPOWER,VARTYPENONE,0);
01166     i = BUILTINSYMBOLS;  /* update this in ftypes.h when we add new symbols */
01167 /*
01168     Next we add a number of dummy symbols for ensuring that the user defined
01169     symbols start at a fixed given number FIRSTUSERSYMBOL
01170     We do want to give them unique names though that the user cannot access.
01171 */
01172     {
01173         char dumstr[20];
01174         for ( ; i < FIRSTUSERSYMBOL; i++ ) {
01175             snprintf(dumstr,20,":%d:",i);
01176             AddSymbol((UBYTE *)dumstr,-MAXPOWER,MAXPOWER,VARTYPENONE,0);
01177         }
01178     }
01179
01180     AddIndex((UBYTE *)"iarg_",4,0);
01181     AddVector((UBYTE *)"parg_",VARTYPENONE,0);
01182
01183     AM.NumFixedFunctions = sizeof(fixedfunctions)/sizeof(struct fixedfun);
01184     for ( i = 0; i < AM.NumFixedFunctions; i++ ) {
01185         ii = AddFunction((UBYTE *)fixedfunctions[i].name
01186                                  ,fixedfunctions[i].commu
01187                                  ,fixedfunctions[i].tensor
01188                                  ,fixedfunctions[i].complx
01189                                  ,fixedfunctions[i].symmetric
01190                                  ,0,-1,-1);
01191         if ( fixedfunctions[i].tensor == GAMMAFUNCTION )
01192                              functions[ii].flags |= COULDCOMMUTE;
01193     }
01194 /*
01195     Next we add a number of dummy functions for ensuring that the user defined
01196     functions start at a fixed given number FIRSTUSERFUNCTION.
01197     We do want to give them unique names though that the user cannot access.
01198 */
```

```
01199     {
01200         char dumstr[20];
01201         for ( ; i < FIRSTUSERFUNCTION-FUNCTION; i++ ) {
01202             snprintf(dumstr,20,"::%d::",i);
01203             AddFunction((UBYTE *)dumstr,0,0,0,0,0,-1,-1);
01204         }
01205     }
01206     AM.NumFixedSets = sizeof(fixedsets)/sizeof(struct fixedset);
01207     for ( i = 0; i < AM.NumFixedSets; i++ ) {
01208         ii = AddSet((UBYTE *)fixedsets[i].name,fixedsets[i].dimension);
01209         Sets[ii].type = fixedsets[i].type;
01210     }
01211     AM.RepMax = MAXREPEAT;
01212 #ifndef WITHPTHREADS
01213     AT.RepCount = (int *)Malloc1((LONG)((AM.RepMax+3)*sizeof(int)),"repeat buffers");
01214     AN.RepPoint = AT.RepCount;
01215     AT.RepTop = AT.RepCount + AM.RepMax;
01216     AN.polysortflag = 0;
01217     AN.subsubveto = 0;
01218 #endif
01219     AC.NumWildcardNames = 0;
01220     AC.WildcardBufferSize = 50;
01221     AC.WildcardNames = (UBYTE *)Malloc1((LONG)AC.WildcardBufferSize,"argument list names");
01222 #ifndef WITHPTHREADS
01223     AT.WildArgTaken = (WORD *)Malloc1((LONG)AC.WildcardBufferSize*sizeof(WORD)/2
01224                 ,"argument list names");
01225     AT.WildcardBufferSize = AC.WildcardBufferSize;
01226     AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
01227     AT.nfac = AT.nBer = 0;
01228     AT.factorials = 0;
01229     AT.bernoullis = 0;
01230     AR.wranfia = 0;
01231     AR.wranfcall = 0;
01232     AR.wranfnpair1 = NPAIR1;
01233     AR.wranfnpair2 = NPAIR2;
01234     AR.wranfseed = 0;
01235
01236     AT.NormData = Malloc1(sizeof(*(AT.NormData)), "NormData pointers");
01237     AT.NormDataSize = 1;
01238     AT.NormData[0] = AllocNormData();
01239     AT.NormDepth = 0;
01240 #endif
01241     AM.atstartup = 1;
01242     AM.oldnumextrasymbols = strDup1((UBYTE *)"OLDNUMEXTRASYMBOLS_","oldnumextrasymbols");
01243     PutPreVar((UBYTE *)"VERSION_",(UBYTE *)STRINGIFY(MAJORVERSION),0,0);
01244     PutPreVar((UBYTE *)"SUBVERSION_",(UBYTE *)STRINGIFY(MINORVERSION),0,0);
01245     PutPreVar((UBYTE *)"SUBSUBVERSION_",(UBYTE *)STRINGIFY(PATCHVERSION),0,0);
01246     PutPreVar((UBYTE *)"DATE_",(UBYTE *)MakeDate(),0,0);
01247     PutPreVar((UBYTE *)"random_",(UBYTE *)"________",0,0);
01248     PutPreVar((UBYTE *)"optimminvar_",(UBYTE *)("0"),0,0);
01249     PutPreVar((UBYTE *)"optimmaxvar_",(UBYTE *)("0"),0,0);
01250     PutPreVar(AM.oldnumextrasymbols,(UBYTE *)("0"),0,0);
01251     PutPreVar((UBYTE *)"optimvalue_",(UBYTE *)("0"),0,0);
01252     PutPreVar((UBYTE *)"optimscheme_",(UBYTE *)("0"),0,0);
01253     PutPreVar((UBYTE *)"tolower_",(UBYTE *)("0"),0,0);
01254     PutPreVar((UBYTE *)"toupper_",(UBYTE *)("0"),0,0);
01255     PutPreVar((UBYTE *)"takeleft_",(UBYTE *)("0"),0,0);
01256     PutPreVar((UBYTE *)"takeright_",(UBYTE *)("0"),0,0);
01257     PutPreVar((UBYTE *)"keepleft_",(UBYTE *)("0"),0,0);
01258     PutPreVar((UBYTE *)"keepright_",(UBYTE *)("0"),0,0);
01259     PutPreVar((UBYTE *)"SYSTEMERROR_",(UBYTE *)("0"),0,0);
01260 /*
01261     Next are the flags to control diagram generation filters
01262 */
01263     #define STR2(x) #x
01264     #define STR(x) STR2(x)
01265     PutPreVar((UBYTE *)"TOPOLOGIESONLY_" ,(UBYTE*)(STR(TOPOLOGIESONLY)) ,0,0);
01266     PutPreVar((UBYTE *)"WITHOUTNODES_"   ,(UBYTE*)(STR(WITHOUTNODES))   ,0,0);
01267     PutPreVar((UBYTE *)"WITHEDGES_"      ,(UBYTE*)(STR(WITHEDGES))      ,0,0);
01268     PutPreVar((UBYTE *)"WITHBLOCKS_"     ,(UBYTE*)(STR(WITHBLOCKS))     ,0,0);
01269     PutPreVar((UBYTE *)"WITHONEPISETS_"  ,(UBYTE*)(STR(WITHONEPISETS))  ,0,0);
01270     PutPreVar((UBYTE *)"WITHSYMMETRIZEI_",(UBYTE*)(STR(WITHSYMMETRIZEI)),0,0);
01271     PutPreVar((UBYTE *)"WITHSYMMETRIZEF_",(UBYTE*)(STR(WITHSYMMETRIZEF)),0,0);
01272     // This is not an "option" preprocessor var but is set by "external" particle definitions
01273 //  PutPreVar((UBYTE *)"CHECKEXTERN_"    ,(UBYTE*)(STR(CHECKEXTERN))    ,0,0);
01274     PutPreVar((UBYTE *)"ONEPI_"          ,(UBYTE*)(STR(ONEPARTI))       ,0,0);
01275     PutPreVar((UBYTE *)"ONEPR_"          ,(UBYTE*)(STR(ONEPARTR))       ,0,0);
01276     PutPreVar((UBYTE *)"ONSHELL_"        ,(UBYTE*)(STR(ONSHELL))        ,0,0);
01277     PutPreVar((UBYTE *)"OFFSHELL_"       ,(UBYTE*)(STR(OFFSHELL))       ,0,0);
01278     PutPreVar((UBYTE *)"NOSIGMA_"        ,(UBYTE*)(STR(NOSIGMA))        ,0,0);
01279     PutPreVar((UBYTE *)"SIGMA_"          ,(UBYTE*)(STR(SIGMA))          ,0,0);
01280     PutPreVar((UBYTE *)"NOSNAIL_"        ,(UBYTE*)(STR(NOSNAIL))        ,0,0);
01281     PutPreVar((UBYTE *)"SNAIL_"          ,(UBYTE*)(STR(SNAIL))          ,0,0);
01282     PutPreVar((UBYTE *)"NOTADPOLE_"      ,(UBYTE*)(STR(NOTADPOLE))      ,0,0);
01283     PutPreVar((UBYTE *)"TADPOLE_"        ,(UBYTE*)(STR(TADPOLE))        ,0,0);
01284     PutPreVar((UBYTE *)"SIMPLE_"         ,(UBYTE*)(STR(SIMPLE))         ,0,0);
01285     PutPreVar((UBYTE *)"NOTSIMPLE_"      ,(UBYTE*)(STR(NOTSIMPLE))      ,0,0);
```

```
01286     PutPreVar((UBYTE *)"BIPART_"          ,(UBYTE*)(STR(BIPART))          ,0,0);
01287     PutPreVar((UBYTE *)"NONBIPART_"       ,(UBYTE*)(STR(NONBIPART))       ,0,0);
01288     PutPreVar((UBYTE *)"CYCLI_"           ,(UBYTE*)(STR(CYCLI))           ,0,0);
01289     PutPreVar((UBYTE *)"CYCLR_"           ,(UBYTE*)(STR(CYCLR))           ,0,0);
01290     PutPreVar((UBYTE *)"FLOOP_"           ,(UBYTE*)(STR(FLOOP))           ,0,0);
01291     PutPreVar((UBYTE *)"NOTFLOOP_"        ,(UBYTE*)(STR(NOTFLOOP))        ,0,0);
01292
01293     {
01294         char buf[41];  /* up to 128-bit */
01295         LONG pid;
01296 #ifndef WITHMPI
01297         pid = GetPID();
01298 #else
01299         pid = ( PF.me == MASTER ) ? GetPID() : (LONG)0;
01300         pid = PF_BroadcastNumber(pid);
01301 #endif
01302         LongCopy(pid,buf);
01303         PutPreVar((UBYTE *)"PID_",(UBYTE *)buf,0,0);
01304     }
01305     AM.atstartup = 0;
01306     AP.MaxPreTypes = 10;
01307     AP.NumPreTypes = 0;
01308     AP.PreTypes = (int *)Malloc1(sizeof(int)*(AP.MaxPreTypes+1),"preprocessor types");
01309     AP.inside.buffer = 0;
01310     AP.inside.size = 0;
01311
01312     AC.SortType = AC.lSortType = AM.gSortType = SORTLOWFIRST;
01313 #ifdef WITHPTHREADS
01314 #else
01315     AR.SortType = AC.SortType;
01316 #endif
01317     AC.LogHandle = -1;
01318     AC.SetList.numtemp        = AC.SetList.num;
01319     AC.SetElementList.numtemp = AC.SetElementList.num;
01320
01321     GetName(AC.varnames,(UBYTE *)"exp_",&AM.expnum,NOAUTO);
01322     GetName(AC.varnames,(UBYTE *)"denom_",&AM.denomnum,NOAUTO);
01323     GetName(AC.varnames,(UBYTE *)"fac_",&AM.facnum,NOAUTO);
01324     GetName(AC.varnames,(UBYTE *)"invfac_",&AM.invfacnum,NOAUTO);
01325     GetName(AC.varnames,(UBYTE *)"sum_",&AM.sumnum,NOAUTO);
01326     GetName(AC.varnames,(UBYTE *)"sump_",&AM.sumpnum,NOAUTO);
01327     GetName(AC.varnames,(UBYTE *)"term_",&AM.termfunnum,NOAUTO);
01328     GetName(AC.varnames,(UBYTE *)"match_",&AM.matchfunnum,NOAUTO);
01329     GetName(AC.varnames,(UBYTE *)"count_",&AM.countfunnum,NOAUTO);
01330     AM.termfunnum += FUNCTION;
01331     AM.matchfunnum += FUNCTION;
01332     AM.countfunnum += FUNCTION;
01333
01334     AC.ThreadStats = AM.gThreadStats = AM.ggThreadStats = 1;
01335     AC.FinalStats = AM.gFinalStats = AM.ggFinalStats = 1;
01336     AC.StatsFlag = AM.gStatsFlag = AM.ggStatsFlag = 1;
01337     AC.ThreadsFlag = AM.gThreadsFlag = AM.ggThreadsFlag = 1;
01338     AC.ThreadBalancing = AM.gThreadBalancing = AM.ggThreadBalancing = 1;
01339     AC.ThreadSortFileSynch = AM.gThreadSortFileSynch = AM.ggThreadSortFileSynch = 0;
01340     AC.ProcessStats = AM.gProcessStats = AM.ggProcessStats = 1;
01341     AC.OldParallelStats = AM.gOldParallelStats = AM.ggOldParallelStats = 0;
01342     AC.OldFactArgFlag = AM.gOldFactArgFlag = AM.ggOldFactArgFlag = NEWFACTARG;
01343     AC.OldGCDflag = AM.gOldGCDflag = AM.ggOldGCDflag = 1;
01344     AC.OldPRFSignFlag = 0;
01345     AC.WTimeStatsFlag = AM.gWTimeStatsFlag = AM.ggWTimeStatsFlag = 0;
01346     AM.gcNumDollars = AP.DollarList.num;
01347     AC.SizeCommuteInSet = AM.gSizeCommuteInSet = 0;
01348 #ifdef WITHFLOAT
01349     AC.MaxWeight = AM.gMaxWeight = AM.ggMaxWeight = MAXWEIGHT;
01350     AC.DefaultPrecision = AM.gDefaultPrecision = AM.ggDefaultPrecision = DEFAULTPRECISION;
01351 #endif
01352     AC.CommuteInSet = 0;
01353
01354     AM.PrintTotalSize = 0;
01355
01356     AO.NoSpacesInNumbers = AM.gNoSpacesInNumbers = AM.ggNoSpacesInNumbers = 0;
01357     AO.IndentSpace = AM.gIndentSpace = AM.ggIndentSpace = INDENTSPACE;
01358     AO.BlockSpaces = 0;
01359     AO.OptimizationLevel = 0;
01360     PUTZERO(AS.MaxExprSize);
01361     PUTZERO(AC.StoreFileSize);
01362
01363 #ifdef WITHPTHREADS
01364     AC.inputnumbers = 0;
01365     AC.pfirstnum = 0;
01366     AC.numpfirstnum = AC.sizepfirstnum = 0;
01367 #endif
01368     AC.MemDebugFlag = 1;
01369
01370 #ifdef WITHEXTERNALCHANNEL
01371     AX.currentExternalChannel=0;
01372     AX.killSignal=SIGKILL;
```

```
01373     AX.killWholeGroup=1;
01374     AX.daemonize=1;
01375     AX.currentPrompt=0;
01376     AX.timeout=1000;/*One second to initialize preset channels*/
01377     AX.shellname=strDup1((UBYTE *)"/bin/sh -c","external channel shellname");
01378     AX.stderrname=strDup1((UBYTE *)"/dev/null","external channel stderrname");
01379 #endif
01380 }
01381
01382 /*
01383         #] StartVariables :
01384         #[ StartMore :
01385 */
01386
01387 void StartMore(void)
01388 {
01389 #ifdef WITHEXTERNALCHANNEL
01390     /*If env.variable "FORM_PIPES" is defined, we have to initialize
01391         corresponding pre-set external channels, see file extcmd.c.*/
01392     /*This line must be after all setup settings: in future, timeout
01393         could be changed at setup.*/
01394     if(AX.timeout>=0)/*if AX.timeout<0, this was done by cmdline option -pipe*/
01395         initPresetExternalChannels((UBYTE*)getenv("FORM_PIPES"),AX.timeout);
01396 #endif
01397
01398 #ifdef WITHMPI
01399 /*
01400     Define preprocessor variable PARALLELTASK_ as a process number, 0 is the master
01401     Define preprocessor variable NPARALLELTASKS_ as a total number of processes
01402 */
01403     {
01404         UBYTE buf[32];
01405         snprintf((char*)buf,32,"%d",PF.me);
01406         PutPreVar((UBYTE *)"PARALLELTASK_",buf,0,0);
01407         snprintf((char*)buf,32,"%d",PF.numtasks);
01408         PutPreVar((UBYTE *)"NPARALLELTASKS_",buf,0,0);
01409     }
01410 #else
01411     PutPreVar((UBYTE *)"PARALLELTASK_",(UBYTE *)"0",0,0);
01412     PutPreVar((UBYTE *)"NPARALLELTASKS_",(UBYTE *)"1",0,0);
01413 #endif
01414
01415     PutPreVar((UBYTE *)"NAME_",AM.InputFileName ? AM.InputFileName : (UBYTE *)"STDIN",0,0);
01416 }
01417
01418 /*
01419         #] StartMore :
01420         #[ IniVars :
01421
01422         This routine initializes the parameters that may change during the run.
01423 */
01424
01425 void IniVars(void)
01426 {
01427 #ifdef WITHPTHREADS
01428     GETIDENTITY
01429 #else
01430     WORD *t;
01431 #endif
01432     WORD *fi, i, one = 1;
01433     CBUF *C = cbuf+AC.cbufnum;
01434
01435 #ifdef WITHPTHREADS
01436     UBYTE buf[32];
01437     snprintf((char*)buf,32,"%d",AM.totalnumberofthreads);
01438     PutPreVar((UBYTE *)"NTHREADS_",buf,0,1);
01439 #else
01440     PutPreVar((UBYTE *)"NTHREADS_",(UBYTE *)"1",0,1);
01441 #endif
01442
01443     AC.ShortStats = 0;
01444     AC.WarnFlag = 1;
01445     AR.SortType = AC.SortType = AC.lSortType = AM.gSortType;
01446     AC.OutputMode = 72;
01447     AC.OutputSpaces = NORMALFORMAT;
01448     AR.Eside = 0;
01449     AC.DumNum = 0;
01450     AC.ncmod = AM.gncmod = 0;
01451     AC.modmode = AM.gmodmode = 0;
01452     AC.npowmod = AM.gnpowmod = 0;
01453     AC.halfmod = 0; AC.nhalfmod = 0;
01454     AC.modinverses = 0;
01455     AC.lPolyFun = AM.gPolyFun = 0;
01456     AC.lPolyFunInv = AM.gPolyFunInv = 0;
01457     AC.lPolyFunType = AM.gPolyFunType = 0;
01458     AC.lPolyFunExp = AM.gPolyFunExp = 0;
01459     AC.lPolyFunVar = AM.gPolyFunVar = 0;
```

```
01460     AC.lPolyFunPow = AM.gPolyFunPow = 0;
01461 #ifdef WITHFLINT
01462     AC.FlintPolyFlag = 1;
01463 #else
01464     AC.FlintPolyFlag = 0;
01465 #endif
01466     AC.DirtPow = 0;
01467     AC.lDefDim = AM.gDefDim = 4;
01468     AC.lDefDim4 = AM.gDefDim4 = 0;
01469     AC.lUnitTrace = AM.gUnitTrace = 4;
01470     AC.NamesFlag = AM.gNamesFlag = 0;
01471     AC.CodesFlag = AM.gCodesFlag = 0;
01472     /* Printing a backtrace on crash is on by default for both normal and debug
01473        modes if FORM has been compiled with backtrace support. */
01474 #ifdef ENABLE_BACKTRACE
01475     AC.PrintBacktraceFlag = 1;
01476 #else
01477     AC.PrintBacktraceFlag = 0;
01478 #endif
01479     /* Human-readable statistics are off by default */
01480     AC.HumanStatsFlag = 0;
01481     AC.extrasymbols = AM.gextrasymbols = AM.ggextrasymbols = 0;
01482     AC.extrasym = (UBYTE *)Malloc1(2*sizeof(UBYTE),"extrasym");
01483     AM.gextrasym = (UBYTE *)Malloc1(2*sizeof(UBYTE),"extrasym");
01484     AM.ggextrasym = (UBYTE *)Malloc1(2*sizeof(UBYTE),"extrasym");
01485     AC.extrasym[0] = AM.gextrasym[0] = AM.ggextrasym[0] = 'Z';
01486     AC.extrasym[1] = AM.gextrasym[1] = AM.ggextrasym[1] = 0;
01487     AC.TokensWriteFlag = AM.gTokensWriteFlag = 0;
01488     AC.SetupFlag = 0;
01489     AC.LineLength = AM.gLineLength = 79;
01490     AC.NwildC = 0;
01491     AC.OutputMode = 0;
01492     AM.gOutputMode = 0;
01493     AC.OutputSpaces = NORMALFORMAT;
01494     AM.gOutputSpaces = NORMALFORMAT;
01495     AC.OutNumberType = RATIONALMODE;
01496     AM.gOutNumberType = RATIONALMODE;
01497 #ifdef WITHZLIB
01498     AR.gzipCompress = GZIPDEFAULT;
01499     AR.FoStage4[0].ziobuffer = 0;
01500     AR.FoStage4[1].ziobuffer = 0;
01501 #ifdef WITHZSTD
01502     /* Zstd compression is on by default, if we have compiled with it */
01503     ZWRAP_useZSTDcompression(1);
01504 #endif
01505 #endif
01506     AR.BracketOn = 0;
01507     AC.bracketindexflag = 0;
01508     AT.bracketindexflag = 0;
01509     AT.bracketinfo = 0;
01510     AO.IsBracket = 0;
01511     AM.gfunpowers = AC.funpowers = COMFUNPOWERS;
01512     AC.parallelflag = AM.gparallelflag;
01513     AC.properorderflag = AM.gproperorderflag = PROPERORDERFLAG;
01514     AC.ProcessBucketSize = AC.mProcessBucketSize = AM.gProcessBucketSize;
01515     AC.ThreadBucketSize = AM.gThreadBucketSize;
01516     AC.ShortStatsMax = 0;
01517     AM.gShortStatsMax = 0;
01518     AM.ggShortStatsMax = 0;
01519
01520     GlobalSymbols     = NumSymbols;
01521     GlobalIndices     = NumIndices;
01522     GlobalVectors     = NumVectors;
01523     GlobalFunctions   = NumFunctions;
01524     GlobalSets        = NumSets;
01525     GlobalSetElements = NumSetElements;
01526     AC.modpowers = (UWORD *)0;
01527
01528     i = AM.OffsetIndex;
01529     fi = AC.FixIndices;
01530     if ( i > 0 ) do { *fi++ = one; } while ( --i >= 0 );
01531     AR.sLevel = -1;
01532     AM.Ordering[0] = 5;
01533     AM.Ordering[1] = 6;
01534     AM.Ordering[2] = 7;
01535     AM.Ordering[3] = 0;
01536     AM.Ordering[4] = 1;
01537     AM.Ordering[5] = 2;
01538     AM.Ordering[6] = 3;
01539     AM.Ordering[7] = 4;
01540     for ( i = 8; i < 15; i++ ) AM.Ordering[i] = i;
01541     AM.gUniTrace[0] =
01542     AC.lUniTrace[0] = SNUMBER;
01543     AM.gUniTrace[1] =
01544     AC.lUniTrace[1] =
01545     AM.gUniTrace[2] =
01546     AC.lUniTrace[2] = 4;
```

```
01547     AM.gUniTrace[3] =
01548     AC.lUniTrace[3] = 1;
01549 #ifdef WITHPTHREADS
01550     AS.Balancing = 0;
01551 #else
01552     AT.MinVecArg[0] = 7+ARGHEAD;
01553     AT.MinVecArg[ARGHEAD] = 7;
01554     AT.MinVecArg[1+ARGHEAD] = INDEX;
01555     AT.MinVecArg[2+ARGHEAD] = 3;
01556     AT.MinVecArg[3+ARGHEAD] = 0;
01557     AT.MinVecArg[4+ARGHEAD] = 1;
01558     AT.MinVecArg[5+ARGHEAD] = 1;
01559     AT.MinVecArg[6+ARGHEAD] = -3;
01560     t = AT.FunArg;
01561     *t++ = 4+ARGHEAD+FUNHEAD;
01562     for ( i = 1; i < ARGHEAD; i++ ) *t++ = 0;
01563     *t++ = 4+FUNHEAD;
01564     *t++ = 0;
01565     *t++ = FUNHEAD;
01566     for ( i = 2; i < FUNHEAD; i++ ) *t++ = 0;
01567     *t++ = 1; *t++ = 1; *t++ = 3;
01568
01569 #ifdef WITHMPI
01570     AS.printflag = 0;
01571 #endif
01572
01573     AT.comsym[0] = 8;
01574     AT.comsym[1] = SYMBOL;
01575     AT.comsym[2] = 4;
01576     AT.comsym[3] = 0;
01577     AT.comsym[4] = 1;
01578     AT.comsym[5] = 1;
01579     AT.comsym[6] = 1;
01580     AT.comsym[7] = 3;
01581     AT.comnum[0] = 4;
01582     AT.comnum[1] = 1;
01583     AT.comnum[2] = 1;
01584     AT.comnum[3] = 3;
01585     AT.comfun[0] = FUNHEAD+4;
01586     AT.comfun[1] = FUNCTION;
01587     AT.comfun[2] = FUNHEAD;
01588     AT.comfun[3] = 0;
01589 #if FUNHEAD == 4
01590     AT.comfun[4] = 0;
01591 #endif
01592     AT.comfun[FUNHEAD+1] = 1;
01593     AT.comfun[FUNHEAD+2] = 1;
01594     AT.comfun[FUNHEAD+3] = 3;
01595     AT.comind[0] = 7;
01596     AT.comind[1] = INDEX;
01597     AT.comind[2] = 3;
01598     AT.comind[3] = 0;
01599     AT.comind[4] = 1;
01600     AT.comind[5] = 1;
01601     AT.comind[6] = 3;
01602     AT.locwildvalue[0] = SUBEXPRESSION;
01603     AT.locwildvalue[1] = SUBEXPSIZE;
01604     for ( i = 2; i < SUBEXPSIZE; i++ ) AT.locwildvalue[i] = 0;
01605     AT.mulpat[0] = TYPEMULT;
01606     AT.mulpat[1] = SUBEXPSIZE+3;
01607     AT.mulpat[2] = 0;
01608     AT.mulpat[3] = SUBEXPRESSION;
01609     AT.mulpat[4] = SUBEXPSIZE;
01610     AT.mulpat[5] = 0;
01611     AT.mulpat[6] = 1;
01612     for ( i = 7; i < SUBEXPSIZE+5; i++ ) AT.mulpat[i] = 0;
01613     AT.proexp[0] = SUBEXPSIZE+4;
01614     AT.proexp[1] = EXPRESSION;
01615     AT.proexp[2] = SUBEXPSIZE;
01616     AT.proexp[3] = -1;
01617     AT.proexp[4] = 1;
01618     for ( i = 5; i < SUBEXPSIZE+1; i++ ) AT.proexp[i] = 0;
01619     AT.proexp[SUBEXPSIZE+1] = 1;
01620     AT.proexp[SUBEXPSIZE+2] = 1;
01621     AT.proexp[SUBEXPSIZE+3] = 3;
01622     AT.proexp[SUBEXPSIZE+4] = 0;
01623     AT.dummysubexp[0] = SUBEXPRESSION;
01624     AT.dummysubexp[1] = SUBEXPSIZE+4;
01625     for ( i = 2; i < SUBEXPSIZE; i++ ) AT.dummysubexp[i] = 0;
01626     AT.dummysubexp[SUBEXPSIZE] = WILDDUMMY;
01627     AT.dummysubexp[SUBEXPSIZE+1] = 4;
01628     AT.dummysubexp[SUBEXPSIZE+2] = 0;
01629     AT.dummysubexp[SUBEXPSIZE+3] = 0;
01630
01631     AT.inprimelist = -1;
01632     AT.sizeprimelist = 0;
01633     AT.primelist = 0;
```

```
01634     AT.LeaveNegative = 0;
01635     AT.TrimPower = 0;
01636     AN.SplitScratch = 0;
01637     AN.SplitScratchSize = AN.InScratch = 0;
01638     AN.SplitScratch1 = 0;
01639     AN.SplitScratchSize1 = AN.InScratch1 = 0;
01640     AN.idfunctionflag = 0;
01641 #endif
01642     AO.OutputLine = AO.OutFill = BufferForOutput;
01643     AO.FactorMode = 0;
01644     C->Pointer = C->Buffer;
01645
01646     AP.PreOut = 0;
01647     AP.ComChar = AP.cComChar;
01648     AC.cbufnum = AM.rbufnum;        /* Select the default compiler buffer */
01649     AC.HideLevel = 0;
01650     AP.PreAssignFlag = 0;
01651 }
01652
01653 /*
01654         #] IniVars :
01655         #[ Signal handlers :
01656 */
01657 /*[28apr2004 mt]:*/
01658 #ifdef TRAPSIGNALS
01659
01660 static int exitInProgress = 0;
01661 static int trappedTerminate = 0;
01662
01663 /*INTSIGHANDLER : some systems require a signal handler to return an integer,
01664   so define the macro INTSIGHANDLER if compiler fails:*/
01665 #ifdef INTSIGHANDLER
01666 static int onErrSig(int i)
01667 #else
01668 static void onErrSig(int i)
01669 #endif
01670 {
01671     if (exitInProgress){
01672         signal(i,SIG_DFL);/* Use default behaviour*/
01673         raise (i);/*reproduce trapped signal*/
01674 #ifdef INTSIGHANDLER
01675         return(i);
01676 #else
01677         return;
01678 #endif
01679     }
01680     trappedTerminate = 1;
01681     /*[13jul2005 mt]*//*TerminateImpl(-1) on signal is here:*/
01682     Terminate(-1);
01683 }
01684
01685 #ifdef INTSIGHANDLER
01686 static void setNewSig(int i, int (*handler)(int))
01687 #else
01688 static void setNewSig(int i, void (*handler)(int))
01689 #endif
01690 {
01691     if(! (i<NSIG) )/* Invalid signal -- see comments in the file */
01692         return;
01693     if ( signal(i,SIG_IGN) !=SIG_IGN)
01694     /* if compiler fails here, try to define INTSIGHANDLER):*/
01695         signal(i,handler);
01696 }
01697
01698 void setSignalHandlers(void)
01699 {
01700     /* Reset various unrecoverable error signals:*/
01701     setNewSig(SIGSEGV,onErrSig);
01702     setNewSig(SIGFPE,onErrSig);
01703     setNewSig(SIGILL,onErrSig);
01704     setNewSig(SIGEMT,onErrSig);
01705     setNewSig(SIGSYS,onErrSig);
01706     setNewSig(SIGPIPE,onErrSig);
01707     setNewSig(SIGLOST,onErrSig);
01708     setNewSig(SIGXCPU,onErrSig);
01709     setNewSig(SIGXFSZ,onErrSig);
01710
01711     /* Reset interrupt signals:*/
01712     setNewSig(SIGTERM,onErrSig);
01713     setNewSig(SIGINT,onErrSig);
01714     setNewSig(SIGQUIT,onErrSig);
01715     setNewSig(SIGHUP,onErrSig);
01716     setNewSig(SIGALRM,onErrSig);
01717     setNewSig(SIGVTALRM,onErrSig);
01718 /*  setNewSig(SIGPROF,onErrSig); */  /* Why did Tentukov forbid profilers?? */
01719 }
01720
```

```
01721 #endif
01722 /*:[28apr2004 mt]*/
01723 /*
01724         #] Signal handlers :
01725         #[ main :
01726 */
01727
01728 #ifdef WITHPTHREADS
01729 ALLPRIVATES *ABdummy[10];
01730 #endif
01731
01732 int main(int argc, char **argv)
01733 {
01734     int retval;
01735     bzero((void *)(&A),sizeof(A)); /* make sure A is initialized at zero */
01736     iniTools();
01737 #ifdef TRAPSIGNALS
01738     setSignalHandlers();
01739 #endif
01740 #ifdef WINDOWS
01741     _setmode(_fileno(stdout),O_BINARY);
01742 #endif
01743
01744 #ifdef WITHPTHREADS
01745     AB = ABdummy;
01746     StartHandleLock();
01747     BeginIdentities();
01748 #else
01749     AM.SumTime = TimeCPU(0);
01750     TimeWallClock(0);
01751 #endif
01752
01753 #ifdef WITHMPI
01754     if ( PF_Init(&argc,&argv) ) exit(-1);
01755 #endif
01756
01757     StartFiles();
01758     StartVariables();
01759 #ifdef WITHMPI
01760     /*
01761      * Here MesPrint() is ready. We turn on AS.printflag to print possible
01762      * errors occurring on slaves in the initialization. With AS.printflag = -1
01763      * MesPrint() does not use the synchronized output. This may lead broken
01764      * texts in the output somewhat, but it is safer to implement in this way
01765      * for the situation in which some of MesPrint() calls use MLOCK()-MUNLOCK()
01766      * and some do not. In future if we set AS.printflag = 1 and modify the
01767      * source code such that all MesPrint() calls are sandwiched by MLOCK()-
01768      * MUNLOCK(), we need also to modify the code for the master to catch
01769      * messages corresponding to MUNLOCK() calls at some point.
01770      *
01771      * AS.printflag will be set to 0 in IniVars() to prevent slaves from
01772      * printing redundant errors in the preprocessor and compiler (e.g., syntax
01773      * errors).
01774      */
01775     AS.printflag = -1;
01776 #endif
01777
01778     if ( ( retval = DoTail(argc,(UBYTE **)argv) ) != 0 ) {
01779         if ( retval > 0 ) Terminate(0);
01780         else              Terminate(-1);
01781     }
01782     if ( DoSetups() ) Terminate(-2);
01783 #ifdef WITHMPI
01784     /* It is messy if all errors in OpenInput() on slaves are printed. */
01785     AS.printflag = 0;
01786 #endif
01787     if ( OpenInput() ) Terminate(-3);
01788 #ifdef WITHMPI
01789     AS.printflag = -1;
01790 #endif
01791     if ( TryEnvironment() ) Terminate(-2);
01792     if ( TryFileSetups() ) Terminate(-2);
01793     if ( MakeSetupAllocs() ) Terminate(-2);
01794     StartMore();
01795     InitRecovery();
01796     CheckRecoveryFile();
01797     if ( AM.totalnumberofthreads == 0 ) AM.totalnumberofthreads = 1;
01798     AS.MultiThreaded = 0;
01799 #ifdef WITHPTHREADS
01800     if ( AM.totalnumberofthreads > 1 ) AS.MultiThreaded = 1;
01801     ReserveTempFiles(1);
01802     StartAllThreads(AM.totalnumberofthreads);
01803     IniFbufs();
01804 #else
01805     ReserveTempFiles(0);
01806     IniFbuffer(AT.fbufnum);
01807 #endif
```

```
01808     if ( !AM.FromStdin ) PrintHeader(1);
01809     IniVars();
01810     Globalize(1);
01811 #ifdef WITH_ALARM
01812     if ( AM.TimeLimit > 0 ) alarm(AM.TimeLimit);
01813 #endif
01814 #ifdef WITHFLINT
01815     flint_check_version();
01816 #endif
01817     TimeCPU(0);
01818     TimeChildren(0);
01819     TimeWallClock(0);
01820     PreProcessor();
01821     Terminate(0);
01822     return(0);
01823 }
01824 /*
01825         #] main :
01826         #[ CleanUp :
01827 
01828         if par < 0 we have to keep the storage file.
01829         when par > 0 we ran into a .clear statement.
01830         In that case we keep the zero level input and the log file.
01831 
01832 */
01833 
01834 void CleanUp(WORD par)
01835 {
01836     GETIDENTITY
01837     int i;
01838 
01839     if ( FG.fname ) {
01840 #ifdef WITHPTHREADS
01841     if ( B ) {
01842 #endif
01843     CleanUpSort(0);
01844     for ( i = 0; i < 3; i++ ) {
01845         if ( AR.Fscr[i].handle >= 0 ) {
01846             if ( AR.Fscr[i].name ) {
01847 /*
01848                 If there are more threads referring to the same file
01849                 only the one with the name is the owner of the file.
01850 */
01851                 CloseFile(AR.Fscr[i].handle);
01852                 remove(AR.Fscr[i].name);
01853             }
01854             AR.Fscr[i].handle = - 1;
01855             AR.Fscr[i].POfill = 0;
01856         }
01857     }
01858 #ifdef WITHPTHREADS
01859     }
01860 #endif
01861     if ( par > 0 ) {
01862 /*
01863     Close all input levels above the lowest?
01864 */
01865     }
01866     if ( AC.StoreHandle >= 0 && par <= 0 ) {
01867 #ifdef TRAPSIGNALS
01868         if ( trappedTerminate ) { /* We don't throw .str if it has contents */
01869             POSITION pos;
01870             PUTZERO(pos);
01871             SeekFile(AC.StoreHandle,&pos,SEEK_END);
01872             if ( ISNOTZEROPOS(pos) ) {
01873                 CloseFile(AC.StoreHandle);
01874                 goto dontremove;
01875             }
01876         }
01877         CloseFile(AC.StoreHandle);
01878 #ifdef WITHPTHREADS
01879     if ( B )
01880 #endif
01881         if ( par >= 0 || AR.StoreData.Handle < 0 || AM.ClearStore ) {
01882             remove(FG.fname);
01883         }
01884 dontremove:;
01885 #else
01886         CloseFile(AC.StoreHandle);
01887 #ifdef WITHPTHREADS
01888     if ( B )
01889 #endif
01890         if ( par >= 0 || AR.StoreData.Handle < 0 || AM.ClearStore > 0 ) {
01891             remove(FG.fname);
01892         }
01893 #endif
01894     }
```

```
01895     }
01896     ClearSpectators(CLEARMODULE);
01897 /*
01898     Remove recovery file on exit if everything went well
01899 */
01900     if ( par == 0 ) {
01901         DeleteRecoveryFile();
01902     }
01903 /*
01904     Now the final message concerning the total time
01905 */
01906     if ( AC.LogHandle >= 0 && par <= 0 ) {
01907         WORD lh = AC.LogHandle;
01908         AC.LogHandle = -1;
01909 #ifdef WITHMPI
01910         if ( PF.me == MASTER )  /* Only the master opened the real file. */
01911 #endif
01912         CloseFile(lh);
01913     }
01914 }
01915
01916 /*
01917         #] CleanUp :
01918         #[ TerminateImpl :
01919 */
01920
01921 static int firstterminate = 1;
01922
01923 void TerminateImpl(int errorcode, const char* file, int line, const char* function)
01924 {
01925 #ifdef WITHMPI
01926     if ( errorcode && firstterminate && !PF.notMyFault ) {
01927 #else
01928     if ( errorcode && firstterminate ) {
01929 #endif
01930         firstterminate = 0;
01931
01932         MLOCK(ErrorMessageLock);
01933 #ifdef WITHPTHREADS
01934         MesPrint("Program terminating in thread %w at &");
01935 #elif defined(WITHMPI)
01936         MesPrint("Program terminating in process %w at &");
01937 #else
01938         MesPrint("Program terminating at &");
01939 #endif
01940         MesPrint("Terminate called from %s:%d (%s)", file, line, function);
01941
01942         if ( AC.PrintBacktraceFlag ) {
01943 #ifdef ENABLE_BACKTRACE
01944             void *stack[64];
01945             int stacksize, stop = 0;
01946             stacksize = backtrace(stack, sizeof(stack)/sizeof(stack[0]));
01947
01948             /* First check whether eu-addr2line is available */
01949             if ( !system("command -v eu-addr2line > /dev/null 2>&1") ) {
01950                 MesPrint("Backtrace:");
01951                 for (int i = 0; i < stacksize && !stop; i++) {
01952                     FILE *fp;
01953                     char cmd[512];
01954                     // Leave an initial space
01955                     cmd[0] = ' ';
01956                     MesPrint("%#%2d:%", i);
01957                     snprintf(cmd+1, sizeof(cmd)-1, "eu-addr2line -s --pretty-print -f -i '%p'
      --pid=%d\n", stack[i], getpid());
01958                     fp = popen(cmd+1, "r");
01959                     while ( fgets(cmd+1, sizeof(cmd)-1, fp) != NULL ) {
01960                         MesPrint("%s", cmd);
01961                         /* Don't show functions lower than "main" (or thread equivalent) */
01962                         if ( strstr(cmd, " main ") || strstr(cmd, " RunThread ") || strstr(cmd, "
      RunSortBot ") ) {
01963                             stop = 1;
01964                         }
01965                     }
01966                     pclose(fp);
01967                 }
01968             }
01969 #ifdef LINUX
01970             else if ( !system("command -v addr2line > /dev/null 2>&1") ) {
01971                 /* Get the executable path. */
01972                 char exe_path[PATH_MAX];
01973                 {
01974                     ssize_t len = readlink("/proc/self/exe", exe_path, sizeof(exe_path) - 1);
01975                     if ( len != -1 ) {
01976                         exe_path[len] = '\0';
01977                     }
01978                     else {
01979                         goto backtrace_fallback;
```

```
01980                   }
01981               }
01982               /* Assume PIE binary and get the base address. */
01983               uintptr_t base_address = 0;
01984               {
01985                   char line[256];
01986                   FILE *maps = fopen("/proc/self/maps", "r");
01987                   if ( !maps ) {
01988                       goto backtrace_fallback;
01989                   }
01990                   /* See the format used by nommu_region_show() in fs/proc/nommu.c of the Linux source. */
01991                   if ( fgets(line, sizeof(line), maps) ) {
01992                       sscanf(line, "%" SCNxPTR "-", &base_address);
01993                   }
01994                   else {
01995                       fclose(maps);
01996                       goto backtrace_fallback;
01997                   }
01998                   fclose(maps);
01999               }
02000               char **strings;
02001               strings = backtrace_symbols(stack, stacksize);
02002               MesPrint("Backtrace:");
02003               for ( int i = 0; i < stacksize && !stop; i++ ) {
02004                   FILE *fp;
02005                   char cmd[PATH_MAX + 512];
02006                   // Leave an initial space
02007                   cmd[0] = ' ';
02008                   uintptr_t addr = (uintptr_t)stack[i] - base_address;
02009                   MesPrint("%#%2d:%", i);
02010                   snprintf(cmd+1, sizeof(cmd)-1, "addr2line -e \"%s\" -i -p -s -f -C 0x%" PRIxPTR, exe_path, addr);
02011                   fp = popen(cmd+1, "r");
02012                   while ( fgets(cmd+1, sizeof(cmd)-1, fp) != NULL ) {
02013                       MesPrint("%s", cmd);
02014                       /* Don't show functions lower than "main" */
02015                       if ( strstr(cmd, " main ") || strstr(cmd, " RunThread ") || strstr(cmd, " RunSortBot ") ) {
02016                           stop = 1;
02017                       }
02018                   }
02019                   pclose(fp);
02020               }
02021               free(strings);
02022           }
02023 #endif
02024           else {
02025               /* eu-addr2line not found */
02026 #ifdef LINUX
02027 backtrace_fallback: ;
02028 #endif
02029               char **strings;
02030               strings = backtrace_symbols(stack, stacksize);
02031               MesPrint("Backtrace:");
02032               for ( int i = 0; i < stacksize && !stop; i++ ) {
02033                   char *p = strings[i];
02034                   while ( *p && *p != '(' ) p++;
02035                   MesPrint("%#%2d: %s\n", i, p);
02036                   /* Don't show functions lower than "main" (or thread equivalent) */
02037                   if ( strstr(p, "(main+") || strstr(p, "(RunThread+") || strstr(p, "(RunSortBot+") ) {
02038                       stop = 1;
02039                   }
02040               }
02041 #ifdef LINUX
02042               MesPrint("Please install addr2line or eu-addr2line for readable stack information.");
02043 #else
02044               MesPrint("Please install eu-addr2line for readable stack information.");
02045 #endif
02046               free(strings);
02047           }
02048 #else
02049           MesPrint("FORM compiled without backtrace support.");
02050 #endif
02051       } /* if ( AC.PrintBacktraceFlag) { */
02052
02053       MUNLOCK(ErrorMessageLock);
02054
02055 #ifdef WITHMPI
02056       PF_PreTerminate(errorcode);
02057 #endif
02058       Crash();
02059   }
02060 #ifdef TRAPSIGNALS
02061   exitInProgress=1;
02062 #endif
```

```
02063 #ifdef WITHEXTERNALCHANNEL
02064 /*
02065     This function can be called from the error handler, so it is better to
02066     clean up all started processes before any activity:
02067 */
02068     closeAllExternalChannels();
02069     AX.currentExternalChannel=0;
02070     /*[08may2006 mt]:*/
02071     AX.killSignal=SIGKILL;
02072     AX.killWholeGroup=1;
02073     AX.daemonize=1;
02074     /*:[08may2006 mt]*/
02075     if(AX.currentPrompt){
02076         M_free(AX.currentPrompt,"external channel prompt");
02077         AX.currentPrompt=0;
02078     }
02079     /*[08may2006 mt]:*/
02080     if(AX.shellname){
02081         M_free(AX.shellname,"external channel shellname");
02082         AX.shellname=0;
02083     }
02084     if(AX.stderrname){
02085         M_free(AX.stderrname,"external channel stderrname");
02086         AX.stderrname=0;
02087     }
02088     /*:[08may2006 mt]*/
02089 #endif
02090 #ifdef WITHPTHREADS
02091     if ( !WhoAmI() && !errorcode ) {
02092         TerminateAllThreads();
02093     }
02094 #endif
02095     if ( AC.FinalStats ) {
02096         if ( AM.PrintTotalSize ) {
02097             MesPrint("Max. space for expressions: %19p bytes",&(AS.MaxExprSize));
02098         }
02099         PrintRunningTime();
02100     }
02101 #ifdef WITHMPI
02102     if ( AM.HoldFlag && PF.me == MASTER ) {
02103         WriteFile(AM.StdOut,(UBYTE *)("Hit any key "),12);
02104         PF_FlushStdOutBuffer();
02105         getchar();
02106     }
02107 #else
02108     if ( AM.HoldFlag ) {
02109         WriteFile(AM.StdOut,(UBYTE *)("Hit any key "),12);
02110         getchar();
02111     }
02112 #endif
02113 #ifdef WITHMPI
02114     PF_Terminate(errorcode);
02115 #endif
02116 /*
02117     We are about to terminate the program. If we are using flint, call the cleanup function.
02118     This keeps valgrind happy.
02119 */
02120 #ifdef WITHFLINT
02121     flint_final_cleanup_master();
02122 #endif
02123     CleanUp(errorcode);
02124     M_print();
02125 #ifdef VMS
02126     P_term(errorcode? 0: 1);
02127 #else
02128     P_term(errorcode);
02129 #endif
02130 }
02131
02132 /*
02133         #] TerminateImpl :
02134         #[ PrintDeprecation :
02135 */
02136
02143 void PrintDeprecation(const char *feature, const char *issue) {
02144 #ifdef WITHMPI
02145     if ( PF.me != MASTER ) return;
02146 #endif
02147     if ( AM.IgnoreDeprecation ) return;
02148
02149     UBYTE *e = (UBYTE *)getenv("FORM_IGNORE_DEPRECATION");
02150     if ( e && e[0] && StrCmp(e, (UBYTE *)"0") && StrICmp(e, (UBYTE *)"false") && StrICmp(e, (UBYTE
      *)"no") ) return;
02151
02152     MesPrint("DeprecationWarning: We are considering deprecating %s.", feature);
02153     MesPrint("If you want this support to continue, leave a comment at:");
02154     MesPrint("");
```

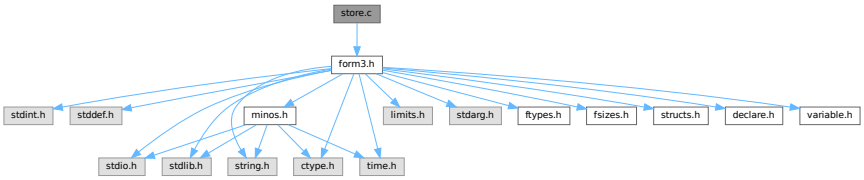
```
02155     MesPrint("    https://github.com/form-dev/form/%s", issue);
02156     MesPrint("");
02157     MesPrint("Otherwise, it will be discontinued in the future.");
02158     MesPrint("To suppress this warning, use the -ignore-deprecation command line option or");
02159     MesPrint("set the environment variable FORM_IGNORE_DEPRECATION=1.");
02160 }
02161
02162 /*
02163         #] PrintDeprecation :
02164         #[ PrintRunningTime :
02165 */
02166
02167 void PrintRunningTime(void)
02168 {
02169 #if (defined(WITHPTHREADS) && (defined(WITHPOSIXCLOCK) || defined(WINDOWS))) || defined(WITHMPI)
02170     LONG mastertime;
02171     LONG workertime;
02172     LONG wallclocktime;
02173     LONG totaltime;
02174 #if defined(WITHPTHREADS)
02175     if ( AB[0] != 0 ) {
02176         workertime = GetWorkerTimes();
02177 #else
02178     workertime = PF_GetSlaveTimes();  /* must be called on all processors */
02179     if ( PF.me == MASTER ) {
02180 #endif
02181         mastertime = AM.SumTime + TimeCPU(1);
02182         wallclocktime = TimeWallClock(1);
02183         totaltime = mastertime+workertime;
02184         if ( !AM.silent ) {
02185         MesPrint("  %l.%2i sec + %l.%2i sec: %l.%2i sec out of %l.%2i sec",
02186             mastertime/1000,(WORD)((mastertime%1000)/10),
02187             workertime/1000,(WORD)((workertime%1000)/10),
02188             totaltime/1000,(WORD)((totaltime%1000)/10),
02189             wallclocktime/100,(WORD)(wallclocktime%100));
02190         }
02191     }
02192 #else
02193     LONG mastertime = AM.SumTime + TimeCPU(1);
02194     LONG wallclocktime = TimeWallClock(1);
02195     if ( !AM.silent ) {
02196     MesPrint("  %l.%2i sec out of %l.%2i sec",
02197         mastertime/1000,(WORD)((mastertime%1000)/10),
02198         wallclocktime/100,(WORD)(wallclocktime%100));
02199     }
02200 #endif
02201 }
02202
02203 /*
02204         #] PrintRunningTime :
02205         #[ GetRunningTime :
02206 */
02207
02208 LONG GetRunningTime(void)
02209 {
02210 #if defined(WITHPTHREADS) && (defined(WITHPOSIXCLOCK) || defined(WINDOWS))
02211     LONG mastertime;
02212     if ( AB[0] != 0 ) {
02213 /*
02214 #if ( defined(APPLE64) || defined(APPLE32) )
02215         mastertime = AM.SumTime + TimeCPU(1);
02216         return(mastertime);
02217 #else
02218 */
02219         LONG workertime = GetWorkerTimes();
02220         mastertime = AM.SumTime + TimeCPU(1);
02221         return(mastertime+workertime);
02222 /*
02223 #endif
02224 */
02225     }
02226     else {
02227         return(AM.SumTime + TimeCPU(1));
02228     }
02229 #elif defined(WITHMPI)
02230     LONG mastertime, t = 0;
02231     LONG workertime = PF_GetSlaveTimes();  /* must be called on all processors */
02232     if ( PF.me == MASTER ) {
02233         mastertime = AM.SumTime + TimeCPU(1);
02234         t = mastertime + workertime;
02235     }
02236     return PF_BroadcastNumber(t);  /* must be called on all processors */
02237 #else
02238     return(AM.SumTime + TimeCPU(1));
02239 #endif
02240 }
02241
```

```
02242 /*
02243         #] GetRunningTime :
02244 */
```

4.137 store.c File Reference

```
#include "form3.h"
```

Include dependency graph for store.c:



Macros

- #define SAVEREVISION 0x02

Functions

- void OpenTemp (void)
- void SeekScratch (FILEHANDLE *fi, POSITION *pos)
- void SetEndScratch (FILEHANDLE *f, POSITION *position)
- void SetEndHScratch (FILEHANDLE *f, POSITION *position)
- void SetScratch (FILEHANDLE *f, POSITION *position)
- int RevertScratch (void)
- int ResetScratch (void)
- int ReadFromScratch (FILEHANDLE *fi, POSITION *pos, UBYTE *buffer, POSITION *length)
- int AddToScratch (FILEHANDLE *fi, POSITION *pos, UBYTE *buffer, POSITION *length, int withflush)
- int CoSave (UBYTE *inp)
- int CoLoad (UBYTE *inp)
- int DeleteStore (WORD par)
- int PutInStore (INDEXENTRY *ind, WORD num)
- WORD GetTerm (PHEAD WORD *term)
- WORD GetOneTerm (PHEAD WORD *term, FILEHANDLE *fi, POSITION *pos, int par)
- WORD GetMoreTerms (WORD *term)
- int GetMoreFromMem (WORD *term, WORD **tpoin)
- WORD GetFromStore (WORD *to, POSITION *position, RENUMBER renumber, WORD *InCompState, WORD nexpr)
- void DetVars (WORD *term, WORD par)
- int ToStorage (EXPRESSIONS e, POSITION *length)
- INDEXENTRY * NextFileIndex (POSITION *indexpos)
- int SetFileIndex (void)
- int VarStore (UBYTE *s, WORD n, WORD name, WORD namesize)
- int TermRenumber (WORD *term, RENUMBER renumber, WORD nexpr)
- WORD FindrNumber (WORD n, VARRENUM *v)
- INDEXENTRY * FindInIndex (WORD expr, FILEDATA *f, WORD par, WORD mode)

- RENUMBER GetTable (WORD expr, POSITION *position, WORD mode)
- int CopyExpression (FILEHANDLE *from, FILEHANDLE *to)
- LONG WriteStoreHeader (WORD handle)
- int ReadSaveHeader (void)
- LONG ReadSaveIndex (FILEINDEX *fileind)
- LONG ReadSaveVariables (UBYTE *buffer, UBYTE *top, LONG *size, LONG *outsize, INDEXENTRY *ind, LONG *stage)
- UBYTE * ReadSaveTerm32 (UBYTE *bin, UBYTE *binend, UBYTE **bout, UBYTE *boutend, UBYTE *top, int terminbuf)
- LONG ReadSaveExpression (UBYTE *buffer, UBYTE *top, LONG *size, LONG *outsize)

4.137.1 Detailed Description

Contains all functions that deal with store-files and the system independent save-files.

Definition in file store.c.

4.137.2 Macro Definition Documentation

4.137.2.1 SAVEREVISION

```
#define SAVEREVISION 0x02
```

Definition at line 4029 of file store.c.

4.137.3 Function Documentation

4.137.3.1 OpenTemp()

```
void OpenTemp (
            void  )
```

Definition at line 48 of file store.c.

4.137.3.2 SeekScratch()

```
void SeekScratch (
            FILEHANDLE * fi,
            POSITION * pos )
```

Definition at line 63 of file store.c.

4.137.3.3 SetEndScratch()

```
void SetEndScratch (
            FILEHANDLE * f,
            POSITION * position )
```

Definition at line 74 of file store.c.

4.137.3.4 SetEndHScratch()

```
void SetEndHScratch (
            FILEHANDLE * f,
            POSITION * position )
```

Definition at line 88 of file store.c.

4.137.3.5 SetScratch()

```
void SetScratch (
            FILEHANDLE * f,
            POSITION * position )
```

Definition at line 114 of file store.c.

4.137.3.6 RevertScratch()

```
int RevertScratch (
            void  )
```

Definition at line 195 of file store.c.

4.137.3.7 ResetScratch()

```
int ResetScratch (
            void  )
```

Definition at line 244 of file store.c.

4.137.3.8 ReadFromScratch()

```
int ReadFromScratch (
            FILEHANDLE * fi,
            POSITION * pos,
            UBYTE * buffer,
            POSITION * length )
```

Definition at line 286 of file store.c.

4.137.3.9 AddToScratch()

```
int AddToScratch (
            FILEHANDLE * fi,
            POSITION * pos,
            UBYTE * buffer,
            POSITION * length,
            int withflush )
```

Definition at line 315 of file store.c.

4.137.3.10 CoSave()

```
int CoSave (
            UBYTE * inp )
```

Definition at line 378 of file store.c.

4.137.3.11 CoLoad()

```
int CoLoad (
            UBYTE * inp )
```

Definition at line 594 of file store.c.

4.137.3.12 DeleteStore()

```
int DeleteStore (
            WORD par )
```

Definition at line 798 of file store.c.

4.137.3.13 PutInStore()

```
int PutInStore (
            INDEXENTRY * ind,
            WORD num )
```

Definition at line 899 of file store.c.

4.137.3.14 GetTerm()

```
WORD GetTerm (
            PHEAD WORD * term )
```

Definition at line 1020 of file store.c.

4.137.3.15 GetOneTerm()

```
WORD GetOneTerm (
            PHEAD WORD * term,
            FILEHANDLE * fi,
            POSITION * pos,
            int par )
```

Definition at line 1297 of file store.c.

4.137.3.16 GetMoreTerms()

```
WORD GetMoreTerms (
            WORD * term )
```

Definition at line 1457 of file store.c.

4.137.3.17 GetMoreFromMem()

```
int GetMoreFromMem (
            WORD * term,
            WORD ** tpoin )
```

Definition at line 1553 of file store.c.

4.137.3.18 GetFromStore()

```
WORD GetFromStore (
            WORD * to,
            POSITION * position,
            RENUMBER renumber,
            WORD * InCompState,
            WORD nexpr )
```

Definition at line 1660 of file store.c.

4.137.3.19 DetVars()

```
void DetVars (
            WORD * term,
            WORD par )
```

Definition at line 1869 of file store.c.

4.137.3.20 ToStorage()

```
int ToStorage (
            EXPRESSIONS e,
            POSITION * length )
```

Definition at line 2056 of file store.c.

4.137.3.21 NextFileIndex()

```
INDEXENTRY * NextFileIndex (
            POSITION * indexpos )
```

Definition at line 2315 of file store.c.

4.137.3.22 SetFileIndex()

```
int SetFileIndex (
            void  )
```

Reads the next file index and puts it into AR.StoreData.Index. TODO

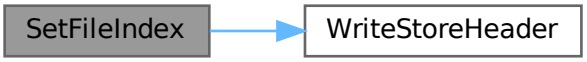
Returns

= 0 everything okay, != 0 an error occurred

Definition at line 2380 of file store.c.

References WriteStoreHeader().

Here is the call graph for this function:



4.137.3.23 VarStore()

```
int VarStore (
            UBYTE * s,
            WORD n,
            WORD name,
            WORD namesize )
```

Definition at line 2441 of file store.c.

4.137.3.24 TermRenumber()

```
int TermRenumber (
            WORD * term,
            RENUMBER renumber,
            WORD nexpr )
```

!! WORD *memterm=term; static LONG ctrap=0; !!!

!! ctrap++; !!!

Definition at line 2499 of file store.c.

References ReNuMbEr::func, ReNuMbEr::funnum, ReNuMbEr::indi, ReNuMbEr::indnum, ReNuMbEr::symb, ReNuMbEr::symnum, TermRenumber(), ReNuMbEr::vecnum, and ReNuMbEr::vect.

Referenced by TermRenumber().

Here is the call graph for this function:

4.137.3.25 FindrNumber()

```
WORD FindrNumber (
              WORD n,
              VARRENUM * v )
```

Definition at line 2659 of file store.c.

4.137.3.26 FindInIndex()

```
INDEXENTRY * FindInIndex (
              WORD expr,
              FILEDATA * f,
              WORD par,
              WORD mode )
```

Definition at line 2740 of file store.c.

4.137.3.27 GetTable()

```
RENUMBER GetTable (
              WORD expr,
              POSITION * position,
              WORD mode )
```

Definition at line 2893 of file store.c.

4.137.3.28 CopyExpression()

```
int CopyExpression (
              FILEHANDLE * from,
              FILEHANDLE * to )
```

Definition at line 3379 of file store.c.

4.137.3.29 WriteStoreHeader()

```
LONG WriteStoreHeader (
              WORD handle )
```

Writes header with information about system architecture and FORM revision to an open store file.

Called by SetFileIndex().

Parameters

handle	specifies open file to which header will be written

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 4040 of file store.c.

References STOREHEADER::endianness, STOREHEADER::lenLONG, STOREHEADER::lenPOINTER, STOREHEADER::lenPOS, STOREHEADER::lenWORD, STOREHEADER::maxpower, STOREHEADER::sFun, STOREHEADER::sInd, STOREHEADER::sSym, STOREHEADER::sVec, and STOREHEADER::wildoffset.

Referenced by SetFileIndex().

4.137.3.30 ReadSaveHeader()

```
int ReadSaveHeader (
            void  )
```

Reads the header in the save file and sets function pointers and flags according to the information found there. Must be called before any other ReadSave... function.

Currently works only for the exchange between 32bit and 64bit machines (WORD size must be 2 or 4 bytes)!

It is called by CoLoad().

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 4136 of file store.c.

4.137.3.31 ReadSaveIndex()

```
LONG ReadSaveIndex (
            FILEINDEX * fileind )
```

Reads a FILEINDEX from the open save file specified by AO.SaveData.Handle. Translations for adjusting endianness and data sizes are done if necessary.

Depends on the assumption that sizeof(FILEINDEX) is the same everywhere. If FILEINDEX or INDEXENTRY change, then this functions has to be adjusted.

Called by CoLoad() and FindInIndex().

Parameters

fileind	contains the read FILEINDEX after successful return. must point to allocated, big enough memory.

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 4257 of file store.c.

References INFILEINDEX, FiLeInDeX::next, FiLeInDeX::number, and FuNcTiOn::number.

4.137.3.32 ReadSaveVariables()

```
LONG ReadSaveVariables (
            UBYTE * buffer,
            UBYTE * top,
            LONG * size,
            LONG * outsize,
            INDEXENTRY * ind,
            LONG * stage )
```

Reads the variables from the open file specified by AO.SaveData.Handle. It reads the ∗size bytes and writes them to the ∗buffer. It is called by PutInStore().

If translation is necessary, the data might shrink or grow in size, then ∗*size* is adjusted so that the reading and writing fits into the memory from the buffer to the top. The actual number of read bytes is returned in ∗*size*, the number of written bytes is returned in ∗*outsize*.

If the ∗size is smaller than the actual size of the variables, this function will be called several times and needs to remember the current position in the variable structure. The parameter *stage* does this job. When ReadSaveVariables() is called for the first time, this parameter should have the value -1.

The parameter *ind* is used to get the number of variables.

Parameters

buffer	read variables are written into this allocated memory
top	upper end of allocated memory
size	number of bytes to read. might return a smaller number of read bytes if translation was necessary
outsize	if translation has be done, outsize contains the number of written bytes
ind	pointer of INDEXENTRY for the current expression. read-only
stage	should be -1 for the first call, will be increased by ReadSaveVariables to memorize the position in the variable structure

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 4445 of file store.c.

References InDeXeNtRy::nfunctions, InDeXeNtRy::nindices, InDeXeNtRy::nsymbols, InDeXeNtRy::nvectors, and FuNcTiOn::spec.

4.137.3.33 ReadSaveTerm32()

```
UBYTE * ReadSaveTerm32 (
             UBYTE * bin,
             UBYTE * binend,
             UBYTE ** bout,
             UBYTE * boutend,
             UBYTE * top,
             int terminbuf )
```

Reads a single term from the given buffer at *bin* and write the translated term back to this buffer at *bout*.

ReadSaveTerm32() is currently the only instantiation of a ReadSaveTerm-function. It only deals with data that already has the correct endianness and that is resized to 32bit words but without being renumbered or translated in any other way. It uses the compress buffer AR.CompressBuffer.

The function is reentrant in order to cope with nested function arguments. It is called by ReadSaveExpression() and itself.

The *return value* indicates the position in the input buffer up to which the data has already been successfully processed. The parameter *bout* returns the corresponding position in the output buffer.

Parameters

bin	start of the input buffer
binend	end of the input buffer
bout	as input points to the beginning of the output buffer, as output points behind the already translated data in the output buffer
boutend	end of already decompressed data in output buffer
top	end of output buffer
terminbuf	flag whether decompressed data is already in the output buffer. used in recursive calls

Returns

pointer to the next unprocessed data in the input buffer

Definition at line 4814 of file store.c.

References ReadSaveTerm32().

Referenced by ReadSaveExpression(), and ReadSaveTerm32().

Here is the call graph for this function:

4.137.3.34 ReadSaveExpression()

```
LONG ReadSaveExpression (
             UBYTE * buffer,
             UBYTE * top,
             LONG * size,
             LONG * outsize )
```

Reads an expression from the open file specified by AO.SaveData.Handle. The endianness flip and a resizing without renumbering is done in this function. Thereafter the buffer consists of chunks with a uniform maximal word size (32bit at the moment). The actual renumbering is then done by calling the function ReadSaveTerm32(). The result is returned in *buffer*.

If the translation at some point doesn't fit into the buffer anymore, the function returns and must be called again. In any case *size* returns the number of successfully read bytes, *outsize* returns the number of successfully written bytes, and the file will be positioned at the next byte after the successfully read data.

It is called by PutInStore().

Parameters

buffer	output buffer, holds the (translated) expression
top	end of buffer
size	number of read bytes
outsize	number of written bytes

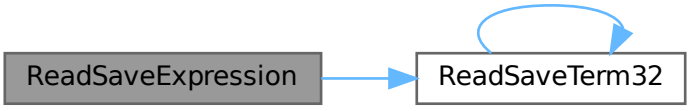
Returns

= 0 everything okay, != 0 an error occurred

Definition at line 5228 of file store.c.

References ReadSaveTerm32().

Here is the call graph for this function:



4.138 store.c

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
```

```
00007 /*
00008  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032 /*
00033 #define HIDEDEBUG
00034     #[ Includes : store.c
00035 */
00036
00037 #include "form3.h"
00038
00039 /*
00040     #] Includes :
00041     #[ StoreExpressions :
00042         #[ OpenTemp :
00043
00044         Opens the scratch files for the input -> output operations.
00045
00046 */
00047
00048 void OpenTemp(void)
00049 {
00050     GETIDENTITY
00051     if ( AR.outfile->handle >= 0 ) {
00052         SeekFile(AR.outfile->handle,&(AR.outfile->filesize),SEEK_SET);
00053         AR.outfile->POposition = AR.outfile->filesize;
00054         AR.outfile->POfill = AR.outfile->PObuffer;
00055     }
00056 }
00057
00058 /*
00059         #] OpenTemp :
00060         #[ SeekScratch :
00061 */
00062
00063 void SeekScratch(FILEHANDLE *fi, POSITION *pos)
00064 {
00065     *pos = fi->POposition;
00066     ADDPOS(*pos,(TOLONG(fi->POfill)-TOLONG(fi->PObuffer)));
00067 }
00068
00069 /*
00070         #] SeekScratch :
00071         #[ SetEndScratch :
00072 */
00073
00074 void SetEndScratch(FILEHANDLE *f, POSITION *position)
00075 {
00076     if ( f->handle < 0 ) {
00077         SETBASEPOSITION(*position,(f->POfull-f->PObuffer)*sizeof(WORD));
00078     }
00079     else *position = f->filesize;
00080     SetScratch(f,position);
00081 }
00082
00083 /*
00084         #] SetEndScratch :
00085         #[ SetEndHScratch :
00086 */
00087
00088 void SetEndHScratch(FILEHANDLE *f, POSITION *position)
00089 {
00090     if ( f->handle < 0 ) {
00091         SETBASEPOSITION(*position,(f->POfull-f->PObuffer)*sizeof(WORD));
00092         f->POfill = f->POfull;
00093     }
```

```
00094     else {
00095 #ifdef HIDEDEBUG
00096         POSITION possize;
00097         PUTZERO(possize);
00098         SeekFile(f->handle,&possize,SEEK_END);
00099         MesPrint("SetEndHScratch: filesize(th) = %12p, filesize(ex) = %12p",&(f->filesize),
00100                 &(possize));
00101 #endif
00102         *position = f->filesize;
00103         f->POposition = f->filesize;
00104         f->POfill = f->POfull = f->PObuffer;
00105     }
00106 /*  SetScratch(f,position); */
00107 }
00108
00109 /*
00110         #] SetEndHScratch :
00111         #[ SetScratch :
00112 */
00113
00114 void SetScratch(FILEHANDLE *f, POSITION *position)
00115 {
00116     GETIDENTITY
00117     POSITION possize;
00118     LONG size, *whichInInBuf;
00119     if ( f == AR.hidefile ) whichInInBuf = &(AR.InHiBuf);
00120     else                    whichInInBuf = &(AR.InInBuf);
00121 #ifdef HIDEDEBUG
00122     if ( f == AR.hidefile ) MesPrint("In the hide file: %s", f->name);
00123     else MesPrint("In the input file: %s", f->name);
00124     MesPrint("SetScratch to position %15p",position);
00125     MesPrint("POposition = %15p, full = %l, fill = %l"
00126         ,&(f->POposition),(f->POfull-f->PObuffer)*sizeof(WORD)
00127         ,(f->POfill-f->PObuffer)*sizeof(WORD));
00128 #endif
00129     if ( ISLESSPOS(*position,f->POposition) ||
00130     ISGEPOSINC(*position,f->POposition,(f->POfull-f->PObuffer)*sizeof(WORD)) ) {
00131         if ( f->handle < 0 ) {
00132             if ( ISEQUALPOSINC(*position,f->POposition,
00133                 (f->POfull-f->PObuffer)*sizeof(WORD)) ) goto endpos;
00134 /* INTERNAL_ERROR_EXCL_START */
00135             MesPrint("!>Illegal position in SetScratch");
00136             Terminate(-1);
00137 /* INTERNAL_ERROR_EXCL_STOP */
00138         }
00139         possize = *position;
00140         LOCK(AS.inputslock);
00141         SeekFile(f->handle,&possize,SEEK_SET);
00142         if ( ISNOTEQUALPOS(possize,*position) ) {
00143 /* INTERNAL_ERROR_EXCL_START */
00144             UNLOCK(AS.inputslock);
00145             MesPrint("!>Cannot position file in SetScratch");
00146             Terminate(-1);
00147 /* INTERNAL_ERROR_EXCL_STOP */
00148         }
00149 #ifdef HIDEDEBUG
00150             MesPrint("SetScratch1(%w): position = %12p, size = %l, address =
    %x",position,f->POsize,f->PObuffer);
00151 #endif
00152         if ( ( size = ReadFile(f->handle,(UBYTE *)(f->PObuffer),f->POsize) ) < 0
00153         || ( size & 1 ) != 0 ) {
00154 /* INTERNAL_ERROR_EXCL_START */
00155             UNLOCK(AS.inputslock);
00156             MesPrint("!>Read error in SetScratch");
00157             Terminate(-1);
00158 /* INTERNAL_ERROR_EXCL_STOP */
00159         }
00160         UNLOCK(AS.inputslock);
00161         if ( size == 0 ) {
00162             f->PObuffer[0] = 0;
00163         }
00164         f->POfill = f->PObuffer;
00165         f->POposition = *position;
00166 #ifdef WORD2
00167         *whichInInBuf = size » 1;
00168 #else
00169         *whichInInBuf = size / TABLESIZE(WORD,UBYTE);
00170 #endif
00171         f->POfull = f->PObuffer + *whichInInBuf;
00172 #ifdef HIDEDEBUG
00173             MesPrint("SetScratch2: size = %l, InInBuf = %l, fill = %l, full = %l"
00174             ,size,*whichInInBuf,(f->POfill-f->PObuffer)*sizeof(WORD)
00175             ,(f->POfull-f->PObuffer)*sizeof(WORD));
00176 #endif
00177     }
00178     else {
00179 endpos:
```

```
00180         DIFPOS(possize,*position,f->POposition);
00181         f->POfill = (WORD *)(BASEPOSITION(possize)+(UBYTE *)(f->PObuffer));
00182         *whichInInBuf = f->POfull-f->POfill;
00183     }
00184 }
00185
00186 /*
00187         #] SetScratch :
00188         #[ RevertScratch :
00189
00190         Reverts the input/output directions. This way input comes
00191         always from AR.infile
00192
00193 */
00194
00195 int RevertScratch(void)
00196 {
00197     GETIDENTITY
00198     FILEHANDLE *f;
00199     if ( AR.infile->handle >= 0 && AR.infile->handle != AR.outfile->handle ) {
00200         CloseFile(AR.infile->handle);
00201         AR.infile->handle = -1;
00202         remove(AR.infile->name);
00203     }
00204     f = AR.infile; AR.infile = AR.outfile; AR.outfile = f;
00205     AR.infile->POfull = AR.infile->POfill;
00206     AR.infile->POfill = AR.infile->PObuffer;
00207     if ( AR.infile->handle >= 0 ) {
00208         POSITION scrpos;
00209         PUTZERO(scrpos);
00210         SeekFile(AR.infile->handle,&scrpos,SEEK_SET);
00211         if ( ISNOTZEROPOS(scrpos) ) {
00212 /* INTERNAL_ERROR_EXCL_START */
00213             return(MesPrint("!>Error with scratch output."));
00214 /* INTERNAL_ERROR_EXCL_STOP */
00215         }
00216         if ( ( AR.InInBuf = ReadFile(AR.infile->handle,(UBYTE *)(AR.infile->PObuffer)
00217             ,AR.infile->POsize) ) < 0 || AR.InInBuf & 1 ) {
00218 /* INTERNAL_ERROR_EXCL_START */
00219             return(MesPrint("!>Error while reading from scratch file"));
00220 /* INTERNAL_ERROR_EXCL_STOP */
00221         }
00222         else {
00223             AR.InInBuf /= TABLESIZE(WORD,UBYTE);
00224         }
00225         AR.infile->POfull = AR.infile->PObuffer + AR.InInBuf;
00226     }
00227     PUTZERO(AR.infile->POposition);
00228     AR.outfile->POfill = AR.outfile->POfull = AR.outfile->PObuffer;
00229     PUTZERO(AR.outfile->POposition);
00230     PUTZERO(AR.outfile->filesize);
00231     return(0);
00232 }
00233
00234 /*
00235         #] RevertScratch :
00236         #[ ResetScratch :
00237
00238         Resets the output scratch file to its beginning in such a way
00239         that the write routines can read it. The output buffers are
00240         left untouched as they may still be needed for extra declarations.
00241
00242 */
00243
00244 int ResetScratch(void)
00245 {
00246     GETIDENTITY
00247     FILEHANDLE *f;
00248     if ( AR.infile->handle >= 0 ) {
00249         CloseFile(AR.infile->handle); AR.infile->handle = -1;
00250         remove(AR.infile->name);
00251         PUTZERO(AR.infile->POposition);
00252         AR.infile->POfill = AR.infile->POfull = AR.infile->PObuffer;
00253     }
00254     if ( AR.outfile->handle >= 0 ) {
00255         POSITION scrpos;
00256         PUTZERO(scrpos);
00257         SeekFile(AR.outfile->handle,&scrpos,SEEK_SET);
00258         if ( ISNOTZEROPOS(scrpos) ) {
00259 /* INTERNAL_ERROR_EXCL_START */
00260             return(MesPrint("!>Error with scratch output."));
00261 /* INTERNAL_ERROR_EXCL_STOP */
00262         }
00263         if ( ( AR.InInBuf = ReadFile(AR.outfile->handle,(UBYTE *)(AR.outfile->PObuffer)
00264         ,AR.outfile->POsize) ) < 0 || AR.InInBuf & 1 ) {
00265 /* INTERNAL_ERROR_EXCL_START */
00266             return(MesPrint("!>Error while reading from scratch file"));
```

```
00267 /* INTERNAL_ERROR_EXCL_STOP */
00268         }
00269         else AR.InInBuf /= TABLESIZE(WORD,UBYTE);
00270         AR.outfile->POfull = AR.outfile->PObuffer + AR.InInBuf;
00271     }
00272     else AR.outfile->POfull = AR.outfile->POfill;
00273     AR.outfile->POfill = AR.outfile->PObuffer;
00274     PUTZERO(AR.outfile->POposition);
00275     f = AR.outfile; AR.outfile = AR.infile; AR.infile = f;
00276     return(0);
00277 }
00278
00279 /*
00280         #] ResetScratch :
00281         #[ ReadFromScratch :
00282
00283     Routine is used to copy files from scratch to hide.
00284 */
00285
00286 int ReadFromScratch(FILEHANDLE *fi, POSITION *pos, UBYTE *buffer, POSITION *length)
00287 {
00288     GETIDENTITY
00289     LONG l = BASEPOSITION(*length);
00290     if ( fi->handle < 0 ) {
00291         memcpy(buffer,fi->POfill,l);
00292     }
00293     else {
00294         SeekFile(fi->handle,pos,SEEK_SET);
00295         if ( ReadFile(fi->handle,buffer,l) != l ) {
00296 /* INTERNAL_ERROR_EXCL_START */
00297             if ( fi == AR.hidefile )
00298                 MesPrint("!>Error reading from hide file.");
00299             else
00300                 MesPrint("!>Error reading from scratch file.");
00301             return(-1);
00302 /* INTERNAL_ERROR_EXCL_STOP */
00303         }
00304     }
00305     return(0);
00306 }
00307
00308 /*
00309         #] ReadFromScratch :
00310         #[ AddToScratch :
00311
00312     Routine is used to copy files from scratch to hide.
00313 */
00314
00315 int AddToScratch(FILEHANDLE *fi, POSITION *pos, UBYTE *buffer, POSITION *length,
00316              int withflush)
00317 {
00318     GETIDENTITY
00319     LONG l = BASEPOSITION(*length), avail;
00320     DUMMYUSE(pos)
00321     fi->POfill = fi->POfull;
00322     while ( fi->POfill+l/sizeof(WORD) > fi->POstop ) {
00323         avail = (fi->POstop-fi->POfill)*sizeof(WORD);
00324         if ( avail > 0 ) {
00325             memcpy(fi->POfill,buffer,avail);
00326             l -= avail; buffer += avail;
00327         }
00328         if ( fi->handle < 0 ) {
00329             if ( ( fi->handle = (WORD)CreateFile(fi->name) ) < 0 ) {
00330                 if ( fi == AR.hidefile )
00331                     MesPrint("Cannot create hide file %s",fi->name);
00332                 else
00333                     MesPrint("Cannot create scratch file %s",fi->name);
00334                 return(-1);
00335             }
00336             PUTZERO(fi->POposition);
00337         }
00338         SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
00339         if ( WriteFile(fi->handle,(UBYTE *)fi->PObuffer,fi->POsize) != fi->POsize )
00340             goto writeerror;
00341         ADDPOS(fi->POposition,fi->POsize);
00342         fi->POfill = fi->POfull = fi->PObuffer;
00343     }
00344     if ( l > 0 ) {
00345         memcpy(fi->POfill,buffer,l);
00346         fi->POfill += l/sizeof(WORD);
00347         fi->POfull = fi->POfill;
00348     }
00349     if ( withflush && fi->handle >= 0 && fi->POfill > fi->PObuffer ) {    /* flush */
00350         l = (LONG)fi->POfill - (LONG)fi->PObuffer;
00351         SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
00352         if ( WriteFile(fi->handle,(UBYTE *)fi->PObuffer,l) != l ) goto writeerror;
00353         ADDPOS(fi->POposition,fi->POsize);
```

```
00354         fi->POfill = fi->POfull = fi->POobuffer;
00355     }
00356     if ( withflush && fi->handle >= 0 )
00357         SETBASEPOSITION(fi->filesize,TellFile(fi->handle));
00358     return(0);
00359 writeerror:
00360     if ( fi == AR.hidefile )
00361         MesPrint("Error writing to hide file. Disk full?");
00362     else
00363         MesPrint("Error writing to scratch file. Disk full?");
00364     return(-1);
00365 }
00366
00367 /*
00368         #] AddToScratch :
00369         #[ CoSave :
00370
00371         The syntax of the save statement is:
00372
00373         save filename
00374         save filename expr1 expr2
00375
00376 */
00377
00378 int CoSave(UBYTE *inp)
00379 {
00380     GETIDENTITY
00381     UBYTE *p, c;
00382     WORD n = 0, i;
00383     WORD error = 0, type, number;
00384     WORD exprInStorageFlag = 0;
00385     LONG RetCode = 0, wSize;
00386     EXPRESSIONS e;
00387     INDEXENTRY *ind;
00388     INDEXENTRY *indold;
00389     WORD TMproto[SUBEXPSIZE];
00390     POSITION scrpos, scrpos1, filesize;
00391     int ii, j = sizeof(FILEINDEX)/(sizeof(LONG));
00392     LONG *lo;
00393     while ( *inp == ',' ) inp++;
00394     p = inp;
00395
00396 #ifdef WITHMPI
00397     if( PF.me != MASTER) return(0);
00398 #endif
00399
00400     if ( !*p ) return(MesPrint("No filename in save statement"));
00401     if ( FG.cTable[*p] > 1 && ( *p != '.' ) && ( *p != SEPARATOR ) && ( *p != ALTSEPARATOR ) )
00402             return(MesPrint("Illegal filename"));
00403     while ( *++p && *p != ',' ) {}
00404     c = *p;
00405     *p = 0;
00406     if ( !AP.preError ) {
00407         if ( ( RetCode = CreateFile((char *)inp) ) < 0 ) {
00408             return(MesPrint("Cannot open file %s",inp));
00409         }
00410     }
00411     AO.SaveData.Handle = (WORD)RetCode;
00412     PUTZERO(filesize);
00413
00414     e = Expressions;
00415     n = NumExpressions;
00416     if ( c ) {          /* There follows a list of expressions */
00417         *p++ = c;
00418         inp = p;
00419         i = (WORD)(INFILEINDEX);
00420         if ( WriteStoreHeader(AO.SaveData.Handle) ) return(MesPrint("Error writing storage file
    header"));
00421 /*      PUTZERO(AO.SaveData.Index.number); */
00422 /*      PUTZERO(AO.SaveData.Index.next); */
00423         lo = (LONG *)(&AO.SaveData.Index);
00424         for ( ii = 0; ii < j; ii++ ) *lo++ = 0;
00425         SETBASEPOSITION(AO.SaveData.Position,(LONG)sizeof(STOREHEADER));
00426         ind = AO.SaveData.Index.expression;
00427         if ( !AP.preError && WriteFile(AO.SaveData.Handle,(UBYTE *)(&(AO.SaveData.Index))
00428         ,(LONG)sizeof(struct FiLeInDeX))!= (LONG)sizeof(struct FiLeInDeX) ) goto SavWrt;
00429         SeekFile(AO.SaveData.Handle,&(filesize),SEEK_END);
00430 /*      ADDPOS(filesize,sizeof(struct FiLeInDeX)); */
00431
00432         do {            /* Scan the list */
00433             if ( !FG.cTable[*p] || *p == '[' ) {
00434                 p = SkipAName(p);
00435                 if ( p == 0 ) return(-1);
00436             }
00437             c = *p; *p = 0;
00438             if ( GetVar(inp,&type,&number,CEXPRESSION,NOAUTO) != NAMENOTFOUND ) {
00439                 if ( e[number].status == STOREDEXPRESSION ) {
```

```
00440                     if ( StrLen(AC.exprnames->namebuffer+e[number].name) > MAXENAME ) {
00441                         char msg[100];
00442                         snprintf(msg, sizeof(msg), "saved expr name over %d char: %s", MAXENAME,
      AC.exprnames->namebuffer+e[number].name);
00443                         Warning(msg);
00444                     }
00445 /*
00446         Here we have to locate the stored expression, copy its index entry
00447         possibly after making a new fileindex and then copy the whole
00448         expression.
00449 */
00450                     if ( AP.preError ) goto NextExpr;
00451                     TMproto[0] = EXPRESSION;
00452                     TMproto[1] = SUBEXPSIZE;
00453                     TMproto[2] = number;
00454                     TMproto[3] = 1;
00455                     { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
00456                     AT.TMaddr = TMproto;
00457                     if ( ( indold = FindInIndex(number,&AR.StoreData,0,0) ) != 0 ) {
00458                         if ( i <= 0 ) {
00459 /*
00460                             AO.SaveData.Index.next = filesize;
00461 */
00462                             SeekFile(AO.SaveData.Handle,&(AO.SaveData.Index.next),SEEK_END);
00463                             scrpos = AO.SaveData.Position;
00464                             SeekFile(AO.SaveData.Handle,&scrpos,SEEK_SET);
00465                             if ( ISNOTEQUALPOS(scrpos,AO.SaveData.Position) ) goto SavWrt;
00466                             if ( WriteFile(AO.SaveData.Handle,(UBYTE *)(&(AO.SaveData.Index))
00467                                     ,(LONG)sizeof(struct FiLeInDeX)) != (LONG)sizeof(struct FiLeInDeX)
      )
00468                                 goto SavWrt;
00469                             i = (WORD)(INFILEINDEX);
00470                             AO.SaveData.Position = AO.SaveData.Index.next;
00471                             lo = (LONG *)(&AO.SaveData.Index);
00472                             for ( ii = 0; ii < j; ii++ ) *lo++ = 0;
00473                             ind = AO.SaveData.Index.expression;
00474                             scrpos = AO.SaveData.Position;
00475                             SeekFile(AO.SaveData.Handle,&scrpos,SEEK_SET);
00476                             if ( ISNOTEQUALPOS(scrpos,AO.SaveData.Position) ) goto SavWrt;
00477                             if ( WriteFile(AO.SaveData.Handle,(UBYTE *)(&(AO.SaveData.Index))
00478                             ,(LONG)sizeof(struct FiLeInDeX)) != (LONG)sizeof(struct FiLeInDeX) )
00479                                 goto SavWrt;
00480                             ADDPOS(filesize,sizeof(struct FiLeInDeX));
00481                         }
00482                         *ind = *indold;
00483 /*
00484                         ind->variables = SeekFile(AO.SaveData.Handle,&(AM.zeropos),SEEK_END);
00485 */
00486                         ind->variables = filesize;
00487                         ind->position = ind->variables;
00488                         ADDPOS(ind->position,DIFBASE(indold->position,indold->variables));
00489                         SeekFile(AR.StoreData.Handle,&(indold->variables),SEEK_SET);
00490                         wSize = TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer);
00491                         scrpos = ind->length;
00492                         ADDPOS(scrpos,DIFBASE(ind->position,ind->variables));
00493                         ADD2POS(filesize,scrpos);
00494                         SETBASEPOSITION(scrpos1,wSize);
00495                         do {
00496                             if ( ISLESSPOS(scrpos,scrpos1) ) wSize = BASEPOSITION(scrpos);
00497                             if ( ReadFile(AR.StoreData.Handle,(UBYTE *)AT.WorkPointer,wSize)
00498                             != wSize ) {
00499 /* INTERNAL_ERROR_EXCL_START */
00500                                 MesPrint("!>ReadError");
00501                                 error = -1;
00502                                 goto EndSave;
00503 /* INTERNAL_ERROR_EXCL_STOP */
00504                             }
00505                             if ( WriteFile(AO.SaveData.Handle,(UBYTE *)AT.WorkPointer,wSize)
00506                             != wSize ) goto SavWrt;
00507                             ADDPOS(scrpos,-wSize);
00508                         } while ( ISPOSPOS(scrpos) );
00509                         ADDPOS(AO.SaveData.Index.number,1);
00510                         ind++;
00511                     }
00512                     else error = -1;
00513                     i--;
00514                 }
00515                 else {
00516                     MesPrint("%s is not a stored expression",inp);
00517                     error = -1;
00518                 }
00519 NextExpr:;
00520             }
00521             else {
00522                 MesPrint("%s is not an expression",inp);
00523                 error = -1;
00524             }
```

```
00525             *p = c;
00526             if ( c != ',' && c ) {
00527                 MesComp("Illegal character",inp,p);
00528                 error = -1;
00529                 goto EndSave;
00530             }
00531             if ( c ) c = *++p;
00532             inp = p;
00533         } while ( c );
00534         if ( !AP.preError ) {
00535             scrpos = AO.SaveData.Position;
00536             SeekFile(AO.SaveData.Handle,&scrpos,SEEK_SET);
00537             if ( ISNOTEQUALPOS(scrpos,AO.SaveData.Position) ) goto SavWrt;
00538         }
00539         if ( !AP.preError &&
00540         WriteFile(AO.SaveData.Handle,(UBYTE *)(&(AO.SaveData.Index))
00541             ,(LONG)sizeof(struct FiLeInDeX)) != (LONG)sizeof(struct FiLeInDeX) ) goto SavWrt;
00542     }
00543     else if ( !AP.preError ) {              /* All stored expressions should be saved. Easy */
00544         /* Make sure there is at least one stored expression: */
00545         if ( n > 0 ) { do {
00546             if ( StrLen(AC.exprnames->namebuffer+e->name) > MAXENAME ) {
00547                 char msg[100];
00548                 snprintf(msg, sizeof(msg), "saved expr name over %d char: %s", MAXENAME,
      AC.exprnames->namebuffer+e->name);
00549                 Warning(msg);
00550             }
00551             if ( e->status == STOREDEXPRESSION ) exprInStorageFlag = 1;
00552             e++;
00553         } while ( --n > 0 ); }
00554         if ( exprInStorageFlag ) {
00555             wSize = TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer);
00556             PUTZERO(scrpos);
00557             SeekFile(AR.StoreData.Handle,&scrpos,SEEK_SET);     /* Start at the beginning */
00558             scrpos = AR.StoreData.Fill;         /* Number of bytes to be copied */
00559             SETBASEPOSITION(scrpos1,wSize);
00560             do {
00561                 if ( ISLESSPOS(scrpos,scrpos1) ) wSize = BASEPOSITION(scrpos);
00562                 if ( ReadFile(AR.StoreData.Handle,(UBYTE *)AT.WorkPointer,wSize) != wSize ) {
00563 /* INTERNAL_ERROR_EXCL_START */
00564                     MesPrint("!>ReadError");
00565                     error = -1;
00566                     goto EndSave;
00567 /* INTERNAL_ERROR_EXCL_STOP */
00568                 }
00569                 if ( WriteFile(AO.SaveData.Handle,(UBYTE *)AT.WorkPointer,wSize) != wSize )
00570                     goto SavWrt;
00571                 ADDPOS(scrpos,-wSize);
00572             } while ( ISPOSPOS(scrpos) );
00573         }
00574     }
00575 EndSave:
00576     if ( !AP.preError ) {
00577         CloseFile(AO.SaveData.Handle);
00578         AO.SaveData.Handle = -1;
00579     }
00580     return(error);
00581 SavWrt:
00582 /* INTERNAL_ERROR_EXCL_START */
00583     MesPrint("!>WriteError");
00584     error = -1;
00585     goto EndSave;
00586 /* INTERNAL_ERROR_EXCL_STOP */
00587 }
00588
00589 /*
00590         #] CoSave :
00591         #[ CoLoad :
00592 */
00593
00594 int CoLoad(UBYTE *inp)
00595 {
00596     GETIDENTITY
00597     INDEXENTRY *ind;
00598     LONG RetCode;
00599     UBYTE *p, c;
00600     WORD num, i, error = 0;
00601     WORD type, number, silentload = 0;
00602     WORD TMproto[SUBEXPSIZE];
00603     POSITION scrpos,firstposition;
00604     while ( *inp == ',' ) inp++;
00605     p = inp;
00606     if ( ( *p == ',' && p[1] == '-' ) || *p == '-' ) {
00607         if ( *p == ',' ) p++;
00608         p++;
00609         if ( *p == 's' || *p == 'S' ) {
00610             silentload = 1;
```

```
00611             while ( *p && ( *p != ',' && *p != '-' && *p != '+'
00612             && *p != SEPARATOR && *p != ALTSEPARATOR && *p != '.' ) ) p++;
00613         }
00614         else if ( *p != ',' ) {
00615             return(MesPrint("Illegal option in Load statement"));
00616         }
00617         while ( *p == ',' ) p++;
00618     }
00619     inp = p;
00620     if ( !*p ) return(MesPrint("No filename in load statement"));
00621     if ( FG.cTable[*p] > 1 && ( *p != '.' ) && ( *p != SEPARATOR ) && ( *p != ALTSEPARATOR ) )
00622                 return(MesPrint("Illegal filename"));
00623     while ( *++p && *p != ',' ) {}
00624     c = *p;
00625     *p = 0;
00626     if ( ( RetCode = OpenFile((char *)inp) ) < 0 ) {
00627         return(MesPrint("Cannot open file %s",inp));
00628     }
00629
00630     if ( SetFileIndex() ) {
00631         MesCall("CoLoad");
00632         SETERROR(-1)
00633     }
00634
00635     AO.SaveData.Handle = (WORD)(RetCode);
00636
00637 #ifdef SYSDEPENDENTSAVE
00638     if ( ReadFile(AO.SaveData.Handle,(UBYTE *)(&(AO.SaveData.Index)),
00639         (LONG)sizeof(struct FiLeInDeX)) != (LONG)sizeof(struct FiLeInDeX) ) goto LoadRead;
00640 #else
00641     if ( ReadSaveHeader() ) goto LoadRead;
00642     TELLFILE(AO.SaveData.Handle,&firstposition);
00643     if ( ReadSaveIndex(&AO.SaveData.Index) ) goto LoadRead;
00644 #endif
00645     if ( c ) {          /* There follows a list of expressions */
00646         *p++ = c;
00647         inp = p;
00648
00649         do {            /* Scan the list */
00650             if ( !FG.cTable[*p] || *p == '[' ) {
00651                 p = SkipAName(p);
00652                 if ( p == 0 ) return(-1);
00653             }
00654             c = *p; *p = 0;
00655             if ( GetVar(inp,&type,&number,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
00656                 MesPrint("Conflicting name: %s",inp);
00657                 error = -1;
00658             }
00659             else {
00660                 if ( ( num = EntVar(CEXPRESSION,inp,STOREDEXPRESSION,0,0,0) ) >= 0 ) {
00661                     TMproto[0] = EXPRESSION;
00662                     TMproto[1] = SUBEXPSIZE;
00663                     TMproto[2] = num;
00664                     TMproto[3] = 1;
00665                     { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
00666                     AT.TMaddr = TMproto;
00667                     SeekFile(AO.SaveData.Handle,&firstposition,SEEK_SET);
00668                     AO.SaveData.Position = firstposition;
00669                     if ( ReadSaveIndex(&AO.SaveData.Index) ) goto LoadRead;
00670                     if ( ( ind = FindInIndex(num,&AO.SaveData,1,0) ) != 0 ) {
00671                         if ( !error ) {
00672                             if ( PutInStore(ind,num) ) error = -1;
00673                             else if ( !AM.silent && silentload == 0 )
00674                                 MesPrint(" %s loaded",ind->name);
00675                         }
00676 /*
00677 !!! Added 1-feb-1998
00678 */
00679                         Expressions[num].counter = -1;
00680                     }
00681                     else {
00682                         MesPrint(" %s not found",inp);
00683                         error = -1;
00684                     }
00685                 }
00686                 else error = -1;
00687             }
00688             *p = c;
00689             if ( c != ',' && c ) {
00690                 MesComp("Illegal character",inp,p);
00691                 error = -1;
00692                 goto EndLoad;
00693             }
00694             if ( c ) c = *++p;
00695             inp = p;
00696         } while ( c );
00697         scrpos = AR.StoreData.Position;
```

```
00698         SeekFile(AR.StoreData.Handle,&scrpos,SEEK_SET);
00699         if ( ISNOTEQUALPOS(scrpos,AR.StoreData.Position) ) goto LoadWrt;
00700         if ( WriteFile(AR.StoreData.Handle,(UBYTE *)(&(AR.StoreData.Index))
00701             ,(LONG)sizeof(struct FiLeInDeX)) != (LONG)sizeof(struct FiLeInDeX) ) goto LoadWrt;
00702     }
00703     else {              /* All saved expressions should be stored. Easy */
00704         i = (WORD)BASEPOSITION(AO.SaveData.Index.number);
00705         ind = AO.SaveData.Index.expression;
00706 #ifdef SYSDEPENDENTSAVE
00707         if ( i > 0 ) { do {
00708             if ( GetVar((UBYTE *)(ind->name),&type,&number,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
00709                 MesPrint("Conflicting name: %s",ind->name);
00710                 error = -1;
00711             }
00712             else {
00713                 if ( ( num = EntVar(CEXPRESSION,(UBYTE *)(ind->name),STOREDEXPRESSION,0,0,0) ) >= 0 )
      {
00714                     if ( !error ) {
00715                         if ( PutInStore(ind,num) ) error = -1;
00716                         else if ( !AM.silent && silentload == 0 )
00717                             MesPrint(" %s loaded",ind->name);
00718                     }
00719                 }
00720                 else error = -1;
00721             }
00722             i--;
00723             if ( i == 0 && ISNOTZEROPOS(AO.SaveData.Index.next) ) {
00724                 SeekFile(AO.SaveData.Handle,&(AO.SaveData.Index.next),SEEK_SET);
00725                 if ( ReadFile(AO.SaveData.Handle,(UBYTE *)(&(AO.SaveData.Index)),
00726                     (LONG)sizeof(struct FiLeInDeX)) != (LONG)sizeof(struct FiLeInDeX) ) goto LoadRead;
00727                 i = (WORD)BASEPOSITION(AO.SaveData.Index.number);
00728                 ind = AO.SaveData.Index.expression;
00729             }
00730             else ind++;
00731         } while ( i > 0 ); }
00732 #else
00733         if ( i > 0 ) {
00734             do {
00735                 if ( GetVar((UBYTE *)(ind->name),&type,&number,ALLVARIABLES,NOAUTO) != NAMENOTFOUND )
      {
00736                     MesPrint("Conflicting name: %s",ind->name);
00737                     error = -1;
00738                 }
00739                 else {
00740                     if ( ( num = EntVar(CEXPRESSION,(UBYTE *)(ind->name),STOREDEXPRESSION,0,0,0) ) >=
      0 ) {
00741                         if ( !error ) {
00742                             if ( PutInStore(ind,num) ) error = -1;
00743                             else if ( !AM.silent && silentload == 0 )
00744                                 MesPrint(" %s loaded",ind->name);
00745                         }
00746                     }
00747                     else error = -1;
00748                 }
00749                 i--;
00750                 if ( i == 0 && (ISNOTZEROPOS(AO.SaveData.Index.next) || AO.bufferedInd) ) {
00751                     SeekFile(AO.SaveData.Handle,&(AO.SaveData.Index.next),SEEK_SET);
00752                     if ( ReadSaveIndex(&AO.SaveData.Index) ) goto LoadRead;
00753                     i = (WORD)BASEPOSITION(AO.SaveData.Index.number);
00754                     ind = AO.SaveData.Index.expression;
00755                 }
00756                 else ind++;
00757             } while ( i > 0 );
00758         }
00759 #endif
00760     }
00761 EndLoad:
00762 #ifndef SYSDEPENDENTSAVE
00763     if ( AO.powerFlag ) {
00764         MesPrint("WARNING: min-/maxpower had to be adjusted!");
00765     }
00766     if ( AO.resizeFlag ) {
00767         MesPrint("ERROR: could not downsize data!");
00768         return ( -2 );
00769     }
00770 #endif
00771     CloseFile(AO.SaveData.Handle);
00772     AO.SaveData.Handle = -1;
00773     SeekFile(AR.StoreData.Handle,&(AC.StoreFileSize),SEEK_END);
00774     return(error);
00775 LoadWrt:
00776 /* INTERNAL_ERROR_EXCL_START */
00777     MesPrint("!>WriteError");
00778     error = -1;
00779     goto EndLoad;
00780 /* INTERNAL_ERROR_EXCL_STOP */
00781 LoadRead:
```

```
00782 /* INTERNAL_ERROR_EXCL_START */
00783     MesPrint("!>ReadError");
00784     error = -1;
00785     goto EndLoad;
00786 /* INTERNAL_ERROR_EXCL_STOP */
00787 }
00788
00789 /*
00790         #] CoLoad :
00791         #[ DeleteStore :
00792
00793         Routine deletes the contents of the entire storage file.
00794         We close the file and recreate it.
00795         If par > 0 we have to remove the expressions from the namelists.
00796 */
00797
00798 int DeleteStore(WORD par)
00799 {
00800     GETIDENTITY
00801     char *s;
00802     WORD j, n = 0;
00803     EXPRESSIONS e_in, e_out;
00804     WORD DidClean = 0;
00805     if ( AR.StoreData.Handle >= 0 ) {
00806         if ( par > 0 ) {
00807             n = NumExpressions;
00808             j = 0;
00809             e_in = e_out = Expressions;
00810             if ( n > 0 ) { do {
00811                 if ( e_in->status == STOREDEXPRESSION ) {
00812                     NAMENODE *node = GetNode(AC.exprnames,
00813                             AC.exprnames->namebuffer+e_in->name);
00814                     node->type = CDELETE;
00815                     DidClean = 1;
00816                 }
00817                 else {
00818                     if ( e_out != e_in ) {
00819                         NAMENODE *node;
00820                         node = GetNode(AC.exprnames,
00821                             AC.exprnames->namebuffer+e_in->name);
00822                         node->number = (WORD)(e_out - Expressions);
00823                         e_out->onfile = e_in->onfile;
00824                         e_out->prototype = e_in->prototype;
00825                         e_out->printflag = 0;
00826                         e_out->status = e_in->status;
00827                         e_out->name = e_in->name;
00828                         e_out->inmem = e_in->inmem;
00829                         e_out->counter = e_in->counter;
00830                         e_out->numfactors = e_in->numfactors;
00831                         e_out->numdummies = e_in->numdummies;
00832                         e_out->compression = e_in->compression;
00833                         e_out->namesize = e_in->namesize;
00834                         e_out->whichbuffer = e_in->whichbuffer;
00835                         e_out->hidelevel = e_in->hidelevel;
00836                         e_out->node = e_in->node;
00837                         e_out->replace = e_in->replace;
00838                         e_out->vflags = e_in->vflags;
00839                         e_out->uflags = e_in->uflags;
00840 #ifdef PARALLELCODE
00841                         e_out->partodo = e_in->partodo;
00842 #endif
00843                     }
00844                     e_out++;
00845                     j++;
00846                 }
00847                 e_in++;
00848             } while ( --n > 0 ); }
00849             NumExpressions = j;
00850             if ( DidClean ) CompactifyTree(AC.exprnames,EXPRNAMES);
00851         }
00852         AR.StoreData.Handle = -1;
00853         CloseFile(AC.StoreHandle);
00854         AC.StoreHandle = -1;
00855         {
00856 /*
00857             Knock out the storage caches (25-apr-1990!)
00858 */
00859             STORECACHE st;
00860             st = (STORECACHE)(AT.StoreCache);
00861             while ( st ) {
00862                 SETBASEPOSITION(st->position,-1);
00863                 SETBASEPOSITION(st->toppos,-1);
00864                 st = st->next;
00865             }
00866 #ifdef WITHPTHREADS
00867             for ( j = 1; j < AM.totalnumberofthreads; j++ ) {
00868                 st = (STORECACHE)(AB[j]->T.StoreCache);
```

```
00869                 while ( st ) {
00870                     SETBASEPOSITION(st->position,-1);
00871                     SETBASEPOSITION(st->toppos,-1);
00872                     st = st->next;
00873                 }
00874             }
00875 #endif
00876         }
00877         PUTZERO(AC.StoreFileSize);
00878         s = FG.fname; while ( *s ) s++;
00879 #ifdef VMS
00880         *s = ';'; s[1] = '*'; s[2] = 0;
00881         remove(FG.fname);
00882         *s = 0;
00883 #endif
00884         return(AC.StoreHandle = CreateFile(FG.fname));
00885     }
00886     else return(0);
00887 }
00888 
00889 /*
00890         #] DeleteStore :
00891         #[ PutInStore :
00892 
00893         Copies the expression indicated by ind from a load file to the
00894         internal storage file. A return value of zero indicates that
00895         everything is OK.
00896 
00897 */
00898 
00899 int PutInStore(INDEXENTRY *ind, WORD num)
00900 {
00901     GETIDENTITY
00902     INDEXENTRY *newind;
00903     LONG wSize;
00904 #ifndef SYSDEPENDENTSAVE
00905     LONG wSizeOut;
00906     LONG stage;
00907 #endif
00908     POSITION scrpos,scrpos1;
00909     newind = NextFileIndex(&(Expressions[num].onfile));
00910     *newind = *ind;
00911 #ifndef SYSDEPENDENTSAVE
00912     SETBASEPOSITION(newind->length, 0);
00913 #endif
00914     newind->variables = AR.StoreData.Fill;
00915     SeekFile(AR.StoreData.Handle,&(newind->variables),SEEK_SET);
00916     if ( ISNOTEQUALPOS(newind->variables,AR.StoreData.Fill) ) goto PutErrS;
00917     newind->position = newind->variables;
00918 #ifdef SYSDEPENDENTSAVE
00919     ADDPOS(newind->position,DIFBASE(ind->position,ind->variables));
00920 #endif
00921     /* set read position to ind->variables */
00922     scrpos = ind->variables;
00923     SeekFile(AO.SaveData.Handle,&scrpos,SEEK_SET);
00924     if ( ISNOTEQUALPOS(scrpos,ind->variables) ) goto PutErrS;
00925     /* set max size for read-in */
00926     wSize = TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer);
00927 #ifdef SYSDEPENDENTSAVE
00928     scrpos = ind->length;
00929     ADDPOS(scrpos,DIFBASE(ind->position,ind->variables));
00930     ADD2POS(AR.StoreData.Fill,scrpos);
00931 #endif
00932     SETBASEPOSITION(scrpos1,wSize);
00933 #ifndef SYSDEPENDENTSAVE
00934     /* prepare look-up table for tensor functions */
00935     if ( ind->nfunctions ) {
00936         AO.tensorList = (UBYTE *)Malloc1(MAXSAVEFUNCTION,"PutInStore");
00937     }
00938     SETBASEPOSITION(scrpos, DIFBASE(ind->position,ind->variables));
00939     /* copy variables first */
00940     stage = -1;
00941     do {
00942         wSize = TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer);
00943         if ( ISLESSPOS(scrpos,scrpos1) ) wSize = BASEPOSITION(scrpos);
00944         wSizeOut = wSize;
00945         if ( ReadSaveVariables(
00946             (UBYTE *)AT.WorkPointer, (UBYTE *)AT.WorkTop, &wSize, &wSizeOut, ind, &stage) ) {
00947             goto PutErrS;
00948         }
00949         if ( WriteFile(AR.StoreData.Handle, (UBYTE *)AT.WorkPointer, wSizeOut)
00950         != wSizeOut ) goto PutErrS;
00951         ADDPOS(scrpos,-wSize);
00952         ADDPOS(newind->position, wSizeOut);
00953         ADDPOS(AR.StoreData.Fill, wSizeOut);
00954     } while ( ISPOSPOS(scrpos) );
00955     /* then copy the expression itself */
```

```
00956     wSize = TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer);
00957     scrpos = ind->length;
00958 #endif
00959     do {
00960         wSize = TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer);
00961         if ( ISLESSPOS(scrpos,scrpos1) ) wSize = BASEPOSITION(scrpos);
00962 #ifdef SYSDEPENDENTSAVE
00963         if ( ReadFile(AO.SaveData.Handle,(UBYTE *)AT.WorkPointer,wSize)
00964         != wSize ) goto PutErrS;
00965         if ( WriteFile(AR.StoreData.Handle,(UBYTE *)AT.WorkPointer,wSize)
00966         != wSize ) goto PutErrS;
00967         ADDPOS(scrpos,-wSize);
00968 #else
00969         wSizeOut = wSize;
00970 
00971         if ( ReadSaveExpression((UBYTE *)AT.WorkPointer, (UBYTE *)AT.WorkTop, &wSize, &wSizeOut) ) {
00972             goto PutErrS;
00973         }
00974 
00975         if ( WriteFile(AR.StoreData.Handle, (UBYTE *)AT.WorkPointer, wSizeOut)
00976         != wSizeOut ) goto PutErrS;
00977         ADDPOS(scrpos,-wSize);
00978         ADDPOS(AR.StoreData.Fill, wSizeOut);
00979         ADDPOS(newind->length, wSizeOut);
00980 #endif
00981     } while ( ISPOSPOS(scrpos) );
00982     /* free look-up table for tensor functions */
00983     if ( ind->nfunctions ) {
00984         M_free(AO.tensorList,"PutInStore");
00985     }
00986     scrpos = AR.StoreData.Position;
00987     SeekFile(AR.StoreData.Handle,&scrpos,SEEK_SET);
00988     if ( ISNOTEQUALPOS(scrpos,AR.StoreData.Position) ) goto PutErrS;
00989     if ( WriteFile(AR.StoreData.Handle,(UBYTE *)(&AR.StoreData.Index),(LONG)sizeof(FILEINDEX))
00990     == (LONG)sizeof(FILEINDEX) ) return(0);
00991 PutErrS:
00992 /* INTERNAL_ERROR_EXCL_START */
00993     return(MesPrint("!>File error"));
00994 /* INTERNAL_ERROR_EXCL_STOP */
00995 }
00996 
00997 /*
00998         #] PutInStore :
00999         #[ GetTerm :
01000 
01001         Gets one term from input scratch stream.
01002         Puts it in 'term'.
01003         Returns the length of the term.
01004 
01005         Used by Processor        (proces.c)
01006                 WriteAll         (sch.c)
01007                 WriteOne         (sch.c)
01008                 GetMoreTerms     (store.c)
01009                 ToStorage        (store.c)
01010                 CoFillExpression (comexpr.c)
01011                 FactorInExpr     (factor.c)
01012                 LoadOpti         (optim.c)
01013                 PF_Processor     (parallel.c)
01014                 ThreadsProcessor (threads.c)
01015         In multi thread/processor mode all calls are done by the master.
01016         Note however that other routines, used by the threads, can use
01017         the same file. Hence we need to be careful about SeekFile and locks.
01018 */
01019 
01020 WORD GetTerm(PHEAD WORD *term)
01021 {
01022     GETBIDENTITY
01023     WORD *inp, i, j = 0, len;
01024     LONG InIn, *whichInInBuf;
01025     WORD *r, *m, *mstop = 0, minsiz = 0, *bra = 0, *from;
01026     WORD first, *start = 0, testing = 0;
01027     FILEHANDLE *fi;
01028     AN.deferskipped = 0;
01029     if ( AR.GetFile == 2 ) {
01030         fi = AR.hidefile;
01031         whichInInBuf = &(AR.InHiBuf);
01032     }
01033     else {
01034         fi = AR.infile;
01035         whichInInBuf = &(AR.InInBuf);
01036     }
01037     InIn = *whichInInBuf;
01038     from = term;
01039     if ( AR.KeptInHold ) {
01040         r = AR.CompressBuffer;
01041         i = *r;
01042         AR.KeptInHold = 0;
```

```
01043         if ( i <= 0 ) { *term = 0; goto RegRet; }
01044         m = term;
01045         NCOPY(m,r,i);
01046         goto RegRet;
01047     }
01048     if ( AR.DeferFlag ) {
01049         m = AR.CompressBuffer;
01050         if ( *m > 0 ) {
01051             mstop = m + *m;
01052             mstop -= ABS(mstop[-1]);
01053             m++;
01054             while ( m < mstop ) {
01055                 if ( *m == HAAKJE ) {
01056                     testing = 1;
01057                     mstop = m + m[1];
01058                     bra = (WORD *)(((UBYTE *)(term)) + 2*AM.MaxTer);
01059                     m = AR.CompressBuffer+1;
01060                     r = bra;
01061                     while ( m < mstop ) *r++ = *m++;
01062                     mstop = r;
01063                     minsiz = WORDDIF(mstop,bra);
01064                     goto ReStart;
01065 /*
01066                     We have the bracket to be tested in bra till mstop
01067 */
01068                 }
01069                 m += m[1];
01070             }
01071         }
01072         bra = (WORD *)(((UBYTE *)(term)) + 2*AM.MaxTer);
01073         mstop = bra+1;
01074         *bra = 0;
01075         minsiz = 1;
01076         testing = 1;
01077     }
01078 ReStart:
01079     first = 0;
01080     r = AR.CompressBuffer;
01081     if ( fi->handle >= 0 ) {
01082         if ( InIn <= 0 ) {
01083             ADDPOS(fi->POposition,(fi->POfull-fi->PObuffer)*sizeof(WORD));
01084             LOCK(AS.inputslock);
01085             SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
01086             InIn = ReadFile(fi->handle,(UBYTE *)(fi->PObuffer),fi->POsize);
01087             UNLOCK(AS.inputslock);
01088             if ( ( InIn < 0 ) || ( InIn & 1 ) ) {
01089                 goto GTerr;
01090             }
01091 #ifdef WORD2
01092             InIn >>= 1;
01093 #else
01094             InIn /= TABLESIZE(WORD,UBYTE);
01095 #endif
01096             *whichInInBuf = InIn;
01097             if ( !InIn ) { *r = 0; *from = 0; goto RegRet; }
01098             fi->POfill = fi->PObuffer;
01099             fi->POfull = fi->PObuffer + InIn;
01100         }
01101         inp = fi->POfill;
01102         if ( ( len = i = *inp ) == 0 ) {
01103             (*whichInInBuf)--;
01104             (fi->POfill)++;
01105             *r = 0;
01106             *from = 0;
01107             goto RegRet;
01108         }
01109         if ( i < 0 ) {
01110             InIn--;
01111             inp++;
01112             r++;
01113             start = term;
01114             *term++ = -i + 1;
01115             while ( ++i <= 0 ) *term++ = *r++;
01116             if ( InIn > 0 ) {
01117                 i = *inp++;
01118                 InIn--;
01119                 *start += i;
01120                 *(AR.CompressBuffer) = len = *start;
01121             }
01122             else {
01123                 first = 1;
01124                 goto NewIn;
01125             }
01126         }
01127         InIn -= i;
01128         if ( InIn < 0 ) {
01129             j = (WORD)(- InIn);
```

```
01130             i -= j;
01131         }
01132         else j = 0;
01133         while ( --i >= 0 ) {
01134                 *r++ = *term++ = *inp++;
01135         }
01136         if ( j ) {
01137 NewIn:
01138             ADDPOS(fi->POposition,(fi->POfull-fi->PObuffer)*sizeof(WORD));
01139             LOCK(AS.inputslock);
01140             SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
01141             InIn = ReadFile(fi->handle,(UBYTE *)(fi->PObuffer),fi->POsize);
01142             UNLOCK(AS.inputslock);
01143             if ( ( InIn <= 0 ) || ( InIn & 1 ) ) {
01144                 goto GTerr;
01145             }
01146 #ifdef WORD2
01147             InIn »= 1;
01148 #else
01149             InIn /= TABLESIZE(WORD,UBYTE);
01150 #endif
01151             inp = fi->PObuffer;
01152             fi->POfull = inp + InIn;
01153 
01154             if ( first ) {
01155                 j = *inp++;
01156                 InIn--;
01157                 *start += j;
01158                 *(AR.CompressBuffer) = len = *start;
01159             }
01160             InIn -= j;
01161             while ( --j >= 0 ) { *r++ = *term++ = *inp++; }
01162         }
01163         fi->POfill = inp;
01164         *whichInInBuf = InIn;
01165         AR.DefPosition = fi->POposition;
01166         ADDPOS(AR.DefPosition,((UBYTE *)(fi->POfill)-(UBYTE *)(fi->PObuffer)));
01167     }
01168     else {
01169         inp = fi->POfill;
01170         if ( inp >= fi->POfull ) { *from = 0; goto RegRet; }
01171         len = j = *inp;
01172         if ( j < 0 ) {
01173             inp++;
01174             *term++ = *r++ = len = - j + 1 + *inp;
01175             while ( ++j <= 0 ) *term++ = *r++;
01176             j = *inp++;
01177         }
01178         else if ( !j ) j = 1;
01179         while ( --j >= 0 ) { *r++ = *term++ = *inp++; }
01180         fi->POfill = inp;
01181 /*%%%%%ADDED 7-apr-2006 for Keep Brackets in bucket */
01182         SETBASEPOSITION(AR.DefPosition,((UBYTE *)(fi->POfill)-(UBYTE *)(fi->PObuffer)));
01183         if ( inp > fi->POfull ) {
01184             goto GTerr;
01185         }
01186     }
01187     if ( r >= AR.ComprTop ) {
01188         MesPrint("CompressSize of %10l is insufficient",AM.CompressSize);
01189         Terminate(-1);
01190     }
01191     AR.CompressPointer = r; *r = 0;
01192 /*
01193         The next *from is a bug fix that made the program read in forbidden
01194         territory.
01195 */
01196     if ( testing && *from != 0 ) {
01197         WORD jj;
01198         r = from;
01199         jj = *r - 1 - ABS(*(r+*r-1));
01200         if ( jj < minsiz ) goto strip;
01201         r++;
01202         m = bra;
01203         while ( m < mstop ) {
01204             if ( *m != *r ) {
01205 strip:          r = from;
01206                 m = r + *r;
01207                 mstop = m - ABS(m[-1]);
01208                 r++;
01209                 while ( r < mstop ) {
01210                     if ( *r == HAAKJE ) {
01211                         *r++ = 1;
01212                         *r++ = 1;
01213                         *r++ = 3;
01214                         len = WORDDIF(r,from);
01215                         *from = len;
01216                         goto RegRet;
```

```
01217                     }
01218                     r += r[1];
01219                 }
01220                 goto RegRet;
01221             }
01222             m++;
01223             r++;
01224         }
01225         term = from;
01226         AN.deferskipped++;
01227         goto ReStart;
01228     }
01229 RegRet:;
01230 /*
01231             #[ debug :
01232 */
01233     {
01234         UBYTE OutBuf[140];
01235 /*      if ( AP.DebugFlag ) { */
01236         if ( ( AP.PreDebug & DUMPINTERMS ) == DUMPINTERMS ) {
01237             MLOCK(ErrorMessageLock);
01238             AO.OutFill = AO.OutputLine = OutBuf;
01239             AO.OutSkip = 3;
01240             FiniLine();
01241             r = from;
01242             i = *r;
01243             TokenToLine((UBYTE *)("Input: "));
01244             if ( i == 0 ) {
01245                 TokenToLine((UBYTE *)"zero");
01246             }
01247             else if ( i < 0 ) {
01248                 TokenToLine((UBYTE *)"negative!!");
01249             }
01250             else {
01251                 while ( --i >= 0 ) {
01252                     TalToLine((UWORD)(*r++)); TokenToLine((UBYTE *)"  ");
01253                 }
01254             }
01255             FiniLine();
01256             MUNLOCK(ErrorMessageLock);
01257         }
01258     }
01259 /*
01260             #] debug :
01261 */
01262     return(*from);
01263 GTerr:
01264 /* INTERNAL_ERROR_EXCL_START */
01265     MesPrint("!>Error while reading scratch file in GetTerm");
01266     Terminate(-1);
01267     return(-1);
01268 /* INTERNAL_ERROR_EXCL_STOP */
01269 }
01270
01271 /*
01272         #] GetTerm :
01273         #[ GetOneTerm :
01274
01275         Gets one term from stream AR.infile->handle.
01276         Puts it in 'term'.
01277         Returns the length of the term.
01278         Input is unbuffered.
01279         Compression via AR.CompressPointer
01280         par is actually in all calls a file handle
01281
01282         Routine is called from
01283             DoOnePow            Get one power of an expression
01284             Deferred            Get the contents of a bracket
01285             GetFirstBracket
01286             FindBracket
01287         We should do something about the lack of buffering.
01288         Maybe a buffer of a few times AM.MaxTer (MaxTermSize*sizeof(WORD)).
01289         Each thread will need its own buffer!
01290
01291         If par == 0 we use ReadPosFile which can fill the whole buffer.
01292         If par == 1 we use ReadFile and do actual read operations.
01293
01294         Note: we cannot use ReadPosFile when running in the master thread.
01295 */
01296
01297 WORD GetOneTerm(PHEAD WORD *term, FILEHANDLE *fi, POSITION *pos, int par)
01298 {
01299     GETBIDENTITY
01300     WORD i, *p;
01301     LONG j, siz;
01302     WORD *r, *rr = AR.CompressPointer;
01303     int error = 0;
```

```
01304     r = rr;
01305     if ( fi->handle >= 0 ) {
01306 #ifdef READONEBYONE
01307 #ifdef WITHPTHREADS
01308 /*
01309         This code needs some investigation.
01310         It may be that we should do this always.
01311         It may be that even for workers it is no good.
01312         We may have to make a variable like AM.ReadDirect with
01313         if ( AM.ReadDirect ) par = 1;
01314         and a user command like
01315         On ReadDirect;
01316 */
01317         if ( AT.identity > 0 ) par = 1;
01318 #endif
01319 #endif
01320 /*
01321         To be changed:
01322         1: check first whether the term lies completely inside the buffer
01323         2: if not a: use old strategy for AT.identity == 0 (master)
01324                   b: for workers, position file and read buffer
01325 */
01326         if ( par == 0 ) {
01327             siz = ReadPosFile(BHEAD fi,(UBYTE *)term,1L,pos);
01328         }
01329         else {
01330             LOCK(AS.inputslock);
01331             SeekFile(fi->handle,pos,SEEK_SET);
01332             siz = ReadFile(fi->handle,(UBYTE *)term,sizeof(WORD));
01333             UNLOCK(AS.inputslock);
01334             ADDPOS(*pos,siz);
01335         }
01336         if ( siz == sizeof(WORD) ) {
01337             p = term;
01338             j = i = *term++;
01339             if ( ( i > AM.MaxTer/((WORD)sizeof(WORD)) ) || ( -i >= AM.MaxTer/((WORD)sizeof(WORD)) ) )
01340             {
01341                 error = 1;
01342                 goto ErrGet;
01343             }
01344             r++;
01345             if ( i < 0 ) { /* Loading a compressed term */
01346                 *p = -i + 1;
01347                 while ( ++i <= 0 ) *term++ = *r++;
01348                 if ( par == 0 ) {
01349                     siz = ReadPosFile(BHEAD fi,(UBYTE *)term,1L,pos);
01350                 }
01351                 else {
01352                     LOCK(AS.inputslock);
01353                     SeekFile(fi->handle,pos,SEEK_SET);
01354                     siz = ReadFile(fi->handle,(UBYTE *)term,sizeof(WORD));
01355                     UNLOCK(AS.inputslock);
01356                     ADDPOS(*pos,sizeof(WORD));
01357                 }
01358                 if ( siz != sizeof(WORD) ) {
01359                     error = 2;
01360                     goto ErrGet;
01361                 }
01362                 *p += *term;
01363                 j = *term;
01364                 if ( ( j > AM.MaxTer/((WORD)sizeof(WORD)) ) || ( j <= 0 ) )
01365                 {
01366                     error = 3;
01367                     goto ErrGet;
01368                 }
01369                 *rr = *p; /* Write the proper term size to *AR.CompressPointer */
01370             }
01371             else { /* Loading a regular term */
01372                 if ( !j ) return(0);
01373                 *rr = i; /* Write the proper term size to *AR.CompressPointer */
01374                 j--;
01375             }
01376             i = (WORD)j;
01377             if ( par == 0 ) {
01378                 siz = ReadPosFile(BHEAD fi,(UBYTE *)term,j,pos);
01379                 j *= TABLESIZE(WORD,UBYTE);
01380             }
01381             else {
01382                 j *= TABLESIZE(WORD,UBYTE);
01383                 LOCK(AS.inputslock);
01384                 SeekFile(fi->handle,pos,SEEK_SET);
01385                 siz = ReadFile(fi->handle,(UBYTE *)term,j);
01386                 UNLOCK(AS.inputslock);
01387                 ADDPOS(*pos,j);
01388             }
01389             if ( siz != j ) {
01390                 error = 4;
```

```
01391                 goto ErrGet;
01392             }
01393             while ( --i >= 0 ) *r++ = *term++;
01394             if ( r >= AR.ComprTop ) {
01395                 MLOCK(ErrorMessageLock);
01396                 MesPrint("CompressSize of %10l is insufficient",AM.CompressSize);
01397                 MUNLOCK(ErrorMessageLock);
01398                 Terminate(-1);
01399             }
01400             AR.CompressPointer = r; *r = 0;
01401             return(*p);
01402         }
01403         error = 5;
01404     }
01405     else {
01406 /*
01407         Here the whole expression is in the buffer.
01408 */
01409         fi->POfill = (WORD *)((UBYTE *)(fi->PObuffer) + BASEPOSITION(*pos));
01410         p = fi->POfill;
01411         if ( p >= fi->POfull ) { *term = 0; return(0); }
01412         j = i = *p;
01413         if ( i < 0 ) {
01414             p++;
01415             j = *r++ = *term++ = -i + 1 + *p;
01416             while ( ++i <= 0 ) *term++ = *r++;
01417             i = *p++;
01418         }
01419         if ( i == 0 ) { i = 1; *r++ = 0; *term++ = 0; }
01420         else { while ( --i >= 0 ) { *r++ = *term++ = *p++; } }
01421         fi->POfill = p;
01422         SETBASEPOSITION(*pos,(UBYTE *)(fi->POfill)-(UBYTE *)(fi->PObuffer));
01423         if ( p <= fi->POfull ) {
01424             if ( r >= AR.ComprTop ) {
01425                 MLOCK(ErrorMessageLock);
01426                 MesPrint("CompressSize of %10l is insufficient",AM.CompressSize);
01427                 MUNLOCK(ErrorMessageLock);
01428                 Terminate(-1);
01429             }
01430             AR.CompressPointer = r; *r = 0;
01431             return((WORD)j);
01432         }
01433         error = 6;
01434     }
01435 ErrGet:
01436 /* INTERNAL_ERROR_EXCL_START */
01437     MLOCK(ErrorMessageLock);
01438     MesPrint("!>Error while reading scratch file in GetOneTerm (%d)",error);
01439     MUNLOCK(ErrorMessageLock);
01440     Terminate(-1);
01441     return(-1);
01442 /* INTERNAL_ERROR_EXCL_STOP */
01443 }
01444
01445 /*
01446         #] GetOneTerm :
01447         #[ GetMoreTerms :
01448     Routine collects more contents of brackets inside a function,
01449     indicated by the number in AC.CollectFun.
01450     The first term is in term already.
01451     We can keep calling GetTerm either till a bracket is finished
01452     or till it would make the term too long (> AM.MaxTer/2)
01453     In all cases this function makes that the routine GetTerm
01454     has a term in 'hold', so the AR.KeptInHold flag must be turned on.
01455 */
01456
01457 WORD GetMoreTerms(WORD *term)
01458 {
01459     GETIDENTITY
01460     WORD *t, *r, *m, *h, *tstop, i, inc, same;
01461     WORD extra;
01462     WORD retval = 0;
01463 /*
01464     We use 23% as a quasi-random default value.
01465 */
01466     extra = ((AM.MaxTer/sizeof(WORD))*((LONG)100-AC.CollectPercentage))/100;
01467     if ( extra < 23 ) extra = 23;
01468 /*
01469     First find the bracket pointer
01470 */
01471     t = term + *term;
01472     tstop = t - ABS(t[-1]);
01473     h = term+1;
01474     while ( *h != HAAKJE && h < tstop ) h += h[1];
01475     if ( h >= tstop ) return(retval);
01476     inc = FUNHEAD+ARGHEAD+1-h[1];
01477     same = WORDDIF(h,term) + h[1] - 1;
```

```
01478     r = m = t + inc;
01479     tstop = h + h[1];
01480     while ( t > tstop ) *--r = *--t;
01481     r--;
01482     *r = WORDDIF(m,r);
01483     while ( GetTerm(BHEAD m) > 0 ) {
01484         r = m + 1;
01485         t = m + *m - 1;
01486         if ( same > ( i = ( *m - ABS(*t) -1 ) ) ) { /* Must fail */
01487             if ( AC.AltCollectFun && AS.CollectOverFlag == 2 ) AS.CollectOverFlag = 3;
01488             break;
01489         }
01490         t = term+1;
01491         i = same;
01492         while ( --i >= 0 ) {
01493             if ( *r != *t ) {
01494                 if ( AC.AltCollectFun && AS.CollectOverFlag == 2 ) AS.CollectOverFlag = 3;
01495                 goto FullTerm;
01496             }
01497             r++; t++;
01498         }
01499         if ( ( WORDDIF(m,term) + i + extra ) > (WORD)(AM.MaxTer/sizeof(WORD)) ) {
01500 /* 23 = 3 +20. The 20 is to have some extra for substitutions or whatever */
01501             if ( AS.CollectOverFlag == 0 && AC.AltCollectFun == 0 ) {
01502                 Warning("Bracket contents too long in Collect statement");
01503                 Warning("Contents spread over more than one term");
01504                 Warning("If possible: increase MaxTermSize in setfile");
01505                 AS.CollectOverFlag = 1;
01506             }
01507             else if ( AC.AltCollectFun ) {
01508                 AS.CollectOverFlag = 2;
01509             }
01510             break;
01511         }
01512         tstop = m + *m;
01513         *m -= same;
01514         m++;
01515         while ( r < tstop ) *m++ = *r++;
01516         retval++;
01517         if ( extra == 23 ) extra = ((AM.MaxTer/sizeof(WORD))/6);
01518     }
01519 FullTerm:
01520     h[1] = WORDDIF(m,h);
01521     if ( AS.CollectOverFlag > 1 ) {
01522         *h = AC.AltCollectFun;
01523         if ( AS.CollectOverFlag == 3 ) AS.CollectOverFlag = 1;
01524     }
01525     else *h = AC.CollectFun;
01526     h[2] |= DIRTYFLAG;
01527     h[FUNHEAD] = h[1] - FUNHEAD;
01528     h[FUNHEAD+1] = 0;
01529     if ( ToFast(h+FUNHEAD,h+FUNHEAD) ) {
01530         if ( h[FUNHEAD] <= -FUNCTION ) {
01531             h[1] = FUNHEAD+1;
01532             m = h + FUNHEAD+1;
01533         }
01534         else {
01535             h[1] = FUNHEAD+2;
01536             m = h + FUNHEAD+2;
01537         }
01538     }
01539     *m++ = 1;
01540     *m++ = 1;
01541     *m++ = 3;
01542     *term = WORDDIF(m,term);
01543     AR.KeptInHold = 1;
01544     return(retval);
01545 }
01546
01547 /*
01548         #] GetMoreTerms :
01549         #[ GetMoreFromMem :
01550
01551 */
01552
01553 int GetMoreFromMem(WORD *term, WORD **tpoin)
01554 {
01555     GETIDENTITY
01556     WORD *t, *r, *m, *h, *tstop, i, j, inc, same;
01557     LONG extra = 23;
01558 /*
01559     First find the bracket pointer
01560 */
01561     t = term + *term;
01562     tstop = t - ABS(t[-1]);
01563     h = term+1;
01564     while ( *h != HAAKJE && h < tstop ) h += h[1];
```

```
01565     if ( h >= tstop ) return(0);
01566     inc = FUNHEAD+ARGHEAD+1-h[1];
01567     same = WORDDIF(h,term) + h[1] - 1;
01568     r = m = t + inc;
01569     tstop = h + h[1];
01570     while ( t > tstop ) *--r = *--t;
01571     r--;
01572     *r = WORDDIF(m,r);
01573     while ( **tpoin ) {
01574         r = *tpoin; j = *r;
01575         for ( i = 0; i < j; i++ ) m[i] = *r++;
01576         *tpoin = r;
01577         r = m + 1;
01578         t = m + *m - 1;
01579         if ( same > ( i = ( *m - ABS(*t) -1 ) ) ) { /* Must fail */
01580             if ( AC.AltCollectFun && AS.CollectOverFlag == 2 ) AS.CollectOverFlag = 3;
01581             break;
01582         }
01583         t = term+1;
01584         i = same;
01585         while ( --i >= 0 ) {
01586             if ( *r != *t ) {
01587                 if ( AC.AltCollectFun && AS.CollectOverFlag == 2 ) AS.CollectOverFlag = 3;
01588                 goto FullTerm;
01589             }
01590             r++; t++;
01591         }
01592         if ( ( WORDDIF(m,term) + i + extra ) > (LONG)(AM.MaxTer/(2*sizeof(WORD))) ) {
01593 /* 23 = 3 +20. The 20 is to have some extra for substitutions or whatever */
01594             if ( AS.CollectOverFlag == 0 && AC.AltCollectFun == 0 ) {
01595                 Warning("Bracket contents too long in Collect statement");
01596                 Warning("Contents spread over more than one term");
01597                 Warning("If possible: increase MaxTermSize in setfile");
01598                 AS.CollectOverFlag = 1;
01599             }
01600             else if ( AC.AltCollectFun ) {
01601                 AS.CollectOverFlag = 2;
01602             }
01603             break;
01604         }
01605         tstop = m + *m;
01606         *m -= same;
01607         m++;
01608         while ( r < tstop ) *m++ = *r++;
01609         if ( extra == 23 ) extra = ((AM.MaxTer/sizeof(WORD))/6);
01610     }
01611 FullTerm:
01612     h[1] = WORDDIF(m,h);
01613     if ( AS.CollectOverFlag > 1 ) {
01614         *h = AC.AltCollectFun;
01615         if ( AS.CollectOverFlag == 3 ) AS.CollectOverFlag = 1;
01616     }
01617     else *h = AC.CollectFun;
01618     h[2] |= DIRTYFLAG;
01619     h[FUNHEAD] = h[1] - FUNHEAD;
01620     h[FUNHEAD+1] = 0;
01621     if ( ToFast(h+FUNHEAD,h+FUNHEAD) ) {
01622         if ( h[FUNHEAD] <= -FUNCTION ) {
01623             h[1] = FUNHEAD+1;
01624             m = h + FUNHEAD+1;
01625         }
01626         else {
01627             h[1] = FUNHEAD+2;
01628             m = h + FUNHEAD+2;
01629         }
01630     }
01631     *m++ = 1;
01632     *m++ = 1;
01633     *m++ = 3;
01634     *term = WORDDIF(m,term);
01635     AR.KeptInHold = 1;
01636     return(0);
01637 }
01638
01639 /*
01640         #] GetMoreFromMem :
01641         #[ GetFromStore :
01642
01643         Gets a single term from the storage file at position and puts
01644         it at 'to'.
01645         The value to be returned is the number of words read.
01646         Renumbering is done also.
01647         This is controlled by the renumber table, given in 'renumber'
01648
01649         This routine should work with a number of cache buffers. The
01650         exact number should be definable in form.set.
01651         The parameters are:
```

```
01652         AM.SizeStoreCache   (4096)
01653         The numbers are the proposed default values.
01654
01655         The cache is a pure read cache.
01656 */
01657
01658 static int gfs = 0;
01659
01660 WORD GetFromStore(WORD *to, POSITION *position, RENUMBER renumber, WORD *InCompState, WORD nexpr)
01661 {
01662     GETIDENTITY
01663     LONG RetCode, num, first = 0;
01664     WORD *from, *m;
01665     struct StOrEcAcHe sc;
01666     STORECACHE s;
01667     STORECACHE snext, sold;
01668     WORD *r, *rr = AR.CompressPointer;
01669     r = rr;
01670     gfs++;
01671     sc.next = AT.StoreCache;
01672     sold = s = &sc;
01673     snext = s->next;
01674     while ( snext ) {
01675         sold = s;
01676         s = snext;
01677         snext = s->next;
01678         if ( BASEPOSITION(s->position) == -1 ) break;
01679         if ( ISLESSPOS(*position,s->toppos) &&
01680         ISGEPOS(*position,s->position) ) {  /* Hit */
01681             if ( AT.StoreCache != s ) {
01682                 sold->next = s->next;
01683                 s->next = AT.StoreCache->next;
01684                 AT.StoreCache = s;
01685             }
01686             from = (WORD *)(((UBYTE *)(s->buffer)) + DIFBASE(*position,s->position));
01687             num = *from;
01688             if ( !num ) { return(*to = 0); }
01689             *InCompState = (WORD)num;
01690             m = to;
01691             if ( num < 0 ) {
01692                 from++;
01693                 ADDPOS(*position,sizeof(WORD));
01694                 *m++ = (WORD)(-num+1);
01695                 r++;
01696                 while ( ++num <= 0 ) *m++ = *r++;
01697                 if ( ISLESSPOS(*position,s->toppos) ) {
01698                     num = *from++;
01699                     *to += (WORD)num;
01700                     ADDPOS(*position,sizeof(WORD));
01701                     *InCompState = (WORD)(num + 2);
01702                 }
01703                 else {
01704                     first = 1;
01705                     goto InNew;
01706                 }
01707             }
01708 PastCon:;
01709             while ( num > 0 && ISLESSPOS(*position,s->toppos) ) {
01710                 *r++ = *m++ = *from++; ADDPOS(*position,sizeof(WORD)); num--;
01711             }
01712             if ( num > 0 ) {
01713 InNew:
01714                 SETBASEPOSITION(s->position,-1);
01715                 SETBASEPOSITION(s->toppos,-1);
01716                 LOCK(AM.storefilelock);
01717                 SeekFile(AR.StoreData.Handle,position,SEEK_SET);
01718                 RetCode = ReadFile(AR.StoreData.Handle,(UBYTE *)(s->buffer),AM.SizeStoreCache);
01719                 UNLOCK(AM.storefilelock);
01720                 if ( RetCode < 0 ) goto PastErr;
01721                 if ( !RetCode ) return( *to = 0 );
01722                 s->position = *position;
01723                 s->toppos = *position;
01724                 ADDPOS(s->toppos,RetCode);
01725                 from = s->buffer;
01726                 if ( first ) {
01727                     num = *from++;
01728                     ADDPOS(*position,sizeof(WORD));
01729                     *to += (WORD)num;
01730                     /* This next line has always been commented, but uncommenting it
01731                        fixes a rare bug when loading certain save files. */
01732                     first = 0;
01733                     *InCompState = (WORD)(num + 2);
01734                 }
01735                 goto PastCon;
01736             }
01737             goto PastEnd;
01738         }
```

```
01739     }
01740     if ( AT.StoreCache ) {      /* Fill the last buffer */
01741         s->position = *position;
01742         LOCK(AM.storefilelock);
01743         SeekFile(AR.StoreData.Handle,position,SEEK_SET);
01744         RetCode = ReadFile(AR.StoreData.Handle,(UBYTE *)(s->buffer),AM.SizeStoreCache);
01745         UNLOCK(AM.storefilelock);
01746         if ( RetCode < 0 ) goto PastErr;
01747         if ( !RetCode ) return( *to = 0 );
01748         s->toppos = *position;
01749         ADDPOS(s->toppos,RetCode);
01750         if ( AT.StoreCache != s ) {
01751             sold->next = s->next;
01752             s->next = AT.StoreCache->next;
01753             AT.StoreCache = s;
01754         }
01755         m = to;
01756         from = s->buffer;
01757         num = *from;
01758         if ( !num ) { return( *to = 0 ); }
01759         *InCompState = (WORD)num;
01760         if ( num < 0 ) {
01761             *m++ = (WORD)(-num+1);
01762             r++;
01763             from++;
01764             ADDPOS(*position,sizeof(WORD));
01765             while ( ++num <= 0 ) *m++ = *r++;
01766             num = *from++;
01767             *to += (WORD)num;
01768             ADDPOS(*position,sizeof(WORD));
01769             *InCompState = (WORD)(num+2);
01770         }
01771         goto PastCon;
01772     }
01773 /*      No caching available */
01774     LOCK(AM.storefilelock);
01775     SeekFile(AR.StoreData.Handle,position,SEEK_SET);
01776     RetCode = ReadFile(AR.StoreData.Handle,(UBYTE *)to,(LONG)sizeof(WORD));
01777     SeekFile(AR.StoreData.Handle,position,SEEK_CUR);
01778     UNLOCK(AM.storefilelock);
01779     if ( RetCode != sizeof(WORD) ) {
01780         *to = 0;
01781         return((WORD)RetCode);
01782     }
01783     if ( !*to ) return(0);
01784     m = to;
01785     if ( *to < 0 ) {
01786         num = *m++;
01787         *to = *r++ = (WORD)(-num + 1);
01788         while ( ++num <= 0 ) *m++ = *r++;
01789         LOCK(AM.storefilelock);
01790         SeekFile(AR.StoreData.Handle,position,SEEK_SET);
01791         RetCode = ReadFile(AR.StoreData.Handle,(UBYTE *)m,(LONG)sizeof(WORD));
01792         SeekFile(AR.StoreData.Handle,position,SEEK_CUR);
01793         UNLOCK(AM.storefilelock);
01794         if ( RetCode != sizeof(WORD) ) {
01795 /* INTERNAL_ERROR_EXCL_START */
01796             MLOCK(ErrorMessageLock);
01797             MesPrint("!>Error in compression of store file");
01798             MUNLOCK(ErrorMessageLock);
01799             return(-1);
01800 /* INTERNAL_ERROR_EXCL_STOP */
01801         }
01802         num = *m;
01803         *to += (WORD)num;
01804         *InCompState = (WORD)(num + 2);
01805     }
01806     else {
01807         *InCompState = *to;
01808         num = *to - 1; m = to + 1; r = rr + 1;
01809     }
01810     first = num;
01811     num *= wsizeof(WORD);
01812     if ( num < 0 ) {
01813 /* INTERNAL_ERROR_EXCL_START */
01814         MLOCK(ErrorMessageLock);
01815         MesPrint("!>Error in stored expressions file at position %9p",position);
01816         MUNLOCK(ErrorMessageLock);
01817         return(-1);
01818 /* INTERNAL_ERROR_EXCL_STOP */
01819     }
01820     LOCK(AM.storefilelock);
01821     SeekFile(AR.StoreData.Handle,position,SEEK_SET);
01822     RetCode = ReadFile(AR.StoreData.Handle,(UBYTE *)m,num);
01823     SeekFile(AR.StoreData.Handle,position,SEEK_CUR);
01824     UNLOCK(AM.storefilelock);
01825     if ( RetCode != num ) {
```

```
01826 /* INTERNAL_ERROR_EXCL_START */
01827         MLOCK(ErrorMessageLock);
01828         MesPrint("!>Error in stored expressions file at position %9p",position);
01829         MUNLOCK(ErrorMessageLock);
01830         return(-1);
01831 /* INTERNAL_ERROR_EXCL_STOP */
01832     }
01833     NCOPY(r,m,first);
01834 PastEnd:
01835     *rr = *to;
01836     if ( r >= AR.ComprTop ) {
01837         MLOCK(ErrorMessageLock);
01838         MesPrint("CompressSize of %10l is insufficient",AM.CompressSize);
01839         MUNLOCK(ErrorMessageLock);
01840         Terminate(-1);
01841     }
01842     AR.CompressPointer = r; *r = 0;
01843     if ( !TermRenumber(to,renumber,nexpr) ) {
01844         MarkDirty(to,DIRTYSYMFLAG);
01845         if ( AR.CurDum > AM.IndDum && Expressions[nexpr].numdummies > 0 )
01846             MoveDummies(BHEAD to,AR.CurDum - AM.IndDum);
01847         return((WORD)*to);
01848     }
01849 PastErr:
01850 /* INTERNAL_ERROR_EXCL_START */
01851     MLOCK(ErrorMessageLock);
01852     MesCall("!>GetFromStore");
01853     MUNLOCK(ErrorMessageLock);
01854     SETERROR(-1)
01855 /* INTERNAL_ERROR_EXCL_STOP */
01856 }
01857
01858 /*
01859         #] GetFromStore :
01860         #[ DetVars :            void DetVars(term)
01861
01862     Determines which variables are used in term.
01863
01864     When par = 1 we are scanning a prototype expression which involves
01865     completely different rules.
01866
01867 */
01868
01869 void DetVars(WORD *term, WORD par)
01870 {
01871     GETIDENTITY
01872     WORD *stopper;
01873     WORD *t, sym;
01874     WORD *sarg;
01875     stopper = term + *term - 1;
01876     stopper = stopper - ABS(*stopper) + 1;
01877     term++;
01878     if ( par ) {        /* Prototype expression */
01879         WORD n;
01880         if ( ( n = NumSymbols ) > 0 ) {
01881             SYMBOLS tt;
01882             tt = symbols;
01883             do {
01884                 (tt++)->flags &= ~INUSE;
01885             } while ( --n > 0 );
01886         }
01887         if ( ( n = NumIndices ) > 0 ) {
01888             INDICES tt;
01889             tt = indices;
01890             do {
01891                 (tt++)->flags &= ~INUSE;
01892             } while ( --n > 0 );
01893         }
01894         if ( ( n = NumVectors ) > 0 ) {
01895             VECTORS tt;
01896             tt = vectors;
01897             do {
01898                 (tt++)->flags &= ~INUSE;
01899             } while ( --n > 0 );
01900         }
01901         if ( ( n = NumFunctions ) > 0 ) {
01902             FUNCTIONS tt;
01903             tt = functions;
01904             do {
01905                 (tt++)->flags &= ~INUSE;
01906             } while ( --n > 0 );
01907         }
01908         term += SUBEXPSIZE;
01909         while ( term < stopper ) {
01910             if ( *term == SYMTOSYM || *term == SYMTONUM ) {
01911                 term += 2;
01912                 AN.UsedSymbol[*term] = 1;
```

```
01913                 symbols[*term].flags |= INUSE;
01914             }
01915             else if ( *term == VECTOVEC ) {
01916                 term += 2;
01917                 AN.UsedVector[*term-AM.OffsetVector] = 1;
01918                 vectors[*term-AM.OffsetVector].flags |= INUSE;
01919             }
01920             else if ( *term == INDTOIND ) {
01921                 term += 2;
01922                 sym = indices[*term - AM.OffsetIndex].dimension;
01923                 if ( sym < 0 ) AN.UsedSymbol[-sym] = 1;
01924                 AN.UsedIndex[(*term) - AM.OffsetIndex] = 1;
01925                 sym = indices[*term-AM.OffsetIndex].nmin4;
01926                 if ( sym < -NMIN4SHIFT ) AN.UsedSymbol[-sym-NMIN4SHIFT] = 1;
01927                 indices[*term-AM.OffsetIndex].flags |= INUSE;
01928             }
01929             else if ( *term == FUNTOFUN ) {
01930                 term += 2;
01931                 AN.UsedFunction[*term-FUNCTION] = 1;
01932                 functions[*term-FUNCTION].flags |= INUSE;
01933             }
01934             term += 2;
01935         }
01936     }
01937     else {
01938         while ( term < stopper ) {
01939             t = term + term[1];
01940             if ( *term == SYMBOL ) {
01941                 term += 2;
01942                 do {
01943                     AN.UsedSymbol[*term] = 1;
01944                     term += 2;
01945                 } while ( term < t );
01946             }
01947             else if ( *term == DOTPRODUCT ) {
01948                 term += 2;
01949                 do {
01950                     AN.UsedVector[(*term++) - AM.OffsetVector] = 1;
01951                     AN.UsedVector[(*term) - AM.OffsetVector] = 1;
01952                     term += 2;
01953                 } while ( term < t );
01954             }
01955             else if ( *term == VECTOR ) {
01956                 term += 2;
01957                 do {
01958                     AN.UsedVector[(*term++) - AM.OffsetVector] = 1;
01959                     if ( *term >= AM.OffsetIndex && *term < AM.DumInd ) {
01960                         sym = indices[*term - AM.OffsetIndex].dimension;
01961                         if ( sym < 0 ) AN.UsedSymbol[-sym] = 1;
01962                         AN.UsedIndex[*term - AM.OffsetIndex] = 1;
01963                         sym = indices[(*term++)-AM.OffsetIndex].nmin4;
01964                         if ( sym < -NMIN4SHIFT ) AN.UsedSymbol[-sym-NMIN4SHIFT] = 1;
01965                     }
01966                     else term++;
01967                 } while ( term < t );
01968             }
01969             else if ( *term == INDEX || *term == LEVICIVITA || *term == GAMMA
01970             || *term == DELTA ) {
01971 /*
01972 Tensors:
01973                 term += 2;
01974 */
01975                 if ( *term == INDEX || *term == DELTA ) term += 2;
01976                 else {
01977 Tensors:
01978                     term += FUNHEAD;
01979                 }
01980                 while ( term < t ) {
01981                     if ( *term  >= AM.OffsetIndex && *term < AM.DumInd ) {
01982                         sym = indices[*term - AM.OffsetIndex].dimension;
01983                         if ( sym < 0 ) AN.UsedSymbol[-sym] = 1;
01984                         AN.UsedIndex[(*term) - AM.OffsetIndex] = 1;
01985                         sym = indices[*term-AM.OffsetIndex].nmin4;
01986                         if ( sym < -NMIN4SHIFT ) AN.UsedSymbol[-sym-NMIN4SHIFT] = 1;
01987                     }
01988                     else if ( *term < (WILDOFFSET+AM.OffsetVector) )
01989                         AN.UsedVector[(*term) - AM.OffsetVector] = 1;
01990                     term++;
01991                 }
01992             }
01993             else if ( *term == HAAKJE ) term = t;
01994             else {
01995                 if ( *term > MAXBUILTINFUNCTION )
01996                     AN.UsedFunction[(*term)-FUNCTION] = 1;
01997                 if ( *term >= FUNCTION && functions[*term-FUNCTION].spec
01998                 >= TENSORFUNCTION && term[1] > FUNHEAD ) goto Tensors;
01999                 term += FUNHEAD;            /* First argument */
```

```
02000                 while ( term < t ) {
02001                     sarg = term;
02002                     NEXTARG(sarg)
02003                     if ( *term > 0 ) {
02004                         sarg = term + *term;    /* End of argument */
02005                         term += ARGHEAD;        /* First term in argument */
02006                         if ( term < sarg ) { do {
02007                             DetVars(term,par);
02008                             term += *term;
02009                         } while ( term < sarg ); }
02010                     }
02011                     else {
02012                         if ( *term < -MAXBUILTINFUNCTION ) {
02013                             AN.UsedFunction[-*term-FUNCTION] = 1;
02014                         }
02015                         else if ( *term == -SYMBOL ) {
02016                             AN.UsedSymbol[term[1]] = 1;
02017                         }
02018                         else if ( *term == -INDEX ) {
02019                             if ( term[1] < (WILDOFFSET+AM.OffsetVector) ) {
02020                                 AN.UsedVector[term[1]-AM.OffsetVector] = 1;
02021                             }
02022                             else if ( term[1] >= AM.OffsetIndex && term[1] < AM.DumInd ) {
02023                                 sym = indices[term[1] - AM.OffsetIndex].dimension;
02024                                 if ( sym < 0 ) AN.UsedSymbol[-sym] = 1;
02025                                 AN.UsedIndex[term[1] - AM.OffsetIndex] = 1;
02026                                 sym = indices[term[1]-AM.OffsetIndex].nmin4;
02027                                 if ( sym < -NMIN4SHIFT ) AN.UsedSymbol[-sym-NMIN4SHIFT] = 1;
02028                             }
02029                         }
02030                         else if ( *term == -VECTOR || *term == -MINVECTOR ) {
02031                             AN.UsedVector[term[1]-AM.OffsetVector] = 1;
02032                         }
02033                     }
02034                     term = sarg;            /* Next argument */
02035                 }
02036                 term = t;
02037             }
02038         }
02039     }
02040 }
02041
02042 /*
02043         #] DetVars :
02044         #[ ToStorage :
02045
02046     This routine takes an expression in the scratch buffer (indicated by e)
02047     and puts it in the storage file. The necessary actions are:
02048
02049     1:  determine the list of the used variables.
02050     2:  make an index entry.
02051     3:  write the namelists.
02052     4:  copy the 'length' bytes of the expression.
02053
02054 */
02055
02056 int ToStorage(EXPRESSIONS e, POSITION *length)
02057 {
02058     GETIDENTITY
02059     WORD *w, i, j;
02060     WORD *term;
02061     INDEXENTRY *indexent;
02062     LONG size;
02063     POSITION indexpos, scrpos;
02064     FILEHANDLE *f;
02065     if ( ( indexent = NextFileIndex(&indexpos) ) == 0 ) {
02066         MesCall("ToStorage");
02067         SETERROR(-1)
02068     }
02069     indexent->CompressSize = 0;     /* thus far no compression */
02070     f = AR.infile; AR.infile = AR.outfile; AR.outfile = f;
02071     if ( e->status == HIDDENGEXPRESSION ) {
02072         AR.InHiBuf = 0; f = AR.hidefile; AR.GetFile = 2;
02073     }
02074     else {
02075         AR.InInBuf = 0; f = AR.infile;   AR.GetFile = 0;
02076     }
02077     if ( f->handle >= 0 ) {
02078         scrpos = e->onfile;
02079         SeekFile(f->handle,&scrpos,SEEK_SET);
02080         if ( ISNOTEQUALPOS(scrpos,e->onfile) ) {
02081 /* INTERNAL_ERROR_EXCL_START */
02082             MesPrint("!>Error in Scratch file");
02083             goto ErrReturn;
02084 /* INTERNAL_ERROR_EXCL_STOP */
02085         }
02086         f->POposition = e->onfile;
```

```
02087         f->POfull = f->PObuffer;
02088         if ( e->status == HIDDENGEXPRESSION ) AR.InHiBuf = 0;
02089         else                                  AR.InInBuf = 0;
02090     }
02091     else {
02092         f->POfill = (WORD *)((UBYTE *)(f->PObuffer)+BASEPOSITION(e->onfile));
02093     }
02094     w = AT.WorkPointer;
02095     AN.UsedSymbol   = w;    w += NumSymbols;
02096     AN.UsedVector   = w;    w += NumVectors;
02097     AN.UsedIndex    = w;    w += NumIndices;
02098     AN.UsedFunction = w;    w += NumFunctions;
02099     term = w;
02100     w = (WORD *)(((UBYTE *)(w)) + AM.MaxTer);
02101     if ( w > AT.WorkTop ) {
02102         MesWork();
02103         goto ErrReturn;
02104     }
02105     w = AN.UsedSymbol;
02106     i = NumSymbols + NumVectors + NumIndices + NumFunctions;
02107     do { *w++ = 0; } while ( --i > 0 );
02108     if ( GetTerm(BHEAD term) > 0 ) {
02109         DetVars(term,1);
02110         if ( GetTerm(BHEAD term) ) {
02111             do { DetVars(term,0); } while ( GetTerm(BHEAD term) > 0 );
02112         }
02113     }
02114     j = 0;
02115     w = AN.UsedSymbol;
02116     i = NumSymbols;
02117     while ( --i >= 0 ) { if ( *w++ ) j++; }
02118     indexent->nsymbols = j;
02119 /*  size = j * sizeof(struct SyMbOl);  */
02120     j = 0;
02121     w = AN.UsedIndex;
02122     i = NumIndices;
02123     while ( --i >= 0 ) { if ( *w++ ) j++; }
02124     indexent->nindices = j;
02125 /*  size += j * sizeof(struct InDeX);  */
02126     j = 0;
02127     w = AN.UsedVector;
02128     i = NumVectors;
02129     while ( --i >= 0 ) { if ( *w++ ) j++; }
02130     indexent->nvectors = j;
02131 /*  size += j * sizeof(struct VeCtOr);  */
02132     j = 0;
02133     w = AN.UsedFunction;
02134     i = NumFunctions;
02135     while ( --i >= 0 ) { if ( *w++ ) j++; }
02136     indexent->nfunctions = j;
02137 /*  size += j * sizeof(struct FuNcTiOn); */
02138     indexent->length = *length;
02139     indexent->variables = AR.StoreData.Fill;
02140 /*  indexent->position = AR.StoreData.Fill + size; */
02141     StrCopy(AC.exprnames->namebuffer+e->name,(UBYTE *)(indexent->name));
02142     SeekFile(AR.StoreData.Handle,&(AR.StoreData.Fill),SEEK_SET);
02143     AO.wlen = 100000;
02144     AO.wpos = (UBYTE *)Malloc1(AO.wlen,"AO.wpos buffer");
02145     AO.wpoin = AO.wpos;
02146     {
02147          SYMBOLS a;
02148         w = AN.UsedSymbol;
02149         a = symbols;
02150         j = 0;
02151         i = indexent->nsymbols;
02152         while ( --i >= 0 ) {
02153             while ( !*w ) { w++; a++; j++; }
02154             a->number = j;
02155             if ( VarStore((UBYTE *)a,(WORD)(sizeof(struct SyMbOl)),a->name,
02156                 a->namesize) ) goto ErrToSto;
02157             w++; j++; a++;
02158         }
02159     }
02160     {
02161          INDICES a;
02162         w = AN.UsedIndex;
02163         a = indices;
02164         j = 0;
02165         i = indexent->nindices;
02166         while ( --i >= 0 ) {
02167             while ( !*w ) { w++; a++; j++; }
02168             a->number = j;
02169             if ( VarStore((UBYTE *)a,(WORD)(sizeof(struct InDeX)),a->name,
02170                 a->namesize) ) goto ErrToSto;
02171             w++; j++; a++;
02172         }
02173     }
```

```
02174     {
02175          VECTORS a;
02176         w = AN.UsedVector;
02177         a = vectors;
02178         j = 0;
02179         i = indexent->nvectors;
02180         while ( --i >= 0 ) {
02181             while ( !*w ) { w++; a++; j++; }
02182             a->number = j;
02183             if ( VarStore((UBYTE *)a,(WORD)(sizeof(struct VeCtOr)),a->name,
02184                 a->namesize) ) goto ErrToSto;
02185             w++; j++; a++;
02186         }
02187     }
02188     {
02189          FUNCTIONS a;
02190         w = AN.UsedFunction;
02191         a = functions;
02192         j = 0;
02193         i = indexent->nfunctions;
02194         while ( --i >= 0 ) {
02195             while ( !*w ) { w++; a++; j++; }
02196             a->number = j;
02197             if ( VarStore((UBYTE *)a,(WORD)(sizeof(struct FuNcTiOn)),a->name,
02198                 a->namesize) ) goto ErrToSto;
02199             w++; a++; j++;
02200         }
02201     }
02202     if ( VarStore((UBYTE *)0L,(WORD)0,(WORD)0,(WORD)0) ) goto ErrToSto; /* Flush buffer */
02203     TELLFILE(AR.StoreData.Handle,&(indexent->position));
02204     indexent->size = (WORD)DIFBASE(indexent->position,indexent->variables);
02205 /*
02206         The following code was added when it became apparent (30-jan-2007)
02207         that we need provisions for extra space without upsetting existing
02208         .sav files. Here we can put as much as we want.
02209         Look in GetTable on how to recover numdummies.
02210         Forgetting numdummies has been in there from the beginning.
02211 */
02212     if ( WriteFile(AR.StoreData.Handle,(UBYTE *)(&(e->numdummies)),(LONG)sizeof(WORD)) !=
02213         sizeof(WORD) ) {
02214 /* INTERNAL_ERROR_EXCL_START */
02215             MesPrint("!>Error while writing storage file");
02216             goto ErrReturn;
02217 /* INTERNAL_ERROR_EXCL_STOP */
02218     }
02219     if ( WriteFile(AR.StoreData.Handle,(UBYTE *)(&(e->numfactors)),(LONG)sizeof(WORD)) !=
02220         sizeof(WORD) ) {
02221 /* INTERNAL_ERROR_EXCL_START */
02222             MesPrint("!>Error while writing storage file");
02223             goto ErrReturn;
02224 /* INTERNAL_ERROR_EXCL_STOP */
02225     }
02226     if ( WriteFile(AR.StoreData.Handle,(UBYTE *)(&(e->vflags)),(LONG)sizeof(WORD)) !=
02227         sizeof(WORD) ) {
02228 /* INTERNAL_ERROR_EXCL_START */
02229             MesPrint("!>Error while writing storage file");
02230             goto ErrReturn;
02231 /* INTERNAL_ERROR_EXCL_STOP */
02232     }
02233     if ( WriteFile(AR.StoreData.Handle,(UBYTE *)(&(e->uflags)),(LONG)sizeof(WORD)) !=
02234         sizeof(WORD) ) {
02235 /* INTERNAL_ERROR_EXCL_START */
02236             MesPrint("!>Error while writing storage file");
02237             goto ErrReturn;
02238 /* INTERNAL_ERROR_EXCL_STOP */
02239     }
02240     TELLFILE(AR.StoreData.Handle,&(indexent->position));
02241     if ( f->handle >= 0 ) {
02242         POSITION llength;
02243         llength = *length;
02244         SeekFile(f->handle,&(e->onfile),SEEK_SET);
02245         while ( ISPOSPOS(llength) ) {
02246             SETBASEPOSITION(scrpos,AO.wlen);
02247             if ( ISLESSPOS(llength,scrpos) ) size = BASEPOSITION(llength);
02248             else                             size = AO.wlen;
02249             if ( ReadFile(f->handle,AO.wpos,size) != size ) {
02250 /* INTERNAL_ERROR_EXCL_START */
02251                 MesPrint("!>Error while reading scratch file");
02252                 goto ErrReturn;
02253 /* INTERNAL_ERROR_EXCL_STOP */
02254             }
02255             if ( WriteFile(AR.StoreData.Handle,AO.wpos,size) != size ) {
02256 /* INTERNAL_ERROR_EXCL_START */
02257                 MesPrint("!>Error while writing storage file");
02258                 goto ErrReturn;
02259 /* INTERNAL_ERROR_EXCL_STOP */
02260             }
```

```
02261             ADDPOS(llength,-size);
02262         }
02263     }
02264     else {
02265         WORD *ppp;
02266         ppp = (WORD *)((UBYTE *)(f->PObuffer) + BASEPOSITION(e->onfile));
02267         if ( WriteFile(AR.StoreData.Handle,(UBYTE *)ppp,BASEPOSITION(*length)) !=
02268         BASEPOSITION(*length) ) {
02269 /* INTERNAL_ERROR_EXCL_START */
02270             MesPrint("!>Error while writing storage file");
02271             goto ErrReturn;
02272 /* INTERNAL_ERROR_EXCL_STOP */
02273         }
02274     }
02275     ADD2POS(*length,indexent->position);
02276     e->onfile = indexpos;
02277 /*
02278     AR.StoreData.Fill = SeekFile(AR.StoreData.Handle,&(AM.zeropos),SEEK_END);
02279 */
02280     AR.StoreData.Fill = *length;
02281     SeekFile(AR.StoreData.Handle,&(AR.StoreData.Fill),SEEK_SET);
02282     scrpos = AR.StoreData.Position;
02283     ADDPOS(scrpos,sizeof(POSITION));
02284     SeekFile(AR.StoreData.Handle,&scrpos,SEEK_SET);
02285     if ( WriteFile(AR.StoreData.Handle,((UBYTE *)&(AR.StoreData.Index.number))
02286         ,(LONG)(sizeof(POSITION))) != sizeof(POSITION) ) goto ErrInSto;
02287     SeekFile(AR.StoreData.Handle,&indexpos,SEEK_SET);
02288     if ( WriteFile(AR.StoreData.Handle,(UBYTE *)indexent,(LONG)(sizeof(INDEXENTRY))) !=
02289         sizeof(INDEXENTRY) ) goto ErrInSto;
02290     FlushFile(AR.StoreData.Handle);
02291     SeekFile(AR.StoreData.Handle,&(AC.StoreFileSize),SEEK_END);
02292     f = AR.infile; AR.infile = AR.outfile; AR.outfile = f;
02293     if ( AO.wpos ) M_free(AO.wpos,"AO.wpos buffer");
02294     AO.wpos = AO.wpoin = 0;
02295     return(0);
02296 /* INTERNAL_ERROR_EXCL_START */
02297 ErrToSto:
02298     MesPrint("!>Error while storing namelists");
02299     goto ErrReturn;
02300 ErrInSto:
02301     MesPrint("!>Error in storage");
02302 /* INTERNAL_ERROR_EXCL_STOP */
02303 ErrReturn:
02304     if ( AO.wpos ) M_free(AO.wpos,"AO.wpos buffer");
02305     AO.wpos = AO.wpoin = 0;
02306     f = AR.infile; AR.infile = AR.outfile; AR.outfile = f;
02307     return(-1);
02308 }
02309
02310 /*
02311         #] ToStorage :
02312         #[ NextFileIndex :
02313 */
02314
02315 INDEXENTRY *NextFileIndex(POSITION *indexpos)
02316 {
02317     GETIDENTITY
02318     INDEXENTRY *ind;
02319     int i, j = sizeof(FILEINDEX)/(sizeof(LONG));
02320     LONG *lo;
02321     if ( AR.StoreData.Handle <= 0 ) {
02322         if ( SetFileIndex() ) {
02323             MesCall("NextFileIndex");
02324             return(0);
02325         }
02326         SETBASEPOSITION(AR.StoreData.Index.number,1);
02327 #ifdef SYSDEPENDENTSAVE
02328         SETBASEPOSITION(*indexpos,(2*sizeof(POSITION)));
02329 #else
02330         SETBASEPOSITION(*indexpos,(2*sizeof(POSITION)+sizeof(STOREHEADER)));
02331 #endif
02332         return(AR.StoreData.Index.expression);
02333     }
02334     while ( BASEPOSITION(AR.StoreData.Index.number) >= (LONG)(INFILEINDEX) ) {
02335         if ( ISNOTZEROPOS(AR.StoreData.Index.next) ) {
02336             SeekFile(AR.StoreData.Handle,&(AR.StoreData.Index.next),SEEK_SET);
02337             AR.StoreData.Position = AR.StoreData.Index.next;
02338             if ( ReadFile(AR.StoreData.Handle,(UBYTE
      *)(&AR.StoreData.Index),(LONG)(sizeof(FILEINDEX))) !=
02339             (LONG)(sizeof(FILEINDEX)) ) goto ErrNextS;
02340         }
02341         else {
02342             PUTZERO(AR.StoreData.Index.number);
02343             SeekFile(AR.StoreData.Handle,&(AR.StoreData.Position),SEEK_SET);
02344             if ( WriteFile(AR.StoreData.Handle,(UBYTE
      *)(&(AR.StoreData.Fill)),(LONG)(sizeof(POSITION)))
02345             != (LONG)(sizeof(POSITION)) ) goto ErrNextS;
```

```
02346             PUTZERO(AR.StoreData.Index.next);
02347             SeekFile(AR.StoreData.Handle,&(AR.StoreData.Fill),SEEK_SET);
02348             AR.StoreData.Position = AR.StoreData.Fill;
02349             lo = (LONG *)(&AR.StoreData.Index);
02350             for ( i = 0; i < j; i++ ) *lo++ = 0;
02351             if ( WriteFile(AR.StoreData.Handle,(UBYTE
     *)(&AR.StoreData.Index),(LONG)(sizeof(FILEINDEX))) !=
02352             (LONG)(sizeof(FILEINDEX)) ) goto ErrNextS;
02353             ADDPOS(AR.StoreData.Fill,sizeof(FILEINDEX));
02354         }
02355     }
02356     *indexpos = AR.StoreData.Position;
02357     ADDPOS(*indexpos,(2*sizeof(POSITION)) +
02358         BASEPOSITION(AR.StoreData.Index.number) * sizeof(INDEXENTRY));
02359     ind = &AR.StoreData.Index.expression[BASEPOSITION(AR.StoreData.Index.number)];
02360     ADDPOS(AR.StoreData.Index.number,1);
02361     return(ind);
02362 ErrNextS:
02363 /* INTERNAL_ERROR_EXCL_START */
02364     MesPrint("!>Error in storage file");
02365     return(0);
02366 /* INTERNAL_ERROR_EXCL_STOP */
02367 }
02368
02369 /*
02370         #] NextFileIndex :
02371         #[ SetFileIndex :
02372 */
02373
02380 int SetFileIndex(void)
02381 {
02382     GETIDENTITY
02383     int i, j = sizeof(FILEINDEX)/(sizeof(LONG));
02384     LONG *lo;
02385     if ( AR.StoreData.Handle < 0 ) {
02386         AR.StoreData.Handle = AC.StoreHandle;
02387         PUTZERO(AR.StoreData.Index.next);
02388         PUTZERO(AR.StoreData.Index.number);
02389 #ifdef SYSDEPENDENTSAVE
02390         SETBASEPOSITION(AR.StoreData.Fill,sizeof(FILEINDEX));
02391 #else
02392         if ( WriteStoreHeader(AR.StoreData.Handle) ) {
02393 /* INTERNAL_ERROR_EXCL_START */
02394             return(MesPrint("!>Error writing storage file header"));
02395 /* INTERNAL_ERROR_EXCL_STOP */
02396         }
02397         SETBASEPOSITION(AR.StoreData.Fill, (LONG)sizeof(FILEINDEX)+(LONG)sizeof(STOREHEADER));
02398 #endif
02399         lo = (LONG *)(&AR.StoreData.Index);
02400         for ( i = 0; i < j; i++ ) *lo++ = 0;
02401         if ( WriteFile(AR.StoreData.Handle,(UBYTE *)(&AR.StoreData.Index),(LONG)(sizeof(FILEINDEX)))
     !=
02402         (LONG)(sizeof(FILEINDEX)) ) {
02403 /* INTERNAL_ERROR_EXCL_START */
02404             return(MesPrint("!>Error writing storage file"));
02405 /* INTERNAL_ERROR_EXCL_STOP */
02406         }
02407     }
02408     else {
02409         POSITION scrpos;
02410 #ifdef SYSDEPENDENTSAVE
02411         PUTZERO(scrpos);
02412 #else
02413         SETBASEPOSITION(scrpos, (LONG)(sizeof(STOREHEADER)));
02414 #endif
02415         SeekFile(AR.StoreData.Handle,&scrpos,SEEK_SET);
02416         if ( ReadFile(AR.StoreData.Handle,(UBYTE *)(&AR.StoreData.Index),(LONG)(sizeof(FILEINDEX))) !=
02417         (LONG)(sizeof(FILEINDEX)) ) {
02418 /* INTERNAL_ERROR_EXCL_START */
02419             return(MesPrint("!>Error reading storage file"));
02420 /* INTERNAL_ERROR_EXCL_STOP */
02421         }
02422     }
02423 #ifdef SYSDEPENDENTSAVE
02424     PUTZERO(AR.StoreData.Position);
02425 #else
02426     SETBASEPOSITION(AR.StoreData.Position, (LONG)(sizeof(STOREHEADER)));
02427 #endif
02428     return(0);
02429 }
02430
02431 /*
02432         #] SetFileIndex :
02433         #[ VarStore :
02434
02435         The n -= sizeof(WORD); makes that the real length comes in the
02436         padding space, provided there is padding space (it seems so).
```

```
02437         The reading of the information assumes this is the case and hence
02438         things work....
02439 */
02440
02441 int VarStore(UBYTE *s, WORD n, WORD name, WORD namesize)
02442 {
02443     GETIDENTITY
02444     UBYTE *t, *u;
02445     if ( s ) {
02446         n -= sizeof(WORD);
02447         t = (UBYTE *)AO.wpoin;
02448 /*
02449         u = (UBYTE *)AT.WorkTop;
02450 */
02451         u = AO.wpos+AO.wlen;
02452         while ( n > 0 && t < u ) { *t++ = *s++; n--; }
02453         while ( t >= u ) {
02454             if ( WriteFile(AR.StoreData.Handle,AO.wpos,AO.wlen) != AO.wlen ) return(-1);
02455             t = AO.wpos;
02456             while ( n > 0 && t < u ) { *t++ = *s++; n--; }
02457         }
02458         s = AC.varnames->namebuffer + name;
02459         n = namesize;
02460         n += sizeof(void *)-1; n &= -(sizeof(void *));
02461         *((WORD *)t) = n;
02462         t += sizeof(WORD);
02463         while ( n > 0 && t < u ) {
02464             if ( namesize > 0 ) { *t++ = *s++; namesize--; }
02465             else { *t++ = 0; }
02466             n--;
02467         }
02468         while ( t >= u ) {
02469             if ( WriteFile(AR.StoreData.Handle,AO.wpos,AO.wlen) != AO.wlen ) return(-1);
02470             t = AO.wpos;
02471             while ( n > 0 && t < u ) {
02472                 if ( namesize > 0 ) { *t++ = *s++; namesize--; }
02473                 else { *t++ = 0; }
02474                 n--;
02475             }
02476         }
02477         AO.wpoin = t;
02478     }
02479     else {
02480         LONG size;
02481         size = AO.wpoin - AO.wpos;
02482         if ( WriteFile(AR.StoreData.Handle,AO.wpos,size) != size ) return(-1);
02483         AO.wpoin = AO.wpos;
02484     }
02485     return(0);
02486 }
02487
02488 /*
02489         #] VarStore :
02490         #[ TermRenumber :
02491
02492         renumbers the variables inside term according to the information
02493         in struct renumber.
02494         The search is binary. This avoided having to read/write the
02495         expression twice when it was stored.
02496
02497 */
02498
02499 int TermRenumber(WORD *term, RENUMBER renumber, WORD nexpr)
02500 {
02501     WORD *stopper;
02506     WORD *t, *sarg, n;
02507     stopper = term + *term - 1;
02508     stopper = stopper - ABS(*stopper) + 1;
02509     term++;
02510     while ( term < stopper ) {
02514         if ( *term == SYMBOL ) {
02515             t = term + term[1];
02516             term += 2;
02517             do {
02518                 if ( ( n = FindrNumber(*term,&(renumber->symb)) ) < 0 ) goto ErrR;
02519                 *term = renumber->symnum[n];
02520                 term += 2;
02521             } while ( term < t );
02522         }
02523         else if ( *term == DOTPRODUCT ) {
02524             t = term + term[1];
02525             term += 2;
02526             do {
02527                 if ( ( n = FindrNumber(*term,&(renumber->vect)) )
02528                      < 0 ) goto ErrR;
02529                 *term++ = renumber->vecnum[n];
02530                 if ( ( n = FindrNumber(*term,&(renumber->vect)) )
```

```
02531                     < 0 ) goto ErrR;
02532                 *term = renumber->vecnum[n];
02533                 term += 2;
02534             } while ( term < t );
02535         }
02536         else if ( *term == VECTOR ) {
02537             t = term + term[1];
02538             term += 2;
02539             do {
02540                 if ( ( n = FindrNumber(*term,&(renumber->vect)) )
02541                      < 0 ) goto ErrR;
02542                 *term++ = renumber->vecnum[n];
02543                 if ( ( *term >= AM.OffsetIndex ) && ( *term < AM.IndDum ) ) {
02544                     if ( ( n = FindrNumber(*term,&(renumber->indi)) )
02545                          < 0 ) goto ErrR;
02546                     *term++ = renumber->indnum[n];
02547                 }
02548                 else term++;
02549             } while ( term < t );
02550         }
02551         else if ( *term == INDEX || *term == LEVICIVITA || *term == GAMMA
02552         || *term == DELTA ) {
02553 Tensors:
02554             t = term + term[1];
02555             if ( *term == INDEX || * term == DELTA ) term += 2;
02556             else term += FUNHEAD;
02557 /*
02558             term += 2;
02559 */
02560             while ( term < t ) {
02561                 if ( *term >= AM.OffsetIndex + WILDOFFSET ) {
02562 /*
02563         Still TOBEDONE
02564 */
02565                 }
02566                 else if ( ( *term  >= AM.OffsetIndex ) && ( *term < AM.IndDum ) ) {
02567                     if ( ( n = FindrNumber(*term,&(renumber->indi)) )
02568                          < 0 ) goto ErrR;
02569                     *term = renumber->indnum[n];
02570                 }
02571                 else if ( *term < (WILDOFFSET+AM.OffsetVector) ) {
02572                     if ( ( n = FindrNumber(*term,&(renumber->vect)) )
02573                          < 0 ) goto ErrR;
02574                     *term = renumber->vecnum[n];
02575                 }
02576                 term++;
02577             }
02578         }
02579         else if ( *term == HAAKJE ) term += term[1];
02580         else {
02581             if ( *term > MAXBUILTINFUNCTION ) {
02582                 if ( ( n = FindrNumber(*term,&(renumber->func)) )
02583                      < 0 ) goto ErrR;
02584                 *term = renumber->funnum[n];
02585             }
02586             if ( *term >= FUNCTION && functions[*term-FUNCTION].spec
02587                     >= TENSORFUNCTION && term[1] > FUNHEAD ) goto Tensors;
02588             t = term + term[1];         /* General stopper */
02589             term += FUNHEAD;            /* First argument */
02590             while ( term < t ) {
02591                 sarg = term;
02592                 NEXTARG(sarg)
02593                 if ( *term > 0 ) {
02594 /*
02595                     Problem here:
02596                     Marking the argument as dirty attacks the heap
02597                     very heavily and costs much computer time.
02598 */
02599                     *++term = 1;
02600                     term += ARGHEAD-1;
02601                     while ( term < sarg ) {
02602                         if ( TermRenumber(term,renumber,nexpr) ) goto ErrR;
02603                         term += *term;
02604                     }
02605                 }
02606                 else {
02607                     if ( *term <= -MAXBUILTINFUNCTION ) {
02608                         if ( ( n = FindrNumber(-*term,&(renumber->func)) )
02609                          < 0 ) goto ErrR;
02610                         *term = -renumber->funnum[n];
02611                     }
02612                     else if ( *term == -SYMBOL ) {
02613                         term++;
02614                         if ( ( n = FindrNumber(*term,
02615                             &(renumber->symb)) ) < 0 ) goto ErrR;
02616                         *term = renumber->symnum[n];
02617                     }
```

```
02618                     else if ( *term == -INDEX ) {
02619                         term++;
02620                         if ( *term >= AM.OffsetIndex + WILDOFFSET ) {
02621 /*
02622             Still TOBEDONE
02623 */
02624                         }
02625                         else if ( ( *term  >= AM.OffsetIndex ) && ( *term < AM.IndDum ) ) {
02626                             if ( ( n = FindrNumber(*term,&(renumber->indi)) )
02627                                  < 0 ) goto ErrR;
02628                             *term = renumber->indnum[n];
02629                         }
02630                         else if ( *term < (WILDOFFSET+AM.OffsetVector) ) {
02631                             if ( ( n = FindrNumber(*term,&(renumber->vect)) )
02632                                  < 0 ) goto ErrR;
02633                             *term = renumber->vecnum[n];
02634                         }
02635                     }
02636                     else if ( *term == -VECTOR || *term == -MINVECTOR ) {
02637                         term++;
02638                         if ( ( n = FindrNumber(*term,&(renumber->vect)) )
02639                          < 0 ) goto ErrR;
02640                         *term = renumber->vecnum[n];
02641                     }
02642                 }
02643                 term = sarg;            /* Next argument */
02644             }
02645             term = t;
02646         }
02647     }
02648     return(0);
02649 ErrR:
02650     MesCall("TermRenumber");
02651     SETERROR(-1)
02652 }
02653
02654 /*
02655         #] TermRenumber :
02656         #[ FindrNumber :
02657 */
02658
02659 WORD FindrNumber(WORD n, VARRENUM *v)
02660 {
02661     WORD *hi,*med,*lo;
02662     hi = v->hi;
02663     lo = v->lo;
02664     med = v->start;
02665     if ( *hi == 0 ) {
02666         if ( n != *hi ) {
02667 /* INTERNAL_ERROR_EXCL_START */
02668             MesPrint("!>Serious problems coming up in FindrNumber");
02669             return(-1);
02670 /* INTERNAL_ERROR_EXCL_STOP */
02671         }
02672         return(*hi);
02673     }
02674     while ( *med != n ) {
02675         if ( *med < n ) {
02676             if ( med == hi ) goto ErrFindr;
02677             lo = med;
02678             med = hi - ((WORDDIF(hi,med))/2);
02679         }
02680         else {
02681             if ( med == lo ) goto ErrFindr;
02682             hi = med;
02683             med = lo + ((WORDDIF(med,lo))/2);
02684         }
02685     }
02686     return(WORDDIF(med,v->lo));
02687 ErrFindr:
02688 /*
02689     Reconstruction:
02690 */
02691     {
02692 /* INTERNAL_ERROR_EXCL_START */
02693         int i;
02694         i = WORDDIF(v->hi,v->lo);
02695         MesPrint("!>FindrNumber: n = %d, list has %d members",n,i);
02696         while ( i >= 0 ) {
02697             MesPrint("v->lo[%d] = %d",i,v->lo[i]); i--;
02698         }
02699         hi = v->hi;
02700         lo = v->lo;
02701         med = v->start;
02702         MesPrint("Start with %d,%d,%d",0,WORDDIF(med,v->lo),WORDDIF(hi,v->lo));
02703         while ( *med != n ) {
02704             if ( *med < n ) {
```

```
02705                 if ( med == hi ) goto ErrFindr2;
02706                 lo = med;
02707                 med = hi - ((WORDDIF(hi,med))/2);
02708             }
02709             else {
02710                 if ( med == lo ) goto ErrFindr2;
02711                 hi = med;
02712                 med = ((WORDDIF(med,lo))/2) + lo;
02713             }
02714             MesPrint("New: %d,%d,%d, *med =
      %d",WORDDIF(lo,v->lo),WORDDIF(med,v->lo),WORDDIF(hi,v->lo),*med);
02715         }
02716     }
02717     return(WORDDIF(med,v->lo));
02718 ErrFindr2:
02719     return(MesPrint("Renumbering problems"));
02720 /* INTERNAL_ERROR_EXCL_STOP */
02721 }
02722
02723 /*
02724         #] FindrNumber :
02725         #[ FindInIndex :
02726
02727         Finds an expression in the storage index if it exists.
02728         If found it returns a pointer to the index entry, otherwise zero.
02729         par = 0     Search by address (--> f == &AR.StoreData, called by GetTable, CoSave )
02730         par = 1     Search by name    (--> f == &AO.SaveData,  called by CoLoad )
02731
02732         When comparing parameter fields the parameters of the expression
02733         to be searched are in AT.TMaddr. This includes the primary expression
02734         and a possible FROMBRAC information. The FROMBRAC is always last.
02735
02736         The parameter mode tells whether we should worry about arguments of
02737         a stored expression.
02738 */
02739
02740 INDEXENTRY *FindInIndex(WORD expr, FILEDATA *f, WORD par, WORD mode)
02741 {
02742     GETIDENTITY
02743     INDEXENTRY *ind;
02744     WORD i, hand, *m;
02745     WORD *start, *stop, *stop2, *m2, nomatch = 0;
02746     POSITION stindex, indexpos, scrpos;
02747     LONG number, num;
02748     stindex = f->Position;
02749     m = AT.TMaddr;
02750     stop = m + m[1];
02751     m += SUBEXPSIZE;
02752     start = m;
02753     while ( m < stop ) {
02754         if ( *m == FROMBRAC || *m == WILDCARDS ) break;
02755         m += m[1];
02756     }
02757     stop = m;
02758     if ( !par ) hand = AR.StoreData.Handle;
02759     else        hand = AO.SaveData.Handle;
02760     for(;;) {
02761         if ( ( i = (WORD)BASEPOSITION(f->Index.number) ) != 0 ) {
02762             indexpos = f->Position;
02763             ADDPOS(indexpos,(2*sizeof(POSITION)));
02764             ind = f->Index.expression;
02765             do {
02766                 if ( ( !par && ISEQUALPOS(indexpos,Expressions[expr].onfile) )
02767                 || ( par && !StrCmp(EXPRNAME(expr),(UBYTE *)(ind->name)) ) ) {
02768                     nomatch = 1;
02769                     if ( par ) return(ind);
02770                     scrpos = ind->position;
02771                     SeekFile(hand,&scrpos,SEEK_SET);
02772                     if ( ISNOTEQUALPOS(scrpos,ind->position) ) goto ErrGt2;
02773                     if ( ReadFile(hand,(UBYTE *)AT.WorkPointer,(LONG)sizeof(WORD)) !=
02774                     sizeof(WORD) || !*AT.WorkPointer ) goto ErrGt2;
02775                     num = *AT.WorkPointer - 1;
02776                     num *= wsizeof(WORD);
02777                     if ( *AT.WorkPointer < 0 ||
02778                     ReadFile(hand,(UBYTE *)(AT.WorkPointer+1),num) != num ) goto ErrGt2;
02779                     m = start;  /* start of parameter field to be searched */
02780                     m2 = AT.WorkPointer + 1;
02781                     stop2 = m2 + m2[1];
02782                     m2 += SUBEXPSIZE;
02783                     while ( m < stop && m2 < stop2 ) {
02784                         if ( *m == SYMBOL ) {
02785                             if ( *m2 != SYMTOSYM ) break;
02786                             m2[3] = m[2];
02787                         }
02788                         else if ( *m == INDEX ) {
02789                             if ( m[2] >= 0 ) {
02790                                 if ( *m2 != INDTOIND ) break;
```

```
02791                               }
02792                               else {
02793                                   if ( *m2 != VECTOVEC ) break;
02794                               }
02795                               m2[3] = m[2];
02796                           }
02797                           else if ( *m >= FUNCTION ) {
02798                               if ( *m2 != FUNTOFUN ) break;
02799                               m2[3] = *m;
02800                           }
02801                           else {}
02802                           m += m[1];
02803                           m2 += m2[1];
02804                       }
02805                       if ( ( m >= stop && m2 >= stop2 ) || mode == 0 ) {
02806                           AT.WorkPointer = stop2;
02807
02808                           return(ind);
02809                       }
02810                   }
02811                   ind++;
02812                   ADDPOS(indexpos,sizeof(INDEXENTRY));
02813               } while ( --i > 0 );
02814           }
02815           f->Position = f->Index.next;
02816 #ifndef SYSDEPENDENTSAVE
02817           if ( !ISNOTZEROPOS(f->Position) ) ADDPOS(f->Position,sizeof(STOREHEADER));
02818           number = sizeof(struct FiLeInDeX);
02819 #endif
02820           if ( ISEQUALPOS(f->Position,stindex) && !AO.bufferedInd ) goto ErrGetTab;
02821           if ( !par ) {
02822               SeekFile(AR.StoreData.Handle,&(f->Position),SEEK_SET);
02823               if ( ISNOTEQUALPOS(f->Position,AR.StoreData.Position) ) goto ErrGt2;
02824 #ifndef SYSDEPENDENTSAVE
02825               if ( ReadFile(f->Handle, (UBYTE *)(&(f->Index)), number) != number ) goto ErrGt2;
02826 #endif
02827           }
02828           else {
02829               SeekFile(AO.SaveData.Handle,&(f->Position),SEEK_SET);
02830               if ( ISNOTEQUALPOS(f->Position,AO.SaveData.Position) ) goto ErrGt2;
02831 #ifndef SYSDEPENDENTSAVE
02832               if ( ReadSaveIndex(&f->Index) ) goto ErrGt2;
02833 #endif
02834           }
02835 #ifdef SYSDEPENDENTSAVE
02836           number = sizeof(struct FiLeInDeX);
02837           if ( ReadFile(f->Handle,(UBYTE *)(&(f->Index)),number) !=
02838           number ) goto ErrGt2;
02839 #endif
02840       }
02841 ErrGetTab:
02842       if ( nomatch ) {
02843           MesPrint("Parameters of expression %s don't match."
02844           ,EXPRNAME(expr));
02845       }
02846       else {
02847           MesPrint("Cannot find expression %s",EXPRNAME(expr));
02848       }
02849       return(0);
02850 ErrGt2:
02851 /* INTERNAL_ERROR_EXCL_START */
02852       MesPrint("!>Readerror in IndexSearch");
02853       return(0);
02854 /* INTERNAL_ERROR_EXCL_STOP */
02855 }
02856
02857 /*
02858           #] FindInIndex :
02859           #[ GetTable :
02860
02861           Locates stored files and constructs the renumbering tables.
02862           They are allocated in the WorkSpace.
02863           First the expression data are located. The Index is treated
02864           as a circularly linked buffer which is paged forwardly.
02865           If the indexentry is located (in ind) the two renumber tables
02866           have to be constructed.
02867           Finally the prototype has to be put in the proper buffer, so
02868           that wildcards can be passed. There should be a test with
02869           an already existing prototype that is constructed by the
02870           pattern matcher. This has not been put in yet.
02871
02872           There is a problem with the parallel processing.
02873           Feeding in the variables that were erased by a .store could in
02874           principle happen in different orders (ParFORM) or simultaneously
02875           (TFORM). The proper resolution is to have the compiler call GetTable
02876           when a stored expression is encountered.
02877
```

```
02878         This has been mended in development of TFORM by reading the
02879         symbol tables during compilation. See the call to GetTable
02880         in the CodeGenerator.
02881
02882         Next is the problem of FindInIndex which writes in AR.StoreData
02883         Copying this is expensive!
02884
02885         This Doesn't work well for TFORM yet.!!!!!!!!
02886         e[x1,x2] versus e[x2,x1] messes up.
02887         For the rest is the reloading during execution not thread safe.
02888
02889         The parameter mode tells whether we should worry about arguments
02890         of a stored expression.
02891 */
02892
02893 RENUMBER GetTable(WORD expr, POSITION *position, WORD mode)
02894 {
02895     GETIDENTITY
02896     WORD i, j;
02897     WORD *w;
02898     RENUMBER r;
02899     LONG num, nsize, xx;
02900     WORD jsym, jind, jvec, jfun;
02901     WORD k, type, error = 0, *oldw, *neww, *oldwork = AT.WorkPointer;
02902     struct SyMbOl SyM;
02903     struct InDeX InD;
02904     struct VeCtOr VeC;
02905     struct FuNcTiOn FuN;
02906     INDEXENTRY *ind;
02907 /*
02908     Prepare for FindInIndex to put the prototype in the WorkSpace.
02909     oldw will point at the "wildcards"
02910 */
02911 /*
02912         Bug fix. Look also in Generator.
02913 #ifndef WITHPTHREADS
02914
02915     if ( ( r = Expressions[expr].renum ) != 0 ) { }
02916     else {
02917         Expressions[expr].renum =
02918         r = (RENUMBER)Malloc1(sizeof(struct ReNuMbEr),"Renumber");
02919     }
02920 #else
02921         r = (RENUMBER)Malloc1(sizeof(struct ReNuMbEr),"Renumber");
02922 #endif
02923 */
02924         r = (RENUMBER)Malloc1(sizeof(struct ReNuMbEr),"Renumber");
02925
02926     oldw = AT.WorkPointer + 1 + SUBEXPSIZE;
02927 /*
02928     The prototype is loaded in the WorkSpace by the Index routine.
02929     After all it has to find an occurrence with the proper arguments.
02930     This sets the WorkPointer. Hence be careful now.
02931 */
02932     LOCK(AM.storefilelock);
02933     if ( ( ind = FindInIndex(expr,&AR.StoreData,0,mode) ) == 0 ) {
02934         UNLOCK(AM.storefilelock);
02935         return(0);
02936     }
02937
02938     xx = ind->nsymbols+ind->nindices+ind->nvectors+ind->nfunctions;
02939     if ( xx == 0 ) {
02940         Expressions[expr].renumlists =
02941         w = AN.dummyrenumlist;
02942     }
02943     else {
02944 /*
02945 #ifndef WITHPTHREADS
02946         Expressions[expr].renumlists =
02947 #endif
02948 */
02949         w = (WORD *)Malloc1(sizeof(WORD)*(xx*2),"VarSpace");
02950     }
02951     r->symb.lo = w;
02952     r->symb.start = w + ind->nsymbols/2;
02953     w += ind->nsymbols;
02954     r->symb.hi = w - 1;
02955     r->symnum = w;
02956     w += ind->nsymbols;
02957
02958     r->indi.lo = w;
02959     r->indi.start = w + ind->nindices/2;
02960     w += ind->nindices;
02961     r->indi.hi = w - 1;
02962     r->indnum = w;
02963     w += ind->nindices;
02964
```

```
02965     r->vect.lo = w;
02966     r->vect.start = w + ind->nvectors/2;
02967     w += ind->nvectors;
02968     r->vect.hi = w - 1;
02969     r->vecnum = w;
02970     w += ind->nvectors;
02971
02972     r->func.lo = w;
02973     r->func.start = w + ind->nfunctions/2;
02974     w += ind->nfunctions;
02975     r->func.hi = w - 1;
02976     r->funnum = w;
02977 /*  w += ind->nfunctions; */
02978
02979     SeekFile(AR.StoreData.Handle,&(ind->variables),SEEK_SET);
02980     *position = ind->position;
02981     jsym = ind->nsymbols;
02982     jvec = ind->nvectors;
02983     jind = ind->nindices;
02984     jfun = ind->nfunctions;
02985 /*
02986             #[ Symbols :
02987 */
02988     {
02989     SYMBOLS s = &SyM;
02990     w = r->symb.lo; j = jsym;
02991     for ( i = 0; i < j; i++ ) {
02992         if ( ReadFile(AR.StoreData.Handle,(UBYTE *)s,(LONG)(sizeof(struct SyMbOl)))
02993         != sizeof(struct SyMbOl) ) goto ErrGt2;
02994         nsize = s->namesize; nsize += sizeof(void *)-1;
02995         nsize &= -sizeof(void *);
02996         if ( ReadFile(AR.StoreData.Handle,(UBYTE *)(AT.WorkPointer),nsize)
02997         != nsize ) goto ErrGt2;
02998         *w = s->number;
02999         if ( ( s->flags & INUSE ) != 0 ) {
03000             /* Find the replacement. It must exist! */
03001             neww = oldw;
03002             while ( *neww != SYMTOSYM || neww[2] != *w ) neww += neww[1];
03003             k = neww[3];
03004         }
03005         else if ( GetVar((UBYTE *)AT.WorkPointer,&type,&k,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
03006             if ( type != CSYMBOL ) {
03007                 MesPrint("Error: Conflicting types for %s",(AT.WorkPointer));
03008                 error = -1;
03009             }
03010             else {
03011                 if ( ( s->complex & (VARTYPEIMAGINARY|VARTYPECOMPLEX) ) !=
03012                 ( symbols[k].complex & (VARTYPEIMAGINARY|VARTYPECOMPLEX) ) ) {
03013                     MesPrint("Warning: Conflicting complexity for %s",AT.WorkPointer);
03014                     error = -1;
03015                 }
03016                 if ( ( s->complex & (VARTYPEROOTOFUNITY) ) !=
03017                 ( symbols[k].complex & (VARTYPEROOTOFUNITY) ) ) {
03018                     MesPrint("Warning: Conflicting root of unity properties for %s",AT.WorkPointer);
03019                     error = -1;
03020                 }
03021                 if ( ( s->complex & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY ) {
03022                     if ( s->maxpower != symbols[k].maxpower ) {
03023                         MesPrint("Warning: Conflicting n in n-th root of unity properties for
      %s",AT.WorkPointer);
03024                         error = -1;
03025                     }
03026                 }
03027                 else if ( ( s->minpower !=
03028                 symbols[k].minpower || s->maxpower !=
03029                 symbols[k].maxpower ) && AC.WarnFlag ) {
03030                     MesPrint("Warning: Conflicting power restrictions for %s",AT.WorkPointer);
03031                 }
03032             }
03033         }
03034         else {
03035             if ( ( k = EntVar(CSYMBOL,(UBYTE *)(AT.WorkPointer),s->complex,s->minpower,
03036             s->maxpower,s->dimension) ) < 0 ) goto GetTcall;
03037         }
03038         *(w+j) = k;
03039         w++;
03040     }
03041     }
03042 /*
03043             #] Symbols :
03044             #[ Indices :
03045 */
03046     {
03047     INDICES s = &InD;
03048     w = r->indi.lo; j = jind;
03049     for ( i = 0; i < j; i++ ) {
03050         if ( ReadFile(AR.StoreData.Handle,(UBYTE *)s,(LONG)(sizeof(struct InDeX)))
```

```
03051         != sizeof(struct InDeX) ) goto ErrGt2;
03052         nsize = s->namesize; nsize += sizeof(void *)-1;
03053         nsize &= -sizeof(void *);
03054         if ( ReadFile(AR.StoreData.Handle,(UBYTE *)(AT.WorkPointer),nsize)
03055         != nsize ) goto ErrGt2;
03056         *w = s->number + AM.OffsetIndex;
03057         if ( s->dimension < 0 ) {   /* Relabel the dimension */
03058             s->dimension = -r->symnum[FindrNumber(-s->dimension,&(r->symb))];
03059             if ( s->nmin4 < -NMIN4SHIFT ) { /* Relabel n-4 */
03060                 s->nmin4 = -r->symnum[FindrNumber(-s->nmin4-NMIN4SHIFT
03061                 ,&(r->symb))]-NMIN4SHIFT;
03062             }
03063         }
03064         if ( ( s->flags & INUSE ) != 0 ) {
03065             /* Find the replacement. It must exist! */
03066             neww = oldw;
03067             while ( *neww != INDTOIND || neww[2] != *w ) neww += neww[1];
03068             k = neww[3] - AM.OffsetIndex;
03069         }
03070         else if ( s->type == DUMMY ) {
03071 /*
03072 --------->           Here we may have to execute some renumbering
03073 */
03074         }
03075         else if ( GetVar((UBYTE *)(AT.WorkPointer),&type,&k,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
03076             if ( type != CINDEX ) {
03077                 MesPrint("Error: Conflicting types for %s",(AT.WorkPointer));
03078                 error = -1;
03079             }
03080             else {
03081                 if ( s->type !=
03082                 indices[k].type ) {
03083                     MesPrint("Warning: %s is also a dummy index",(AT.WorkPointer));
03084                     error = -1;
03085                     goto GetTb3;
03086                 }
03087                 if ( s->dimension != indices[k].dimension ) {
03088                     MesPrint("Warning: Conflicting dimensions for %s",(AT.WorkPointer));
03089                     error = -1;
03090                 }
03091             }
03092         }
03093         else {
03094 GetTb3:
03095             if ( ( k = EntVar(CINDEX,(UBYTE *)(AT.WorkPointer),
03096             s->dimension,0,s->nmin4,0) ) < 0 ) goto GetTcall;
03097
03098         }
03099         *(w+j) = k + AM.OffsetIndex;
03100         w++;
03101     }
03102     }
03103 /*
03104             #] Indices :
03105             #[ Vectors :
03106 */
03107     {
03108     VECTORS s = &VeC;
03109     w = r->vect.lo; j = jvec;
03110     for ( i = 0; i < j; i++ ) {
03111         if ( ReadFile(AR.StoreData.Handle,(UBYTE *)s,(LONG)(sizeof(struct VeCtOr)))
03112         != sizeof(struct VeCtOr) ) goto ErrGt2;
03113         nsize = s->namesize; nsize += sizeof(void *)-1;
03114         nsize &= -sizeof(void *);
03115         if ( ReadFile(AR.StoreData.Handle,(UBYTE *)(AT.WorkPointer),nsize)
03116         != nsize ) goto ErrGt2;
03117         *w = s->number + AM.OffsetVector;
03118         if ( ( s->flags & INUSE ) != 0 ) {
03119             /* Find the replacement. It must exist! */
03120             neww = oldw;
03121             while ( *neww != VECTOVEC || neww[2] != *w ) neww += neww[1];
03122             k = neww[3] - AM.OffsetVector;
03123         }
03124         else if ( GetVar((UBYTE *)(AT.WorkPointer),&type,&k,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
03125             if ( type != CVECTOR ) {
03126                 MesPrint("Error: Conflicting types for %s",(AT.WorkPointer));
03127                 error = -1;
03128             }
03129             else {
03130                 if ( ( s->complex & (VARTYPEIMAGINARY|VARTYPECOMPLEX) ) !=
03131                 ( vectors[k].complex & (VARTYPEIMAGINARY|VARTYPECOMPLEX) ) ) {
03132                     MesPrint("Warning: Conflicting complexity for %s",(AT.WorkPointer));
03133                     error = -1;
03134                 }
03135             }
03136         }
03137         else {
```

```
03138             if ( ( k = EntVar(CVECTOR,(UBYTE *)(AT.WorkPointer),
03139             s->complex,0,0,s->dimension) ) < 0 ) goto GetTcall;
03140         }
03141         *(w+j) = k + AM.OffsetVector;
03142         w++;
03143     }
03144     }
03145 /*
03146             #] Vectors :
03147             #[ Functions :
03148 */
03149     {
03150     FUNCTIONS s = &FuN;
03151     w = r->func.lo; j = jfun;
03152     for ( i = 0; i < j; i++ ) {
03153         if ( ReadFile(AR.StoreData.Handle,(UBYTE *)s,(LONG)(sizeof(struct FuNcTiOn)))
03154         != sizeof(struct FuNcTiOn) ) goto ErrGt2;
03155         nsize = s->namesize; nsize += sizeof(void *)-1;
03156         nsize &= -sizeof(void *);
03157         if ( ReadFile(AR.StoreData.Handle,(UBYTE *)(AT.WorkPointer),nsize)
03158         != nsize ) goto ErrGt2;
03159         *w = s->number + FUNCTION;
03160         if ( ( s->flags & INUSE ) != 0 ) {
03161             /* Find the replacement. It must exist! */
03162             neww = oldw;
03163             while ( *neww != FUNTOFUN || neww[2] != *w ) neww += neww[1];
03164             k = neww[3] - FUNCTION;
03165         }
03166         else if ( GetVar((UBYTE *)(AT.WorkPointer),&type,&k,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
03167             if ( type != CFUNCTION ) {
03168                 MesPrint("Error: Conflicting types for %s",(AT.WorkPointer));
03169                 error = -1;
03170             }
03171             else {
03172                 if ( s->complex != functions[k].complex ) {
03173                     MesPrint("Warning: Conflicting complexity for %s",(AT.WorkPointer));
03174                     error = -1;
03175                 }
03176                 else if ( s->symmetric != functions[k].symmetric ) {
03177                     MesPrint("Warning: Conflicting symmetry properties for %s",(AT.WorkPointer));
03178                     error = -1;
03179                 }
03180                 else if ( ( s->maxnumargs != functions[k].maxnumargs )
03181                 || ( s->minnumargs != functions[k].minnumargs ) ) {
03182                     MesPrint("Warning: Conflicting argument restriction properties for
      %s",(AT.WorkPointer));
03183                     error = -1;
03184                 }
03185             }
03186         }
03187         else {
03188             if ( ( k = EntVar(CFUNCTION,(UBYTE *)(AT.WorkPointer),
03189             s->complex,s->commute,s->spec,s->dimension) ) < 0 ) goto GetTcall;
03190             functions[k].symmetric = s->symmetric;
03191             functions[k].maxnumargs = s->maxnumargs;
03192             functions[k].minnumargs = s->minnumargs;
03193         }
03194         *(w+j) = k + FUNCTION;
03195         w++;
03196     }
03197     }
03198 /*
03199             #] Functions :
03200
03201     Now we skip the prototype. This sets the start position at the first term
03202 */
03203     if ( error ) {
03204         UNLOCK(AM.storefilelock);
03205         AT.WorkPointer = oldwork;
03206         return(0);
03207     }
03208
03209     {
03210 /*
03211         For clarity we look where we are.
03212         We want to know: is this position already known?
03213         Could we have inserted extra information here?
03214
03215         nummystery indicates extra words. We have currently in order
03216         (if they exist)
03217             numdummies
03218             numfactors
03219             vflags
03220             uflags
03221 */
03222         POSITION pos;
03223         int nummystery;
```

```
03224         TELLFILE(AR.StoreData.Handle,&pos);
03225         nummystery = DIFBASE(ind->position,pos);
03226 /*
03227         MesPrint("--> We are at position       %8p",&pos);
03228         MesPrint("--> The index says        at %8p",&(ind->position));
03229         MesPrint("--> There are %d mystery bytes",nummystery);
03230 */
03231         if ( nummystery > 0 ) {
03232             if ( ReadFile(AR.StoreData.Handle,(UBYTE *)AT.WorkPointer,(LONG)sizeof(WORD)) !=
03233             sizeof(WORD) ) {
03234                 UNLOCK(AM.storefilelock);
03235                 AT.WorkPointer = oldwork;
03236                 return(0);
03237             }
03238             Expressions[expr].numdummies = *AT.WorkPointer;
03239 /*
03240             MesPrint("--> numdummies = %d",Expressions[expr].numdummies);
03241 */
03242             nummystery -= sizeof(WORD);
03243         }
03244         else {
03245             Expressions[expr].numdummies = 0;
03246         }
03247         if ( nummystery > 0 ) {
03248             if ( ReadFile(AR.StoreData.Handle,(UBYTE *)AT.WorkPointer,(LONG)sizeof(WORD)) !=
03249             sizeof(WORD) ) {
03250                 UNLOCK(AM.storefilelock);
03251                 AT.WorkPointer = oldwork;
03252                 return(0);
03253             }
03254             if ( ( AS.OldNumFactors == 0 ) || ( AS.NumOldNumFactors < NumExpressions ) ) {
03255                 WORD *buffer;
03256                 int capacity = 20;
03257                 if (capacity < NumExpressions) capacity = NumExpressions * 2;
03258
03259                 buffer = (WORD *)Malloc1(capacity * sizeof(WORD), "numfactors pointers");
03260                 if (AS.OldNumFactors) {
03261                     WCOPY(buffer, AS.OldNumFactors, AS.NumOldNumFactors);
03262                     M_free(AS.OldNumFactors, "numfactors pointers");
03263                 }
03264                 AS.OldNumFactors = buffer;
03265
03266                 buffer = (WORD *)Malloc1(capacity * sizeof(WORD), "vflags pointers");
03267                 if (AS.Oldvflags) {
03268                     WCOPY(buffer, AS.Oldvflags, AS.NumOldNumFactors);
03269                     M_free(AS.Oldvflags, "vflags pointers");
03270                 }
03271                 AS.Oldvflags = buffer;
03272
03273                 buffer = (WORD *)Malloc1(capacity * sizeof(WORD), "uflags pointers");
03274                 if (AS.Olduflags) {
03275                     WCOPY(buffer, AS.Olduflags, AS.NumOldNumFactors);
03276                     M_free(AS.Olduflags, "uflags pointers");
03277                 }
03278                 AS.Olduflags = buffer;
03279
03280                 AS.NumOldNumFactors = capacity;
03281             }
03282
03283             AS.OldNumFactors[expr] =
03284             Expressions[expr].numfactors = *AT.WorkPointer;
03285 /*
03286             MesPrint("--> numfactors = %d",Expressions[expr].numfactors);
03287 */
03288             nummystery -= sizeof(WORD);
03289         }
03290         else {
03291             Expressions[expr].numfactors = 0;
03292         }
03293         if ( nummystery > 0 ) {
03294             if ( ReadFile(AR.StoreData.Handle,(UBYTE *)AT.WorkPointer,(LONG)sizeof(WORD)) !=
03295             sizeof(WORD) ) {
03296                 UNLOCK(AM.storefilelock);
03297                 AT.WorkPointer = oldwork;
03298                 return(0);
03299             }
03300             AS.Oldvflags[expr] =
03301             Expressions[expr].vflags = *AT.WorkPointer;
03302 /*
03303             MesPrint("--> vflags = %d",Expressions[expr].vflags);
03304 */
03305             nummystery -= sizeof(WORD);
03306         }
03307         else {
03308             Expressions[expr].vflags = 0;
03309         }
03310         if ( nummystery > 0 ) {
```

```
03311             if ( ReadFile(AR.StoreData.Handle,(UBYTE *)AT.WorkPointer,(LONG)sizeof(WORD)) !=
03312             sizeof(WORD) ) {
03313                 UNLOCK(AM.storefilelock);
03314                 AT.WorkPointer = oldwork;
03315                 return(0);
03316             }
03317             AS.Olduflags[expr] =
03318             Expressions[expr].uflags = *AT.WorkPointer;
03319 /*
03320             MesPrint("--> uflags = %d",Expressions[expr].uflags);
03321 */
03322             nummystery -= sizeof(WORD);
03323         }
03324         else {
03325             Expressions[expr].uflags = 0;
03326         }
03327     }
03328
03329     SeekFile(AR.StoreData.Handle,&(ind->position),SEEK_SET);
03330     if ( ReadFile(AR.StoreData.Handle,(UBYTE *)AT.WorkPointer,(LONG)sizeof(WORD)) !=
03331     sizeof(WORD) || !*AT.WorkPointer ) {
03332         UNLOCK(AM.storefilelock);
03333         AT.WorkPointer = oldwork;
03334         return(0);
03335     }
03336     num = *AT.WorkPointer - 1;
03337     num *= sizeof(WORD);
03338     if ( *AT.WorkPointer < 0 ||
03339     ReadFile(AR.StoreData.Handle,(UBYTE *)(AT.WorkPointer+1),num) != num ) {
03340 /* INTERNAL_ERROR_EXCL_START */
03341         MesPrint("!>Error in stored expressions file at position %10p",*position);
03342         UNLOCK(AM.storefilelock);
03343         AT.WorkPointer = oldwork;
03344         return(0);
03345 /* INTERNAL_ERROR_EXCL_STOP */
03346     }
03347     UNLOCK(AM.storefilelock);
03348     ADDPOS(*position,num+sizeof(WORD));
03349     r->startposition = *position;
03350     AT.WorkPointer = oldwork;
03351     return(r);
03352 /* INTERNAL_ERROR_EXCL_START */
03353 GetTcall:
03354     UNLOCK(AM.storefilelock);
03355     AT.WorkPointer = oldwork;
03356     MesCall("GetTable");
03357     return(0);
03358 ErrGt2:
03359     UNLOCK(AM.storefilelock);
03360     AT.WorkPointer = oldwork;
03361     MesPrint("!>Readerror in GetTable");
03362     return(0);
03363 /* INTERNAL_ERROR_EXCL_STOP */
03364 }
03365
03366 /*
03367         #] GetTable :
03368         #[ CopyExpression :
03369
03370         Copies from one scratch buffer to another.
03371         We assume here that the complete 'from' scratch buffer is taken.
03372         We also assume that the 'from' buffer is positioned at the end of
03373         the expression.
03374
03375         The locks should be placed in the calling routine. We need basically
03376         AS.outputslock.
03377 */
03378
03379 int CopyExpression(FILEHANDLE *from, FILEHANDLE *to)
03380 {
03381     POSITION posfrom, poscopy;
03382     LONG fullsize,i;
03383     WORD *t1, *t2;
03384     int RetCode;
03385     SeekScratch(from,&posfrom);
03386     if ( from->handle < 0 ) { /* input is in memory */
03387         fullsize = (BASEPOSITION(posfrom))/sizeof(WORD);
03388         if ( ( to->POstop - to->POfull ) >= fullsize ) {
03389 /*
03390             Fits inside the buffer of the output. This will be fast.
03391 */
03392             t1 = from->PObuffer;
03393             t2 = to->POfull;
03394             NCOPY(t2,t1,fullsize)
03395             to->POfull = to->POfill = t2;
03396             goto WriteTrailer;
03397         }
```

```
03398         if ( to->handle < 0 ) {       /* First open the file */
03399             if ( ( RetCode = CreateFile(to->name) ) >= 0 ) {
03400                 to->handle = (WORD)RetCode;
03401                 PUTZERO(to->filesize);
03402                 PUTZERO(to->POposition);
03403             }
03404             else {
03405                 MLOCK(ErrorMessageLock);
03406                 MesPrint("Cannot create scratch file %s",to->name);
03407                 MUNLOCK(ErrorMessageLock);
03408                 return(-1);
03409             }
03410         }
03411         t1 = from->POobuffer;
03412         while ( fullsize > 0 ) {
03413             i = to->POstop - to->POfull;
03414             if ( i > fullsize ) i = fullsize;
03415             fullsize -= i;
03416             t2 = to->POfull;
03417             NCOPY(t2,t1,i)
03418             if ( fullsize > 0 ) {
03419                 SeekFile(to->handle,&(to->POposition),SEEK_SET);
03420                 if ( WriteFile(to->handle,((UBYTE *)(to->POobuffer)),to->POsize) != to->POsize ) {
03421                     MLOCK(ErrorMessageLock);
03422                     MesPrint("Error while writing to disk. Disk full?");
03423                     MUNLOCK(ErrorMessageLock);
03424                     return(-1);
03425                 }
03426                 ADDPOS(to->POposition,to->POsize);
03427 /*              SeekFile(to->handle,&(to->POposition),SEEK_CUR); */
03428                 to->filesize = to->POposition;
03429                 to->POfill = to->POfull = to->POobuffer;
03430             }
03431             else {
03432                 to->POfill = to->POfull = t2;
03433             }
03434         }
03435         goto WriteTrailer;
03436     }
03437 /*
03438     Now the input involves a file. This needs the use of the POobuffer of from.
03439     First make sure the tail of the buffer has been written
03440 */
03441     if ( ((UBYTE *)(from->POfill)-(UBYTE *)(from->POobuffer)) > 0 ) {
03442         if ( WriteFile(from->handle,((UBYTE *)(from->POobuffer)),((UBYTE *)(from->POfill)-(UBYTE
      *)(from->POobuffer)))
03443         != ((UBYTE *)(from->POfill)-(UBYTE *)(from->POobuffer)) ) {
03444             MLOCK(ErrorMessageLock);
03445             MesPrint("Error while writing to disk. Disk full?");
03446             MUNLOCK(ErrorMessageLock);
03447             return(-1);
03448         }
03449         SeekFile(from->handle,&(from->POposition),SEEK_CUR);
03450         posfrom = from->filesize = from->POposition;
03451         from->POfill = from->POfull = from->POobuffer;
03452     }
03453 /*
03454     Now copy the complete contents
03455 */
03456     PUTZERO(poscopy);
03457     SeekFile(from->handle,&poscopy,SEEK_SET);
03458     while ( ISLESSPOS(poscopy,posfrom) ) {
03459         fullsize = ReadFile(from->handle,((UBYTE *)(from->POobuffer)),from->POsize);
03460         if ( fullsize < 0 || ( fullsize % sizeof(WORD) ) != 0 ) {
03461 /* INTERNAL_ERROR_EXCL_START */
03462             MLOCK(ErrorMessageLock);
03463             MesPrint("!>Error while reading from disk while copying expression.");
03464             MUNLOCK(ErrorMessageLock);
03465             return(-1);
03466 /* INTERNAL_ERROR_EXCL_STOP */
03467         }
03468         fullsize /= sizeof(WORD);
03469         from->POfull = from->POobuffer + fullsize;
03470         t1 = from->POobuffer;
03471
03472         if ( ( to->POstop - to->POfull ) >= fullsize ) {
03473 /*
03474             Fits inside the buffer of the output. This will be fast.
03475 */
03476             t2 = to->POfull;
03477             NCOPY(t2,t1,fullsize)
03478             to->POfill = to->POfull = t2;
03479         }
03480         else {
03481           if ( to->handle < 0 ) {     /* First open the file */
03482             if ( ( RetCode = CreateFile(to->name) ) >= 0 ) {
03483                 to->handle = (WORD)RetCode;
```

```
03484                 PUTZERO(to->POposition);
03485                 PUTZERO(to->filesize);
03486               }
03487               else {
03488                 MLOCK(ErrorMessageLock);
03489                 MesPrint("Cannot create scratch file %s",to->name);
03490                 MUNLOCK(ErrorMessageLock);
03491                 return(-1);
03492               }
03493             }
03494             while ( fullsize > 0 ) {
03495               i = to->POstop - to->POfull;
03496               if ( i > fullsize ) i = fullsize;
03497               fullsize -= i;
03498               t2 = to->POfull;
03499               NCOPY(t2,t1,i)
03500               if ( fullsize > 0 ) {
03501                 SeekFile(to->handle,&(to->POposition),SEEK_SET);
03502                 if ( WriteFile(to->handle,((UBYTE *)(to->PObuffer)),to->POsize) != to->POsize ) {
03503                   MLOCK(ErrorMessageLock);
03504                   MesPrint("Error while writing to disk. Disk full?");
03505                   MUNLOCK(ErrorMessageLock);
03506                   return(-1);
03507                 }
03508                 ADDPOS(to->POposition,to->POsize);
03509 /*              SeekFile(to->handle,&(to->POposition),SEEK_CUR); */
03510                 to->filesize = to->POposition;
03511                 to->POfill = to->POfull = to->PObuffer;
03512               }
03513               else {
03514                 to->POfill = to->POfull = t2;
03515               }
03516             }
03517           }
03518           SeekFile(from->handle,&poscopy,SEEK_CUR);
03519       }
03520 WriteTrailer:
03521       if ( ( to->handle >= 0 ) && ( to->POfill > to->PObuffer ) ) {
03522           fullsize = (UBYTE *)(to->POfill) - (UBYTE *)(to->PObuffer);
03523 /*
03524           PUTZERO(to->POposition);
03525           SeekFile(to->handle,&(to->POposition),SEEK_END);
03526 */
03527           SeekFile(to->handle,&(to->filesize),SEEK_SET);
03528           if ( WriteFile(to->handle,((UBYTE *)(to->PObuffer)),fullsize) != fullsize ) {
03529               MLOCK(ErrorMessageLock);
03530               MesPrint("Error while writing to disk. Disk full?");
03531               MUNLOCK(ErrorMessageLock);
03532               return(-1);
03533           }
03534           ADDPOS(to->filesize,fullsize);
03535           to->POposition = to->filesize;
03536           to->POfill = to->POfull = to->PObuffer;
03537       }
03538
03539       return(0);
03540 }
03541
03542 /*
03543           #] CopyExpression :
03544           #[ ExprStatus :
03545 */
03546
03547 #ifdef HIDEDEBUG
03548
03549 static UBYTE *statusexpr[] = {
03550       (UBYTE *)"LOCALEXPRESSION"
03551     ,(UBYTE *)"SKIPLEXPRESSION"
03552     ,(UBYTE *)"DROPLEXPRESSION"
03553     ,(UBYTE *)"DROPPEDEXPRESSION"
03554     ,(UBYTE *)"GLOBALEXPRESSION"
03555     ,(UBYTE *)"SKIPGEXPRESSION"
03556     ,(UBYTE *)"DROPGEXPRESSION"
03557     ,(UBYTE *)"UNKNOWN"
03558     ,(UBYTE *)"STOREDEXPRESSION"
03559     ,(UBYTE *)"HIDDENLEXPRESSION"
03560     ,(UBYTE *)"HIDELEXPRESSION"
03561     ,(UBYTE *)"DROPHLEXPRESSION"
03562     ,(UBYTE *)"UNHIDELEXPRESSION"
03563     ,(UBYTE *)"HIDDENGEXPRESSION"
03564     ,(UBYTE *)"HIDEGEXPRESSION"
03565     ,(UBYTE *)"DROPHGEXPRESSION"
03566     ,(UBYTE *)"UNHIDEGEXPRESSION"
03567     ,(UBYTE *)"INTOHIDELEXPRESSION"
03568     ,(UBYTE *)"INTOHIDEGEXPRESSION"
03569 };
03570
```

```
03571 void ExprStatus(EXPRESSIONS e)
03572 {
03573     MesPrint("Expression %s(%d) has status %s(%d,%d). Buffer: %d, Position: %15p",
03574         AC.exprnames->namebuffer+e->name,(WORD)(e-Expressions),
03575         statusexpr[e->status],e->status,e->hidelevel,
03576         e->whichbuffer,&(e->onfile));
03577 }
03578
03579 #endif
03580
03581 /*
03582         #] ExprStatus :
03583     #] StoreExpressions :
03584     #[ System Independent Saved Expressions :
03585
03586     All functions concerned with the system independent reading of save-files
03587     are here. They are called by the functions CoLoad, PutInStore,
03588     SetFileIndex, FindInIndex. In case no translation (endianness flip,
03589     resizing of words, renumbering) has to be done, they just do simple file
03590     reading. The function SaveFileHeader() for writing a header with
03591     information about the system architecture, FORM version, etc. is also
03592     located here.
03593
03594         #[ Flip :
03595 */
03596
03606 static void FlipN(UBYTE *p, int length)
03607 {
03608     UBYTE *q, buf;
03609     q = p + length;
03610     do {
03611         --q;
03612         buf = *p; *p = *q; *q = buf;
03613     } while ( ++p != q );
03614 }
03615
03625 static void Flip16(UBYTE *p)
03626 {
03627     uint16_t in = *((uint16_t *)p);
03628     uint16_t out = (uint16_t)( (((in) » 8) & UINT16_C(0x00FF)) | (((in) « 8) & UINT16_C(0xFF00)) );
03629     *((uint16_t *)p) = out;
03630 }
03631
03633 static void Flip32(UBYTE *p)
03634 {
03635     uint32_t in = *((uint32_t *)p);
03636     uint32_t out =
03637         ( (((in) » 24) & UINT32_C(0x000000FF)) | (((in) »  8) & UINT32_C(0x0000FF00)) | \
03638           (((in) «  8) & UINT32_C(0x00FF0000)) | (((in) « 24) & UINT32_C(0xFF000000)) );
03639     *((uint32_t *)p) = out;
03640 }
03641
03643 #ifdef UINT64_C
03644 static void Flip64(UBYTE *p)
03645 {
03646     uint64_t in = *((uint64_t *)p);
03647     uint64_t out =
03648         ( (((in) » 56) & UINT64_C(0x00000000000000FF)) | (((in) » 40) & UINT64_C(0x000000000000FF00))
    | \
03649           (((in) » 24) & UINT64_C(0x0000000000FF0000)) | (((in) »  8) & UINT64_C(0x00000000FF000000))
    | \
03650           (((in) «  8) & UINT64_C(0x000000FF00000000)) | (((in) « 24) & UINT64_C(0x0000FF0000000000))
    | \
03651           (((in) « 40) & UINT64_C(0x00FF000000000000)) | (((in) « 56) & UINT64_C(0xFF00000000000000))
    );
03652     *((uint64_t *)p) = out;
03653 }
03654 #else
03655 static void Flip64(UBYTE *p) { FlipN(p, 8); }
03656 #endif /* UINT64_C */
03657
03659 static void Flip128(UBYTE *p) { FlipN(p, 16); }
03660
03661 /*
03662         #] Flip :
03663         #[ Resize :
03664 */
03665
03677 static void ResizeDataBE(UBYTE *src, int slen, UBYTE *dst, int dlen)
03678 {
03679     if ( slen > dlen ) {
03680         src += slen - dlen;
03681         while ( dlen-- ) { *dst++ = *src++; }
03682     }
03683     else {
03684         int i = dlen - slen;
03685         while ( i-- ) { *dst++ = 0; }
```

```
03686         while ( slen-- ) { *dst++ = *src++; }
03687     }
03688 }
03689
03693 static void ResizeDataLE(UBYTE *src, int slen, UBYTE *dst, int dlen)
03694 {
03695     if ( slen > dlen ) {
03696         while ( dlen-- ) { *dst++ = *src++; }
03697     }
03698     else {
03699         int i = dlen - slen;
03700         while ( slen-- ) { *dst++ = *src++; }
03701         while ( i-- ) { *dst++ = 0; }
03702     }
03703 }
03704
03717 static void Resize16t16(UBYTE *src, UBYTE *dst)
03718 {
03719     *((int16_t *)dst) = *((int16_t *)src);
03720 }
03721
03723 static void Resize16t32(UBYTE *src, UBYTE *dst)
03724 {
03725     int16_t in = *((int16_t *)src);
03726     int32_t out = (int32_t)in;
03727     *((int32_t *)dst) = out;
03728 }
03729
03731 #ifdef INT64_C
03732 static void Resize16t64(UBYTE *src, UBYTE *dst)
03733 {
03734     int16_t in = *((int16_t *)src);
03735     int64_t out = (int64_t)in;
03736     *((int64_t *)dst) = out;
03737 }
03738 #else
03739 static void Resize16t64(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 2, dst, 8); }
03740 #endif /* INT64_C */
03741
03743 static void Resize16t128(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 2, dst, 16); }
03744
03746 static void Resize32t32(UBYTE *src, UBYTE *dst)
03747 {
03748     *((int32_t *)dst) = *((int32_t *)src);
03749 }
03750
03752 #ifdef INT64_C
03753 static void Resize32t64(UBYTE *src, UBYTE *dst)
03754 {
03755     int32_t in = *((int32_t *)src);
03756     int64_t out = (int64_t)in;
03757     *((int64_t *)dst) = out;
03758 }
03759 #else
03760 static void Resize32t64(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 4, dst, 8); }
03761 #endif /* INT64_C */
03762
03764 static void Resize32t128(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 4, dst, 16); }
03765
03767 #ifdef INT64_C
03768 static void Resize64t64(UBYTE *src, UBYTE *dst)
03769 {
03770     *((int64_t *)dst) = *((int64_t *)src);
03771 }
03772 #else
03773 static void Resize64t64(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 8, dst, 8); }
03774 #endif /* INT64_C */
03775
03777 static void Resize64t128(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 8, dst, 16); }
03778
03780 static void Resize128t128(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 16); }
03781
03783 static void Resize32t16(UBYTE *src, UBYTE *dst)
03784 {
03785     int32_t in = *((int32_t *)src);
03786     int16_t out = (int16_t)in;
03787     if ( in > (1«15)-1 || in < -(1«15)+1 ) AO.resizeFlag |= 1;
03788     *((int16_t *)dst) = out;
03789 }
03790
03797 static void Resize32t16NC(UBYTE *src, UBYTE *dst)
03798 {
03799     int32_t in = *((int32_t *)src);
03800     int16_t out = (int16_t)in;
03801     *((int16_t *)dst) = out;
03802 }
03803
```

```
03804 #ifdef INT64_C
03806 static void Resize64t16(UBYTE *src, UBYTE *dst)
03807 {
03808     int64_t in = *((int64_t *)src);
03809     int16_t out = (int16_t)in;
03810     if ( in > (1«15)-1 || in < -(1«15)+1 ) AO.resizeFlag |= 1;
03811     *((int16_t *)dst) = out;
03812 }
03814 static void Resize64t16NC(UBYTE *src, UBYTE *dst)
03815 {
03816     int64_t in = *((int64_t *)src);
03817     int16_t out = (int16_t)in;
03818     *((int16_t *)dst) = out;
03819 }
03820 #else
03822 static void Resize64t16(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 8, dst, 2); }
03824 static void Resize64t16NC(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 8, dst, 2); }
03825 #endif /* INT64_C */
03826
03827 #ifdef INT64_C
03829 static void Resize64t32(UBYTE *src, UBYTE *dst)
03830 {
03831     int64_t in = *((int64_t *)src);
03832     int32_t out = (int32_t)in;
03833     if ( in > (INT64_C(1)«31)-1 || in < -(INT64_C(1)«31)+1 ) AO.resizeFlag |= 1;
03834     *((int32_t *)dst) = out;
03835 }
03837 static void Resize64t32NC(UBYTE *src, UBYTE *dst)
03838 {
03839     int64_t in = *((int64_t *)src);
03840     int32_t out = (int32_t)in;
03841     *((int32_t *)dst) = out;
03842 }
03843 #else
03845 static void Resize64t32(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 8, dst, 4); }
03847 static void Resize64t32NC(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 8, dst, 4); }
03848 #endif /* INT64_C */
03849
03851 static void Resize128t16(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 2); }
03852
03854 static void Resize128t16NC(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 2); }
03855
03857 static void Resize128t32(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 4); }
03858
03860 static void Resize128t32NC(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 4); }
03861
03863 static void Resize128t64(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 8); }
03864
03866 static void Resize128t64NC(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 8); }
03867
03868 /*
03869         #] Resize :
03870         #[ CheckPower and RenumberVec :
03871 */
03872
03879 static void CheckPower32(UBYTE *p)
03880 {
03881     if ( *((int32_t *)p) < -MAXPOWER ) {
03882         AO.powerFlag |= 0x01;
03883         *((int32_t *)p) = -MAXPOWER;
03884     }
03885     p += sizeof(int32_t);
03886     if ( *((int32_t *)p) > MAXPOWER ) {
03887         AO.powerFlag |= 0x02;
03888         *((int32_t *)p) = MAXPOWER;
03889     }
03890 }
03891
03899 static void RenumberVec32(UBYTE *p)
03900 {
03901 /*  int32_t wildoffset = *((int32_t *)AO.SaveHeader.wildoffset); */
03902     void *dummy = (void *)AO.SaveHeader.wildoffset;  /* to remove a warning about strict-aliasing
      rules in gcc */
03903     int32_t wildoffset = *(int32_t *)dummy;
03904     int32_t in = *((int32_t *)p);
03905     in = in + 2*wildoffset;
03906     in = in - 2*WILDOFFSET;
03907     *((int32_t *)p) = in;
03908 }
03909
03910 /*
03911         #] CheckPower and RenumberVec :
03912         #[ ResizeCoeff :
03913 */
03914
03928 static void ResizeCoeff32(UBYTE **bout, UBYTE *bend, UBYTE *top)
03929 {
```

```
03930     int i;
03931     int32_t sign;
03932     int32_t *in, *p;
03933     int32_t *out = (int32_t *)*bout;
03934     int32_t *end = (int32_t *)bend;
03935 
03936     if ( sizeof(WORD) == 2 ) {
03937         /* 4 -> 2 */
03938         int32_t len = (end - 1 - out) / 2;
03939         int zeros = 2;
03940         p = out + len - 1;
03941 
03942         if ( *p & 0xFFFF0000 ) --zeros;
03943         p += len;
03944         if ( *p & 0xFFFF0000 ) --zeros;
03945 
03946         in = end - 1;
03947         sign = ( *in-- > 0 ) ? 1 : -1;
03948         p = out + 4*len;
03949         if ( zeros == 2 ) p -= 2;
03950         out = p--;
03951 
03952         if ( zeros < 2 ) *p-- = *in » 16;
03953         *p-- = *in-- & 0x0000FFFF;
03954         for ( i = 1; i < len; ++i ) {
03955             *p-- = *in » 16;
03956             *p-- = *in-- & 0x0000FFFF;
03957         }
03958         if ( zeros < 2 ) *p-- = *in » 16;
03959         *p-- = *in-- & 0x0000FFFF;
03960         for ( i = 1; i < len; ++i ) {
03961             *p-- = *in » 16;
03962             *p-- = *in-- & 0x0000FFFF;
03963         }
03964 
03965         *out = (out - p) * sign;
03966         *bout = (UBYTE *)(out+1);
03967 
03968     }
03969     else {
03970         /* 2 -> 4 */
03971         int32_t len = (end - 1 - out) / 2;
03972         if ( len == 1 ) {
03973             *out = *(uint16_t *)out;
03974             ++out;
03975             *out = *(uint16_t *)out;
03976             ++out;
03977             ++out;
03978         }
03979         else {
03980             p = out;
03981             *out = *(uint16_t *)out;
03982             in = out + 1;
03983             for ( i = 1; i < len; ++i ) {
03984                 /* shift */
03985                 *out = (uint32_t)(*(uint16_t *)out)
03986                     + ((uint32_t)(*(uint16_t *)in) « 16);
03987                 ++in;
03988                 if ( ++i == len ) break;
03989                 /* copy */
03990                 ++out;
03991                 *out = *(uint16_t *)in;
03992                 ++in;
03993             }
03994             ++out;
03995             *out = *(uint16_t *)in;
03996             ++in;
03997             for ( i = 1; i < len; ++i ) {
03998                 /* shift */
03999                 *out = (uint32_t)(*(uint16_t *)out)
04000                     + ((uint32_t)(*(uint16_t *)in) « 16);
04001                 ++in;
04002                 if ( ++i == len ) break;
04003                 /* copy */
04004                 ++out;
04005                 *out = *(uint16_t *)in;
04006                 ++in;
04007             }
04008             ++out;
04009             if ( *in < 0 ) *out = -(out - p + 1);
04010             else *out = out - p + 1;
04011             ++out;
04012         }
04013 
04014         if ( out > (int32_t *)top ) {
04015 /* INTERNAL_ERROR_EXCL_START */
04016             MesPrint("!>Error in resizing coefficient!");
```

```
04017 /* INTERNAL_ERROR_EXCL_STOP */
04018         }
04019
04020         *bout = (UBYTE *)out;
04021     }
04022 }
04023
04024 /*
04025         #] ResizeCoeff :
04026         #[ WriteStoreHeader :
04027 */
04028
04029 #define SAVEREVISION 0x02
04030
04040 LONG WriteStoreHeader(WORD handle)
04041 {
04042     /* template of the STOREHEADER */
04043     static STOREHEADER sh = {
04044         { 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF }, /* store header mark */
04045         0, 0, 0, 0,                                         /* sizeof of WORD,LONG,POSITION,void* */
04046         { 0 },                                              /* endianness check number */
04047         0, 0, 0, 0,                                         /* sizeof variable structs */
04048         { 0 },                                              /* maxpower */
04049         { 0 },                                              /* wildoffset */
04050         SAVEREVISION,                                       /* revision */
04051         { 0 } };                                            /* reserved */
04052     int endian, i;
04053
04054     /* if called for the first time ... */
04055     if ( sh.lenWORD == 0 ) {
04056         sh.lenWORD = sizeof(WORD);
04057         sh.lenLONG = sizeof(LONG);
04058         sh.lenPOS = sizeof(POSITION);
04059         sh.lenPOINTER = sizeof(void *);
04060
04061         endian = 1;
04062         for ( i = 1; i < (int)sizeof(int); ++i ) {
04063             endian <<= 8;
04064             endian += i+1;
04065         }
04066         for ( i = 0; i < (int)sizeof(int); ++i ) sh.endianness[i] = ((char *)&endian)[i];
04067
04068         sh.sSym = sizeof(struct SyMbOl);
04069         sh.sInd = sizeof(struct InDeX);
04070         sh.sVec = sizeof(struct VeCtOr);
04071         sh.sFun = sizeof(struct FuNcTiOn);
04072
04073 /*      *((WORD *)sh.maxpower) = MAXPOWER;
04074         *((WORD *)sh.wildoffset) = WILDOFFSET; */
04075         {
04076             WORD dumw[8];
04077             UBYTE *dummy;
04078             for ( i = 0; i < 8; i++ ) dumw[i] = 0;
04079             dummy = (UBYTE *)dumw;
04080             dumw[0] = (WORD)MAXPOWER;
04081             for ( i = 0; i < 16; i++ ) sh.maxpower[i] = dummy[i];
04082             dumw[0] = (WORD)WILDOFFSET;
04083             for ( i = 0; i < 16; i++ ) sh.wildoffset[i] = dummy[i];
04084         }
04085     }
04086
04087     return ( WriteFile(handle,(UBYTE *)(&sh),(LONG)(sizeof(STOREHEADER)))
04088              != (LONG)(sizeof(STOREHEADER)) );
04089 }
04090
04091 /*
04092         #] WriteStoreHeader :
04093         #[ CompactifySizeof :
04094 */
04095
04103 static unsigned int CompactifySizeof(unsigned int size)
04104 {
04105     switch ( size ) {
04106         case  2: return 0;
04107         case  4: return 1;
04108         case  8: return 2;
04109         case 16: return 3;
04110         default:
04111 /* INTERNAL_ERROR_EXCL_START */
04112             MesPrint("!>Error compactifying size.");
04113           return 3;
04114 /* INTERNAL_ERROR_EXCL_STOP */
04115     }
04116 }
04117
04118 /*
04119         #] CompactifySizeof :
```

```
04120         #[ ReadSaveHeader :
04121 */
04122
04136 int ReadSaveHeader(void)
04137 {
04138     /* Read-only tables of function pointers for conversions. */
04139     static void (*flipJumpTable[4])(UBYTE *) =
04140         { Flip16, Flip32, Flip64, Flip128 };
04141     static void (*resizeJumpTable[4][4])(UBYTE *, UBYTE *) = /* "own x saved"-sizes  */
04142         { { Resize16t16,  Resize32t16,  Resize64t16,  Resize128t16  },
04143           { Resize16t32,  Resize32t32,  Resize64t32,  Resize128t32  },
04144           { Resize16t64,  Resize32t64,  Resize64t64,  Resize128t64  },
04145           { Resize16t128, Resize32t128, Resize64t128, Resize128t128 } };
04146     static void (*resizeNCJumpTable[4][4])(UBYTE *, UBYTE *) = /* "own x saved"-sizes  */
04147         { { Resize16t16,  Resize32t16NC,  Resize64t16NC,  Resize128t16NC  },
04148           { Resize16t32,  Resize32t32,  Resize64t32NC,  Resize128t32NC  },
04149           { Resize16t64,  Resize32t64,  Resize64t64,  Resize128t64NC  },
04150           { Resize16t128, Resize32t128, Resize64t128, Resize128t128 } };
04151
04152     int endian, i;
04153     WORD idxW = CompactifySizeof(sizeof(WORD));
04154     WORD idxL = CompactifySizeof(sizeof(LONG));
04155     WORD idxP = CompactifySizeof(sizeof(POSITION));
04156     WORD idxVP = CompactifySizeof(sizeof(void *));
04157
04158     AO.transFlag = 0;
04159     AO.powerFlag = 0;
04160     AO.resizeFlag = 0;
04161     AO.bufferedInd = 0;
04162
04163     if ( ReadFile(AO.SaveData.Handle,(UBYTE *)(&AO.SaveHeader),
04164         (LONG)sizeof(STOREHEADER)) != (LONG)sizeof(STOREHEADER) ) {
04165 /* INTERNAL_ERROR_EXCL_START */
04166             return(MesPrint("!>Error reading save file header"));
04167 /* INTERNAL_ERROR_EXCL_STOP */
04168         }
04169
04170     /* check whether save-file has no header. if yes then it is an old version
04171        of FORM -> go back to position 0 in file which then contains the first
04172        index and skip the rest. */
04173     for ( i = 0; i < 8; ++i ) {
04174         if ( AO.SaveHeader.headermark[i] != 0xFF ) {
04175             POSITION p;
04176             PUTZERO(p);
04177             SeekFile(AO.SaveData.Handle, &p, SEEK_SET);
04178             return ( 0 );
04179         }
04180     }
04181
04182     if ( AO.SaveHeader.revision != SAVEREVISION ) {
04183         return(MesPrint("Save file header from an old version. Cannot read this file."));
04184     }
04185
04186     endian = 1;
04187     for ( i = 1; i < (int)sizeof(int); ++i ) {
04188         endian <<= 8;
04189         endian += i+1;
04190     }
04191     if ( ((char *)&endian)[0] < ((char *)&endian)[1] ) {
04192         /* this machine is big-endian */
04193         AO.ResizeData = ResizeDataBE;
04194     }
04195     else {
04196         /* this machine is little-endian */
04197         AO.ResizeData = ResizeDataLE;
04198     }
04199
04200     /* set AO.transFlag if ANY conversion has to be done later */
04201     if ( AO.SaveHeader.endianness[0] > AO.SaveHeader.endianness[1] ) {
04202         AO.transFlag = ( ((char *)&endian)[0] < ((char *)&endian)[1] );
04203     }
04204     else {
04205         AO.transFlag = ( ((char *)&endian)[0] > ((char *)&endian)[1] );
04206     }
04207     if ( (WORD)AO.SaveHeader.lenWORD != sizeof(WORD) ) AO.transFlag |= 0x02;
04208     if ( (WORD)AO.SaveHeader.lenLONG != sizeof(LONG) ) AO.transFlag |= 0x04;
04209     if ( (WORD)AO.SaveHeader.lenPOS != sizeof(POSITION) ) AO.transFlag |= 0x08;
04210     if ( (WORD)AO.SaveHeader.lenPOINTER != sizeof(void *) ) AO.transFlag |= 0x10;
04211
04212     AO.FlipWORD = flipJumpTable[idxW];
04213     AO.FlipLONG = flipJumpTable[idxL];
04214     AO.FlipPOS = flipJumpTable[idxP];
04215     AO.FlipPOINTER = flipJumpTable[idxVP];
04216
04217     /* Works only for machines where WORD is not greater than 32bit ! */
04218     AO.CheckPower = CheckPower32;
04219     AO.RenumberVec = RenumberVec32;
```

```
04220
04221     AO.ResizeWORD = resizeJumpTable[idxW][CompactifySizeof(AO.SaveHeader.lenWORD)];
04222     AO.ResizeNCWORD = resizeNCJumpTable[idxW][CompactifySizeof(AO.SaveHeader.lenWORD)];
04223     AO.ResizeLONG = resizeJumpTable[idxL][CompactifySizeof(AO.SaveHeader.lenLONG)];
04224     AO.ResizePOS = resizeJumpTable[idxP][CompactifySizeof(AO.SaveHeader.lenPOS)];
04225     AO.ResizePOINTER = resizeJumpTable[idxVP][CompactifySizeof(AO.SaveHeader.lenPOINTER)];
04226
04227     {
04228         WORD dumw[8];
04229         UBYTE *dummy;
04230         for ( i = 0; i < 8; i++ ) dumw[i] = 0;
04231         dummy = (UBYTE *)dumw;
04232         for ( i = 0; i < 16; i++ ) dummy[i] = AO.SaveHeader.maxpower[i];
04233         AO.mpower = dumw[0];
04234     }
04235
04236     return ( 0 );
04237 }
04238
04239 /*
04240         #] ReadSaveHeader :
04241         #[ ReadSaveIndex :
04242 */
04243
04257 LONG ReadSaveIndex(FILEINDEX *fileind)
04258 {
04259     /* do we need some translation for the FILEINDEX? */
04260     if ( AO.transFlag ) {
04261         /* if a translated FILEINDEX can hold less entries than the original
04262            FILEINDEX, then we need to buffer the extra entries in this static
04263            variable (can happen going from 32bit to 64bit */
04264         static FILEINDEX sbuffer;
04265
04266         FILEINDEX buffer;
04267         UBYTE *p, *q;
04268         int i;
04269
04270         /* shortcuts */
04271         int lenW = AO.SaveHeader.lenWORD;
04272         int lenL = AO.SaveHeader.lenLONG;
04273         int lenP = AO.SaveHeader.lenPOS;
04274
04275         /* if we have a buffered FILEINDEX then just return it */
04276         if ( AO.bufferedInd ) {
04277             *fileind = sbuffer;
04278             AO.bufferedInd = 0;
04279             return ( 0 );
04280         }
04281
04282         if ( ReadFile(AO.SaveData.Handle, (UBYTE *)fileind, sizeof(FILEINDEX))
04283                 != sizeof(FILEINDEX) ) {
04284 /* INTERNAL_ERROR_EXCL_START */
04285             return ( MesPrint("!>Error(1) reading stored expression.") );
04286 /* INTERNAL_ERROR_EXCL_STOP */
04287         }
04288
04289         /* do we need to flip the endianness? */
04290         if ( AO.transFlag & 1 ) {
04291             LONG number;
04292             /* padding bytes */
04293             int padp = lenL - ((lenW*5+(MAXENAME + 1)) & (lenL-1));
04294             p = (UBYTE *)fileind;
04295             AO.FlipPOS(p); p += lenP;           /* next */
04296             AO.FlipPOS(p);                      /* number */
04297             AO.ResizePOS(p, (UBYTE *)&number);
04298             p += lenP;
04299             for ( i = 0; i < number; ++i ) {
04300                 AO.FlipPOS(p); p += lenP;       /* position */
04301                 AO.FlipPOS(p); p += lenP;       /* length */
04302                 AO.FlipPOS(p); p += lenP;       /* variables */
04303                 AO.FlipLONG(p); p += lenL;      /* CompressSize */
04304                 AO.FlipWORD(p); p += lenW;      /* nsymbols */
04305                 AO.FlipWORD(p); p += lenW;      /* nindices */
04306                 AO.FlipWORD(p); p += lenW;      /* nvectors */
04307                 AO.FlipWORD(p); p += lenW;      /* nfunctions */
04308                 AO.FlipWORD(p); p += lenW;      /* size */
04309                 p += padp;
04310             }
04311         }
04312
04313         /* do we need to resize data? */
04314         if ( AO.transFlag > 1 ) {
04315             LONG number, maxnumber;
04316             int n;
04317             /* padding bytes */
04318             int padp = lenL - ((lenW*5+(MAXENAME + 1)) & (lenL-1));
04319             int padq = sizeof(LONG) - ((sizeof(WORD)*5+(MAXENAME + 1)) & (sizeof(LONG)-1));
```

```
04320
04321             p = (UBYTE *)fileind; q = (UBYTE *)&buffer;
04322             AO.ResizePOS(p, q);                       /* next */
04323             p += lenP; q += sizeof(POSITION);
04324             AO.ResizePOS(p, q);                  /* number */
04325             p += lenP;
04326             number = BASEPOSITION(*((POSITION *)q));
04327             /* if FILEINDEX in file contains more entries than the FILEINDEX in
04328                memory can contain, then adjust the numbers and prepare for
04329                buffering */
04330             if ( number > (LONG)INFILEINDEX ) {
04331                 AO.bufferedInd = number-INFILEINDEX;
04332                 if ( AO.bufferedInd > (WORD)INFILEINDEX ) {
04333                     /* can happen when reading 32bit and writing >=128bit.
04334                        Fix: more than one static buffer for FILEINDEX */
04335                     return ( MesPrint("Too many index entries.") );
04336                 }
04337                 maxnumber = INFILEINDEX;
04338                 SETBASEPOSITION(*((POSITION *)q),INFILEINDEX);
04339             }
04340             else {
04341                 maxnumber = number;
04342             }
04343             q += sizeof(POSITION);
04344             /* read all INDEXENTRY that fit into the output buffer */
04345             for ( i = 0; i < maxnumber; ++i ) {
04346                 AO.ResizePOS(p, q);                   /* position */
04347                 p += lenP; q += sizeof(POSITION);
04348                 AO.ResizePOS(p, q);                   /* length */
04349                 p += lenP; q += sizeof(POSITION);
04350                 AO.ResizePOS(p, q);                   /* variables */
04351                 p += lenP; q += sizeof(POSITION);
04352                 AO.ResizeLONG(p, q);                  /* CompressSize */
04353                 p += lenL; q += sizeof(LONG);
04354                 AO.ResizeWORD(p, q);                  /* nsymbols */
04355                 p += lenW; q += sizeof(WORD);
04356                 AO.ResizeWORD(p, q);                  /* nindices */
04357                 p += lenW; q += sizeof(WORD);
04358                 AO.ResizeWORD(p, q);                  /* nvectors */
04359                 p += lenW; q += sizeof(WORD);
04360                 AO.ResizeWORD(p, q);                  /* nfunctions */
04361                 p += lenW; q += sizeof(WORD);
04362                 AO.ResizeWORD(p, q);                  /* size (unchanged!) */
04363                 p += lenW; q += sizeof(WORD);
04364                 n = MAXENAME + 1;
04365                 NCOPYB(q, p, n)
04366                 p += padp;
04367                 q += padq;
04368             }
04369             /* read all the remaining INDEXENTRY and put them into the static buffer */
04370             if ( AO.bufferedInd ) {
04371                 sbuffer.next = buffer.next;
04372                 SETBASEPOSITION(sbuffer.number,AO.bufferedInd);
04373                 q = (UBYTE *)&sbuffer + sizeof(POSITION) + sizeof(LONG);
04374                 for ( i = maxnumber; i < number; ++i ) {
04375                     AO.ResizePOS(p, q);               /* position */
04376                     p += lenP; q += sizeof(POSITION);
04377                     AO.ResizePOS(p, q);               /* length */
04378                     p += lenP; q += sizeof(POSITION);
04379                     AO.ResizePOS(p, q);               /* variables */
04380                     p += lenP; q += sizeof(POSITION);
04381                     AO.ResizeLONG(p, q);              /* CompressSize */
04382                     p += lenL; q += sizeof(LONG);
04383                     AO.ResizeWORD(p, q);              /* nsymbols */
04384                     p += lenW; q += sizeof(WORD);
04385                     AO.ResizeWORD(p, q);              /* nindices */
04386                     p += lenW; q += sizeof(WORD);
04387                     AO.ResizeWORD(p, q);              /* nvectors */
04388                     p += lenW; q += sizeof(WORD);
04389                     AO.ResizeWORD(p, q);              /* nfunctions */
04390                     p += lenW; q += sizeof(WORD);
04391                     AO.ResizeWORD(p, q);              /* size (unchanged!) */
04392                     p += lenW; q += sizeof(WORD);
04393                     n = MAXENAME + 1;
04394                     NCOPYB(q, p, n)
04395                     p += padp;
04396                     q += padq;
04397                 }
04398             }
04399             /* copy to output */
04400             p = (UBYTE *)fileind; q = (UBYTE *)&buffer; n = sizeof(FILEINDEX);
04401             NCOPYB(p, q, n)
04402         }
04403         return ( 0 );
04404     } else {
04405         return ( ReadFile(AO.SaveData.Handle, (UBYTE *)fileind, sizeof(FILEINDEX))
04406                  != sizeof(FILEINDEX) );
```

```
04407     }
04408 }
04409
04410 /*
04411         #] ReadSaveIndex :
04412         #[ ReadSaveVariables :
04413 */
04414
04445 LONG ReadSaveVariables(UBYTE *buffer, UBYTE *top, LONG *size, LONG *outsize,\
04446                        INDEXENTRY *ind, LONG *stage)
04447 {
04448     /* do we need some translation for the variables? */
04449     if ( AO.transFlag ) {
04450         /* counters for the number of already read symbols, indices, ... that
04451            need to remain valid between different calls to ReadSaveVariables().
04452            are initialized if stage == -1 */
04453         static WORD numReadSym;
04454         static WORD numReadInd;
04455         static WORD numReadVec;
04456         static WORD numReadFun;
04457
04458         POSITION pos;
04459         UBYTE *in, *out, *pp = 0, *end, *outbuf;
04460         LONG numread;
04461         WORD namelen, realnamelen;
04462         /* shortcuts */
04463         WORD lenW = AO.SaveHeader.lenWORD;
04464         WORD lenL = AO.SaveHeader.lenLONG;
04465         WORD lenP = AO.SaveHeader.lenPOINTER;
04466         WORD flip = AO.transFlag & 1;
04467
04468         /* remember file position in case we have to rewind */
04469         TELLFILE(AO.SaveData.Handle,&pos);
04470
04471         /* decide on the position of the in and out buffers.
04472            if the input is "bigger" than the output, we resize in-place, i.e.
04473            we immediately overwrite the source data by the translated data. in
04474            and out buffers start at the same place.
04475            if not, we read from the end of the given buffer and write at the
04476            beginning. */
04477         if ( (lenW > (WORD)sizeof(WORD))
04478         || ( (lenW == (WORD)sizeof(WORD))
04479               && ( (lenL > (WORD)sizeof(LONG))
04480                    || ( (lenL == (WORD)sizeof(LONG)) && lenP > (WORD)sizeof(void *))
04481                  )
04482            ) ) {
04483             in = out = buffer;
04484             end = buffer + *size;
04485         }
04486         else {
04487             /* data will grow roughly by sizeof(WORD)/lenW. the exact value is
04488                not important. if reading and writing areas start to overlap, the
04489                reading will already be near the end of the data and overwriting
04490                doesn't matter. */
04491             LONG newsize = (top - buffer) / (1 + sizeof(WORD)/lenW);
04492             end = top;
04493             out = buffer;
04494             in = end - newsize;
04495             if ( *size > newsize ) *size = newsize;
04496         }
04497
04498         if ( ( numread = ReadFile(AO.SaveData.Handle, in, *size) ) != *size ) {
04499 /* INTERNAL_ERROR_EXCL_START */
04500             return ( MesPrint("!>Error(2) reading stored expression.") );
04501 /* INTERNAL_ERROR_EXCL_STOP */
04502         }
04503
04504         *size = 0;
04505         *outsize = 0;
04506
04507         /* first time in ReadSaveVariables(). initialize counters. */
04508         if ( *stage == -1 ) {
04509             numReadSym = 0;
04510             numReadInd = 0;
04511             numReadVec = 0;
04512             numReadFun = 0;
04513             ++*stage;
04514         }
04515
04516         while ( in < end ) {
04517             /* Symbols */
04518             if ( *stage == 0 ) {
04519                 if ( ind->nsymbols <= numReadSym ) {
04520                     ++*stage;
04521                     continue;
04522                 }
04523                 if ( end - in < AO.SaveHeader.sSym ) {
```

```
04524                   goto RSVEnd;
04525               }
04526               if ( flip ) {
04527                   pp = in;
04528                   AO.FlipLONG(pp); pp += lenL;
04529                   while ( pp < in + AO.SaveHeader.sSym ) {
04530                       AO.FlipWORD(pp); pp += lenW;
04531                   }
04532               }
04533               pp = in + AO.SaveHeader.sSym;
04534               AO.ResizeLONG(in, out); in += lenL; out += sizeof(LONG); /* name     */
04535               AO.CheckPower(in);
04536               AO.ResizeWORD(in, out); in += lenW;
04537               if ( *((WORD *)out) == -AO.mpower ) *((WORD *)out) = -MAXPOWER;
04538               out += sizeof(WORD);                                     /* minpower */
04539               AO.ResizeWORD(in, out); in += lenW;
04540               if ( *((WORD *)out) == AO.mpower ) *((WORD *)out) = MAXPOWER;
04541               out += sizeof(WORD);                                     /* maxpower */
04542               AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* complex  */
04543               AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* number   */
04544               AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* flags    */
04545               AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* node     */
04546               AO.ResizeWORD(in, out); in += lenW;                      /* namesize */
04547               realnamelen = *((WORD *)out);
04548               realnamelen += sizeof(void *)-1; realnamelen &= -(sizeof(void *));
04549               out += sizeof(WORD);
04550               AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* dimension */
04551               while ( in < pp ) {
04552                   AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD);
04553               }
04554               namelen = *((WORD *)out-1); /* cares for padding "bug" */
04555               if ( end - in < namelen ) {
04556                   goto RSVEnd;
04557               }
04558               *((WORD *)out-1) = realnamelen;
04559               *size += AO.SaveHeader.sSym + namelen;
04560               *outsize += sizeof(struct SyMbOl) + realnamelen;
04561               if ( realnamelen > namelen ) {
04562                   int j = namelen;
04563                   NCOPYB(out, in, j);
04564                   out += realnamelen - namelen;
04565               }
04566               else {
04567                   int j = realnamelen;
04568                   NCOPYB(out, in, j);
04569                   in += namelen - realnamelen;
04570               }
04571               ++numReadSym;
04572               continue;
04573           }
04574           /* Indices */
04575           if ( *stage == 1 ) {
04576               if ( ind->nindices <= numReadInd ) {
04577                   ++*stage;
04578                   continue;
04579               }
04580               if ( end - in < AO.SaveHeader.sInd ) {
04581                   goto RSVEnd;
04582               }
04583               if ( flip ) {
04584                   pp = in;
04585                   AO.FlipLONG(pp); pp += lenL;
04586                   while ( pp < in + AO.SaveHeader.sInd ) {
04587                       AO.FlipWORD(pp); pp += lenW;
04588                   }
04589               }
04590               pp = in + AO.SaveHeader.sInd;
04591               AO.ResizeLONG(in, out); in += lenL; out += sizeof(LONG); /* name      */
04592               AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* type      */
04593               AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* dimension */
04594               AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* number    */
04595               AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* flags     */
04596               AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* nmin4     */
04597               AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* node      */
04598               AO.ResizeWORD(in, out); in += lenW;                      /* namesize  */
04599               realnamelen = *((WORD *)out);
04600               realnamelen += sizeof(void *)-1; realnamelen &= -(sizeof(void *));
04601               out += sizeof(WORD);
04602               while ( in < pp ) {
04603                   AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD);
04604               }
04605               namelen = *((WORD *)out-1); /* cares for padding "bug" */
04606               if ( end - in < namelen ) {
04607                   goto RSVEnd;
04608               }
04609               *((WORD *)out-1) = realnamelen;
04610               *size += AO.SaveHeader.sInd + namelen;
```

```
04611                 *outsize += sizeof(struct InDeX) + realnamelen;
04612                 if ( realnamelen > namelen ) {
04613                     int j = namelen;
04614                     NCOPYB(out, in, j);
04615                     out += realnamelen - namelen;
04616                 }
04617                 else {
04618                     int j = realnamelen;
04619                     NCOPYB(out, in, j);
04620                     in += namelen - realnamelen;
04621                 }
04622                 ++numReadInd;
04623                 continue;
04624             }
04625             /* Vectors */
04626             if ( *stage == 2 ) {
04627                 if ( ind->nvectors <= numReadVec ) {
04628                     ++*stage;
04629                     continue;
04630                 }
04631                 if ( end - in < AO.SaveHeader.sVec ) {
04632                     goto RSVEnd;
04633                 }
04634                 if ( flip ) {
04635                     pp = in;
04636                     AO.FlipLONG(pp); pp += lenL;
04637                     while ( pp < in + AO.SaveHeader.sVec ) {
04638                         AO.FlipWORD(pp); pp += lenW;
04639                     }
04640                 }
04641                 pp = in + AO.SaveHeader.sVec;
04642                 AO.ResizeLONG(in, out); in += lenL; out += sizeof(LONG); /* name     */
04643                 AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* complex  */
04644                 AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* number   */
04645                 AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* flags    */
04646                 AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* node     */
04647                 AO.ResizeWORD(in, out); in += lenW;                      /* namesize */
04648                 realnamelen = *((WORD *)out);
04649                 realnamelen += sizeof(void *)-1; realnamelen &= -(sizeof(void *));
04650                 out += sizeof(WORD);
04651                 AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* dimension */
04652                 while ( in < pp ) {
04653                     AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD);
04654                 }
04655                 namelen = *((WORD *)out-1); /* cares for padding "bug" */
04656                 if ( end - in < namelen ) {
04657                     goto RSVEnd;
04658                 }
04659                 *((WORD *)out-1) = realnamelen;
04660                 *size += AO.SaveHeader.sVec + namelen;
04661                 *outsize += sizeof(struct VeCtOr) + realnamelen;
04662                 if ( realnamelen > namelen ) {
04663                     int j = namelen;
04664                     NCOPYB(out, in, j)
04665                     out += realnamelen - namelen;
04666                 }
04667                 else {
04668                     int j = realnamelen;
04669                     NCOPYB(out, in, j)
04670                     in += namelen - realnamelen;
04671                 }
04672                 ++numReadVec;
04673                 continue;
04674             }
04675             /* Functions */
04676             if ( *stage == 3 ) {
04677                 if ( ind->nfunctions <= numReadFun ) {
04678                     ++*stage;
04679                     continue;
04680                 }
04681                 if ( end - in < AO.SaveHeader.sFun ) {
04682                     goto RSVEnd;
04683                 }
04684                 if ( flip ) {
04685                     pp = in;
04686                     AO.FlipPOINTER(pp); pp += lenP;
04687                     AO.FlipLONG(pp); pp += lenL;
04688                     AO.FlipLONG(pp); pp += lenL;
04689                     while ( pp < in + AO.SaveHeader.sFun ) {
04690                         AO.FlipWORD(pp); pp += lenW;
04691                     }
04692                 }
04693                 pp = in + AO.SaveHeader.sFun;
04694                 outbuf = out;
04695                 AO.ResizePOINTER(in, out); in += lenP; out += sizeof(void *); /* tabl */
04696                 AO.ResizeLONG(in, out); in += lenL; out += sizeof(LONG); /* symminfo  */
04697                 AO.ResizeLONG(in, out); in += lenL; out += sizeof(LONG); /* name      */
```

```
04698                 AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* commute   */
04699                 AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* complex   */
04700                 AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* number    */
04701                 AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* flags     */
04702                 AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* spec      */
04703                 AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* symmetric */
04704                 AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* numargs */
04705                 AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* node      */
04706                 AO.ResizeWORD(in, out); in += lenW;                      /* namesize  */
04707                 realnamelen = *((WORD *)out);
04708                 realnamelen += sizeof(void *)-1; realnamelen &= -(sizeof(void *));
04709                 out += sizeof(WORD);
04710                 AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* dimension */
04711                 while ( in < pp ) {
04712                     AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD);
04713                 }
04714                 namelen = *((WORD *)out-1); /* cares for padding "bug" */
04715                 if ( end - in < namelen ) {
04716                     goto RSVEnd;
04717                 }
04718                 *((WORD *)out-1) = realnamelen;
04719                 *size += AO.SaveHeader.sFun + namelen;
04720                 *outsize += sizeof(struct FuNcTiOn) + realnamelen;
04721                 if ( realnamelen > namelen ) {
04722                     int j = namelen;
04723                     NCOPYB(out, in, j);
04724                     out += realnamelen - namelen;
04725                 }
04726                 else {
04727                     int j = realnamelen;
04728                     NCOPYB(out, in, j);
04729                     in += namelen - realnamelen;
04730                 }
04731                 ++numReadFun;
04732                 /* we use the information whether a function is tensorial later in ReadSaveTerm */
04733                 AO.tensorList[((FUNCTIONS)outbuf)->number+FUNCTION] =
04734                     (UBYTE)(((FUNCTIONS)outbuf)->spec == TENSORFUNCTION);
04735                 continue;
04736             }
04737             /* handle numdummies */
04738             if ( end - in >= lenW ) {
04739                 if ( flip ) AO.FlipWORD(in);
04740                 AO.ResizeWORD(in, out);
04741                 *size += lenW;
04742                 *outsize += sizeof(WORD);
04743             }
04744             /* handle numfactors */
04745             if ( end - in >= lenW ) {
04746                 if ( flip ) AO.FlipWORD(in);
04747                 AO.ResizeWORD(in, out);
04748                 *size += lenW;
04749                 *outsize += sizeof(WORD);
04750             }
04751             /* handle vflags */
04752             if ( end - in >= lenW ) {
04753                 if ( flip ) AO.FlipWORD(in);
04754                 AO.ResizeWORD(in, out);
04755                 *size += lenW;
04756                 *outsize += sizeof(WORD);
04757             }
04758             /* handle uflags */
04759             if ( end - in >= lenW ) {
04760                 if ( flip ) AO.FlipWORD(in);
04761                 AO.ResizeWORD(in, out);
04762                 *size += lenW;
04763                 *outsize += sizeof(WORD);
04764             }
04765             return ( 0 );
04766         }
04767
04768 RSVEnd:
04769         /* we are here because the remaining buffer cannot hold the next
04770            struct. we position the file behind the last successfully translated
04771            struct and return. */
04772         ADDPOS(pos, *size);
04773         SeekFile(AO.SaveData.Handle, &pos, SEEK_SET);
04774         return ( 0 );
04775     } else {
04776         return ( ReadFile(AO.SaveData.Handle, buffer, *size) != *size );
04777     }
04778 }
04779
04780 /*
04781         #] ReadSaveVariables :
04782         #[ ReadSaveTerm :
04783 */
04784
```

```
04813 UBYTE *
04814 ReadSaveTerm32(UBYTE *bin, UBYTE *binend, UBYTE **bout, UBYTE *boutend, UBYTE *top, int terminbuf)
04815 {
04816     GETIDENTITY
04817
04818     UBYTE *boutbuf;
04819     int32_t len, j, id;
04820     int32_t *r, *t, *coeff, *end, *newtermsize, *rend;
04821     int32_t *newsubtermp;
04822     int32_t *in = (int32_t *)bin;
04823     int32_t *out = (int32_t *)*bout;
04824
04825     /* if called recursively the term is already decompressed in buffer.
04826        is this the case? */
04827     if ( terminbuf ) {
04828         /* don't do any decompression, just adjust the pointers */
04829         len = *out;
04830         end = out + len;
04831         r = in + 1;
04832         rend = (int32_t *)boutend;
04833         coeff = end - ABS(*(end-1));
04834         newtermsize = (int32_t *)*bout;
04835         out = newtermsize + 1;
04836     }
04837     else {
04838         /* do decompression of necessary. always return if the space in the
04839            buffer is not sufficient */
04840         int32_t rbuf;
04841         r = (int32_t *)AR.CompressBuffer;
04842         rbuf = *r;
04843         len = j = *in;
04844         /* first copy from AR.CompressBuffer if necessary */
04845         if ( j < 0 ) {
04846             ++in;
04847             if ( (UBYTE *)in >= binend ) {
04848                 return ( bin );
04849             }
04850             *out = len = -j + 1 + *in;
04851             end = out + *out;
04852             if ( (UBYTE *)end >= top ) {
04853                 return ( bin );
04854             }
04855             ++out;
04856             *r++ = len;
04857             while ( ++j <= 0 ) {
04858                 int32_t bb = *r++;
04859                 *out++ = bb;
04860             }
04861             j = *in++;
04862         }
04863         else if ( j == 0 ) {
04864             /* care for padding words */
04865             while ( (UBYTE *)in < binend ) {
04866                 *out++ = 0;
04867                 if ( (UBYTE *)out > top ) {
04868                     return ( (UBYTE *)bin );
04869                 }
04870
04871                 *r++ = 0;
04872                 ++in;
04873             }
04874             *bout = (UBYTE *)out;
04875             return ( (UBYTE *)in );
04876         }
04877         else {
04878             end = out + len;
04879             if ( (UBYTE *)end >= top ) {
04880                 return ( bin );
04881             }
04882         }
04883         if ( (UBYTE *)(in + j) >= binend ) {
04884             *(AR.CompressBuffer) = rbuf;
04885             return ( bin );
04886         }
04887         if ( (UBYTE *)out + j >= top ) {
04888             return ( bin );
04889         }
04890         /* second copy from input buffer */
04891         while ( --j >= 0 ) {
04892             int32_t bb = *in++;
04893             *r++ = *out++ = bb;
04894         }
04895
04896         rend = r;
04897         r = (int32_t *)AR.CompressBuffer + 1;
04898         coeff = end - ABS(*(end-1));
04899         newtermsize = (int32_t *)*bout;
```

```
04900         out = newtermsize + 1;
04901     }
04902
04903     /* iterate over subterms */
04904     while ( out < coeff ) {
04905
04906         id = *out++;
04907         ++r;
04908         t = out + *out - 1;
04909         newsubtermp = out;
04910         ++out; ++r;
04911
04912         if ( id == SYMBOL ) {
04913             while ( out < t ) {
04914                 ++out; ++r; /* symbol number */
04915                 /* if exponent is too big, rewrite as exponent function */
04916                 if ( ABS(*out) >= MAXPOWER ) {
04917                     int32_t *a, *b;
04918                     int32_t n;
04919                     int32_t num = *(out-1);
04920                     int32_t exp = *out;
04921                     coeff += 9;
04922                     end += 9;
04923                     t += 9;
04924                     if ( (UBYTE *)end > top ) return ( bin );
04925                     out -= 3;
04926                     *out++ = EXPONENT;      /* id */
04927                     *out++ = 13;            /* size */
04928                     *out++ = 1;             /* dirtyflag */
04929                     *out++ = -SYMBOL;       /* first short arg */
04930                     *out++ = num;
04931                     *out++ = 8;             /* second arg, size */
04932                     *out++ = 0;             /* dirtyflag */
04933                     *out++ = 6;             /* term size */
04934                     *out++ = ABS(exp) & 0x0000FFFF;
04935                     *out++ = ABS(exp) >> 16;
04936                     *out++ = 1;
04937                     *out++ = 0;
04938                     *out++ = ( exp < 0 ) ? -5 : 5;
04939                     a = ++r;
04940                     b = out;
04941                     n = rend - r;
04942                     NCOPYI32(b, a, n)
04943                 }
04944                 else {
04945                     ++out; ++r;
04946                 }
04947             }
04948         }
04949         else if ( id == DOTPRODUCT ) {
04950             while ( out < t ) {
04951                 AO.RenumberVec((UBYTE *)out); /* vector 1 */
04952                 ++out; ++r;
04953                 AO.RenumberVec((UBYTE *)out); /* vector 2 */
04954                 ++out; ++r;
04955                 /* if exponent is too big, rewrite as exponent function */
04956                 if ( ABS(*out) >= MAXPOWER ) {
04957                     int32_t *a, *b;
04958                     int32_t n;
04959                     int32_t num1 = *(out-2);
04960                     int32_t num2 = *(out-1);
04961                     int32_t exp = *out;
04962                     coeff += 17;
04963                     end += 17;
04964                     t += 17;
04965                     if ( (UBYTE *)end > top ) return ( bin );
04966                     out -= 4;
04967                     *out++ = EXPONENT;      /* id */
04968                     *out++ = 22;            /* size */
04969                     *out++ = 1;             /* dirtyflag */
04970                     *out++ = 11;            /* first arg, size */
04971                     *out++ = 0;             /* dirtyflag */
04972                     *out++ = 9;             /* term size */
04973                     *out++ = DOTPRODUCT;    /* p1.p2 */
04974                     *out++ = 5;             /* subterm size */
04975                     *out++ = num1;          /* p1 */
04976                     *out++ = num2;          /* p2 */
04977                     *out++ = 1;             /* exponent */
04978                     *out++ = 1;             /* coeff */
04979                     *out++ = 1;
04980                     *out++ = 3;
04981                     *out++ = 8;             /* second arg, size */
04982                     *out++ = 0;             /* dirtyflag */
04983                     *out++ = 6;             /* term size */
04984                     *out++ = ABS(exp) & 0x0000FFFF;
04985                     *out++ = ABS(exp) >> 16;
04986                     *out++ = 1;
```

```
04987                       *out++ = 0;
04988                       *out++ = ( exp < 0 ) ? -5 : 5;
04989                       a = ++r;
04990                       b = out;
04991                       n = rend - r;
04992                       NCOPYI32(b, a, n)
04993                   }
04994                   else {
04995                       ++out; ++r;
04996                   }
04997               }
04998           }
04999           else if ( id == VECTOR ) {
05000               while ( out < t ) {
05001                   AO.RenumberVec((UBYTE *)out); /* vector number */
05002                   ++out; ++r;
05003                   ++out; ++r; /* index, do nothing */
05004               }
05005           }
05006           else if ( id == INDEX ) {
05007 /*            int32_t vectoroffset = -2 * *((int32_t *)AO.SaveHeader.wildoffset); */
05008               void *dummy = (void *)AO.SaveHeader.wildoffset;  /* to remove a warning about
      strict-aliasing rules in gcc */
05009               int32_t vectoroffset = -2 * *((int32_t *)dummy);
05010               while ( out < t ) {
05011                   /* if there is a vector, renumber it */
05012                   if ( *out < vectoroffset ) {
05013                       AO.RenumberVec((UBYTE *)out);
05014                   }
05015                   ++out; ++r;
05016               }
05017           }
05018           else if ( id == SUBEXPRESSION ) {
05019               /* nothing to translate */
05020               while ( out < t ) {
05021                   ++out; ++r;
05022               }
05023           }
05024           else if ( id == DELTA ) {
05025               /* nothing to translate */
05026               r += t - out;
05027               out = t;
05028           }
05029           else if ( id == HAAKJE ) {
05030               /* nothing to translate */
05031               r += t - out;
05032               out = t;
05033           }
05034           else if ( id == GAMMA || id == LEVICIVITA || (id >= FUNCTION && AO.tensorList[id]) ) {
05035 /*            int32_t vectoroffset = -2 * *((int32_t *)AO.SaveHeader.wildoffset); */
05036               void *dummy = (void *)AO.SaveHeader.wildoffset;  /* to remove a warning about
      strict-aliasing rules in gcc */
05037               int32_t vectoroffset = -2 * *((int32_t *)dummy);
05038               while ( out < t ) {
05039                   /* if there is a vector as an argument, renumber it */
05040                   if ( *out < vectoroffset ) {
05041                       AO.RenumberVec((UBYTE *)out);
05042                   }
05043                   ++out; ++r;
05044               }
05045           }
05046           else if ( id >= FUNCTION ) {
05047               int32_t *argEnd;
05048               UBYTE *newbin;
05049
05050               ++out; ++r; /* dirty flags */
05051
05052               /* loop over arguments */
05053               while ( out < t ) {
05054                   if ( *out < 0 ) {
05055                       /* short notation arguments */
05056                       switch ( -*out ) {
05057                           case SYMBOL:
05058                               ++out; ++r;
05059                               ++out; ++r;
05060                               break;
05061                           case SNUMBER:
05062                               argEnd = out+2;
05063                               ++out; ++r;
05064                               if ( sizeof(WORD) == 2 ) {
05065                                   /* resize if needed */
05066                                   if ( *out > (1«15)-1 || *out < -(1«15)+1 ) {
05067                                       int32_t *a, *b;
05068                                       int32_t n;
05069                                       int32_t num = *out;
05070                                       coeff += 6;
05071                                       end += 6;
```

```
05072                                   argEnd += 6;
05073                                   t += 6;
05074                                   if ( (UBYTE *)end > top ) return ( bin );
05075                                   --out;
05076                                   *out++ = 8;         /* argument size */
05077                                   *out++ = 0;         /* dirtyflag */
05078                                   *out++ = 6;         /* term size */
05079                                   *out++ = ABS(num) & 0x0000FFFF;
05080                                   *out++ = ABS(num) » 16;
05081                                   *out++ = 1;
05082                                   *out++ = 0;
05083                                   *out++ = ( num < 0 ) ? -5 : 5;
05084                                   a = ++r;
05085                                   b = out;
05086                                   n = rend - r;
05087                                   NCOPYI32(b, a, n)
05088                               }
05089                               else {
05090                                   ++out; ++r;
05091                               }
05092                           }
05093                           else {
05094                               ++out; ++r;
05095                           }
05096                           break;
05097                       case VECTOR:
05098                           ++out; ++r;
05099                           AO.RenumberVec((UBYTE *)out);
05100                           ++out; ++r;
05101                           break;
05102                       case INDEX:
05103                           ++out; ++r;
05104                           ++out; ++r;
05105                           break;
05106                       case MINVECTOR:
05107                           ++out; ++r;
05108                           AO.RenumberVec((UBYTE *)out);
05109                           ++out; ++r;
05110                           break;
05111                       default:
05112                           if ( -*out >= FUNCTION ) {
05113                               ++out; ++r;
05114                               break;
05115                           } else {
05116 /* INTERNAL_ERROR_EXCL_START */
05117                               MesPrint("!>short function code %d not implemented.", *out);
05118                               return ( (UBYTE *)in );
05119 /* INTERNAL_ERROR_EXCL_STOP */
05120                           }
05121                   }
05122               }
05123               else {
05124                   /* long arguments */
05125                   int32_t *newargsize = out;
05126                   argEnd = out + *out;
05127                   ++out; ++r;
05128                   ++out; ++r; /* dirty flags */
05129                   while ( out < argEnd ) {
05130                       int32_t *keepsizep = out + *out;
05131                       int32_t lenbuf = *out;
05132                       int32_t **ppp = &out; /* to avoid a compiler warning */
05133                       /* recursion */
05134                       newbin = ReadSaveTerm32((UBYTE *)r, binend, (UBYTE **)ppp, (UBYTE *)rend, top,
      1);
05135                       r += lenbuf;
05136                       if ( newbin == (UBYTE *)r ) {
05137                           return ( (UBYTE *)in );
05138                       }
05139                       /* if the term done by recursion has changed in size,
05140                          we need to move the rest of the data accordingly */
05141                       if ( out > keepsizep ) {
05142                           int32_t *a, *b;
05143                           int32_t n;
05144                           int32_t extention = out - keepsizep;
05145                           a = r;
05146                           b = out;
05147                           n = rend - r;
05148                           NCOPYI32(b, a, n)
05149                           coeff += extention;
05150                           end += extention;
05151                           argEnd += extention;
05152                           t += extention;
05153                       }
05154                       else if ( out < keepsizep ) {
05155                           int32_t *a, *b;
05156                           int32_t n;
05157                           int32_t extention = keepsizep - out;
```

```
05158                               a = keepsizep;
05159                               b = out;
05160                               n = rend - r;
05161                               NCOPYI32(b, a, n)
05162                               coeff -= extention;
05163                               end -= extention;
05164                               argEnd -= extention;
05165                               t -= extention;
05166                           }
05167                       }
05168                       *newargsize = out - newargsize;
05169                   }
05170               }
05171           }
05172           else {
05173 /* INTERNAL_ERROR_EXCL_START */
05174               MesPrint("!>ID %d not recognized.", id);
05175               return ( (UBYTE *)in );
05176 /* INTERNAL_ERROR_EXCL_STOP */
05177           }
05178
05179           *newsubtermp = out - newsubtermp + 1;
05180       }
05181
05182       if ( (UBYTE *)end >= top ) {
05183           return ( bin );
05184       }
05185
05186       /* do coefficient and adjust term size */
05187       boutbuf = *bout;
05188       *bout = (UBYTE *)out;
05189
05190       ResizeCoeff32(bout, (UBYTE *)end, top);
05191
05192       if ( *bout >= top ) {
05193           *bout = boutbuf;
05194           return ( bin );
05195       }
05196
05197       *newtermsize = (int32_t *)*bout - newtermsize;
05198
05199       return ( (UBYTE *)in );
05200 }
05201
05202 /*
05203           #] ReadSaveTerm :
05204           #[ ReadSaveExpression :
05205 */
05206
05228 LONG ReadSaveExpression(UBYTE *buffer, UBYTE *top, LONG *size, LONG *outsize)
05229 {
05230       if ( AO.transFlag ) {
05231           UBYTE *in, *end, *out, *outend, *p;
05232           POSITION pos;
05233           LONG half;
05234           WORD lenW = AO.SaveHeader.lenWORD;
05235
05236           /* remember the last file position in case an expression cannot be
05237              fully processed */
05238           TELLFILE(AO.SaveData.Handle,&pos);
05239
05240           /* adjust 'size' depending on whether the translated data is bigger or
05241              smaller */
05242           half = (top-buffer)/2;
05243           if ( *size > half ) *size = half;
05244           if ( lenW < (WORD)sizeof(WORD) ) {
05245               if ( *size * (LONG)sizeof(WORD)/lenW > half ) *size = half*lenW/(LONG)sizeof(WORD);
05246           }
05247           else {
05248               if ( *size > half ) *size = half;
05249           }
05250
05251           /* depending on the necessary resizing we position the input pointer
05252              either at the start of the buffer or in the middle. if the data will
05253              roughly remain the same size, we need only one processing step, so
05254              we put the 'in' at the middle and 'out' and the beginning. in the
05255              other cases we need two processing steps, so first we put 'in' at
05256              the beginning and write at the middle. the second step can then read
05257              from the middle and put its results at the beginning. */
05258           in = out = buffer;
05259           if ( lenW == sizeof(WORD) ) in += half;
05260           else out += half;
05261           end = in + *size;
05262           outend = out + *size;
05263
05264           if ( ReadFile(AO.SaveData.Handle, in, *size) != *size ) {
05265 /* INTERNAL_ERROR_EXCL_START */
```

```
05266             return ( MesPrint("!>Error(3) reading stored expression.") );
05267 /* INTERNAL_ERROR_EXCL_STOP */
05268         }
05269
05270         if ( AO.transFlag & 1 ) {
05271             p = in;
05272             end -= lenW;
05273             while ( p <= end ) {
05274                 AO.FlipWORD(p); p += lenW;
05275             }
05276             end += lenW;
05277         }
05278
05279         if ( lenW > (WORD)sizeof(WORD) ) {
05280             /* renumber first */
05281             do {
05282                 outend = out+*size;
05283                 if ( outend > top ) outend = top;
05284                 p = ReadSaveTerm32(in, end, &out, outend, top, 0);
05285                 if ( p == in ) break;
05286                 in = p;
05287             } while ( in <= end - lenW );
05288             /* then resize */
05289             *size = in - buffer;
05290             in = buffer + half;
05291             end = out;
05292             out = buffer;
05293
05294             while ( in < end ) {
05295                 /* resize without checking */
05296                 AO.ResizeNCWORD(in, out);
05297                 in += lenW; out += sizeof(WORD);
05298             }
05299         }
05300         else {
05301             if ( lenW < (WORD)sizeof(WORD) ) {
05302                 /* resize first */
05303                 while ( in < end ) {
05304                     AO.ResizeWORD(in, out);
05305                     in += lenW; out += sizeof(WORD);
05306                 }
05307                 in = buffer + half;
05308                 end = out;
05309                 out = buffer;
05310             }
05311             /* then renumber */
05312             do {
05313                 p = ReadSaveTerm32(in, end, &out,  buffer+half, buffer+half, 0);
05314                 if ( p == in ) break;
05315                 in = p;
05316             } while ( in <= end - sizeof(WORD) );
05317             *size = (in - buffer - half) * lenW / (ULONG)sizeof(WORD);
05318         }
05319         *outsize = out - buffer;
05320         ADDPOS(pos, *size);
05321         SeekFile(AO.SaveData.Handle, &pos, SEEK_SET);
05322
05323         return ( 0 );
05324     }
05325     else {
05326         return ( ReadFile(AO.SaveData.Handle, buffer, *size) != *size );
05327     }
05328 }
05329
05330 /*
05331         #] ReadSaveExpression :
05332     #] System Independent Saved Expressions :
05333 */
```

4.139 structs.h File Reference

This graph shows which files directly or indirectly include this file:

Data Structures

- struct PoSiTiOn
- struct STOREHEADER
- struct InDeXeNtRy
- struct FiLeInDeX
- struct FiLeDaTa
- struct VaRrEnUm
- struct ReNuMbEr
- struct LIST
- struct KEYWORD
- struct KEYWORDV
- struct NaMeNode
- struct NaMeTree
- struct tree
- struct MiNmAx
- struct BrAcKeTiNdEx
- struct BrAcKeTiNfO
- struct TaBlEs
- struct ExPrEsSiOn
- struct SyMbOl
- struct InDeX
- struct VeCtOr
- struct FuNcTiOn
- struct SeTs
- struct DuBiOuS
- struct FaCdOlLaR
- struct DoLlArS
- struct MoDoPtDoLlArS
- struct fixedset
- struct TaBlEbAsEsUbInDeX
- struct TaBlEbAsE
- struct FUN_INFO
- struct PaRtIcLe
- struct VeRtEx
- struct MoDeL
- struct NaMeSpAcE
- struct FiLe
- struct StreaM
- struct SpecTatoR
- struct TrAcEs
- struct TrAcEn
- struct pReVaR
- struct INSIDEINFO
- struct PRELOAD
- struct PROCEDURE
- struct DoLoOp
- struct bit_field
- struct HANDLERS
- struct CbUf
- struct ChAnNeL
- struct SETUPPARAMETERS
- struct NeStInG
- struct StOrEcAcHe
- struct PeRmUtE

- struct PeRmUtEp
- struct DiStRiBuTe
- struct PaRtI
- struct sOrT
- struct POLYMOD
- struct SHvariables
- struct COST
- struct MODNUM
- struct OPTIMIZE
- struct OPTIMIZERESULT
- struct DICTIONARY_ELEMENT
- struct DICTIONARY
- struct SWITCHTABLE
- struct SWITCH
- struct TERMINFO
- struct NoRmDaTa
- struct M_const
- struct P_const
- struct C_const
- struct S_const
- struct R_const
- struct T_const
- struct N_const
- struct O_const
- struct X_const
- struct AllGlobals
- struct FixedGlobals

Macros

- #define INFILEINDEX ((512-2∗sizeof(POSITION))/sizeof(INDEXENTRY))
- #define EMPTYININDEX (512-2∗sizeof(POSITION)-INFILEINDEX∗sizeof(INDEXENTRY))
- #define PHEAD
- #define PHEAD0 void
- #define BHEAD
- #define BHEAD0

Typedefs

- typedef struct PoSiTiOn **POSITION**
- typedef struct InDeXeNtRy INDEXENTRY
- typedef struct FiLeInDeX FILEINDEX
- typedef struct FiLeDaTa **FILEDATA**
- typedef struct VaRrEnUm VARRENUM
- typedef struct ReNuMbEr ∗ RENUMBER
- typedef struct NaMeNode NAMENODE
- typedef struct NaMeTree NAMETREE
- typedef struct tree COMPTREE
- typedef struct MiNmAx **MINMAX**
- typedef struct BrAcKeTiNdEx **BRACKETINDEX**
- typedef struct BrAcKeTiNfO **BRACKETINFO**
- typedef struct TaBlEs ∗ TABLES
- typedef struct ExPrEsSiOn ∗ **EXPRESSIONS**

- typedef struct SyMbOl * **SYMBOLS**
- typedef struct InDeX * **INDICES**
- typedef struct VeCtOr * **VECTORS**
- typedef struct FuNcTiOn * FUNCTIONS
- typedef struct SeTs * **SETS**
- typedef struct DuBiOuS * **DUBIOUSV**
- typedef struct FaCdOlLaR **FACDOLLAR**
- typedef struct DoLlArS * **DOLLARS**
- typedef struct MoDoPtDoLlArS **MODOPTDOLLAR**
- typedef struct fixedset **FIXEDSET**
- typedef struct TaBlEbAsEsUbInDeX **TABLEBASESUBINDEX**
- typedef struct TaBlEbAsE **TABLEBASE**
- typedef struct PaRtIcLe **PARTICLE**
- typedef struct VeRtEx **VERTEX**
- typedef struct MoDeL **MODEL**
- typedef struct NaMeSpAcE **NAMESPACE**
- typedef struct FiLe FILEHANDLE
- typedef struct StreaM STREAM
- typedef struct SpecTatoR **SPECTATOR**
- typedef struct TrAcEs TRACES
- typedef struct TrAcEn * TRACEN
- typedef struct pReVaR PREVAR
- typedef struct DoLoOp DOLOOP
- typedef struct bit_field set_of_char[32]
- typedef struct bit_field * one_byte
- typedef int(* FINISHUFFLE) (WORD *)
- typedef int(* DO_UFFLE) (WORD *, WORD, WORD, WORD)
- typedef WORD(* COMPAREDUMMY) (WORD *, WORD *, WORD)
- typedef struct CbUf CBUF
- typedef struct ChAnNeL CHANNEL
- typedef struct NeStInG * NESTING
- typedef struct StOrEcAcHe * STORECACHE
- typedef struct PeRmUtE PERM
- typedef struct PeRmUtEp PERMP
- typedef struct DiStRiBuTe DISTRIBUTE
- typedef struct PaRtI PARTI
- typedef struct sOrT SORTING
- typedef struct NoRmDaTa **NORMDATA**
- typedef struct AllGlobals ALLGLOBALS
- typedef int(* WCN) (PHEAD WORD *, WORD *, WORD, WORD)
- typedef int(* WCN2) (PHEAD WORD *, WORD *)
- typedef WORD(* COMPARE) (PHEAD WORD *, WORD *, WORD)
- typedef struct FixedGlobals FIXEDGLOBALS

Functions

- **STATIC_ASSERT** (sizeof(STOREHEADER)==512)
- **STATIC_ASSERT** (sizeof(FILEINDEX)==512)

4.139.1 Detailed Description

Contains definitions for global structs.

!!!CAUTION!!! Changes in this file will most likely have consequences for the recovery mechanism (see checkpoint.c). You need to care for the code in checkpoint.c as well and modify the code there accordingly!

The marker [D] is used in comments in this file to mark pointers to which dynamically allocated memory is assigned by a call to malloc() during runtime (in contrast to pointers that point into already allocated memory). This information is especially helpful if one needs to know which pointers need to be freed (cf. checkpoint.c).

Definition in file structs.h.

4.139.2 Macro Definition Documentation

4.139.2.1 INFILEINDEX

```
#define INFILEINDEX ((512-2*sizeof(POSITION))/sizeof(INDEXENTRY))
```

Maximum number of entries (struct InDeXeNtRy) in a file index (struct FiLeInDeX). Number is calculated such that the size of a file index is no more than 512 bytes.

Definition at line 118 of file structs.h.

4.139.2.2 EMPTYININDEX

```
#define EMPTYININDEX (512-2*sizeof(POSITION)-INFILEINDEX*sizeof(INDEXENTRY))
```

Number of empty filling bytes for a file index (struct FiLeInDeX). It is calculated such that the size of a file index is always 512 bytes.

Definition at line 123 of file structs.h.

4.139.2.3 PHEAD

```
#define PHEAD
```

Definition at line 2529 of file structs.h.

4.139.2.4 PHEAD0

```
#define PHEAD0 void
```

Definition at line 2530 of file structs.h.

4.139.2.5 BHEAD

```
#define BHEAD
```

Definition at line 2531 of file structs.h.

4.139.2.6 BHEAD0

```
#define BHEAD0
```

Definition at line 2532 of file structs.h.

4.139.3 Typedef Documentation

4.139.3.1 INDEXENTRY

```
typedef struct InDeXeNtRy INDEXENTRY
```

Defines the structure of an entry in a file index (see struct FiLeInDeX).

It represents one expression in the file.

4.139.3.2 FILEINDEX

```
typedef struct FiLeInDeX FILEINDEX
```

Defines the structure of a file index in store-files and save-files.

It contains several entries (see struct InDeXeNtRy) up to a maximum of INFILEINDEX.

The variable number has been made of type POSITION to avoid padding problems with some types of computers/↩ OS and keep system independence of the .sav files.

This struct is always 512 bytes long.

4.139.3.3 VARRENUM

```
typedef struct VaRrEnUm VARRENUM
```

Contains the pointers to an array in which a binary search will be performed.

4.139.3.4 RENUMBER

```
typedef struct ReNuMbEr * RENUMBER
```

Only symb.lo gets dynamically allocated. All other pointers points into this memory.

4.139.3.5 NAMENODE

```
typedef struct NaMeNode NAMENODE
```

The names of variables are kept in an array. Elements of type NAMENODE define a tree (that is kept balanced) that make it easy and fast to look for variables. See also NAMETREE.

4.139.3.6 NAMETREE

```
typedef struct NaMeTree NAMETREE
```

A struct of type NAMETREE controls a complete (balanced) tree of names for the compiler. The compiler maintains several of such trees and the system has been set up in such a way that one could define more of them if we ever want to work with local name spaces.

4.139.3.7 COMPTREE

```
typedef struct tree COMPTREE
```

The subexpressions in the compiler are kept track of in a (balanced) tree to reduce the need for subexpressions and hence save much space in large rhs expressions (like when we have xxxxxxx occurrences of objects like f(x+1,x+1) in which each x+1 becomes a subexpression. The struct that controls this tree is COMPTREE.

4.139.3.8 TABLES

```
typedef struct TaBlEs * TABLES
```

buffers, mm, flags, and prototype are always dynamically allocated, tablepointers only if needed (=0 if unallocated), boomlijst and argtail only for sparse tables.

Allocation is done for both the normal and the stub instance (spare), except for prototype and argtail which share memory.

4.139.3.9 FUNCTIONS

```
typedef struct FuNcTiOn * FUNCTIONS
```

Contains all information about a function. Also used for tables. It is used in the LIST elements of AC.

4.139.3.10 FILEHANDLE

```
typedef struct FiLe FILEHANDLE
```

The type FILEHANDLE is the struct that controls all relevant information of a file, whether it is open or not. The file may even not yet exist. There is a system of caches (PObuffer) and as long as the information to be written still fits inside the cache the file may never be created. There are variables that can store information about different types of files, like scratch files or sort files. Depending on what is available in the system we may also have information about gzip compression (currently sort file only) or locks (TFORM).

4.139.3.11 STREAM

```
typedef struct StreaM STREAM
```

Input is read from 'streams' which are represented by objects of type STREAM. A stream can be a file, a do-loop, a procedure, the string value of a preprocessor variable ..... When a new stream is opened we have to keep information about where to fall back in the parent stream to allow this to happen even in the middle of reading names etc as would be the case with a`i'b

4.139.3.12 TRACES

```
typedef struct TrAcEs TRACES
```

The struct TRACES keeps track of the progress during the expansion of a 4-dimensional trace. Each time a term gets generated the expansion tree continues in the next statement. When it returns it has to know where to continue. The 4-dimensional traces are more complicated than the n-dimensional traces (see TRACEN) because of the extra tricks that can be used. They are responsible for the shorter final expressions.

4.139.3.13 TRACEN

```
typedef struct TrAcEn * TRACEN
```

The struct TRACEN keeps track of the progress during the expansion of a 4-dimensional trace. Each time a term gets generated the expansion tree continues in the next statement. When it returns it has to know where to continue.

4.139.3.14 PREVAR

```
typedef struct pReVaR PREVAR
```

An element of the type PREVAR is needed for each preprocessor variable.

4.139.3.15 DOLOOP

```
typedef struct DoLoOp DOLOOP
```

Each preprocessor do loop has a struct of type DOLOOP to keep track of all relevant parameters like where the beginning of the loop is, what the boundaries, increment and value of the loop parameter are, etc. Also we keep the whole loop inside a buffer of type PRELOAD

4.139.3.16 set_of_char

```
typedef struct bit_field set_of_char[32]
```

Used in set_in, set_set, set_del and set_sub.

Definition at line 930 of file structs.h.

4.139.3.17 one_byte

```
typedef struct bit_field* one_byte
```

Used in set_in, set_set, set_del and set_sub.

Definition at line 936 of file structs.h.

4.139.3.18 FINISHUFFLE

```
typedef int(* FINISHUFFLE) (WORD *)
```

Definition at line 956 of file structs.h.

4.139.3.19 DO_UFFLE

```
typedef int(* DO_UFFLE) (WORD *, WORD, WORD, WORD)
```

Definition at line 957 of file structs.h.

4.139.3.20 COMPAREDUMMY

```
typedef WORD(* COMPAREDUMMY) (WORD *, WORD *, WORD)
```

Definition at line 962 of file structs.h.

4.139.3.21 CBUF

```
typedef struct CbUf CBUF
```

The CBUF struct is used by the compiler. It is a compiler buffer of which since version 3.0 there can be many.

4.139.3.22 CHANNEL

```
typedef struct ChAnNeL CHANNEL
```

When we read input from text files we have to remember not only their handle but also their name. This is needed for error messages. Hence we call such a file a channel and reserve a struct of type CHANNEL to allow to lay this link.

4.139.3.23 NESTING

```
typedef struct NeStInG * NESTING
```

The NESTING struct is used when we enter the argument of functions and there is the possibility that we have to change something there. Because functions can be nested we have to keep track of all levels of functions in case we have to move the outer layers to make room for a larger function argument.

4.139.3.24 STORECACHE

```
typedef struct StOrEcAcHe * STORECACHE
```

The struct of type STORECACHE is used by a caching system for reading terms from stored expressions. Each thread should have its own system of caches.

4.139.3.25 PERM

```
typedef struct PeRmUtE PERM
```

The struct PERM is used to generate all permutations when the pattern matcher has to try to match (anti)symmetric functions.

4.139.3.26 PERMP

```
typedef struct PeRmUtEp PERMP
```

Like struct PERM but works with pointers.

4.139.3.27 DISTRIBUTE

```
typedef struct DiStRiBuTe DISTRIBUTE
```

The struct DISTRIBUTE is used to help the pattern matcher when matching antisymmetric tensors.

4.139.3.28 PARTI

```
typedef struct PaRtI PARTI
```

The struct PARTI is used to help determining whether a partition_ function can be replaced.

4.139.3.29 SORTING

```
typedef struct sOrT SORTING
```

The struct SORTING is used to control a sort operation. It includes a small and a large buffer and arrays for keeping track of various stages of the (merge) sorts. Each sort level has its own struct and different levels can have different sizes for its arrays. Also different threads have their own set of SORTING structs.

4.139.3.30 ALLGLOBALS

```
typedef struct AllGlobals ALLGLOBALS
```

Without pthreads (FORM) the ALLGLOBALS struct has all the global variables

4.139.3.31 WCN

```
typedef int(* WCN) (PHEAD WORD *, WORD *, WORD, WORD)
```

Definition at line 2535 of file structs.h.

4.139.3.32 WCN2

```
typedef int(* WCN2) (PHEAD WORD *, WORD *)
```

Definition at line 2536 of file structs.h.

4.139.3.33 COMPARE

```
typedef WORD(* COMPARE) (PHEAD WORD *, WORD *, WORD)
```

Definition at line 2538 of file structs.h.

4.139.3.34 FIXEDGLOBALS

```
typedef struct FixedGlobals FIXEDGLOBALS
```

The FIXEDGLOBALS struct is an anachronism. It started as the struct with global variables that needed initialization. It contains the elements Operation and OperaFind which define a very early way of automatically jumping to the proper operation. We find the results of it in parts of the file opera.c Later operations were treated differently in a more transparent way. We never changed the existing code. The most important part is currently the cTable which is used intensively in the compiler.

4.140 structs.h

Go to the documentation of this file.

```
00001
00017 /* #[ License : */
00018 /*
00019  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00020  *   When using this file you are requested to refer to the publication
00021  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00022  *   This is considered a matter of courtesy as the development was paid
00023  *   for by FOM the Dutch physics granting agency and we would like to
00024  *   be able to track its scientific use to convince FOM of its value
00025  *   for the community.
00026  *
00027  *   This file is part of FORM.
00028  *
00029  *   FORM is free software: you can redistribute it and/or modify it under the
00030  *   terms of the GNU General Public License as published by the Free Software
00031  *   Foundation, either version 3 of the License, or (at your option) any later
00032  *   version.
00033  *
00034  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00035  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00036  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00037  *   details.
00038  *
00039  *   You should have received a copy of the GNU General Public License along
00040  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00041  */
00042 /* #] License : */
00043
00044 #ifndef __STRUCTS__
00045
00046 #define __STRUCTS__
00047 #ifdef _MSC_VER
00048 #include <wchar.h>  /* off_t */
00049 #endif
00050
00051 /*
00052     #[ sav&store :
00053 */
00054
00059 typedef struct PoSiTiOn {
```

```
00060     off_t p1;
00061 } POSITION;
00062
00063 /*  Next are the index structs for stored and saved expressions */
00064
00075 typedef struct {
00076     UBYTE headermark[8];
00078     UBYTE lenWORD;
00079     UBYTE lenLONG;
00080     UBYTE lenPOS;
00081     UBYTE lenPOINTER;
00082     UBYTE endianness[16];
00083     UBYTE sSym;
00084     UBYTE sInd;
00085     UBYTE sVec;
00086     UBYTE sFun;
00087     UBYTE maxpower[16];
00088     UBYTE wildoffset[16];
00089     UBYTE revision;
00090     UBYTE reserved[512-8-4-16-4-16-16-1];
00091 } STOREHEADER;
00092
00093 STATIC_ASSERT(sizeof(STOREHEADER) == 512);
00094
00100 typedef struct InDeXeNtRy {
00101     POSITION    position;
00102     POSITION    length;
00103     POSITION    variables;
00104     LONG    CompressSize;
00105     WORD    nsymbols;
00106     WORD    nindices;
00107     WORD    nvectors;
00108     WORD    nfunctions;
00109     WORD    size;
00110     SBYTE   name[MAXENAME+1];
00111 } INDEXENTRY;
00112
00118 #define INFILEINDEX ((512-2*sizeof(POSITION))/sizeof(INDEXENTRY))
00123 #define EMPTYININDEX (512-2*sizeof(POSITION)-INFILEINDEX*sizeof(INDEXENTRY))
00124
00137 typedef struct FiLeInDeX {
00138     POSITION    next;
00139     POSITION    number;
00140     INDEXENTRY expression[INFILEINDEX];
00141     SBYTE   empty[EMPTYININDEX];
00142 } FILEINDEX;
00143
00144 STATIC_ASSERT(sizeof(FILEINDEX) == 512);
00145
00150 typedef struct FiLeDaTa {
00151     FILEINDEX Index;
00152     POSITION Fill;
00153     POSITION Position;
00154     WORD Handle;
00155     WORD dirtyflag;
00156 } FILEDATA;
00157
00164 typedef struct VaRrEnUm {
00165     WORD  *start;
00166     WORD  *lo;
00167     WORD  *hi;
00168 } VARRENUM;
00169
00176 typedef struct ReNuMbEr {
00177     POSITION   startposition;
00178     /* First stage renumbering */
00179     VARRENUM   symb;
00180     VARRENUM   indi;
00181     VARRENUM   vect;
00182     VARRENUM   func;
00183     /* Second stage renumbering */
00184     WORD       *symnum;
00185     WORD       *indnum;
00186     WORD       *vecnum;
00187     WORD       *funnum;
00188 } *RENUMBER;
00189
00190 /*
00191     #] sav&store :
00192     #[ Variables :
00193 */
00194
00202 typedef struct {
00203     void *lijst;
00204     char *message;
00205     int num;
00206     int maxnum;
```

```
00207     int size;
00208     int numglobal;
00209     int numtemp;
00210     int numclear;
00211 } LIST;
00212
00218 typedef struct {
00219     char *name;
00220     TFUN func;
00221     int type;
00222     int flags;
00223 } KEYWORD;
00224
00230 typedef struct {
00231     char *name;
00232     int *var;
00233     int type;
00234     int flags;
00235 } KEYWORDV;
00236
00243 typedef struct NaMeNode {
00244     LONG name;
00245     WORD parent;
00246     WORD left;
00247     WORD right;
00248     WORD balance;
00249     WORD type;
00250     WORD number;
00251 } NAMENODE;
00252
00260 typedef struct NaMeTree {
00261     NAMENODE *namenode;
00263     UBYTE    *namebuffer;
00266     LONG     nodesize;
00267     LONG     nodefill;
00268     LONG     namesize;
00269     LONG     namefill;
00270     LONG     oldnamefill;
00271     LONG     oldnodefill;
00272     LONG     globalnamefill;
00274     LONG     globalnodefill;
00275     LONG     clearnamefill;
00276     LONG     clearnodefill;
00277     WORD     headnode;
00278 } NAMETREE;
00279
00288 typedef struct tree {
00289     int parent;
00290     int left;
00291     int right;
00292     int value;
00293     int blnce;
00294     int usage;
00295 } COMPTREE;
00296
00301 typedef struct MiNmAx {
00302     WORD mini;
00303     WORD maxi;
00304     WORD size;
00305 } MINMAX;
00306
00311 typedef struct BrAcKeTiNdEx {   /* For indexing brackets in local expressions */
00312     POSITION start;             /* Place where bracket starts - start of expr */
00313     POSITION next;              /* Place of next indexed bracket in expr */
00314     LONG bracket;               /* Offset of position in bracketbuffer */
00315     LONG termsinbracket;
00316 } BRACKETINDEX;
00317
00322 typedef struct BrAcKeTiNfO {
00323     BRACKETINDEX *indexbuffer;
00324     WORD  *bracketbuffer;
00325     LONG  bracketbuffersize;
00326     LONG  indexbuffersize;
00327     LONG  bracketfill;
00328     LONG  indexfill;
00329     WORD  SortType;
00330 } BRACKETINFO;
00331
00342 typedef struct TaBlEs {
00343     WORD    *tablepointers;
00344 #ifdef WITHPTHREADS
00345     WORD    **prototype;
00346     WORD    **pattern;
00347 #else
00348     WORD    *prototype;
00349     WORD    *pattern;
00350 #endif
```

```
00351     MINMAX  *mm;
00352     WORD    *flags;
00353     COMPTREE *boomlijst;
00354     UBYTE   *argtail;
00356     struct TaBlEs *spare;
00357     WORD    *buffers;
00358     LONG    totind;
00359     LONG    reserved;
00360     LONG    defined;
00361     LONG    mdefined;
00362     int     prototypeSize;
00363     int     numind;
00364     int     bounds;
00365     int     strict;
00366     int     sparse;
00367     int     numtree;
00368     int     rootnum;
00369     int     MaxTreeSize;
00370     WORD    bufnum;
00371     WORD    bufferssize;
00372     WORD    buffersfill;
00373     WORD    tablenum;
00374     WORD    mode;
00375     WORD    numdummies;
00376 } *TABLES;
00377
00382 typedef struct ExPrEsSiOn {
00383     POSITION    onfile;
00384     POSITION    prototype;
00385     POSITION    size;
00386     RENUMBER renum;         /* For Renumbering of global stored expressions */
00387     BRACKETINFO *bracketinfo;
00388     BRACKETINFO *newbracketinfo;
00389     WORD    *renumlists;
00391     WORD    *inmem;         /* If in memory like e.g. a polynomial */
00392     LONG    counter;
00393     LONG    name;
00394     WORD    hidelevel;
00395     WORD    vflags;         /* Various flags */
00396     WORD    printflag;
00397     WORD    status;
00398     WORD    replace;
00399     WORD    node;
00400     WORD    whichbuffer;
00401     WORD    namesize;
00402     WORD    compression;
00403     WORD    numdummies;
00404     WORD    numfactors;
00405     WORD    sizeprototype;
00406     WORD    uflags;         /* User flags */
00407 #ifdef PARALLELCODE
00408     WORD    partodo;        /* Whether to be done in parallel mode */
00409 #endif
00410 } *EXPRESSIONS;
00411
00416 typedef struct SyMbOl {           /* Don't change unless altering .sav too */
00417     LONG    name;              /* Location in names buffer */
00418     WORD    minpower;          /* Minimum power admissible */
00419     WORD    maxpower;          /* Maximum power admissible */
00420     WORD    complex;           /* Properties wrt complex conjugation */
00421     WORD    number;            /* Number when stored in file */
00422     WORD    flags;             /* Used to indicate usage when storing */
00423     WORD    node;
00424     WORD    namesize;
00425     WORD    dimension;         /* For dimensionality checks */
00426 #if BITSINWORD == 32
00427     UBYTE d_u_m_m_y[8];     /* Padding for sav compatibility */
00428 #elif BITSINWORD == 16
00429     UBYTE d_u_m_m_y[4];     /* Padding for sav compatibility */
00430 #endif
00431 } *SYMBOLS;
00432 #if BITSINWORD == 32
00433 STATIC_ASSERT(sizeof(struct SyMbOl) == 48);
00434 #elif BITSINWORD == 16
00435 STATIC_ASSERT(sizeof(struct SyMbOl) == 24);
00436 #endif
00437
00442 typedef struct InDeX {         /* Don't change unless altering .sav too */
00443     LONG    name;              /* Location in names buffer */
00444     WORD    type;              /* Regular or dummy */
00445     WORD    dimension;         /* Value of d_(n,n) or -number of symbol */
00446     WORD    number;            /* Number when stored in file */
00447     WORD    flags;             /* Used to indicate usage when storing */
00448     WORD    nmin4;             /* Used for n-4 if dimension < 0 */
00449     WORD    node;
00450     WORD    namesize;
00451 } *INDICES;
```

```
00452 #if BITSINWORD == 32
00453 STATIC_ASSERT(sizeof(struct InDeX) == 40);
00454 #elif BITSINWORD == 16
00455 STATIC_ASSERT(sizeof(struct InDeX) == 20);
00456 #endif
00457
00462 typedef struct VeCtOr {           /* Don't change unless altering .sav too */
00463     LONG    name;               /* Location in names buffer */
00464     WORD    complex;            /* Properties under complex conjugation */
00465     WORD    number;             /* Number when stored in file */
00466     WORD    flags;              /* Used to indicate usage when storing */
00467     WORD    node;
00468     WORD    namesize;
00469     WORD    dimension;          /* For dimensionality checks */
00470 #if BITSINWORD == 32
00471     UBYTE d_u_m_m_y[8];     /* Padding for sav compatibility */
00472 #elif BITSINWORD == 16
00473     UBYTE d_u_m_m_y[4];     /* Padding for sav compatibility */
00474 #endif
00475 } *VECTORS;
00476 #if BITSINWORD == 32
00477 STATIC_ASSERT(sizeof(struct VeCtOr) == 40);
00478 #elif BITSINWORD == 16
00479 STATIC_ASSERT(sizeof(struct VeCtOr) == 20);
00480 #endif
00481
00487 typedef struct FuNcTiOn {  /* Don't change unless altering .sav too */
00488     TABLES  tabl;
00489     LONG    symminfo;
00490     LONG    name;
00491     WORD    commute;
00492     WORD    complex;
00493     WORD    number;
00494     WORD    flags;
00495     WORD    spec;
00496     WORD    symmetric;
00497     WORD    node;
00498     WORD    namesize;
00499     WORD    dimension;     /* For dimensionality checks */
00500     WORD    maxnumargs;
00501     WORD    minnumargs;
00502 } *FUNCTIONS;
00503 #if BITSINWORD == 32
00504 STATIC_ASSERT(sizeof(struct FuNcTiOn) == 72);
00505 #elif BITSINWORD == 16
00506 STATIC_ASSERT(sizeof(struct FuNcTiOn) == 36);
00507 #endif
00508
00513 typedef struct SeTs {
00514     LONG    name;               /* Location in names buffer */
00515     WORD    type;               /* Symbol, vector, index or function */
00516     WORD    first;              /* First element in setstore */
00517     WORD    last;               /* Last element in setstore (excluding) */
00518     WORD    node;
00519     WORD    namesize;
00520     WORD    dimension;          /* For dimensionality checks */
00521     WORD    flags;              /* Like: ordered */
00522 } *SETS;
00523
00528 typedef struct DuBiOuS {     /* Undeclared objects. Just for compiler. */
00529     LONG    name;               /* Location in names buffer */
00530     WORD    node;
00531     WORD    dummy;
00532 } *DUBIOUSV;
00533
00534 typedef struct FaCdOlLaR {
00535     WORD    *where;             /* A pointer(!) to the content */
00536     LONG    size;
00537     WORD    type;               /* Type can be DOLNUMBER or DOLTERMS */
00538     WORD    value;              /* in case it is a (short) number */
00539 } FACDOLLAR;
00540
00541 typedef struct DoLlArS {
00542     WORD    *where;             /* A pointer(!) to the object */
00543     FACDOLLAR *factors;            /* an array of factors. nfactors elements */
00544 #ifdef WITHPTHREADS
00545     pthread_mutex_t pthreadslock;
00546 #endif
00547     LONG    size;               /* The number of words */
00548     LONG    name;
00549     WORD    type;
00550     WORD    node;
00551     WORD    index;
00552     WORD    zero;
00553     WORD    numdummies;
00554     WORD    nfactors;
00555 } *DOLLARS;
```

```
00556
00561 typedef struct MoDoPtDoLlArS {
00562 #ifdef WITHPTHREADS
00563     DOLLARS  dstruct;    /* If local,min,max dollar: list of DOLLARS for each thread */
00564 #endif
00565     WORD    number;
00566     WORD    type;
00567 } MODOPTDOLLAR;
00568
00573 typedef struct fixedset {
00574     char *name;
00575     char *description;
00576     int type;
00577     int dimension;
00578 } FIXEDSET;
00579
00584 typedef struct TaBlEbAsEsUbInDeX {
00585     POSITION where;
00586     LONG size;
00587 } TABLEBASESUBINDEX;
00588
00593 typedef struct TaBlEbAsE {
00594     POSITION fillpoint;
00595     POSITION current;
00596     UBYTE *name;
00597     int *tablenumbers;      /* Number of each table */
00598     TABLEBASESUBINDEX *subindex;       /* For each table */
00599     int numtables;
00600 } TABLEBASE;
00601
00608 typedef struct {
00609     WORD *location;
00610     int numargs;
00611     int numfunnies;
00612     int numwildcards;
00613     int symmet;
00614     int tensor;
00615     int commute;
00616 } FUN_INFO;
00617
00618 typedef struct PaRtIcLe {
00619     WORD number; /* Number of the function */
00620     WORD spin;   /* +/- dimension of SU(2) representation */
00621     WORD mass;   /* Number of symbol or 0 */
00622     WORD type;   /* -1: anti particle, 1: particle, 0: own antiparticle */
00623 } PARTICLE;
00624
00625 typedef struct VeRtEx {
00626     PARTICLE particles[MAXPARTICLES];
00627     WORD couplings[2*MAXCOUPLINGS];
00628     WORD nparticles;
00629     WORD ncouplings;
00630     WORD type;
00631     WORD error;
00632     WORD externonly;
00633     WORD spare;
00634 } VERTEX;
00635
00636 typedef struct MoDeL {
00637     VERTEX **vertices;
00638     WORD *couplings;
00639     UBYTE *name;
00640     void *grccmodel;
00641     WORD legcouple[MAXLEGS+2];
00642     WORD nparticles;
00643     WORD nvertices;
00644     WORD invertices;
00645     WORD sizevertices;
00646     WORD sizecouplings;
00647     WORD ncouplings;
00648     WORD error;
00649     WORD dummy;
00650 } MODEL;
00651
00652 /*
00653  *  The struct NAMESPACE is used for the namespace info. It is a
00654  *  doubly linked list of nested namespaces.
00655  */
00656
00657 typedef struct NaMeSpAcE {
00658     struct NaMeSpAcE *previous;
00659     struct NaMeSpAcE *next;
00660     NAMETREE  *usenames;
00661     UBYTE     *name;
00662 } NAMESPACE;
00663
00664
```

```
00665 /*
00666     #] Variables :
00667     #[ Files :
00668 */
00669
00681 typedef struct FiLe {
00682     POSITION POposition;        /* File position */
00683     POSITION filesize;          /* Because SEEK_END is unsafe on IBM */
00684     WORD *PObuffer;             /* Address of the intermediate buffer */
00685     WORD *POstop;               /* End of the buffer */
00686     WORD *POfill;               /* Fill position of the buffer */
00687     WORD *POfull;               /* Full buffer when only cached */
00688 #ifdef WITHPTHREADS
00689     WORD *wPObuffer;             /* Address of the intermediate worker buffer */
00690     WORD *wPOstop;               /* End of the worker buffer */
00691     WORD *wPOfill;               /* Fill position of the worker buffer */
00692     WORD *wPOfull;               /* Full buffer when only worker cached */
00693 #endif
00694     char *name;                 /* name of the file */
00695 #ifdef WITHZLIB
00696     z_streamp zsp;              /* The pointer to the stream struct for gzip */
00697     Bytef *ziobuffer;           /* The output buffer for compression */
00698 #endif
00699     ULONG numblocks;            /* Number of blocks in file */
00700     ULONG inbuffer;             /* Block in the buffer */
00701     LONG POsize;                /* size of the buffer */
00702 #ifdef WITHZLIB
00703     LONG ziosize;               /* size of the zoutbuffer */
00704 #endif
00705 #ifdef WITHPTHREADS
00706     LONG wPOsize;                /* size of the worker buffer */
00707     pthread_mutex_t pthreadslock;
00708 #endif
00709     int handle;
00710     int active;                 /* File is open or closed. Not used. */
00711 } FILEHANDLE;
00712
00722 typedef struct StreaM {
00723     off_t fileposition;
00724     off_t linenumber;
00725     off_t prevline;
00726     UBYTE *buffer;
00727     UBYTE *pointer;
00728     UBYTE *top;
00729     UBYTE *FoldName;
00730     UBYTE *name;
00731     UBYTE *pname;
00733     LONG buffersize;
00734     LONG bufferposition;
00735     LONG inbuffer;
00736     int previous;
00737     int handle;
00738     int type;
00739     int prevars;
00740     int previousNoShowInput;
00741     int eqnum;
00742     int afterwards;
00743     int olddelay;
00744     int oldnoshowinput;
00745     UBYTE isnextchar;
00746     UBYTE nextchar[2];
00747     UBYTE reserved;
00748 } STREAM;
00749
00750 typedef struct SpecTatoR {
00751     POSITION position;   /* The place where we will be writing */
00752     POSITION readpos;    /* The place from which we read */
00753     FILEHANDLE *fh;
00754     char *name;          /* We identify the spectator by the name of the expression */
00755     WORD exprnumber;     /* During running we use the number. */
00756     WORD flags;          /* local, global? */
00757 } SPECTATOR;
00758
00759 /*
00760     #] Files :
00761     #[ Traces :
00762 */
00763
00773 typedef struct TrAcEs {           /* For computing 4 dimensional traces */
00774     WORD        *accu;      /* NUMBER * 2 */
00775     WORD        *accup;
00776     WORD        *termp;
00777     WORD        *perm;      /* number */
00778     WORD        *inlist;    /* number */
00779     WORD        *nt3;       /* number/2 */
00780     WORD        *nt4;       /* number/2 */
00781     WORD        *j3;        /* number*2 */
```

```
00782     WORD        *j4;        /* number*2 */
00783     WORD        *e3;        /* number*2 */
00784     WORD        *e4;        /* number */
00785     WORD        *eers;      /* number/2 */
00786     WORD        *mepf;      /* number/2 */
00787     WORD        *mdel;      /* number/2 */
00788     WORD        *pepf;      /* number*2 */
00789     WORD        *pdel;      /* number*3/2 */
00790     WORD        sgn;
00791     WORD        stap;
00792     WORD        step1,kstep,mdum;
00793     WORD        gamm,ad,a3,a4,lc3,lc4;
00794     WORD        sign1,sign2,gamma5,num,level,factor,allsign;
00795     WORD        finalstep;
00796 } TRACES;
00797
00805 typedef struct TrAcEn {           /* For computing n dimensional traces */
00806     WORD        *accu;      /* NUMBER */
00807     WORD        *accup;
00808     WORD        *termp;
00809     WORD        *perm;      /* number */
00810     WORD        *inlist;    /* number */
00811     WORD        sgn,num,level,factor,allsign;
00812 } *TRACEN;
00813
00814 /*
00815     #] Traces :
00816     #[ Preprocessor :
00817 */
00818
00823 typedef struct pReVaR {
00824     UBYTE *name;
00825     UBYTE *value;
00826     UBYTE *argnames;
00827     int nargs;
00828     int wildarg;
00829 } PREVAR;
00830
00835 typedef struct {
00836     WORD *buffer;
00837     int oldcompiletype;
00838     int oldparallelflag;
00839     int oldnumpotmoddollars;
00840     WORD size;
00841     WORD numdollars;
00842     WORD oldcbuf;
00843     WORD oldrbuf;
00844     WORD inscbuf;
00845     WORD oldcnumlhs;
00846 } INSIDEINFO;
00847
00853 typedef struct {
00854     UBYTE   *buffer;
00855     LONG    size;
00856 } PRELOAD;
00857
00862 typedef struct {
00863     PRELOAD p;
00864     UBYTE   *name;
00865     int     loadmode;
00866     int     mustfree;
00867 } PROCEDURE;
00868
00876 typedef struct DoLoOp {
00877     PRELOAD p;
00878     UBYTE *name;
00879     UBYTE *vars;        /* for {} or name of expression */
00880     UBYTE *contents;
00881     UBYTE *dollarname;
00882     LONG startlinenumber;
00883     LONG firstnum;
00884     LONG lastnum;
00885     LONG incnum;
00886     int type;
00887     int NoShowInput;
00888     int errorsinloop;
00889     int firstloopcall;
00890     WORD firstdollar;   /* When >= 0 we have to get the value from a dollar */
00891     WORD lastdollar;    /* When >= 0 we have to get the value from a dollar */
00892     WORD incdollar;     /* When >= 0 we have to get the value from a dollar */
00893     WORD NumPreTypes;
00894     WORD PreIfLevel;
00895     WORD PreSwitchLevel;
00896 } DOLOOP;
00897
00905 struct  bit_field {    /* Assume 8 bits per byte */
00906     UBYTE bit_0        : 1;
```

```
00907     UBYTE bit_1        : 1;
00908     UBYTE bit_2        : 1;
00909     UBYTE bit_3        : 1;
00910     UBYTE bit_4        : 1;
00911     UBYTE bit_5        : 1;
00912     UBYTE bit_6        : 1;
00913     UBYTE bit_7        : 1;
00914 /*
00915     UINT bit_0        : 1;
00916     UINT bit_1        : 1;
00917     UINT bit_2        : 1;
00918     UINT bit_3        : 1;
00919     UINT bit_4        : 1;
00920     UINT bit_5        : 1;
00921     UINT bit_6        : 1;
00922     UINT bit_7        : 1;
00923 */
00924 };
00925
00930 typedef struct bit_field set_of_char[32];
00931
00936 typedef struct bit_field *one_byte;
00937
00942 typedef struct {
00943     WORD newlogonly;
00944     WORD newhandle;
00945     WORD oldhandle;
00946     WORD oldlogonly;
00947     WORD oldprinttype;
00948     WORD oldsilent;
00949 } HANDLERS;
00950
00951 /*
00952     #] Preprocessor :
00953     #[ Varia :
00954 */
00955
00956 typedef int (*FINISHUFFLE)(WORD *);
00957 typedef int (*DO_UFFLE)(WORD *,WORD,WORD,WORD);
00958
00959 #ifdef WITHPTHREADS
00960 typedef WORD (*COMPAREDUMMY)(void *,WORD *,WORD *,WORD);
00961 #else
00962 typedef WORD (*COMPAREDUMMY)(WORD *,WORD *,WORD);
00963 #endif
00964
00970 typedef struct CbUf {
00971     WORD *Buffer;
00972     WORD *Top;
00973     WORD *Pointer;
00974     WORD **lhs;
00975     WORD **rhs;
00976     LONG *CanCommu;
00977     LONG *NumTerms;
00978     WORD *numdum;
00979     WORD *dimension;
00980     COMPTREE *boomlijst;
00981     LONG BufferSize;
00982     int numlhs;
00983     int numrhs;
00984     int maxlhs;
00985     int maxrhs;
00986     int mnumlhs;
00987     int mnumrhs;
00988     int numtree;
00989     int rootnum;
00990     int MaxTreeSize;
00991 } CBUF;
00992
01000 typedef struct ChAnNeL {
01001     char *name;
01002     int handle;
01003 } CHANNEL;
01004
01014 typedef struct {
01015     UBYTE *parameter;
01016     int type;
01017     int flags;
01018     // This ordering is assumed by initializer lists in many places. In this
01019     // case the struct should anyway not have additional padding between the
01020     // two int and the LONG members.
01021     LONG value;
01022 } SETUPPARAMETERS;
01023
01032 typedef struct NeStInG {
01033     WORD *termsize;
01034     WORD *funsize;
```

```
01035     WORD *argsize;
01036 } *NESTING;
01037
01044 typedef struct StOrEcAcHe {
01045     POSITION position;
01046     POSITION toppos;
01047     struct StOrEcAcHe *next;
01048     WORD buffer[2];
01049 } *STORECACHE;
01050
01056 typedef struct PeRmUtE {
01057     WORD *objects;
01058     WORD sign;
01059     WORD n;
01060     WORD cycle[MAXMATCH];
01061 } PERM;
01062
01067 typedef struct PeRmUtEp {
01068     WORD **objects;
01069     WORD sign;
01070     WORD n;
01071     WORD cycle[MAXMATCH];
01072 } PERMP;
01073
01079 typedef struct DiStRiBuTe {
01080     WORD *obj1;
01081     WORD *obj2;
01082     WORD *out;
01083     WORD sign;
01084     WORD n1;
01085     WORD n2;
01086     WORD n;
01087     WORD cycle[MAXMATCH];
01088 } DISTRIBUTE;
01089
01095 typedef struct PaRtI {
01096     WORD *psize;  /* the sizes of the partitions */
01097     WORD *args;   /* the offsets of the arguments to be partitioned */
01098     WORD *nargs;  /* argument numbers (different number = different argument) */
01099     WORD *nfun;   /* the functions into which the partitions go */
01100     WORD numargs; /* the number of arguments to be partitioned */
01101     WORD numpart; /* the number of partitions */
01102     WORD where;   /* offset of the function in the term */
01103 } PARTI;
01104
01114 typedef struct sOrT {
01115     FILEHANDLE file;            /* The own sort file */
01116     POSITION SizeInFile[3];     /* Sizes in the various files */
01117     POSITION OldPosIn;          /* Sort file fill positions */
01118     POSITION OldPosOut;
01119     WORD *lBuffer;              /* The large buffer */
01120     WORD *lTop;                 /* End of the large buffer */
01121     WORD *lFill;                /* The filling point of the large buffer */
01122     WORD *used;                 /*  auxiliary during actual sort */
01123     WORD *sBuffer;              /* The small buffer */
01124     WORD *sTop;                 /* End of the small buffer */
01125     WORD *sTop2;                /* End of the extension of the small buffer */
01126     WORD *sHalf;                /* Halfway point in the extension */
01127     WORD *sFill;                /* Filling point in the small buffer */
01128     WORD **sPointer;            /* Pointers to terms in the small buffer */
01129     WORD **PoinFill;            /* Filling point for pointers to the sm.buf */
01130     WORD *cBuffer;              /* Compress buffer (if it exists) */
01131     WORD **Patches;             /* Positions of patches in large buffer */
01132     WORD **pStop;               /* Ends of patches in the large buffer */
01133     WORD **poina;               /*  auxiliary during actual sort */
01134     WORD **poin2a;              /*  auxiliary during actual sort */
01135     WORD *ktoi;                 /*  auxiliary during actual sort */
01136     WORD *tree;                 /*  auxiliary during actual sort */
01137 #ifdef WITHZLIB
01138     WORD *fpcompressed;         /*  is output filepatch compressed? */
01139     WORD *fpincompressed;       /*  is input filepatch compressed? */
01140     z_streamp zsparray;         /*  the array of zstreams for decompression */
01141 #endif
01142     POSITION *fPatches;         /* Positions of output file patches */
01143     POSITION *inPatches;        /* Positions of input file patches */
01144     POSITION *fPatchesStop;     /* Positions of output file patches */
01145     POSITION *iPatches;         /* Input file patches, Points to fPatches or inPatches */
01146     FILEHANDLE *f;              /* The actual output file */
01147     FILEHANDLE **ff;            /* Handles for a staged sort */
01148     LONG sTerms;                /* Terms in small buffer */
01149     LONG LargeSize;             /* Size of large buffer (in words) */
01150     LONG SmallSize;             /* Size of small buffer (in words) */
01151     LONG SmallEsize;            /* Size of small + extension (in words) */
01152     LONG TermsInSmall;          /* Maximum number of terms in small buffer */
01153     LONG Terms2InSmall;         /* with extension for polyfuns etc. */
01154     LONG GenTerms;              /* Number of generated terms */
01155     LONG TermsLeft;             /* Number of terms still in existence */
```

```
01156     LONG GenSpace;              /* Amount of space of generated terms */
01157     LONG SpaceLeft;             /* Space needed for still existing terms */
01158     LONG putinsize;             /* Size of buffer in putin */
01159     LONG ninterms;              /* Which input term ? */
01160     LONG verbComparisons;       /* Counters for "On SortVerbose;" statistics */
01161     LONG verbSBsortTerms;
01162     LONG verbSBsortCap;
01163     LONG verbLBsortPatches;
01164     LONG verbLBsortCap;
01165     LONG verbUnsortedSize;
01166     LONG verbMaxTermSize;
01167     int MaxPatches;             /* Maximum number of patches in large buffer */
01168     int MaxFpatches;            /* Maximum number of patches in one filesort */
01169     int type;                   /* Main, function or sub(routine) */
01170     int lPatch;                 /* Number of patches in the large buffer */
01171     int fPatchN1;               /* Number of patches in input file */
01172     int PolyWise;               /* Is there a polyfun and if so, where? */
01173     int PolyFlag;               /*  */
01174     int cBufferSize;            /* Size of the compress buffer */
01175     int maxtermsize;            /* Keeps track for buffer allocations */
01176     int newmaxtermsize;         /* Auxiliary for maxtermsize */
01177     int outputmode;             /* Tells where the output is going */
01178     int stagelevel;             /* In case we have a 'staged' sort */
01179     WORD fPatchN;               /* Number of patches on file (output) */
01180     WORD inNum;                 /* Number of patches on file (input) */
01181     WORD stage4;                /* Are we using stage4? */
01182 } SORTING;
01183
01184 #ifdef WITHPTHREADS
01185
01191 typedef struct SoRtBlOcK {
01192     pthread_mutex_t *MasterBlockLock;
01193     WORD    **MasterStart;
01194     WORD    **MasterFill;
01195     WORD    **MasterStop;
01196     LONG    *BlockTerms;
01197     int     MasterNumBlocks;
01198     int     MasterBlock;
01199     int     FillBlock;
01200 } SORTBLOCK;
01201 #endif
01202
01203 #ifdef DEBUGGER
01204 typedef struct DeBuGgInG {
01205     int eflag;
01206     int printflag;
01207     int logfileflag;
01208     int stdoutflag;
01209 } DEBUGSTR;
01210 #endif
01211
01212 #ifdef WITHPTHREADS
01213
01219 typedef struct ThReAdBuCkEt {
01220     POSITION *deferbuffer;      /* For Keep Brackets: remember position */
01221     WORD *threadbuffer;         /* Here are the (primary) terms */
01222     WORD *compressbuffer;       /* For keep brackets we need the compressbuffer */
01223     LONG threadbuffersize;      /* Number of words in threadbuffer */
01224     LONG compressbuffersize;    /* Number of words in compressbuffer */
01225     LONG ddterms;               /* Number of primary+secondary terms represented */
01226     LONG firstterm;             /* The number of the first term in the bucket */
01227     LONG firstbracket;          /* When doing complete brackets */
01228     LONG lastbracket;           /* When doing complete brackets */
01229     pthread_mutex_t lock;       /* For the load balancing phase */
01230     int  free;                  /* Status of the bucket */
01231     int  totnum;                /* Total number of primary terms */
01232     int  usenum;                /* Which is the term being used at the moment */
01233     int  busy;                  /*  */
01234     int  type;                  /* Doing brackets? */
01235 } THREADBUCKET;
01236
01237 #endif
01238
01246 typedef struct {
01247     WORD    *coefs;             /* The array of coefficients */
01248     WORD    numsym;             /* The number of the symbol in the polynomial */
01249     WORD    arraysize;          /* The size of the allocation of coefs */
01250     WORD    polysize;           /* The maximum power in the polynomial */
01251     WORD    modnum;             /* The prime number of the modulus */
01252 } POLYMOD;
01253
01254 typedef struct {
01255     WORD    *outterm;           /* Used in DoShuffle/Merge/FinishShuffle system */
01256     WORD    *outfun;
01257     WORD    *incoef;
01258     WORD    *stop1;
01259     WORD    *stop2;
```

```
01260     WORD    *ststop1;
01261     WORD    *ststop2;
01262     FINISHUFFLE finishuf;
01263     DO_UFFLE    do_uffle;
01264     LONG    combilast;
01265     WORD    nincoef;
01266     WORD    level;
01267     WORD    thefunction;
01268     WORD    option;
01269 } SHvariables;
01270
01271 typedef struct {               /* Used for computing calculational cost in optim.c */
01272     LONG    add;
01273     LONG    mul;
01274     LONG    div;
01275     LONG    pow;
01276 } COST;
01277
01278 typedef struct {
01279     UWORD   *a;     /* The number array */
01280     UWORD   *m;     /* The modulus array */
01281     WORD    na;     /* Size of the number */
01282     WORD    nm;     /* size of the number in the modulus array */
01283 } MODNUM;
01284
01285 /*
01286     Struct for optimizing outputs. If changed, do not forget to change
01287     the padding information in the AO struct.
01288 */
01289 typedef struct {
01290     union { /* we do this to allow padding */
01291         float fval;
01292         int ival[2]; /* This should be enough */
01293     } mctsconstant;
01294     int   horner;
01295     int   hornerdirection;
01296     int   method;
01297     int   mctstimelimit;
01298     int   mctsnumexpand;
01299     int   mctsnumkeep;
01300     int   mctsnumrepeat;
01301     int   greedytimelimit;
01302     int   greedyminnum;
01303     int   greedymaxperc;
01304     int   printstats;
01305     int   debugflags;
01306     int   schemeflags;
01307     int   mctsdecaymode;
01308     int   saIter; /* Simulated annealing updates */
01309     union {
01310         float fval;
01311         int ival[2];
01312     } saMaxT; /* Maximum temperature of SA */
01313     union {
01314         float fval;
01315         int ival[2];
01316     } saMinT; /* Minimum temperature of SA */
01317     int    spare;
01318 } OPTIMIZE;
01319
01320 typedef struct {
01321     WORD  *code;
01322     UBYTE *nameofexpr;  /* It is easier to remember an expression by name */
01323     LONG  codesize;     /* We need this for the checkpoints */
01324     WORD  exprnr;       /* Problem here is: we renumber them in execute.c */
01325     WORD  minvar;
01326     WORD  maxvar;
01327 } OPTIMIZERESULT;
01328
01329 typedef struct {
01330     WORD *lhs;      /* Object to be replaced */
01331     WORD *rhs;      /* Depending on the type it will be UBYTE* or WORD* */
01332     int type;
01333     int size;       /* Size of the lhs */
01334 } DICTIONARY_ELEMENT;
01335
01336 typedef struct {
01337     DICTIONARY_ELEMENT **elements;
01338     UBYTE *name;
01339     int sizeelements;
01340     int numelements;
01341     int numbers;        /* deal with numbers */
01342     int variables;      /* deal with single variables */
01343     int characters;     /* deal with special characters */
01344     int funwith;        /* deal with functions with arguments */
01345     int gnumelements;   /* if .global shrinks the dictionary */
01346     int ranges;
```

```
01347 } DICTIONARY;
01348
01349 typedef struct {
01350     WORD ncase;
01351     WORD value;
01352     WORD compbuffer;
01353 } SWITCHTABLE;
01354
01355 typedef struct {
01356     SWITCHTABLE *table;
01357     SWITCHTABLE defaultcase;
01358     SWITCHTABLE endswitch;
01359     WORD typetable;
01360     WORD maxcase;
01361     WORD mincase;
01362     WORD numcases;
01363     WORD tablesize;
01364     WORD caseoffset;
01365     WORD iflevel;
01366     WORD whilelevel;
01367     WORD nestingsum;
01368     WORD padding;
01369 } SWITCH;
01370
01371 typedef struct {
01372     WORD *term;
01373     void *currentModel;
01374     void *currentMODEL;
01375     WORD *legcouple[MAXLEGS+1];
01376     LONG numtopo;
01377     LONG numdia;
01378     WORD diaoffset;
01379     WORD level;
01380     WORD externalset;
01381     WORD internalset;
01382     WORD numextern;
01383     WORD flags;
01384 } TERMINFO;
01385
01386 /*
01387     Struct to hold temporary data in Normalize, so that it is not allocated
01388     on the stack each function call. The sizes of these arrays introduces a
01389     maximum complexity of terms which can be normalized. Each array is
01390     allocated dynamically, by AllocNormData, such that valgrind can detect
01391     errors if they are over-run.
01392 */
01393 typedef struct NoRmDaTa {
01394     WORD  *psym;
01395     WORD  *pvec;
01396     WORD  *pdot;
01397     WORD  *pdel;
01398     WORD  *pind;
01399     WORD **peps;
01400     WORD **pden;
01401     WORD **pcom;
01402     WORD **pnco;
01403     WORD **pcon;
01404 } NORMDATA;
01405
01406 /*
01407     #] Varia :
01408     #[ A :
01409         #[ M : The M struct is for global settings at startup or .clear
01410 */
01411
01419 struct M_const {
01420     POSITION zeropos;              /* (M) is zero */
01421     SORTING *S0;
01422     UWORD   *gcmod;
01423     UWORD   *gpowmod;
01424     UBYTE   *TempDir;              /* (M) Path with where the temporary files go */
01425     UBYTE   *TempSortDir;          /* (M) Path with where the sort files go */
01426     UBYTE   *IncDir;               /* (M) Directory path for include files */
01427     UBYTE   *InputFileName;        /* (M) */
01428     UBYTE   *LogFileName;          /* (M) */
01429     UBYTE   *OutBuffer;            /* (M) Output buffer in pre.c */
01430     UBYTE   *Path;                 /* (M) */
01431     UBYTE   *SetupDir;             /* (M) Directory with setup file */
01432     UBYTE   *SetupFile;            /* (M) Name of setup file */
01433     UBYTE   *gFortran90Kind;
01434     UBYTE   *gextrasym;
01435     UBYTE   *ggextrasym;
01436     UBYTE   *oldnumextrasymbols;
01437     SPECTATOR *SpectatorFiles;
01438 #ifdef WITHPTHREADS
01439     pthread_rwlock_t handlelock;   /* (M) */
01440     pthread_mutex_t storefilelock; /* (M) */
```

```
01441     pthread_mutex_t sbuflock;      /* (M) Lock for writing in the AM.sbuffer */
01442     LONG    ThreadScratSize;       /* (M) Size of Fscr[0/2] buffers of the workers */
01443     LONG    ThreadScratOutSize;    /* (M) Size of Fscr[1] buffers of the workers */
01444 #endif
01445     LONG    MaxTer;                /* (M) Maximum term size. Fixed at setup. In Bytes!!!*/
01446     LONG    CompressSize;          /* (M) Size of Compress buffer */
01447     LONG    ScratSize;             /* (M) Size of Fscr[] buffers */
01448     LONG    HideSize;              /* (M) Size of Fscr[2] buffer */
01449     LONG    SizeStoreCache;        /* (M) Size of the caches for reading global expr. */
01450     LONG    MaxStreamSize;         /* (M) Maximum buffer size in reading streams */
01451     LONG    SIOsize;               /* (M) Sort InputOutput buffer size */
01452     LONG    SLargeSize;            /* (M) */
01453     LONG    SSmallEsize;           /* (M) */
01454     LONG    SSmallSize;            /* (M) */
01455     LONG    STermsInSmall;         /* (M) */
01456     LONG    MaxBracketBufferSize;  /* (M) Max Size for B+ or AB+ per expression */
01457     LONG    hProcessBucketSize;    /* (M) */
01458     LONG    gProcessBucketSize;    /* (M) */
01459     LONG    shmWinSize;            /* (M) size for shared memory window used in communications */
01460     LONG    OldChildTime;          /* (M) Zero time. Needed in timer. */
01461     LONG    OldSecTime;            /* (M) Zero time for measuring wall clock time */
01462     LONG    OldMilliTime;          /* (M) Same, but milli seconds */
01463     LONG    WorkSize;              /* (M) Size of WorkSpace */
01464     LONG    gThreadBucketSize;     /* (C) */
01465     LONG    ggThreadBucketSize;    /* (C) */
01466     LONG    SumTime;               /*     Used in .clear */
01467     LONG    SpectatorSize;         /*     Size of the buffer in bytes */
01468     LONG    TimeLimit;             /*     Limit in sec to the total real time */
01469 #ifdef WITHFLOAT
01470     LONG    gDefaultPrecision;     /* (M) Default precision in bits for float_ */
01471     LONG    gMaxWeight;            /* (M) Maximum weight for MZV or Euler */
01472     LONG    ggDefaultPrecision;    /* (M) Default precision in bits for float_ */
01473     LONG    ggMaxWeight;           /* (M) Maximum weight for MZV or Euler */
01474 #endif
01475     int     FileOnlyFlag;          /* (M) Writing only to file */
01476     int     Interact;              /* (M) Interactive mode flag */
01477     int     MaxParLevel;           /* (M) Maximum nesting of parentheses */
01478     int     OutBufSize;            /* (M) size of OutBuffer */
01479     int     SMaxFpatches;          /* (M) */
01480     int     SMaxPatches;           /* (M) */
01481     int     StdOut;                /* (M) Regular output channel */
01482     int     ginsidefirst;          /* (M) Not used yet */
01483     int     gDefDim;               /* (M) */
01484     int     gDefDim4;              /* (M) */
01485     int     NumFixedSets;          /* (M) Number of the predefined sets */
01486     int     NumFixedFunctions;     /* (M) Number of built in functions */
01487     int     rbufnum;               /* (M) startup compiler buffer */
01488     int     dbufnum;               /* (M) dollar variables */
01489     int     sbufnum;               /* (M) subterm variables */
01490     int     zbufnum;               /* (M) special values */
01491     int     SkipClears;            /* (M) Number of .clear to skip at start */
01492     int     gTokensWriteFlag;      /* (M) */
01493     int     gfunpowers;            /* (M) */
01494     int     gStatsFlag;            /* (M) */
01495     int     gNamesFlag;            /* (M) */
01496     int     gCodesFlag;            /* (M) */
01497     int     gSortType;             /* (M) */
01498     int     gproperorderflag;      /* (M) */
01499     int     hparallelflag;         /* (M) */
01500     int     gparallelflag;         /* (M) */
01501     int     totalnumberofthreads;  /* (M) */
01502     int     gSizeCommuteInSet;
01503     int     gThreadStats;
01504     int     ggThreadStats;
01505     int     gFinalStats;
01506     int     ggFinalStats;
01507     int     gThreadsFlag;
01508     int     ggThreadsFlag;
01509     int     gThreadBalancing;
01510     int     ggThreadBalancing;
01511     int     gThreadSortFileSynch;
01512     int     ggThreadSortFileSynch;
01513     int     gProcessStats;
01514     int     ggProcessStats;
01515     int     gOldParallelStats;
01516     int     ggOldParallelStats;
01517     int     maxFlevels;            /* () maximum function levels */
01518     int     resetTimeOnClear;      /* (M) */
01519     int     gcNumDollars;          /* () number of dollars for .clear */
01520     int     MultiRun;
01521     int     gNoSpacesInNumbers;    /* For very long numbers */
01522     int     ggNoSpacesInNumbers;   /* For very long numbers */
01523     int     gIsFortran90;
01524     int     PrintTotalSize;
01525     int     fbuffersize;           /* Size for the AT.fbufnum factorization caches */
01526     int     gOldFactArgFlag;
01527     int     ggOldFactArgFlag;
```

```
01528     int     gnumextrasym;
01529     int     ggnumextrasym;
01530     int     NumSpectatorFiles;     /* Elements used in AM.spectatorfiles; */
01531     int     SizeForSpectatorFiles; /* Size in AM.spectatorfiles; */
01532     int     gOldGCDflag;
01533     int     ggOldGCDflag;
01534     int     gWTimeStatsFlag;
01535     int     ggWTimeStatsFlag;
01536     int     jumpratio;
01537     WORD    MaxTal;                /* (M) Maximum number of words in a number */
01538     WORD    IndDum;                /* (M) Basis value for dummy indices */
01539     WORD    DumInd;                /* (M) */
01540     WORD    WilInd;                /* (M) Offset for wildcard indices */
01541     WORD    gncmod;                /* (M) Global setting of modulus. size of gcmod */
01542     WORD    gnpowmod;              /* (M) Global printing as powers. size gpowmod */
01543     WORD    gmodmode;              /* (M) Global mode for modulus */
01544     WORD    gUnitTrace;            /* (M) Global value of Tr[1] */
01545     WORD    gOutputMode;           /* (M) */
01546     WORD    gOutputSpaces;         /* (M) */
01547     WORD    gOutNumberType;        /* (M) */
01548     WORD    gCnumpows;             /* (M) */
01549     WORD    gUniTrace[4];          /* (M) */
01550     WORD    MaxWildcards;          /* (M) Maximum number of wildcards */
01551     WORD    mTraceDum;             /* (M) Position/Offset for generated dummies */
01552     WORD    OffsetIndex;           /* (M) */
01553     WORD    OffsetVector;          /* (M) */
01554     WORD    RepMax;                /* (M) Max repeat levels */
01555     WORD    LogType;               /* (M) Type of writing to log file */
01556     WORD    ggStatsFlag;           /* (M) */
01557     WORD    gLineLength;           /* (M) */
01558     WORD    qError;                /* (M) Only error checking {-c option} */
01559     WORD    FortranCont;           /* (M) Fortran Continuation character */
01560     WORD    HoldFlag;              /* (M) Exit on termination? */
01561     WORD    Ordering[15];          /* (M) Auxiliary for ordering wildcards */
01562     WORD    silent;                /* (M) Silent flag. Only results in output. */
01563     WORD    tracebackflag;         /* (M) For tracing errors */
01564     WORD    expnum;                /* (M) internal number of ^ function */
01565     WORD    denomnum;              /* (M) internal number of / function */
01566     WORD    facnum;                /* (M) internal number of fac_ function */
01567     WORD    invfacnum;             /* (M) internal number of invfac_ function */
01568     WORD    sumnum;                /* (M) internal number of sum_ function */
01569     WORD    sumpnum;               /* (M) internal number of sump_ function */
01570     WORD    OldOrderFlag;          /* (M) Flag for allowing old statement order */
01571     WORD    termfunnum;            /* (M) internal number of term_ function */
01572     WORD    matchfunnum;           /* (M) internal number of match_ function */
01573     WORD    countfunnum;           /* (M) internal number of count_ function */
01574     WORD    gPolyFun;              /* (M) global value of PolyFun */
01575     WORD    gPolyFunInv;           /* (M) global value of Inverse of PolyFun */
01576     WORD    gPolyFunType;          /* (M) global value of PolyFun */
01577     WORD    gPolyFunExp;
01578     WORD    gPolyFunVar;
01579     WORD    gPolyFunPow;
01580     WORD    dollarzero;            /* (M) for dollars with zero value */
01581     WORD    atstartup;             /* To protect against DATE_ ending in \n */
01582     WORD    exitflag;              /* (R) For the exit statement */
01583     WORD    NumStoreCaches;        /* () Number of storage caches per processor */
01584     WORD    gIndentSpace;          /* For indentation in output */
01585     WORD    ggIndentSpace;
01586     WORD    gShortStatsMax;
01587     WORD    ggShortStatsMax;
01588     WORD    gextrasymbols;
01589     WORD    ggextrasymbols;
01590     WORD    zerorhs;
01591     WORD    onerhs;
01592     WORD    havesortdir;
01593     WORD    vectorzero;            /* p0_ */
01594     WORD    ClearStore;
01595     WORD    BracketFactors[8];
01596     BOOL    FromStdin;             /* read the input from STDIN */
01597     BOOL    IgnoreDeprecation;     /* ignore deprecation warning */
01598 };
01599 /*
01600         #] M :
01601         #[ P : The P struct defines objects set by the preprocessor
01602 */
01610 struct P_const {
01611     LIST DollarList;               /* (R) Dollar variables. Contains pointers
01612                                       to contents of the variables.*/
01613     LIST PreVarList;               /* (R) List of preprocessor variables
01614                                       Points to contents. Can be changed */
01615     LIST LoopList;                 /* (P) List of do loops */
01616     LIST ProcList;                 /* (P) List of procedures */
01617     INSIDEINFO inside;             /*     Information during #inside/#endinside */
01618     NAMESPACE *firstnamespace;     /* is zero when no namespace specified */
01619     NAMESPACE *lastnamespace;      /* is zero when no namespace specified */
01620     UBYTE *fullname;               /* buffer for namespace expanded names */
01621     UBYTE **PreSwitchStrings;      /* (P) The string in a switch */
```

```
01622     UBYTE *preStart;              /* (P) Preprocessor instruction buffer */
01623     UBYTE *preStop;               /* (P) end of preStart */
01624     UBYTE *preFill;               /* (P) Filling point in preStart */
01625     UBYTE *procedureExtension;    /* (P) Extension for procedure files (prc) */
01626     UBYTE *cprocedureExtension;   /* (P) Extension after .clear */
01627     LONG  *PreAssignStack;        /* For nesting #$name assignments */
01628     int *PreIfStack;              /* (P) Tracks nesting of #if */
01629     int *PreSwitchModes;          /* (P) Stack of switch status */
01630     int *PreTypes;                /* (P) stack of #call, #do etc nesting */
01631 #ifdef WITHPTHREADS
01632     pthread_mutex_t PreVarLock;   /* (P) */
01633 #endif
01634     LONG    StopWatchZero;        /* For `timer_' and #reset timer */
01635     LONG    InOutBuf;             /* (P) Characters in the output buf in pre.c */
01636     LONG    pSize;                /* (P) size of preStart */
01637     int     PreAssignFlag;        /* (C) Indicates #assign -> catch dollar */
01638     int     PreContinuation;      /* (C) Indicates whether the statement is new */
01639     int     PreproFlag;           /* (P) Internal use to mark work on prepro instr. */
01640     int     iBufError;            /* (P) Flag for errors with input buffer */
01641     int     PreOut;               /* (P) Flag for #+ #- */
01642     int     PreSwitchLevel;       /* (P) Nesting of #switch */
01643     int     NumPreSwitchStrings;  /* (P) Size of PreSwitchStrings */
01644     int     MaxPreTypes;          /* (P) Size of PreTypes */
01645     int     NumPreTypes;          /* (P) Number of nesting objects in PreTypes */
01646     int     MaxPreIfLevel;        /* (C) Maximum number of nested #if. Dynamic */
01647     int     PreIfLevel;           /* (C) Current position if PreIfStack */
01648     int     PreInsideLevel;       /* (C) #inside active? */
01649     int     DelayPrevar;          /* (P) Delaying prevar substitution */
01650     int     AllowDelay;           /* (P) Allow delayed prevar substitution */
01651     int     lhdollarerror;        /* (R) */
01652     int     eat;                  /* () */
01653     int     gNumPre;              /* (P) Number of preprocessor variables for .clear */
01654     int     PreDebug;             /* (C) */
01655     int     OpenDictionary;
01656     int     PreAssignLevel;       /* For nesting #$name = ...; assignments */
01657     int     MaxPreAssignLevel;    /* For nesting #$name = ...; assignments */
01658     int     fullnamesize;         /* size of the fullname buffer */
01659     int     FoundFileSetupCount;  /* The number of "file setup" (#:) lines */
01660     WORD    DebugFlag;            /* (P) For debugging purposes */
01661     WORD    preError;             /* (P) Blocks certain types of execution */
01662     UBYTE   ComChar;              /* (P) Commentary character */
01663     UBYTE   cComChar;             /* (P) Old commentary character for .clear */
01664 };
01665
01666 /*
01667         #] P :
01668         #[ C : The C struct defines objects changed by the compiler
01669 */
01670
01679 struct C_const {
01680     set_of_char separators;
01681     POSITION StoreFileSize;       /* () Size of store file */
01682     NAMETREE *dollarnames;
01683     NAMETREE *exprnames;
01684     NAMETREE *varnames;
01685     LIST    ChannelList;
01686                                   /*     Later also for write? */
01687     LIST    DubiousList;
01689     LIST    FunctionList;
01690     LIST    ExpressionList;
01691     LIST    IndexList;
01692     LIST    SetElementList;
01693     LIST    SetList;
01694     LIST    SymbolList;
01695     LIST    VectorList;
01696     LIST    PotModDolList;
01697     LIST    ModOptDolList;
01698     LIST    TableBaseList;
01699 /*
01700     Compile buffer variables
01701 */
01702     LIST    cbufList;
01703 /*
01704     Objects for auto declarations
01705 */
01706     LIST    AutoSymbolList;       /* (C) */
01707     LIST    AutoIndexList;        /* (C) */
01708     LIST    AutoVectorList;       /* (C) */
01709     LIST    AutoFunctionList;     /* (C) */
01710     NAMETREE *autonames;
01712     LIST    *Symbols;             /* (C) Pointer for autodeclare. Which list is
01713                                      it searching. Later also for subroutines */
01714     LIST    *Indices;             /* (C) id. */
01715     LIST    *Vectors;             /* (C) id. */
01716     LIST    *Functions;           /* (C) id. */
01717     NAMETREE **activenames;
01719     STREAM  *Streams;
```

```
01720     STREAM  *CurrentStream;
01722     SWITCH  *SwitchArray;
01723     MODEL   **models;
01724     WORD    *SwitchHeap;
01725     LONG    *termstack;
01726     LONG    *termsortstack;
01727     UWORD   *cmod;
01728     UWORD   *powmod;
01729     UWORD   *modpowers;
01730     UWORD   *halfmod;             /* (C) half the modulus when not zero */
01731     WORD    *ProtoType;           /* (C) The subexpression prototype {wildcards} */
01732     WORD    *WildC;               /* (C) Filling point for wildcards. */
01733     LONG    *IfHeap;
01734     LONG    *IfCount;
01735     LONG    *IfStack;
01736     UBYTE   *iBuffer;
01737     UBYTE   *iPointer;
01738     UBYTE   *iStop;
01739     UBYTE   **LabelNames;
01740     WORD    *FixIndices;
01741     WORD    *termsumcheck;
01742     UBYTE   *WildcardNames;
01743     int     *Labels;
01744     SBYTE   *tokens;
01745     SBYTE   *toptokens;
01746     SBYTE   *endoftokens;
01747     WORD    *tokenarglevel;
01748     UWORD   *modinverses;         /* Table for inverses of primes */
01749     UBYTE   *Fortran90Kind;       /* The kind of number in Fortran 90 as in _ki */
01750     WORD    **MultiBracketBuf;    /* Array of buffers for multi-level brackets */
01751     UBYTE   *extrasym;            /* Array with the name for extra symbols in ToPolynomial */
01752     WORD    *doloopstack;         /* To keep track of begin and end of doloops */
01753     WORD    *doloopnest;          /* To keep track of nesting of doloops etc */
01754     char    *CheckpointRunAfter;
01756     char    *CheckpointRunBefore;
01758     WORD    *IfSumCheck;
01759     WORD    *CommuteInSet;        /* groups of noncommuting functions that can commute */
01760     UBYTE   *TestValue;           /* For debugging */
01761 #ifdef PARALLELCODE
01762     LONG    *inputnumbers;
01763     WORD    *pfirstnum;
01764 #endif
01765 #ifdef WITHPTHREADS
01766     pthread_mutex_t halfmodlock;  /* () Lock for adding buffer for halfmod */
01767 #endif
01768     LONG    argstack[MAXNEST];    /* (C) {contents} Stack for nesting of Argument */
01769     LONG    insidestack[MAXNEST]; /* (C) {contents} Stack for Argument or Inside. */
01770     LONG    inexprstack[MAXNEST]; /* (C) {contents} Stack for Argument or Inside. */
01771     LONG    iBufferSize;          /* (C) Size of the input buffer */
01772     LONG    TransEname;           /* (C) Used when a new definition overwrites
01773                                      an old expression. */
01774     LONG    ProcessBucketSize;    /* (C) */
01775     LONG    mProcessBucketSize;   /* (C) */
01776     LONG    CModule;              /* (C) Counter of current module */
01777     LONG    ThreadBucketSize;     /* (C) Roughly the maximum number of input terms */
01778     LONG    CheckpointStamp;
01779     LONG    CheckpointInterval;
01781 #ifdef WITHFLOAT
01782     LONG    DefaultPrecision;     /* (C) Default precision in bits for float_ */
01783     LONG    MaxWeight;            /* (C) Maximum weight for MZV or Euler */
01784     LONG    tDefaultPrecision;    /* (C) Default precision in bits for float_ */
01785     LONG    tMaxWeight;           /* (C) Maximum weight for MZV or Euler */
01786 #endif
01787     int     cbufnum;
01788     int     AutoDeclareFlag;
01790     int     NoShowInput;          /* (C) No listing of input as in .prc, #do */
01791     int     ShortStats;           /* (C) */
01792     int     compiletype;          /* (C) type of statement {DECLARATION etc} */
01793     int     firstconstindex;      /* (C) flag for giving first error message */
01794     int     insidefirst;          /* (C) Not used yet */
01795     int     minsidefirst;         /* (?) Not used yet */
01796     int     wildflag;             /* (C) Flag for marking use of wildcards */
01797     int     NumLabels;            /* (C) Number of labels {in Labels} */
01798     int     MaxLabels;            /* (C) Size of Labels array */
01799     int     lDefDim;              /* (C) */
01800     int     lDefDim4;             /* (C) */
01801     int     NumWildcardNames;     /* (C) Number of ?a variables */
01802     int     WildcardBufferSize;   /* (C) size of WildcardNames buffer */
01803     int     MaxIf;                /* (C) size of IfHeap, IfSumCheck, IfCount */
01804     int     NumStreams;           /* (C) */
01805     int     MaxNumStreams;        /* (C) */
01806     int     firstctypemessage;    /* (C) Flag for giving first st order error */
01807     int     tablecheck;           /* (C) For table checking */
01808     int     idoption;             /* (C) */
01809     int     BottomLevel;          /* (C) For propercount. Not used!!! */
01810     int     CompileLevel;         /* (C) Subexpression level */
01811     int     TokensWriteFlag;      /* (C) */
```

```
01812     int     UnsureDollarMode;      /* (C)?Controls error messages undefined $'s */
01813     int     outsidefun;            /* (C) Used for writing Tables to file */
01814     int     funpowers;             /* (C) */
01815     int     WarnFlag;              /* (C) */
01816     int     StatsFlag;             /* (C) */
01817     int     NamesFlag;             /* (C) */
01818     int     CodesFlag;             /* (C) */
01819     int     SetupFlag;             /* (C) */
01820     int     SortType;              /* (C) */
01821     int     lSortType;             /* (C) */
01822     int     ThreadStats;           /* (C) */
01823     int     FinalStats;            /* (C) */
01824     int     OldParallelStats;      /* (C) */
01825     int     ThreadsFlag;
01826     int     ThreadBalancing;
01827     int     ThreadSortFileSynch;
01828     int     ProcessStats;          /* (C) */
01829     int     BracketNormalize;      /* (C) Indicates whether the bracket st is normalized */
01830     int     maxtermlevel;          /* (C) Size of termstack */
01831     int     dumnumflag;            /* (C) Where there dummy indices in tokenizer? */
01832     int     bracketindexflag;      /* (C) Are brackets going to be indexed? */
01833     int     parallelflag;          /* (C) parallel allowed? */
01834     int     mparallelflag;         /* (C) parallel allowed in this module? */
01835     int     inparallelflag;        /* (C) inparallel allowed? */
01836     int     partodoflag;           /* (C) parallel allowed? */
01837     int     properorderflag;       /* (C) clean normalizing. */
01838     int     vetofilling;           /* (C) vetoes overwriting in tablebase stubs */
01839     int     tablefilling;          /* (C) to notify AddRHS we are filling a table */
01840     int     vetotablebasefill;     /* (C) For the load in tablebase */
01841     int     exprfillwarning;       /* (C) Warning has been printed for expressions in fill statements
      */
01842     int     lhdollarflag;          /* (R) left hand dollar present */
01843     int     NoCompress;            /* (R) Controls native compression */
01844     int     IsFortran90;           /* Tells whether the Fortran is Fortran90 */
01845     int     MultiBracketLevels;    /* Number of elements in MultiBracketBuf */
01846     int     topolynomialflag;      /* To avoid ToPolynomial and FactArg together */
01847     int     ffbufnum;              /* Buffer number for user defined factorizations */
01848     int     OldFactArgFlag;
01849     int     MemDebugFlag;          /* Only used when MALLOCDEBUG in tools.c */
01850     int     OldGCDflag;
01851     int     OldPRFSignFlag;
01852     int     WTimeStatsFlag;
01853     int     SortReallocateFlag;    /* Controls reallocation of large+small buffer at module end.
01854                                       0 : Off
01855                                       1 : On, every module (set by On sortreallocate;)
01856                                       2 : On, single module (set by #sortreallocate) */
01857     int     PrintBacktraceFlag;    /* Print backtrace on terminate? */
01858     int    FlintPolyFlag;          /* Use Flint for polynomial arithmetic */
01859     int     HumanStatsFlag;        /* Print human-readable stats in the stats print? */
01860     int     GrccVerbose;           /* Enable extra print statements in grcc? */
01861     int     SortVerbose;           /* Enable extra sort stats information? */
01862     int     doloopstacksize;
01863     int     doolooplevel;
01864     int     CheckpointFlag;
01868     int     SizeCommuteInSet;      /* Size of the CommuteInSet buffer */
01869 #ifdef PARALLELCODE
01870     int     numpfirstnum;          /* For redefine */
01871     int     sizepfirstnum;         /* For redefine */
01872 #endif
01873     int     origin;                /* Determines whether .sort or ModuleOption */
01874     int     vectorlikeLHS;
01875     int     nummodels;
01876     int     modelspace;
01877     int     ModelLevel;
01878     int     ShortStatsMax;         /* For  On FewerStatistics 10; */
01879     int     InnerTest;             /* For debugging */
01880     WORD    argsumcheck[MAXNEST];  /* (C) Checking of nesting */
01881     WORD    insidesumcheck[MAXNEST];/* (C) Checking of nesting */
01882     WORD    inexprsumcheck[MAXNEST];/* (C) Checking of nesting */
01883     WORD    RepSumCheck[MAXREPEAT];/* (C) Checks nesting of repeat, if, argument */
01884     WORD    lUniTrace[4];          /* (C) */
01885     WORD    RepLevel;              /* (C) Tracks nesting of repeat. */
01886     WORD    arglevel;              /* (C) level of nested argument statements */
01887     WORD    insidelevel;           /* (C) level of nested inside statements */
01888     WORD    inexprlevel;           /* (C) level of nested inexpr statements */
01889     WORD    termlevel;             /* (C) level of nested term statements */
01890     WORD    MustTestTable;         /* (C) Indicates whether tables have been changed */
01891     WORD    DumNum;                /* (C) */
01892     WORD    ncmod;                 /* (C) Local setting of modulus. size of cmod */
01893     WORD    npowmod;               /* (C) Local printing as powers. size powmod */
01894     WORD    modmode;               /* (C) Mode for modulus calculus */
01895     WORD    nhalfmod;              /* relevant word size of halfmod when defined */
01896     WORD    DirtPow;               /* (C) Flag for changes in printing mod powers */
01897     WORD    lUnitTrace;            /* (C) Local value of Tr[1] */
01898     WORD    NwildC;                /* (C) Wildcard counter */
01899     WORD    ComDefer;              /* (C) defer brackets */
01900     WORD    CollectFun;            /* (C) Collect function number */
```

```
01901     WORD    AltCollectFun;          /* (C) Alternate Collect function number */
01902     WORD    OutputMode;             /* (C) */
01903     WORD    Cnumpows;
01904     WORD    OutputSpaces;           /* (C) */
01905     WORD    OutNumberType;          /* (C) Controls RATIONAL/FLOAT output */
01906     WORD    DidClean;               /* (C) Test whether nametree needs cleaning */
01907     WORD    IfLevel;                /* (C) */
01908     WORD    WhileLevel;             /* (C) */
01909     WORD    SwitchLevel;
01910     WORD    SwitchInArray;
01911     WORD    MaxSwitch;
01912     WORD    LogHandle;              /* (C) The Log File */
01913     WORD    LineLength;             /* (C) */
01914     WORD    StoreHandle;            /* (C) Handle of .str file */
01915     WORD    HideLevel;              /* (C) Hiding indicator */
01916     WORD    lPolyFun;               /* (C) local value of PolyFun */
01917     WORD    lPolyFunInv;            /* (C) local value of Inverse of PolyFun */
01918     WORD    lPolyFunType;           /* (C) local value of PolyFunType */
01919     WORD    lPolyFunExp;
01920     WORD    lPolyFunVar;
01921     WORD    lPolyFunPow;
01922     WORD    SymChangeFlag;          /* (C) */
01923     WORD    CollectPercentage;      /* (C) Collect function percentage */
01924     WORD    extrasymbols;           /* Flag for the extra symbols output mode */
01925     WORD    PolyRatFunChanged;      /* Keeps track whether we changed in the compiler */
01926     WORD    ToBeInFactors;
01927 #ifdef WITHMPI
01928     WORD    RhsExprInModuleFlag;    /* (C) Set by the compiler if RHS expressions exists. */
01929 #endif
01930     UBYTE   Commercial[COMMERCIALSIZE+2]; /* (C) Message to be printed in statistics */
01931     UBYTE   debugFlags[MAXFLAGS+2];    /* On/Off Flag number(s) */
01932 };
01933 /*
01934         #] C :
01935         #[ S : The S struct defines objects changed at the start of the run (Processor)
01936                Basically only set by the master.
01937 */
01945 struct S_const {
01946     POSITION MaxExprSize;           /* ( ) Maximum size of in/out/sort */
01947 #ifdef WITHPTHREADS
01948     pthread_mutex_t inputslock;
01949     pthread_mutex_t outputslock;
01950     pthread_mutex_t MaxExprSizeLock;
01951 #endif
01952     POSITION *OldOnFile;            /* (S) File positions of expressions */
01953     WORD    *OldNumFactors;         /* ( ) NumFactors in (old) expression */
01954     WORD    *Oldvflags;             /* ( ) vflags in (old) expression */
01955     WORD    *Olduflags;             /* ( ) uflags in (old) expression */
01956 #ifdef WITHFLOAT
01957     void    *delta_1;
01958 #endif
01959     int     NumOldOnFile;           /* (S) Number of expressions in OldOnFile */
01960     int     NumOldNumFactors;       /* (S) Number of expressions in OldNumFactors */
01961     int     MultiThreaded;          /* (S) Are we running multi-threaded? */
01962 #ifdef WITHPTHREADS
01963     int     MasterSort;             /* Final stage of sorting to the master */
01964 #endif
01965 #ifdef WITHMPI
01966     int     printflag;              /* controls MesPrint() on each slave */
01967 #endif
01968     int     Balancing;              /* For second stage loadbalancing */
01969     WORD    ExecMode;               /* (S) */
01970
01971     WORD    CollectOverFlag;        /* (R) Indicates overflow at Collect */
01972 #ifdef WITHPTHREADS
01973     WORD    sLevel;                 /* Copy of AR0.sLevel because it can get messy */
01974 #endif
01975 };
01976 /*
01977         #] S :
01978         #[ R : The R struct defines objects changed at run time.
01979                They determine the environment that has to be transferred
01980                together with a term during multithreaded execution.
01981 */
01990 struct R_const {
01991     FILEDATA    StoreData;          /* (O) */
01992     FILEHANDLE  Fscr[3];            /* (R) Dollars etc play with it too */
01993     FILEHANDLE  FoStage4[2];        /* (R) In Sort. Stage 4. */
01994     POSITION DefPosition;           /* (R) Deferred position of keep brackets. */
01995     FILEHANDLE *infile;             /* (R) Points alternatingly to Fscr[0] or Fscr[1] */
01996     FILEHANDLE *outfile;            /* (R) Points alternatingly to Fscr[1] or Fscr[0] */
01997     FILEHANDLE *hidefile;           /* (R) Points to Fscr[2] */
01998
01999     WORD    *CompressBuffer;        /* (M) */
02000     WORD    *ComprTop;              /* (M) */
02001     WORD    *CompressPointer;       /* (R) */
02002     COMPAREDUMMY CompareRoutine;
```

```
02003     ULONG   *wranfia;
02004     SBYTE   *moebiustable;
02005
02006     LONG    OldTime;              /* (R) Zero time. Needed in timer. */
02007     LONG    InInBuf;              /* (R) Characters in input buffer. Scratch files. */
02008     LONG    InHiBuf;              /* (R) Characters in hide buffer. Scratch file. */
02009     LONG    pWorkSize;            /* (R) Size of pWorkSpace */
02010     LONG    lWorkSize;            /* (R) Size of lWorkSpace */
02011     LONG    posWorkSize;          /* (R) Size of posWorkSpace */
02012     ULONG   wranfseed;
02013     int     NoCompress;           /* (R) Controls native compression */
02014     int     gzipCompress;         /* (R) Controls gzip compression */
02015     int     Cnumlhs;              /* Local copy of cbuf[rbufnum].numlhs */
02016     int     outtohide;            /* Indicates that output is directly to hide */
02017 #ifdef WITHPTHREADS
02018     int     exprtodo;             /* The expression to do in parallel mode */
02019 #endif
02020     int     wranfcall;
02021     int     wranfnpair1;
02022     int     wranfnpair2;
02023 #if ( BITSINWORD == 32 )
02024     WORD    PrimeList[5000];
02025     WORD    numinprimelist;
02026     WORD    notfirstprime;
02027 #endif
02028     WORD    GetFile;              /* (R) Where to get the terms {like Hide} */
02029     WORD    KeptInHold;           /* (R) */
02030     WORD    BracketOn;            /* (R) Intensely used in poly_ */
02031     WORD    MaxBracket;           /* (R) Size of BrackBuf. Changed by poly_ */
02032     WORD    CurDum;               /* (R) Current maximum dummy number */
02033     WORD    DeferFlag;            /* (R) For deferred brackets */
02034     WORD    TePos;                /* (R) */
02035     WORD    sLevel;               /* (R) Sorting level */
02036     WORD    Stage4Name;           /* (R) Sorting only */
02037     WORD    GetOneFile;           /* (R) Getting from hide or regular */
02038     WORD    PolyFun;              /* (C) Number of the PolyFun function */
02039     WORD    PolyFunInv;           /* (C) Number of the Inverse of the PolyFun function */
02040     WORD    PolyFunType;          /* () value of PolyFunType */
02041     WORD    PolyFunExp;
02042     WORD    PolyFunVar;
02043     WORD    PolyFunPow;
02044     WORD    Eside;                /* () Tells which side of = sign */
02045     WORD    MaxDum;               /* Maximum dummy value in an expression */
02046     WORD    level;                /* Running level in Generator */
02047     WORD    expchanged;           /* (R) Info about expression */
02048     WORD    expflags;             /* (R) Info about expression */
02049     WORD    CurExpr;              /* (S) Number of current expression */
02050     WORD    SortType;             /* A copy of AC.SortType to play with */
02051     WORD    ShortSortCount;       /* For On FewerStatistics 10; */
02052     WORD    modeloptions;
02053     WORD    funoffset;
02054     WORD    moebiustablesize;
02055 };
02056
02057 /*
02058         #] R :
02059         #[ T : These are variables that stay in each thread during multi threaded execution.
02060 */
02069 struct T_const {
02070 #ifdef WITHPTHREADS
02071     SORTBLOCK SB;
02072 #endif
02073     SORTING *S0;                  /* (-) The thread specific sort buffer */
02074     SORTING *SS;                  /* (R) Current sort buffer */
02075     NESTING     Nest;             /* (R) Nesting of function levels etc. */
02076     NESTING     NestStop;         /* (R) */
02077     NESTING     NestPoin;         /* (R) */
02078     WORD    *BrackBuf;            /* (R) Bracket buffer. Used by poly_ at runtime. */
02079     STORECACHE  StoreCache;       /* (R) Cache for picking up stored expr. */
02080     STORECACHE  StoreCacheAlloc;  /* (R) Permanent address of StoreCache to keep valgrind happy */
02081     WORD    **pWorkSpace;         /* (R) Workspace for pointers. Dynamic. */
02082     LONG    *lWorkSpace;          /* (R) WorkSpace for LONG. Dynamic. */
02083     POSITION *posWorkSpace;       /* (R) WorkSpace for file positions */
02084     WORD    *WorkSpace;           /* (M) */
02085     WORD    *WorkTop;             /* (M) */
02086     WORD    *WorkPointer;         /* (R) Pointer in the WorkSpace heap. */
02087     int     *RepCount;            /* (M) Buffer for repeat nesting */
02088     int     *RepTop;              /* (M) Top of RepCount buffer */
02089     WORD    *WildArgTaken;        /* (N) Stack for wildcard pattern matching */
02090     UWORD   *factorials;          /* (T) buffer of factorials. Dynamic. */
02091     WORD    *small_power_n;       /*     length of the number */
02092     UWORD  **small_power;         /*     the number*/
02093     UWORD   *bernoullis;          /* (T) The buffer with Bernoulli numbers. Dynamic. */
02094     WORD    *primelist;
02095     LONG    *pfac;                /* (T) array of positions of factorials. Dynamic. */
02096     LONG    *pBer;                /* (T) array of positions of Bernoulli's. Dynamic. */
02097     WORD    *TMaddr;              /* (R) buffer for TestSub */
```

```
02098     WORD    *WildMask;              /* (N) Wildcard info during pattern matching */
02099     WORD    *previousEfactor;       /* () Cache for factors in expressions */
02100     WORD    **TermMemHeap;         /* For TermMalloc. Set zero in Checkpoint */
02101     UWORD    **NumberMemHeap;       /* For NumberMalloc. Set zero in Checkpoint */
02102     UWORD    **CacheNumberMemHeap; /* For CacheNumberMalloc. Set zero in Checkpoint */
02103     BRACKETINFO *bracketinfo;
02104     WORD    **ListPoly;
02105     WORD    *ListSymbols;
02106     UWORD   *NumMem;
02107     WORD    *TopologiesTerm;
02108     WORD    *TopologiesStart;
02109 #ifdef WITHFLOAT
02110     WORD    *indi1;
02111     WORD    *indi2;
02112     void    *mpf_tab1;
02113     void    *mpf_tab2;
02114     void    *aux_;
02115     void    *auxr_;
02116 #endif
02117     NORMDATA **NormData;
02118     LONG    NormDataSize;
02119     LONG    NormDepth;
02120     PARTI   partitions;
02121     LONG    sBer;                   /* (T) Size of the Bernoullis buffer */
02122     LONG    pWorkPointer;           /* (R) Offset-pointer in pWorkSpace */
02123     LONG    lWorkPointer;           /* (R) Offset-pointer in lWorkSpace */
02124     LONG    posWorkPointer;         /* (R) Offset-pointer in posWorkSpace */
02125     LONG    InNumMem;
02126     int     sfact;                  /* (T) size of the factorials buffer */
02127     int     mfac;                   /* (T) size of the pfac array. */
02128     int     ebufnum;                /* (R) extra compiler buffer */
02129     int     fbufnum;                /* extra compiler buffer for factorization cache */
02130     int     allbufnum;              /* extra compiler buffer for id,all */
02131     int     aebufnum;               /* extra compiler buffer for id,all */
02132     int     idallflag;              /* indicates use of id,all buffers */
02133     int     idallnum;
02134     int     idallmaxnum;
02135     int     WildcardBufferSize;     /* () local copy for updates */
02136 #ifdef WITHPTHREADS
02137     int     identity;               /* () When we work with B->T */
02138     int     LoadBalancing;          /* Needed for synchronization */
02139 #ifdef WITHSORTBOTS
02140     int     SortBotIn1;             /* Input stream 1 for a SortBot */
02141     int     SortBotIn2;             /* Input stream 2 for a SortBot */
02142 #endif
02143 #endif
02144     int     TermMemMax;             /* For TermMalloc. Set zero in Checkpoint */
02145     int     TermMemTop;             /* For TermMalloc. Set zero in Checkpoint */
02146     int     NumberMemMax;           /* For NumberMalloc. Set zero in Checkpoint */
02147     int     NumberMemTop;           /* For NumberMalloc. Set zero in Checkpoint */
02148     int     CacheNumberMemMax;      /* For CacheNumberMalloc. Set zero in Checkpoint */
02149     int     CacheNumberMemTop;      /* For CacheNumberMalloc. Set zero in Checkpoint */
02150     int     bracketindexflag;       /* Are brackets going to be indexed? */
02151     int     optimtimes;             /* Number of the evaluation of the MCTS tree */
02152     int     ListSymbolsSize;
02153     int     NumListSymbols;
02154     int     numpoly;
02155     int     LeaveNegative;
02156     int     TrimPower;              /* Indicates trimming in polyratfun expansion */
02157     WORD    small_power_maxx;       /*     size of the cache for small powers  */
02158     WORD    small_power_maxn;       /*     size of the cache for small powers */
02159     WORD    dummysubexp[SUBEXPSIZE+4]; /* () used in normal.c */
02160     WORD    comsym[8];              /* () Used in tools.c = {8,SYMBOL,4,0,1,1,1,3} */
02161     WORD    comnum[4];              /* () Used in tools.c = { 4,1,1,3 } */
02162     WORD    comfun[FUNHEAD+4];      /* () Used in tools.c = {7,FUNCTION,3,0,1,1,3} */
02163                                     /*            or { 8,FUNCTION,4,0,0,1,1,3 } */
02164     WORD    comind[7];              /* () Used in tools.c = {7,INDEX,3,0,1,1,3} */
02165     WORD    MinVecArg[7+ARGHEAD];   /* (N) but should be more local */
02166     WORD    FunArg[4+ARGHEAD+FUNHEAD]; /* (N) but can be more local */
02167     WORD    locwildvalue[SUBEXPSIZE]; /* () Used in argument.c = {SUBEXPRESSION,SUBEXPSIZE,0,0,0} */
02168     WORD    mulpat[SUBEXPSIZE+5];   /* () Used in argument.c = {TYPEMULT, SUBEXPSIZE+3, 0, */
02169                                     /* SUBEXPRESSION, SUBEXPSIZE, 0, 1, 0, 0, 0 } */
02170     WORD    proexp[SUBEXPSIZE+5];   /* () Used in poly.c */
02171     WORD    TMout[40];              /* (R) Passing info */
02172     WORD    TMbuff;                 /* (R) Communication between TestSub and Genera */
02173     WORD    TMdolfac;               /* factor number for dollar */
02174     WORD    nfac;                   /* (T) Number of highest stored factorial */
02175     WORD    nBer;                   /* (T) Number of highest Bernoulli number. */
02176     WORD    mBer;                   /* (T) Size of buffer pBer. */
02177     WORD    PolyAct;                /* (R) Used for putting the PolyFun at end. ini at 0 */
02178     WORD    RecFlag;                /* (R) Used in TestSub. ini at zero. */
02179     WORD    inprimelist;
02180     WORD    sizeprimelist;
02181     WORD    fromindex;              /* Tells the compare routine whether call from index */
02182     WORD    setinterntopo;          /* Set of internal momenta for topogen */
02183     WORD    setexterntopo;          /* Set of external momenta for topogen */
02184     WORD    TopologiesLevel;
```

```
02185     WORD    TopologiesOptions[2];
02186 #ifdef WITHFLOAT
02187     WORD    FloatPos;
02188     WORD    SortFloatMode;
02189 #endif
02190 };
02191 /*
02192         #] T :
02193         #[ N : The N struct contains variables used in running information
02194                that is inside blocks that should not be split, like pattern
02195                matching, traces etc. They are local for each thread.
02196                They don't need initializations.
02197 */
02206 struct N_const {
02207     POSITION theposition;          /* () Used in index.c */
02208     WORD    *EndNest;              /* (R) Nesting of function levels etc. */
02209     WORD    *Frozen;               /* (R) Bracket info */
02210     WORD    *FullProto;            /* (R) Prototype of a subexpression or table */
02211     WORD    *cTerm;                /* (R) Current term for coef_ and term_ */
02212     int     *RepPoint;             /* (R) Pointer in RepCount buffer. Tracks repeat */
02213     WORD    *WildValue;            /* (N) Wildcard info during pattern matching */
02214     WORD    *WildStop;             /* (N) Wildcard info during pattern matching */
02215     WORD    *argaddress;           /* (N) Used in pattern matching of arguments */
02216     WORD    *RepFunList;           /* (N) For pattern matching */
02217     WORD    *patstop;              /* (N) Used in pattern matching */
02218     WORD    *terstop;              /* (N) Used in pattern matching */
02219     WORD    *terstart;             /* (N) Used in pattern matching */
02220     WORD    *terfirstcomm;         /* (N) Used in pattern matching */
02221     WORD    *DumFound;             /* (N) For renumbering indices {make local?} */
02222     WORD    **DumPlace;            /* (N) For renumbering indices {make local?} */
02223     WORD    **DumFunPlace;         /* (N) For renumbering indices {make local?} */
02224     WORD    *UsedSymbol;           /* (N) When storing terms of a global expr. */
02225     WORD    *UsedVector;           /* (N) When storing terms of a global expr. */
02226     WORD    *UsedIndex;            /* (N) When storing terms of a global expr. */
02227     WORD    *UsedFunction;         /* (N) When storing terms of a global expr. */
02228     WORD    *MaskPointer;          /* (N) For wildcard pattern matching */
02229     WORD    *ForFindOnly;          /* (N) For wildcard pattern matching */
02230     WORD    *findTerm;             /* (N) For wildcard pattern matching */
02231     WORD    *findPattern;          /* (N) For wildcard pattern matching */
02232 #ifdef WITHZLIB
02233     Bytef   **ziobufnum;           /* () Used in compress.c */
02234     Bytef   *ziobuffers;           /* () Used in compress.c */
02235 #endif
02236     WORD    *dummyrenumlist;       /* () Used in execute.c and store.c */
02237     int     *funargs;              /* () Used in lus.c */
02238     WORD    **funlocs;             /* () Used in lus.c */
02239     int     *funinds;              /* () Used in lus.c */
02240     UWORD   *NoScrat2;             /* () Used in normal.c */
02241     WORD    *ReplaceScrat;         /* () Used in normal.c */
02242     TRACES    *tracestack;           /* () used in opera.c */
02243     WORD    *selecttermundo;       /* () Used in pattern.c */
02244     WORD    *patternbuffer;        /* () Used in pattern.c */
02245     WORD    *termbuffer;           /* () Used in pattern.c */
02246     WORD    **PoinScratch;         /* () used in reshuf.c */
02247     WORD    **FunScratch;          /* () used in reshuf.c */
02248     WORD    *RenumScratch;         /* () used in reshuf.c */
02249     FUN_INFO *FunInfo;             /* () Used in smart.c */
02250     WORD    **SplitScratch;        /* () Used in sort.c */
02251     WORD    **SplitScratch1;       /* () Used in sort.c */
02252     SORTING **FunSorts;            /* () Used in sort.c */
02253     UWORD   *SoScratC;             /* () Used in sort.c */
02254     WORD    *listinprint;          /* () Used in proces.c and message.c */
02255     WORD    *currentTerm;          /* () Used in proces.c and message.c */
02256     WORD    **arglist;             /* () Used in function.c */
02257     int     *tlistbuf;             /* () used in lus.c */
02258 #ifdef WHICHSUBEXPRESSION
02259     UWORD   *BinoScrat;            /* () Used in proces.c */
02260 #endif
02261     WORD    *compressSpace;        /* () Used in sort.c */
02262 #ifdef WITHPTHREADS
02263     THREADBUCKET *threadbuck;
02264     EXPRESSIONS expr;
02265 #endif
02266     UWORD   *SHcombi;
02267     WORD    *poly_vars;
02268     UWORD   *cmod;                 /* Local setting of modulus. Pointer to value. */
02269     SHvariables SHvar;
02270     LONG    deferskipped;          /* () Used in proces.c store.c and parallel.c */
02271     LONG    InScratch;             /* () Used in sort.c */
02272     LONG    SplitScratchSize;      /* () Used in sort.c */
02273     LONG    InScratch1;            /* () Used in sort.c */
02274     LONG    SplitScratchSize1;     /* () Used in sort.c */
02275     LONG    ninterms;              /* () Used in proces.c and sort.c */
02276 #ifdef WITHPTHREADS
02277     LONG    inputnumber;           /* () For use in redefine */
02278     LONG    lastinindex;
02279 #endif
```

```
02280 #ifdef WHICHSUBEXPRESSION
02281     LONG    last2;                  /* () Used in proces.c */
02282     LONG    last3;                  /* () Used in proces.c */
02283 #endif
02284     LONG    SHcombisize;
02285     int     NumTotWildArgs;         /* (N) Used in pattern matching */
02286     int     UseFindOnly;            /* (N) Controls pattern routines */
02287     int     UsedOtherFind;          /* (N) Controls pattern routines */
02288     int     ErrorInDollar;          /* (R) */
02289     int     numfargs;               /* () Used in lus.c */
02290     int     numflocs;               /* () Used in lus.c */
02291     int     nargs;                  /* () Used in lus.c */
02292     int     tohunt;                 /* () Used in lus.c */
02293     int     numoffuns;              /* () Used in lus.c */
02294     int     funisize;               /* () Used in lus.c */
02295     int     RSsize;                 /* () Used in normal.c */
02296     int     numtracesctack;         /* () used in opera.c */
02297     int     intracestack;           /* () used in opera.c */
02298     int     numfuninfo;             /* () Used in smart.c */
02299     int     NumFunSorts;            /* () Used in sort.c */
02300     int     MaxFunSorts;            /* () Used in sort.c */
02301     int     arglistsize;            /* () Used in function.c */
02302     int     tlistsize;              /* () used in lus.c */
02303     int     filenum;                /* () used in setfile.c */
02304     int     compressSize;           /* () Used in sort.c */
02305     int     polysortflag;
02306     int     nogroundlevel;          /* () Used to see whether pattern matching at groundlevel */
02307     int     subsubveto;             /* () Sabotage combining subexpressions in TestSub */
02308     WORD    MaxRenumScratch;        /* () used in reshuf.c */
02309     WORD    oldtype;                /* (N) WildCard info at pattern matching */
02310     WORD    oldvalue;               /* (N) WildCard info at pattern matching */
02311     WORD    NumWild;                /* (N) Used in Wildcard */
02312     WORD    RepFunNum;              /* (N) Used in pattern matching */
02313     WORD    DisOrderFlag;           /* (N) Disorder option? Used in pattern matching */
02314     WORD    WildDirt;               /* (N) dirty in wildcard substitution. */
02315     WORD    NumFound;               /* (N) in reshuf only. Local? */
02316     WORD    WildReserve;            /* (N) Used in the wildcards */
02317     WORD    TeInFun;                /* (R) Passing type of action */
02318     WORD    TeSuOut;                /* (R) Passing info. Local? */
02319     WORD    WildArgs;               /* (R) */
02320     WORD    WildEat;                /* (R) */
02321     WORD    PolyNormFlag;           /* (R) For polynomial arithmetic */
02322     WORD    PolyFunTodo;            /* deals with expansions and multiplications */
02323     WORD    sizeselecttermundo;     /* () Used in pattern.c */
02324     WORD    patternbuffersize;      /* () Used in pattern.c */
02325     WORD    numlistinprint;         /* () Used in process.c */
02326     WORD    ncmod;                  /* () used as some type of flag to disable */
02327     WORD    ExpectedSign;
02328     WORD    SignCheck;
02329     WORD    IndDum;                 /* Active dummy indices */
02330     WORD    poly_num_vars;
02331     WORD    idfunctionflag;
02332     WORD    poly_vars_type;         /* type of allocation. For free. */
02333     WORD    tryterm;                /* For EndSort(...,2) */
02334 #ifdef WHICHSUBEXPRESSION
02335     WORD    nbino;                  /* () Used in proces.c */
02336     WORD    last1;                  /* () Used in proces.c */
02337 #endif
02338 };
02339
02340 /*
02341         #] N :
02342         #[ O : The O struct concerns output variables
02343 */
02352 struct O_const {
02353     FILEDATA    SaveData;           /* (O) */
02354     STOREHEADER SaveHeader;         /* ()  System Independent save-Files */
02355     OPTIMIZERESULT OptimizeResult;
02356     UBYTE   *OutputLine;            /* (O) Sits also in debug statements */
02357     UBYTE   *OutStop;               /* (O) Top of OutputLine buffer */
02358     UBYTE   *OutFill;               /* (O) Filling point in OutputLine buffer */
02359     WORD    *bracket;               /* (O) For writing brackets */
02360     WORD    *termbuf;               /* (O) For writing terms */
02361     WORD    *tabstring;
02362     UBYTE   *wpos;                  /* (O) Only when storing file {local?} */
02363     UBYTE   *wpoin;                 /* (O) Only when storing file {local?} */
02364     UBYTE   *DollarOutBuffer;       /* (O) Outputbuffer for Dollars */
02365     UBYTE   *CurBufWrt;             /* (O) Name of currently written expr. */
02366     void    (*FlipWORD)(UBYTE *);   /* ()  Function pointers for translations. Initialized by
      ReadSaveHeader() */
02367     void    (*FlipLONG)(UBYTE *);
02368     void    (*FlipPOS)(UBYTE *);
02369     void    (*FlipPOINTER)(UBYTE *);
02370     void    (*ResizeData)(UBYTE *,int,UBYTE *,int);
02371     void    (*ResizeWORD)(UBYTE *,UBYTE *);
02372     void    (*ResizeNCWORD)(UBYTE *,UBYTE *);
02373     void    (*ResizeLONG)(UBYTE *,UBYTE *);
```

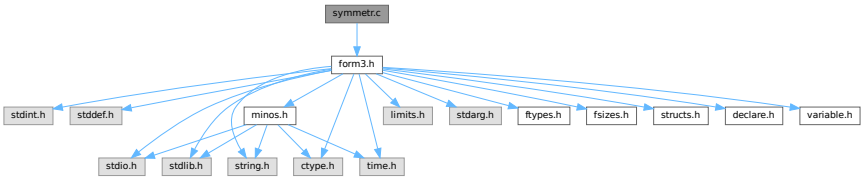
```
02374     void    (*ResizePOS)(UBYTE *,UBYTE *);
02375     void    (*ResizePOINTER)(UBYTE *,UBYTE *);
02376     void    (*CheckPower)(UBYTE *);
02377     void    (*RenumberVec)(UBYTE *);
02378     DICTIONARY **Dictionaries;
02379     UBYTE   *tensorList;           /* Dynamically allocated list with functions that are tensorial. */
02380     WORD    *inscheme;             /* for feeding a Horner scheme to Optimize */
02381 #ifdef WITHFLOAT
02382     UBYTE   *floatspace;
02383     LONG    floatsize;
02384 #endif
02385 /*----Leave NumInBrack as first non-pointer. This is used by the checkpoints--*/
02386     LONG    NumInBrack;            /* (O) For typing [] option in print */
02387     LONG    wlen;                  /* (O) Used to store files. */
02388     LONG    DollarOutSizeBuffer;   /* (O) Size of DollarOutBuffer */
02389     LONG    DollarInOutBuffer;     /* (O) Characters in DollarOutBuffer */
02390 #if defined(mBSD) && defined(MICROTIME)
02391     LONG    wrap;                  /* (O) For statistics time. wrap around */
02392     LONG    wrapnum;               /* (O) For statistics time. wrap around */
02393 #endif
02394     OPTIMIZE Optimize;
02395     int     OutInBuffer;           /* (O) Which routine does the writing */
02396     int     NoSpacesInNumbers;     /*     For very long numbers */
02397     int     BlockSpaces;           /*     For very long numbers */
02398     int     CurrentDictionary;
02399     int     SizeDictionaries;
02400     int     NumDictionaries;
02401     int     CurDictNumbers;
02402     int     CurDictVariables;
02403     int     CurDictSpecials;
02404     int     CurDictFunWithArgs;
02405     int     CurDictNumberWarning;
02406     int     CurDictNotInFunctions;
02407     int     CurDictInDollars;
02408     int     gNumDictionaries;
02409     int     IndentSpace;           /* For indentation in output */
02410 #ifdef WITHFLOAT
02411     int     FloatPrec;
02412 #endif
02413     WORD    schemenum;             /* for feeding a Horner scheme to Optimize */
02414     WORD    transFlag;             /* ()  >0 indicates that translations have to be done */
02415     WORD    powerFlag;             /* ()  >0 indicates that some exponents/powers had to be adjusted
      */
02416     WORD    mpower;                /*     For maxpower adjustment to larger value */
02417     WORD    resizeFlag;            /* ()  >0 indicates that something went wrong when resizing words
      */
02418     WORD    bufferedInd;           /* ()  Contains extra INDEXENTRIES, see ReadSaveIndex() for an
     explanation */
02419     WORD    OutSkip;               /* (O) How many chars to skip in output line */
02420     WORD    IsBracket;             /* (O) Controls brackets */
02421     WORD    InFbrack;              /* (O) For writing only */
02422     WORD    PrintType;             /* (O) */
02423     WORD    FortFirst;             /* (O) Only in sch.c */
02424     WORD    DoubleFlag;            /* (O) Output in double precision */
02425     WORD    FactorMode;            /* When the output should be written as factors */
02426     WORD    FactorNum;             /* Number of factor currently treated */
02427     WORD    ErrorBlock;
02428     WORD    OptimizationLevel;     /* Level of optimization in the output */
02429     UBYTE   FortDotChar;           /* (O) */
02430 /*
02431     For the padding, please count also the number of int's in the OPTIMIZE struct.
02432 */
02433 };
02434 /*
02435         #] O :
02436         #[ X : The X struct contains variables that deal with the external channel
02437 */
02445 struct X_const {
02446     UBYTE *currentPrompt;
02447     UBYTE *shellname;          /* if !=NULL (default is "/bin/sh -c"), start in
02448                                   the specified subshell*/
02449     UBYTE *stderrname;         /* If !=NULL (default if "/dev/null"), stderr is
02450                                   redirected to the specified file*/
02451     int timeout;               /* timeout to initialize preset channels.
02452                                   If timeout<0, the preset channels are
02453                                   already initialized*/
02454     int killSignal;            /* signal number, SIGKILL by default*/
02455     int killWholeGroup;        /* if 0, the signal is sent only to a process,
02456                                   if !=0 (default) is sent to a whole process group*/
02457     int daemonize;             /* if !=0 (default), start in a daemon mode */
02458     int currentExternalChannel;
02459 };
02460 /*
02461         #] X :
02462         #[ Definitions :
02463
02464     Note: we changed the definition from C to Cc in version 5.
```

```
02465     The reason is that everywhere the pointer to the compiler
02466     buffer was called C as well. Somehow that gave no problems but
02467     who knows what the future might bring.
02468 */
02469
02470 #ifdef WITHPTHREADS
02471
02477 typedef struct AllGlobals {
02478     struct M_const M;
02479     struct C_const Cc;
02480     struct S_const S;
02481     struct O_const O;
02482     struct P_const P;
02483     struct X_const X;
02484 } ALLGLOBALS;
02485
02491 typedef struct AllPrivates {
02492     struct R_const R;
02493     struct N_const N;
02494     struct T_const T;
02495 } ALLPRIVATES;
02496
02497 #else
02498
02503 typedef struct AllGlobals {
02504     struct M_const M;
02505     struct C_const Cc;
02506     struct S_const S;
02507     struct R_const R;
02508     struct N_const N;
02509     struct O_const O;
02510     struct P_const P;
02511     struct T_const T;
02512     struct X_const X;
02513 } ALLGLOBALS;
02514
02515 #endif
02516
02517 /*
02518         #] Definitions :
02519     #] A :
02520     #[ FG :
02521 */
02522
02523 #ifdef WITHPTHREADS
02524 #define PHEAD  ALLPRIVATES *B,
02525 #define PHEAD0 ALLPRIVATES *B
02526 #define BHEAD  B,
02527 #define BHEAD0 B
02528 #else
02529 #define PHEAD
02530 #define PHEAD0 void
02531 #define BHEAD
02532 #define BHEAD0
02533 #endif
02534
02535 typedef int (*WCN)(PHEAD WORD *,WORD *,WORD,WORD);
02536 typedef int (*WCN2)(PHEAD WORD *,WORD *);
02537
02538 typedef WORD (*COMPARE)(PHEAD WORD *,WORD *,WORD);
02539
02551 typedef struct FixedGlobals {
02552     WCN     Operation[8];
02553     WCN2    OperaFind[6];
02554     char    *VarType[10];
02555     char    *ExprStat[21];
02556     char    *FunNam[2];
02557     char    *swmes[3];
02558     char    *fname;
02559     char    *fname2;
02560     UBYTE   *s_one;
02561     WORD    fnamebase;
02562     WORD    fname2base;
02563     WORD    fnamesize;
02564     WORD    fname2size;
02565     UINT    cTable[256];
02566 } FIXEDGLOBALS;
02567
02568 /*
02569     #] FG :
02570 */
02571
02572 #endif
```

4.141 symmetr.c File Reference

```
#include "form3.h"
```

Include dependency graph for symmetr.c:



Functions

- int MatchE (PHEAD WORD *pattern, WORD *fun, WORD *inter, WORD par)
- int Permute (PERM *perm, WORD first)
- int PermuteP (PERMP *perm, WORD first)
- int Distribute (DISTRIBUTE *d, WORD first)
- int MatchCy (PHEAD WORD *pattern, WORD *fun, WORD *inter, WORD par)
- int FunMatchCy (PHEAD WORD *pattern, WORD *fun, WORD *inter, WORD par)
- int FunMatchSy (PHEAD WORD *pattern, WORD *fun, WORD *inter, WORD par)
- int MatchArgument (PHEAD WORD *arg, WORD *pat)

4.141.1 Detailed Description

The routines that deal with the pattern matching of functions with symmetric properties.

Definition in file symmetr.c.

4.141.2 Function Documentation

4.141.2.1 MatchE()

```
int MatchE (
            PHEAD WORD * pattern,
            WORD * fun,
            WORD * inter,
            WORD par )
```

Definition at line 56 of file symmetr.c.

4.141.2.2 Permute()

```
int Permute (
            PERM * perm,
            WORD first )
```

Definition at line 444 of file symmetr.c.

4.141.2.3 PermuteP()

```
int PermuteP (
              PERMP * perm,
              WORD first )
```

Definition at line 478 of file symmetr.c.

4.141.2.4 Distribute()

```
int Distribute (
              DISTRIBUTE * d,
              WORD first )
```

Definition at line 510 of file symmetr.c.

4.141.2.5 MatchCy()

```
int MatchCy (
              PHEAD WORD * pattern,
              WORD * fun,
              WORD * inter,
              WORD par )
```

Definition at line 571 of file symmetr.c.

4.141.2.6 FunMatchCy()

```
int FunMatchCy (
              PHEAD WORD * pattern,
              WORD * fun,
              WORD * inter,
              WORD par )
```

Definition at line 1067 of file symmetr.c.

4.141.2.7 FunMatchSy()

```
int FunMatchSy (
              PHEAD WORD * pattern,
              WORD * fun,
              WORD * inter,
              WORD par )
```

Definition at line 1525 of file symmetr.c.

4.141.2.8 MatchArgument()

```
int MatchArgument (
              PHEAD WORD * arg,
              WORD * pat )
```

Definition at line 2052 of file symmetr.c.

4.142 symmetr.c

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032 /*
00033     #[ Includes : function.c
00034 */
00035
00036 #include "form3.h"
00037
00038 /*
00039     #] Includes :
00040     #[ MatchE :                 WORD MatchE(pattern,fun,inter,par)
00041
00042         Matches symmetric and antisymmetric tensors.
00043         Pattern and fun point at a tensor.
00044         Problem is the wildcarding and all its possible permutations.
00045         This routine loops over all of them and calls for each
00046         possible wildcarding the recursion in ScanFunctions.
00047         Note that this can be very costly.
00048
00049         Originally this routine did only Levi Civita tensors and hence
00050         it dealt only with commuting objects.
00051         Because of the backtracking we cannot fall back to the calling
00052         ScanFunctions routine and check the sequence of functions when
00053         non-commuting objects are involved.
00054 */
00055
00056 int MatchE(PHEAD WORD *pattern, WORD *fun, WORD *inter, WORD par)
00057 {
00058     GETBIDENTITY
00059     WORD *m, *t, *r, i;
00060     int retval;
00061     WORD *mstop, *tstop, j, newvalue, newfun;
00062     WORD fixvec[MAXMATCH],wcvec[MAXMATCH],fixind[MAXMATCH],wcind[MAXMATCH];
00063     WORD tfixvec[MAXMATCH],tfixind[MAXMATCH];
00064     WORD vwc,vfix,ifix,iwc,tvfix,tifix,nv,ni;
00065     WORD sign = 0, *rstop, first1, first2, first3, funwild;
00066     WORD *OldWork, nwstore, oRepFunNum;
00067     PERM perm1,perm2;
00068     DISTRIBUTE distr;
00069     WORD *newpat, /* *newter, *instart, */ offset;
00070 /*  instart = fun; */
```

```
00071     offset = WORDDIF(fun,AN.terstart);
00072     if ( pattern[1] != fun[1] ) return(0);
00073     if ( *pattern >= FUNCTION+WILDOFFSET ) {
00074         if ( CheckWild(BHEAD *pattern-WILDOFFSET,FUNTOFUN,*fun,&newfun) ) return(0);
00075         funwild = 1;
00076     }
00077     else funwild = 0;
00078     mstop = pattern + pattern[1];
00079     tstop = fun + fun[1];
00080     m = pattern + FUNHEAD;
00081     t = fun + FUNHEAD;
00082     while ( m < mstop ) {
00083         if ( *m != *t ) break;
00084         m++; t++;
00085     }
00086     if ( m >= mstop ) {
00087         AN.RepFunList[AN.RepFunNum++] = offset;
00088         AN.RepFunList[AN.RepFunNum++] = 0;
00089         newpat = pattern + pattern[1];
00090         if ( funwild ) {
00091             m = AN.WildValue;
00092             t = OldWork = AT.WorkPointer;
00093             nwstore = i = (m[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
00094             r = AT.WildMask;
00095             if ( i > 0 ) {
00096                 do {
00097                     *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
00098                 } while ( --i > 0 );
00099             }
00100             if ( t >= AT.WorkTop ) {
00101                 MLOCK(ErrorMessageLock);
00102                 MesWork();
00103                 MUNLOCK(ErrorMessageLock);
00104                 return(-1);
00105             }
00106             AT.WorkPointer = t;
00107             AddWild(BHEAD *pattern-WILDOFFSET,FUNTOFUN,newfun);
00108             if ( newpat >= AN.patstop ) {
00109                 if ( AN.UseFindOnly == 0 ) {
00110                     if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
00111                         AN.UsedOtherFind = 1;
00112                         return(1);
00113                     }
00114                     retval = 0;
00115                 }
00116                 else return(1);
00117             }
00118             else {
00119 /*              newter = instart; */
00120                 retval = ScanFunctions(BHEAD newpat,inter,par);
00121             }
00122             if ( retval == 0 ) {
00123                 m = AN.WildValue;
00124                 t = OldWork; r = AT.WildMask; i = nwstore;
00125                 if ( i > 0 ) {
00126                     do {
00127                         *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
00128                     } while ( --i > 0 );
00129                 }
00130             }
00131             AT.WorkPointer = OldWork;
00132             return(retval);
00133         }
00134         else {
00135             if ( newpat >= AN.patstop ) {
00136                 if ( AN.UseFindOnly == 0 ) {
00137                     if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
00138                         AN.UsedOtherFind = 1;
00139                         return(1);
00140                     }
00141                     else return(0);
00142                 }
00143                 else return(1);
00144             }
00145 /*          newter = instart; */
00146             retval = ScanFunctions(BHEAD newpat,inter,par);
00147             return(retval);
00148         }
00149 /*
00150         Now the recursion
00151 */
00152     }
00153 /*
00154     Strategy:
00155     1: match the fixed arguments
00156     2: match, permuting the wildcards if needed.
00157     3: keep track of sign.
```

```
00158 */
00159     vwc = 0;
00160     vfix = 0;
00161     ifix = 0;
00162     iwc = 0;
00163     r = pattern+FUNHEAD;
00164     while ( r < mstop ) {
00165         if ( *r < (AM.OffsetVector+WILDOFFSET) ) {
00166             fixvec[vfix++] = *r;            /* Fixed vectors */
00167             sign += vwc + ifix + iwc;
00168         }
00169         else if ( *r < MINSPEC ) {
00170             wcvec[vwc++] = *r;              /* Wildcard vectors */
00171             sign += ifix + iwc;
00172         }
00173         else if ( *r < (AM.OffsetIndex+WILDOFFSET) ) {
00174             fixind[ifix++] = *r;            /* Fixed indices */
00175             sign += iwc;
00176         }
00177         else if ( *r < (AM.OffsetIndex+(WILDOFFSET«1)) ) {
00178             wcind[iwc++] = *r;              /* Wildcard indices */
00179         }
00180         else {
00181             fixind[ifix++] = *r;            /* Generated indices ~ fixed */
00182             sign += iwc;
00183         }
00184         r++;
00185     }
00186     if ( iwc == 0 && vwc == 0 ) return(0);
00187     tvfix = tifix = 0;
00188     t = fun + FUNHEAD;
00189     m = fixvec;
00190     mstop = m + vfix;
00191     r = fixind;
00192     rstop = r + ifix;
00193     nv = 0; ni = 0;
00194     while ( t < tstop ) {
00195         if ( *t < 0 ) {
00196             nv++;
00197             if ( m < mstop && *t == *m ) {
00198                 m++;
00199             }
00200             else {
00201                 sign += WORDDIF(mstop,m);
00202                 tfixvec[tvfix++] = *t;
00203             }
00204         }
00205         else {
00206             ni++;
00207             if ( r < rstop && *r == *t ) {
00208                 r++;
00209             }
00210             else {
00211                 sign += WORDDIF(rstop,r);
00212                 tfixind[tifix++] = *t;
00213             }
00214         }
00215         t++;
00216     }
00217     if ( m < mstop || r < rstop ) return(0);
00218     if ( tvfix < vwc || (tvfix+tifix) < (vwc+iwc) ) return(0);
00219     sign += ( nv - vfix - vwc ) & ni;
00220 /*
00221     Take now the wildcards that have an assignment already.
00222     See whether they match.
00223 */
00224     {
00225         WORD *wv, *wm, n;
00226         wm = AT.WildMask;
00227         wv = AN.WildValue;
00228         n = AN.NumWild;
00229         do {
00230             if ( *wm ) {
00231                 if ( *wv == VECTOVEC ) {
00232                     for ( ni = 0; ni < vwc; ni++ ) {
00233                         if ( wcvec[ni]-WILDOFFSET == wv[2] ) {  /* Has been assigned */
00234                             sign += ni;
00235                             vwc--;
00236                             while ( ni < vwc ) {
00237                                 wcvec[ni] = wcvec[ni+1];
00238                                 ni++;
00239                             }
00240 /* TryVect: */
00241                             for ( ni = 0; ni < tvfix; ni++ ) {
00242                                 if ( tfixvec[ni] == wv[3] ) {
00243                                     sign += ni;
00244                                     tvfix--;
```

```
00245                                   while ( ni < tvfix ) {
00246                                       tfixvec[ni] = tfixvec[ni+1];
00247                                       ni++;
00248                                   }
00249                                   goto NextWV;
00250                               }
00251                           }
00252                           return(0);
00253                       }
00254                   }
00255               }
00256               else if ( *wv == INDTOIND ) {
00257                   for ( ni = 0; ni < iwc; ni++ ) {
00258                       if ( wcind[ni]-WILDOFFSET == wv[2] ) {  /* Has been assigned */
00259                           sign += ni;
00260                           iwc--;
00261                           while ( ni < iwc ) {
00262                               wcind[ni] = wcind[ni+1];
00263                               ni++;
00264                           }
00265                           for ( ni = 0; ni < tifix; ni++ ) {
00266                               if ( tfixind[ni] == wv[3] ) {
00267                                   sign += ni;
00268                                   tifix--;
00269                                   while ( ni < tifix ) {
00270                                       tfixind[ni] = tfixind[ni+1];
00271                                       ni++;
00272                                   }
00273                                   goto NextWV;
00274                               }
00275                           }
00276 /*                        goto TryVect; */
00277                           return(0);
00278
00279                       }
00280                   }
00281               }
00282               else if ( *wv == VECTOSUB ) {
00283                   for ( ni = 0; ni < vwc; ni++ ) {
00284                       if ( wcvec[ni]-WILDOFFSET == wv[2] ) return(0);
00285                   }
00286               }
00287               else if ( *wv == INDTOSUB ) {
00288                   for ( ni = 0; ni < iwc; ni++ ) {
00289                       if ( wcind[ni]-WILDOFFSET == wv[2] ) return(0);
00290                   }
00291               }
00292           }
00293 NextWV:
00294           wm++;
00295           wv += wv[1];
00296           n--;
00297           if ( n > 0 ) {
00298               while ( n > 0 && ( *wv == FROMSET || *wv == SETTONUM
00299               || *wv == LOADDOLLAR ) ) { wv += wv[1]; wm++; n--; }
00300 /*
00301               Freak problem: doesn't test for n and ran into a remaining
00302               code equal to SETTONUM followed by a big number and then
00303               ran out of the memory.
00304
00305               while ( *wv == FROMSET || *wv == SETTONUM
00306               || ( *wv == LOADDOLLAR && n > 0 ) ) { wv += wv[1]; wm++; n--; }
00307 */
00308           }
00309       } while ( n > 0 );
00310   }
00311 /*
00312   Now there are only free wildcards left.
00313   Possibly the assigned values ate too many vectors.
00314   The rest has to be done the 'hard way' via permutations.
00315   This is too bad when there are 10 indices.
00316   This could cause 10! tries.
00317   We try to avoid the worst case by using a very special
00318   (somewhat slow) permutation routine that has as its worst
00319   cases some rather unlikely configurations, rather than some
00320   common ones (as would have been the case with the conventional
00321   permutation routine).
00322       assume:
00323       vvvvvvvvvvvviiiiiii     (tvfix in tfixvec and tifix in tfixind)
00324       VVVVVVVVVIIIIIIIII      (vwc in wcvec and iwc in wcind)
00325   Note: all further assignments are possible at this point!
00326   Strategy:
00327       permute v
00328       permute i
00329       loop over the ordered distribution of the leftover v's
00330       through the i's.
00331 */
```

```
00332     if ( tvfix < vwc ) { return(0); }
00333     perm1.n = tvfix;
00334     perm1.sign = 0;
00335     perm1.objects = tfixvec;
00336     perm2.n = tifix;
00337     perm2.sign = 0;
00338     perm2.objects = tfixind;
00339     distr.n1 = tvfix - vwc;
00340     distr.n2 = tifix;
00341     distr.obj1 = tfixvec + vwc;
00342     distr.obj2 = tfixind;
00343     distr.out = fixvec;             /* For scratch */
00344     first1 = 1;
00345 /*
00346             Store the current Wildcard assignments
00347 */
00348     m = AN.WildValue;
00349     t = OldWork = AT.WorkPointer;
00350     nwstore = i = (m[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
00351     r = AT.WildMask;
00352     if ( i > 0 ) {
00353         do {
00354             *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
00355         } while ( --i > 0 );
00356     }
00357     if ( t >= AT.WorkTop ) {
00358         MLOCK(ErrorMessageLock);
00359         MesWork();
00360         MUNLOCK(ErrorMessageLock);
00361         return(-1);
00362     }
00363     AT.WorkPointer = t;
00364     while ( (first1 = Permute(&perm1,first1) ) == 0 ) {
00365         first2 = 1;
00366         while ( (first2 = Permute(&perm2,first2) ) == 0 ) {
00367             first3 = 1;
00368             while ( (first3 = Distribute(&distr,first3) ) == 0 ) {
00369 /*
00370                 Make now the wildcard assignments
00371 */
00372                 for ( i = 0; i < vwc; i++ ) {
00373                     j = wcvec[i] - WILDOFFSET;
00374                     if ( CheckWild(BHEAD j,VECTOVEC,tfixvec[i],&newvalue) )
00375                         goto NoCaseB;
00376                     AddWild(BHEAD j,VECTOVEC,newvalue);
00377                 }
00378                 for ( i = 0; i < iwc; i++ ) {
00379                     j = wcind[i] - WILDOFFSET;
00380                     if ( CheckWild(BHEAD j,INDTOIND,fixvec[i],&newvalue) )
00381                         goto NoCaseB;
00382                     AddWild(BHEAD j,INDTOIND,newvalue);
00383                 }
00384 /*
00385                 Go into the recursion
00386 */
00387                 oRepFunNum = AN.RepFunNum;
00388                 AN.RepFunList[AN.RepFunNum++] = offset;
00389                 AN.RepFunList[AN.RepFunNum++] =
00390                     ( perm1.sign + perm2.sign + distr.sign + sign ) & 1;
00391                 newpat = pattern + pattern[1];
00392                 if ( funwild ) AddWild(BHEAD *pattern-WILDOFFSET,FUNTOFUN,newfun);
00393                 if ( newpat >= AN.patstop ) {
00394                     if ( AN.UseFindOnly == 0 ) {
00395                         if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
00396                             AN.UsedOtherFind = 1;
00397                             return(1);
00398                         }
00399                     }
00400                     else return(1);
00401                 }
00402                 else {
00403 /*                  newter = instart; */
00404                     if ( ScanFunctions(BHEAD newpat,inter,par) ) { return(1); }
00405                 }
00406 /*
00407                 Restore the old Wildcard assignments
00408 */
00409                 AN.RepFunNum = oRepFunNum;
00410 NoCaseB:        m = AN.WildValue;
00411                 t = OldWork; r = AT.WildMask; i = nwstore;
00412                 if ( i > 0 ) {
00413                     do {
00414                         *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
00415                     } while ( --i > 0 );
00416                 }
00417                 AT.WorkPointer = t;
00418             }
```

```
00419         }
00420     }
00421     AT.WorkPointer = OldWork;
00422     return(0);
00423 }
00424
00425 /*
00426     #] MatchE :
00427     #[ Permute :                WORD Permute(perm,first)
00428
00429         Special permutation function.
00430         Works recursively.
00431         The aim is to cycle through in as fast a way as possible,
00432         to take care that each object hits the various positions
00433         already early in the game.
00434
00435         Start at two: -> cycle of two
00436         then three -> cycle of three
00437         etc;
00438         The innermost cycle is the longest. This is the opposite
00439         of the usual way of generating permutations and it is
00440         certainly not the fastest one. It allows for the fastest
00441         hit in the assignment of wildcards though.
00442 */
00443
00444 int Permute(PERM *perm, WORD first)
00445 {
00446     WORD *s, c, i, j;
00447     if ( first ) {
00448         perm->sign = ( perm->sign <= 1 ) ? 0: 1;
00449         for ( i = 0; i < perm->n; i++ ) perm->cycle[i] = 0;
00450         return(0);
00451     }
00452     i = perm->n;
00453     while ( --i > 0 ) {
00454         s = perm->objects;
00455         c = s[0];
00456         j = i;
00457         while ( --j >= 0 ) { *s = s[1]; s++; }
00458         *s = c;
00459         if ( ( i & 1 ) != 0 ) perm->sign ^= 1;
00460         if ( perm->cycle[i] < i ) {
00461             (perm->cycle[i])++;
00462             return(0);
00463         }
00464         else {
00465             perm->cycle[i] = 0;
00466         }
00467     }
00468     return(1);
00469 }
00470
00471 /*
00472     #] Permute :
00473     #[ PermuteP :               WORD PermuteP(perm,first)
00474
00475     Like Permute, but works on an array of pointers
00476 */
00477
00478 int PermuteP(PERMP *perm, WORD first)
00479 {
00480     WORD **s, *c, i, j;
00481     if ( first ) {
00482         perm->sign = ( perm->sign <= 1 ) ? 0: 1;
00483         for ( i = 0; i < perm->n; i++ ) perm->cycle[i] = 0;
00484         return(0);
00485     }
00486     i = perm->n;
00487     while ( --i > 0 ) {
00488         s = perm->objects;
00489         c = s[0];
00490         j = i;
00491         while ( --j >= 0 ) { *s = s[1]; s++; }
00492         *s = c;
00493         if ( ( i & 1 ) != 0 ) perm->sign ^= 1;
00494         if ( perm->cycle[i] < i ) {
00495             (perm->cycle[i])++;
00496             return(0);
00497         }
00498         else {
00499             perm->cycle[i] = 0;
00500         }
00501     }
00502     return(1);
00503 }
00504
00505 /*
```

```
00506     #] PermuteP :
00507     #[ Distribute :
00508 */
00509
00510 int Distribute(DISTRIBUTE *d, WORD first)
00511 {
00512     WORD *to, *from, *inc, *from2, i, j;
00513     if ( first ) {
00514         d->n = d->n1 + d->n2;
00515         to = d->out;
00516         from = d->obj2;
00517         for ( i = 0; i < d->n2; i++ ) {
00518             d->cycle[i] = 0;
00519             *to++ = *from++;
00520         }
00521         from = d->obj1;
00522         while ( i < d->n ) {
00523             d->cycle[i++] = 1;
00524             *to++ = *from++;
00525         }
00526         d->sign = 0;
00527         return(0);
00528     }
00529     if ( d->n1 == 0 || d->n2 == 0 ) return(1);
00530     j = 0;
00531     i = 0;
00532     inc = d->cycle;
00533     from = inc + d->n;
00534     while ( *inc ) { j++; inc++; }
00535     while ( inc < from && !*inc ) { i++; inc++; }
00536     if ( inc >= from ) return(1);
00537     d->sign ^= ((i&j)-j+1) & 1;
00538     *inc = 0;
00539     *--inc = 1;
00540     while ( --j >= 0 ) *--inc = 1;
00541     while ( --i > 0 ) *--inc = 0;
00542     to = d->out;
00543     from = d->obj1;
00544     from2 = d->obj2;
00545     for ( i = 0; i < d->n; i++ ) {
00546         if ( *inc++ ) {
00547             *to++ = *from++;
00548         }
00549         else {
00550             *to++ = *from2++;
00551         }
00552     }
00553     return(0);
00554 }
00555
00556 /*
00557     #] Distribute :
00558     #[ MatchCy :
00559
00560         Matching of (r)cyclic tensors.
00561         Parameters like in MatchE.
00562         The structure of the routine is much simpler, because the number
00563         of possibilities is much more limited.
00564         The major complication is the ?a-type wildcards
00565         We need a strategy for T(i1?,?a,i1?,?b). Which is the shorter
00566         match: ?a or ?b ? (if possible of course)
00567         This is also relevant in the case of the shortest match if there
00568         is more than one choice for i1.
00569 */
00570
00571 int MatchCy(PHEAD WORD *pattern, WORD *fun, WORD *inter, WORD par)
00572 {
00573     GETBIDENTITY
00574     WORD *t, *tstop, *p, *pstop, *m, *r, *oldworkpointer = AT.WorkPointer;
00575     WORD *thewildcards, *multiplicity, *renum, wc, newvalue, oldwilval = 0;
00576     WORD *params, *lowlevel = 0;
00577     int argcount = 0, funnycount = 0, tcount = fun[1] - FUNHEAD;
00578     int type = 0, pnum, i, j, k, nwstore, iraise, itop, sumeat;
00579     CBUF *C = cbuf+AT.ebufnum;
00580     int ntwa = 3*AN.NumTotWildArgs+1;
00581     LONG oldcpointer = C->Pointer - C->Buffer;
00582     WORD offset = fun-AN.terstart, *newpat;
00583
00584     if ( (functions[fun[0]-FUNCTION].symmetric & ~REVERSEORDER) == RCYCLESYMMETRIC ) type = 1;
00585     pnum = pattern[0];
00586     nwstore = (AN.WildValue[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
00587     if ( pnum > FUNCTION + WILDOFFSET ) {
00588         pnum -= WILDOFFSET;
00589         if ( CheckWild(BHEAD pnum,FUNTOFUN,fun[0],&newvalue) ) return(0);
00590         oldwilval = 1;
00591         t = lowlevel = AT.WorkPointer;
00592         m = AN.WildValue;
```

```
00593         i = nwstore;
00594         r = AT.WildMask;
00595         if ( i > 0 ) {
00596             do {
00597                 *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
00598             } while ( --i > 0 );
00599         }
00600         *t++ = C->numrhs;
00601         if ( t >= AT.WorkTop ) {
00602             MLOCK(ErrorMessageLock);
00603             MesWork();
00604             MUNLOCK(ErrorMessageLock);
00605             return(-1);
00606         }
00607         AT.WorkPointer = t;
00608         AddWild(BHEAD pnum,FUNTOFUN,newvalue);
00609     }
00610     if ( (functions[pnum-FUNCTION].symmetric & ~REVERSEORDER) == RCYCLESYMMETRIC ) type = 1;
00611 
00612     /* First we have to make an inventory. Are there FUNNYWILD pointers? */
00613 
00614     p = pattern + FUNHEAD;
00615     pstop = pattern + pattern[1];
00616     while ( p < pstop ) {
00617         if ( *p == FUNNYWILD ) { p += 2; funnycount++; }
00618         else { p++; argcount++; }
00619     }
00620     if ( argcount > tcount ) goto NoSuccess;
00621     if ( argcount < tcount && funnycount == 0 ) goto NoSuccess;
00622     if ( argcount == 0 && tcount == 0 && funnycount == 0 ) {
00623         AN.RepFunList[AN.RepFunNum++] = offset;
00624         AN.RepFunList[AN.RepFunNum++] = 0;
00625         newpat = pattern + pattern[1];
00626         if ( newpat >= AN.patstop ) {
00627             if ( AN.UseFindOnly == 0 ) {
00628                 if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
00629                     AT.WorkPointer = oldworkpointer;
00630                     AN.UsedOtherFind = 1;
00631                     return(1);
00632                 }
00633                 j = 0;
00634             }
00635             else {
00636                 AT.WorkPointer = oldworkpointer;
00637                 return(1);
00638             }
00639         }
00640         else j = ScanFunctions(BHEAD newpat,inter,par);
00641         if ( j ) return(j);
00642         goto NoSuccess;
00643     }
00644     tstop = fun + fun[1];
00645 
00646     /* Store the wildcard assignments */
00647 
00648     params = AT.WorkPointer;
00649     thewildcards = t = params + tcount;
00650     t += ntwa;
00651     if ( oldwilval ) lowlevel = oldworkpointer;
00652     else lowlevel = t;
00653     m = AN.WildValue;
00654     i = nwstore;
00655     if ( i > 0 ) {
00656         r = AT.WildMask;
00657         do {
00658             *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
00659         } while ( --i > 0 );
00660         *t++ = C->numrhs;
00661     }
00662     if ( t >= AT.WorkTop ) {
00663         MLOCK(ErrorMessageLock);
00664         MesWork();
00665         MUNLOCK(ErrorMessageLock);
00666         return(-1);
00667     }
00668     AT.WorkPointer = t;
00669 /*
00670     #[ Case 1: no funnies or all funnies must be empty. We just cycle through.
00671 */
00672     if ( argcount == tcount ) {
00673         if ( funnycount > 0 ) { /* Test all funnies first */
00674             p = pattern + FUNHEAD;
00675             t = fun + FUNHEAD;
00676             while ( p < pstop ) {
00677                 if ( *p != FUNNYWILD ) { p++; continue; }
00678                 AN.argaddress = t;
00679                 if ( CheckWild(BHEAD p[1],ARGTOARG,0,t) ) goto nomatch;
```

```
00680                 AddWild(BHEAD p[1],ARGTOARG,0);
00681                 p += 2;
00682             }
00683             oldwilval = 1;
00684         }
00685         for ( k = 0; k <= type; k++ ) {
00686             if ( k == 0 ) {
00687                 p = params; t = fun + FUNHEAD;
00688                 while ( t < tstop ) *p++ = *t++;
00689             }
00690             else {
00691                 p = params+tcount; t = fun + FUNHEAD;
00692                 while ( t < tstop ) *--p = *t++;
00693             }
00694             for ( i = 0; i < tcount; i++ ) { /* The various cycles */
00695                 p = pattern + FUNHEAD;
00696                 wc = 0;
00697                 for ( j = 0; j < tcount; j++, p++ ) { /* The arguments */
00698                     while ( *p == FUNNYWILD ) p += 2;
00699                     t = params + (i+j)%tcount;
00700                     if ( *t == *p ) continue;
00701                     if ( *p >= AM.OffsetIndex + WILDOFFSET
00702                     && *p < AM.OffsetIndex + 2*WILDOFFSET ) {
00703
00704                         /* Test wildcard index */
00705
00706                         wc = *p - WILDOFFSET;
00707                         if ( CheckWild(BHEAD wc,INDTOIND,*t,&newvalue) ) break;
00708                         AddWild(BHEAD wc,INDTOIND,newvalue);
00709                     }
00710                     else if ( *t < MINSPEC && p[j] < MINSPEC
00711                         && *p >= AM.OffsetVector + WILDOFFSET ) {
00712
00713                         /* Test wildcard vector */
00714
00715                         wc = *p - WILDOFFSET;
00716                         if ( CheckWild(BHEAD wc,VECTOVEC,*t,&newvalue) ) break;
00717                         AddWild(BHEAD wc,VECTOVEC,newvalue);
00718                     }
00719                     else break;
00720                 }
00721                 if ( j >= tcount ) { /* Match! */
00722
00723                     /* Continue with other functions. Make sure of the funnies */
00724
00725                     AN.RepFunList[AN.RepFunNum++] = offset;
00726                     AN.RepFunList[AN.RepFunNum++] = 0;
00727
00728                     if ( funnycount > 0 ) {
00729                         p = pattern + FUNHEAD;
00730                         t = fun + FUNHEAD;
00731                         while ( p < pstop ) {
00732                             if ( *p != FUNNYWILD ) { p++; continue; }
00733                             AN.argaddress = t;
00734                             AddWild(BHEAD p[1],ARGTOARG,0);
00735                             p += 2;
00736                         }
00737                     }
00738                     newpat = pattern + pattern[1];
00739                     if ( newpat >= AN.patstop ) {
00740                         if ( AN.UseFindOnly == 0 ) {
00741                             if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
00742                                 AT.WorkPointer = oldworkpointer;
00743                                 AN.UsedOtherFind = 1;
00744                                 return(1);
00745                             }
00746                             j = 0;
00747                         }
00748                         else {
00749                             AT.WorkPointer = oldworkpointer;
00750                             return(1);
00751                         }
00752                     }
00753                     else j = ScanFunctions(BHEAD newpat,inter,par);
00754                     if ( j ) {
00755                         AT.WorkPointer = oldworkpointer;
00756                         return(j); /* Full match. Return our success */
00757                     }
00758                     AN.RepFunNum -= 2;
00759                 }
00760
00761                 /* No (deeper) match. -> reset wildcards and continue */
00762
00763                 if ( wc && nwstore > 0 ) {
00764                     j = nwstore;
00765                     m = AN.WildValue;
00766                     t = thewildcards + ntwa; r = AT.WildMask;
```

```
00767                     if ( j > 0 ) {
00768                         do {
00769                             *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
00770                         } while ( --j > 0 );
00771                     }
00772                     C->numrhs = *t++;
00773                     C->Pointer = C->Buffer + oldcpointer;
00774                 }
00775             }
00776         }
00777         goto NoSuccess;
00778     }
00779 /*
00780     #] Case 1:
00781     #[ Case 2: One FUNNYWILD. Fix its length.
00782 */
00783     if ( funnycount == 1 ) {
00784         funnycount = tcount - argcount; /* Number or arguments to be eaten */
00785         for ( k = 0; k <= type; k++ ) {
00786             if ( k == 0 ) {
00787                 p = params; t = fun + FUNHEAD;
00788                 while ( t < tstop ) *p++ = *t++;
00789             }
00790             else {
00791                 p = params+tcount; t = fun + FUNHEAD;
00792                 while ( t < tstop ) *--p = *t++;
00793             }
00794             for ( i = 0; i < tcount; i++ ) { /* The various cycles */
00795                 p = pattern + FUNHEAD;
00796                 t = params;
00797                 wc = 0;
00798                 for ( j = 0; j < tcount; j++, p++, t++ ) { /* The arguments */
00799                     if ( *t == *p ) continue;
00800                     if ( *p == FUNNYWILD ) {
00801                         p++; wc = 1;
00802                         AN.argaddress = t;
00803                         if ( CheckWild(BHEAD *p,ARGTOARG|EATTENSOR,funnycount,t) ) break;
00804                         AddWild(BHEAD *p,ARGTOARG|EATTENSOR,funnycount);
00805                         j += funnycount-1; t += funnycount-1;
00806                     }
00807                     else if ( *p >= AM.OffsetIndex + WILDOFFSET
00808                     && *p < AM.OffsetIndex + 2*WILDOFFSET ) {
00809
00810                         /* Test wildcard index */
00811
00812                         wc = *p - WILDOFFSET;
00813                         if ( CheckWild(BHEAD wc,INDTOIND,*t,&newvalue) ) break;
00814                         AddWild(BHEAD wc,INDTOIND,newvalue);
00815                     }
00816                     else if ( *t < MINSPEC && *p < MINSPEC
00817                         && *p >= AM.OffsetVector + WILDOFFSET ) {
00818
00819                         /* Test wildcard vector */
00820
00821                         wc = *p - WILDOFFSET;
00822                         if ( CheckWild(BHEAD wc,VECTOVEC,*t,&newvalue) ) break;
00823                         AddWild(BHEAD wc,VECTOVEC,newvalue);
00824                     }
00825                     else break;
00826                 }
00827                 if ( j >= tcount ) { /* Match! */
00828
00829                     /* Continue with other functions. Make sure of the funnies */
00830
00831                     AN.RepFunList[AN.RepFunNum++] = offset;
00832                     AN.RepFunList[AN.RepFunNum++] = 0;
00833                     newpat = pattern + pattern[1];
00834                     if ( newpat >= AN.patstop ) {
00835                         if ( AN.UseFindOnly == 0 ) {
00836                             if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
00837                                 AT.WorkPointer = oldworkpointer;
00838                                 AN.UsedOtherFind = 1;
00839                                 return(1);
00840                             }
00841                             j = 0;
00842                         }
00843                         else {
00844                             AT.WorkPointer = oldworkpointer;
00845                             return(1);
00846                         }
00847                     }
00848                     else j = ScanFunctions(BHEAD newpat,inter,par);
00849                     if ( j ) {
00850                         AT.WorkPointer = oldworkpointer;
00851                         return(j); /* Full match. Return our success */
00852                     }
00853                     AN.RepFunNum -= 2;
```

```
00854                 }
00855
00856                 /* No (deeper) match. -> reset wildcards and continue */
00857
00858                 if ( wc ) {
00859                     j = nwstore;
00860                     m = AN.WildValue;
00861                     t = thewildcards + ntwa; r = AT.WildMask;
00862                     if ( j > 0 ) {
00863                         do {
00864                             *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
00865                         } while ( --j > 0 );
00866                     }
00867                     C->numrhs = *t++;
00868                     C->Pointer = C->Buffer + oldcpointer;
00869                 }
00870                 t = params;
00871                 wc = *t;
00872                 for ( j = 1; j < tcount; j++ ) { *t = t[1]; t++; }
00873                 *t = wc;
00874             }
00875         }
00876         goto NoSuccess;
00877     }
00878 /*
00879     #] Case 2:
00880     #[ Case 3: More than one FUNNYWILD. Complicated.
00881 */
00882
00883     sumeat = tcount - argcount; /* Total number to be eaten by Funnies */
00884 /*
00885     In the first funnycount elements of 'thewildcards' we arrange
00886     for the summing over the various possibilities.
00887     The renumbering table is in thewildcards[2*funnycount]
00888     The multiplicity table is in thewildcards[funnycount]
00889     The number of arguments for each is in thewildcards[]
00890 */
00891     p = pattern+FUNHEAD;
00892     for ( i = funnycount; i < ntwa; i++ ) thewildcards[i] = -1;
00893     multiplicity = thewildcards + funnycount;
00894     renum = multiplicity + funnycount;
00895     j = 0;
00896     while ( p < pstop ) {
00897         if ( *p != FUNNYWILD ) { p++; continue; }
00898         p++;
00899         if ( renum[*p] < 0 ) {
00900             renum[*p] = j;
00901             multiplicity[j] = 1;
00902             j++;
00903         }
00904         else multiplicity[renum[*p]]++;
00905         p++;
00906     }
00907 /*
00908     Strategy: First 'declared' has a tendency to be smaller
00909 */
00910     for ( i = 1; i < AN.NumTotWildArgs; i++ ) {
00911         if ( renum[i] < 0 ) continue;
00912         for ( j = i+1; j <= AN.NumTotWildArgs; j++ ) {
00913             if ( renum[j] < 0 ) continue;
00914             if ( renum[i] < renum[j] ) continue;
00915             k = multiplicity[renum[i]];
00916             multiplicity[renum[i]] = multiplicity[renum[j]];
00917             multiplicity[renum[j]] = k;
00918             k = renum[i]; renum[i] = renum[j]; renum[j] = k;
00919         }
00920     }
00921     for ( i = 0; i < funnycount; i++ ) thewildcards[i] = 0;
00922     iraise = funnycount-1;
00923     for ( ;; ) {
00924         for ( i = 0, j = sumeat; i < iraise; i++ )
00925                 j -= thewildcards[i]*multiplicity[i];
00926         if ( j < 0 || j % multiplicity[iraise] != 0 ) {
00927             if ( j > 0 ) {
00928                 thewildcards[iraise-1]++;
00929                 continue;
00930             }
00931             itop = iraise-1;
00932             while ( itop > 0 && j < 0 ) {
00933                 j += thewildcards[itop]*multiplicity[itop];
00934                 thewildcards[itop] = 0;
00935                 itop--;
00936             }
00937             if ( itop <= 0 && j <= 0 ) break;
00938             thewildcards[itop]++;
00939             continue;
00940         }
```

```
00941         thewildcards[iraise] = j / multiplicity[iraise];
00942
00943         for ( k = 0; k <= type; k++ ) {
00944             if ( k == 0 ) {
00945                 p = params; t = fun + FUNHEAD;
00946                 while ( t < tstop ) *p++ = *t++;
00947             }
00948             else {
00949                 p = params+tcount; t = fun + FUNHEAD;
00950                 while ( t < tstop ) *--p = *t++;
00951             }
00952             for ( i = 0; i < tcount; i++ ) { /* The various cycles */
00953                 p = pattern + FUNHEAD;
00954                 t = params;
00955                 wc = 0;
00956                 for ( j = 0; j < tcount; j++, p++, t++ ) { /* The arguments */
00957                     if ( *t == *p ) continue;
00958                     if ( *p == FUNNYWILD ) {
00959                         p++; wc = thewildcards[renum[*p]];
00960                         AN.argaddress = t;
00961                         if ( CheckWild(BHEAD *p,ARGTOARG|EATTENSOR,wc,t) ) break;
00962                         AddWild(BHEAD *p,ARGTOARG|EATTENSOR,wc);
00963                         j += wc-1; t += wc-1; wc = 1;
00964                     }
00965                     else if ( *p >= AM.OffsetIndex + WILDOFFSET
00966                     && *p < AM.OffsetIndex + 2*WILDOFFSET ) {
00967
00968                         /* Test wildcard index */
00969
00970                         wc = *p - WILDOFFSET;
00971                         if ( CheckWild(BHEAD wc,INDTOIND,*t,&newvalue) ) break;
00972                         AddWild(BHEAD wc,INDTOIND,newvalue);
00973                     }
00974                     else if ( *t < MINSPEC && *p < MINSPEC
00975                         && *p >= AM.OffsetVector + WILDOFFSET ) {
00976
00977                         /* Test wildcard vector */
00978
00979                         wc = *p - WILDOFFSET;
00980                         if ( CheckWild(BHEAD wc,VECTOVEC,*t,&newvalue) ) break;
00981                         AddWild(BHEAD wc,VECTOVEC,newvalue);
00982                     }
00983                     else break;
00984                 }
00985                 if ( j >= tcount ) { /* Match! */
00986
00987                     /* Continue with other functions. Make sure of the funnies */
00988
00989                     AN.RepFunList[AN.RepFunNum++] = offset;
00990                     AN.RepFunList[AN.RepFunNum++] = 0;
00991                     newpat = pattern + pattern[1];
00992                     if ( newpat >= AN.patstop ) {
00993                         if ( AN.UseFindOnly == 0 ) {
00994                             if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
00995                                 AT.WorkPointer = oldworkpointer;
00996                                 AN.UsedOtherFind = 1;
00997                                 return(1);
00998                             }
00999                             j = 0;
01000                         }
01001                         else {
01002                             AT.WorkPointer = oldworkpointer;
01003                             return(1);
01004                         }
01005                     }
01006                     else j = ScanFunctions(BHEAD newpat,inter,par);
01007                     if ( j ) {
01008                         AT.WorkPointer = oldworkpointer;
01009                         return(j); /* Full match. Return our success */
01010                     }
01011                     AN.RepFunNum -= 2;
01012                 }
01013
01014                 /* No (deeper) match. -> reset wildcards and continue */
01015
01016                 if ( wc ) {
01017                     j = nwstore;
01018                     m = AN.WildValue;
01019                     t = thewildcards + ntwa; r = AT.WildMask;
01020                     if ( j > 0 ) {
01021                         do {
01022                             *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01023                         } while ( --j > 0 );
01024                     }
01025                     C->numrhs = *t++;
01026                     C->Pointer = C->Buffer + oldcpointer;
01027                 }
```

```
01028                 t = params;
01029                 wc = *t;
01030                 for ( j = 1; j < tcount; j++ ) { *t = t[1]; t++; }
01031                 *t = wc;
01032             }
01033         }
01034         (thewildcards[iraise-1])++;
01035     }
01036 /*
01037     #] Case 3:
01038 */
01039 NoSuccess:
01040     if ( oldwilval > 0 ) {
01041 nomatch:;
01042         j = nwstore;
01043         if ( j > 0 ) {
01044             m = AN.WildValue;
01045             t = lowlevel; r = AT.WildMask;
01046             if ( j > 0 ) {
01047                 do {
01048                     *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01049                 } while ( --j > 0 );
01050             }
01051             C->numrhs = *t++;
01052             C->Pointer = C->Buffer + oldcpointer;
01053         }
01054     }
01055     AT.WorkPointer = oldworkpointer;
01056     return(0);
01057 }
01058
01059 /*
01060     #] MatchCy :
01061     #[ FunMatchCy :
01062
01063         Matching of (r)cyclic functions.
01064         Like MatchCy, but now for general functions.
01065 */
01066
01067 int FunMatchCy(PHEAD WORD *pattern, WORD *fun, WORD *inter, WORD par)
01068 {
01069     GETBIDENTITY
01070     WORD *t, *tstop, *p, *pstop, *m, *r, *oldworkpointer = AT.WorkPointer;
01071     WORD **a, *thewildcards, *multiplicity, *renum, wc, wcc, oldwilval = 0;
01072     LONG oww = AT.pWorkPointer;
01073     WORD newvalue, *lowlevel = 0;
01074     int argcount = 0, funnycount = 0, tcount = 0;
01075     int type = 0, pnum, i, j, k, nwstore, iraise, itop, sumeat;
01076     CBUF *C = cbuf+AT.ebufnum;
01077     int ntwa = 3*AN.NumTotWildArgs+1;
01078     LONG oldcpointer = C->Pointer - C->Buffer;
01079     WORD offset = fun-AN.terstart, *newpat;
01080
01081     if ( (functions[fun[0]-FUNCTION].symmetric & ~REVERSEORDER) == RCYCLESYMMETRIC ) type = 1;
01082     pnum = pattern[0];
01083     nwstore = (AN.WildValue[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
01084     if ( pnum > FUNCTION + WILDOFFSET ) {
01085         pnum -= WILDOFFSET;
01086         if ( CheckWild(BHEAD pnum,FUNTOFUN,fun[0],&newvalue) ) return(0);
01087         oldwilval = 1;
01088         t = lowlevel = oldworkpointer;
01089         m = AN.WildValue;
01090         i = nwstore;
01091         r = AT.WildMask;
01092         if ( i > 0 ) {
01093             do {
01094                 *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
01095             } while ( --i > 0 );
01096         }
01097         *t++ = C->numrhs;
01098         if ( t >= AT.WorkTop ) {
01099             MLOCK(ErrorMessageLock);
01100             MesWork();
01101             MUNLOCK(ErrorMessageLock);
01102             return(-1);
01103         }
01104         AT.WorkPointer = t;
01105         AddWild(BHEAD pnum,FUNTOFUN,newvalue);
01106     }
01107     if ( (functions[pnum-FUNCTION].symmetric & ~REVERSEORDER) == RCYCLESYMMETRIC ) type = 1;
01108
01109     /* First we have to make an inventory. Are there -ARGWILD pointers? */
01110
01111     p = pattern + FUNHEAD;
01112     pstop = pattern + pattern[1];
01113     while ( p < pstop ) {
01114         if ( *p == -ARGWILD ) { p += 2; funnycount++; }
```

```
01115         else { NEXTARG(p); argcount++; }
01116     }
01117     t = fun + FUNHEAD;
01118     tstop = fun + fun[1];
01119     while ( t < tstop ) { NEXTARG(t); tcount++; }
01120
01121     if ( argcount > tcount ) return(0);
01122     if ( argcount < tcount && funnycount == 0 ) return(0);
01123     if ( argcount == 0 && tcount == 0 && funnycount == 0 ) {
01124         AN.RepFunList[AN.RepFunNum++] = offset;
01125         AN.RepFunList[AN.RepFunNum++] = 0;
01126         newpat = pattern + pattern[1];
01127         if ( newpat >= AN.patstop ) {
01128             if ( AN.UseFindOnly == 0 ) {
01129                 if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
01130                     AT.WorkPointer = oldworkpointer;
01131                     AN.UsedOtherFind = 1;
01132                     return(1);
01133                 }
01134                 j = 0;
01135             }
01136             else {
01137                 AT.WorkPointer = oldworkpointer;
01138                 return(1);
01139             }
01140         }
01141         else j = ScanFunctions(BHEAD newpat,inter,par);
01142         if ( j ) return(j);
01143         goto NoSuccess;
01144     }
01145
01146     /* Store the wildcard assignments */
01147
01148     WantAddPointers(tcount);
01149     AT.pWorkPointer += tcount;
01150     thewildcards = t = AT.WorkPointer;
01151     t += ntwa;
01152     if ( oldwilval ) lowlevel = oldworkpointer;
01153     else lowlevel = t;
01154     m = AN.WildValue;
01155     i = nwstore;
01156     if ( i > 0 ) {
01157         r = AT.WildMask;
01158         do {
01159             *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
01160         } while ( --i > 0 );
01161         *t++ = C->numrhs;
01162     }
01163     if ( t >= AT.WorkTop ) {
01164         MLOCK(ErrorMessageLock);
01165         MesWork();
01166         MUNLOCK(ErrorMessageLock);
01167         return(-1);
01168     }
01169     AT.WorkPointer = t;
01170 /*
01171     #[ Case 1: no funnies or all funnies must be empty. We just cycle through.
01172 */
01173     if ( argcount == tcount ) {
01174         if ( funnycount > 0 ) { /* Test all funnies first */
01175             p = pattern + FUNHEAD;
01176             t = fun + FUNHEAD;
01177             while ( p < pstop ) {
01178                 if ( *p != -ARGWILD ) { p++; continue; }
01179                 AN.argaddress = t;
01180                 if ( CheckWild(BHEAD p[1],ARGTOARG,0,t) ) goto nomatch;
01181                 AddWild(BHEAD p[1],ARGTOARG,0);
01182                 p += 2;
01183             }
01184             oldwilval = 1;
01185         }
01186         for ( k = 0; k <= type; k++ ) {
01187             if ( k == 0 ) {
01188                 a = AT.pWorkSpace+oww; t = fun + FUNHEAD;
01189                 while ( t < tstop ) { *a++ = t; NEXTARG(t); }
01190             }
01191             else {
01192                 a = AT.pWorkSpace+oww+tcount; t = fun + FUNHEAD;
01193                 while ( t < tstop ) { *--a = t; NEXTARG(t); }
01194             }
01195             for ( i = 0; i < tcount; i++ ) { /* The various cycles */
01196                 p = pattern + FUNHEAD;
01197                 wc = 0;
01198                 for ( j = 0; j < tcount; j++ ) { /* The arguments */
01199                     while ( *p == -ARGWILD ) p += 2;
01200                     t = AT.pWorkSpace[oww+((i+j)%tcount)];
01201                     if ( ( wcc =  MatchArgument(BHEAD t,p) ) == 0 ) break;
```

```
01202                   if ( wcc > 1 ) wc = 1;
01203                   NEXTARG(p);
01204               }
01205               if ( j >= tcount ) { /* Match! */
01206
01207                   /* Continue with other functions. Make sure of the funnies */
01208
01209                   AN.RepFunList[AN.RepFunNum++] = offset;
01210                   AN.RepFunList[AN.RepFunNum++] = 0;
01211
01212                   if ( funnycount > 0 ) {
01213                       p = pattern + FUNHEAD;
01214                       t = fun + FUNHEAD;
01215                       while ( p < pstop ) {
01216                           if ( *p != -ARGWILD ) { p++; continue; }
01217                           AN.argaddress = t;
01218                           AddWild(BHEAD p[1],ARGTOARG,0);
01219                           p += 2;
01220                       }
01221                   }
01222                   newpat = pattern + pattern[1];
01223                   if ( newpat >= AN.patstop ) {
01224                       if ( AN.UseFindOnly == 0 ) {
01225                           if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
01226                               AT.WorkPointer = oldworkpointer;
01227                               AT.pWorkPointer = oww;
01228                               AN.UsedOtherFind = 1;
01229                               return(1);
01230                           }
01231                           j = 0;
01232                       }
01233                       else {
01234                           AT.WorkPointer = oldworkpointer;
01235                           AT.pWorkPointer = oww;
01236                           return(1);
01237                       }
01238                   }
01239                   else j = ScanFunctions(BHEAD newpat,inter,par);
01240                   if ( j ) {
01241                       AT.WorkPointer = oldworkpointer;
01242                       AT.pWorkPointer = oww;
01243                       return(j); /* Full match. Return our success */
01244                   }
01245                   AN.RepFunNum -= 2;
01246               }
01247
01248               /* No (deeper) match. -> reset wildcards and continue */
01249
01250               if ( wc && nwstore > 0 ) {
01251                   j = nwstore;
01252                   m = AN.WildValue;
01253                   t = thewildcards + ntwa; r = AT.WildMask;
01254                   if ( j > 0 ) {
01255                       do {
01256                           *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01257                       } while ( --j > 0 );
01258                   }
01259                   C->numrhs = *t++;
01260                   C->Pointer = C->Buffer + oldcpointer;
01261               }
01262           }
01263       }
01264       goto NoSuccess;
01265   }
01266 /*
01267   #] Case 1:
01268   #[ Case 2: One -ARGWILD. Fix its length.
01269 */
01270   if ( funnycount == 1 ) {
01271       funnycount = tcount - argcount; /* Number or arguments to be eaten */
01272       for ( k = 0; k <= type; k++ ) {
01273           if ( k == 0 ) {
01274               a =  AT.pWorkSpace+oww; t = fun + FUNHEAD;
01275               while ( t < tstop ) { *a++ = t; NEXTARG(t); }
01276           }
01277           else {
01278               a =  AT.pWorkSpace+oww+tcount; t = fun + FUNHEAD;
01279               while ( t < tstop ) { *--a = t; NEXTARG(t); }
01280           }
01281           for ( i = 0; i < tcount; i++ ) { /* The various cycles */
01282               p = pattern + FUNHEAD;
01283               a = AT.pWorkSpace+oww;
01284               wc = 0;
01285               for ( j = 0; j < tcount; j++, a++ ) { /* The arguments */
01286                   t = *a;
01287                   if ( *p == -ARGWILD ) {
01288                       wc = 1;
```

```
01289                       AN.argaddress = (WORD *)a;
01290                       if ( CheckWild(BHEAD p[1],ARLTOARL,funnycount,(WORD *)a) ) break;
01291                       AddWild(BHEAD p[1],ARLTOARL,funnycount);
01292                       j += funnycount-1; a += funnycount-1;
01293                   }
01294                   else if ( MatchArgument(BHEAD t,p) == 0 ) break;
01295                   NEXTARG(p);
01296               }
01297               if ( j >= tcount ) { /* Match! */
01298
01299                   /* Continue with other functions. Make sure of the funnies */
01300
01301                   AN.RepFunList[AN.RepFunNum++] = offset;
01302                   AN.RepFunList[AN.RepFunNum++] = 0;
01303                   newpat = pattern + pattern[1];
01304                   if ( newpat >= AN.patstop ) {
01305                       if ( AN.UseFindOnly == 0 ) {
01306                           if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
01307                               AT.WorkPointer = oldworkpointer;
01308                               AT.pWorkPointer = oww;
01309                               AN.UsedOtherFind = 1;
01310                               return(1);
01311                           }
01312                           j = 0;
01313                       }
01314                       else {
01315                           AT.WorkPointer = oldworkpointer;
01316                           AT.pWorkPointer = oww;
01317                           return(1);
01318                       }
01319                   }
01320                   else j = ScanFunctions(BHEAD newpat,inter,par);
01321                   if ( j ) {
01322                       AT.WorkPointer = oldworkpointer;
01323                       AT.pWorkPointer = oww;
01324                       return(j); /* Full match. Return our success */
01325                   }
01326                   AN.RepFunNum -= 2;
01327               }
01328
01329               /* No (deeper) match. -> reset wildcards and continue */
01330
01331               if ( wc ) {
01332                   j = nwstore;
01333                   m = AN.WildValue;
01334                   t = thewildcards + ntwa; r = AT.WildMask;
01335                   if ( j > 0 ) {
01336                       do {
01337                           *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01338                       } while ( --j > 0 );
01339                   }
01340                   C->numrhs = *t++;
01341                   C->Pointer = C->Buffer + oldcpointer;
01342               }
01343               a = AT.pWorkSpace+oww;
01344               t = *a;
01345               for ( j = 1; j < tcount; j++ ) { *a = a[1]; a++; }
01346               *a = t;
01347           }
01348       }
01349       goto NoSuccess;
01350   }
01351 /*
01352       #] Case 2:
01353       #[ Case 3: More than one -ARGWILD. Complicated.
01354 */
01355
01356       sumeat = tcount - argcount; /* Total number to be eaten by Funnies */
01357 /*
01358       In the first funnycount elements of 'thewildcards' we arrange
01359       for the summing over the various possibilities.
01360       The renumbering table is in thewildcards[2*funnycount]
01361       The multiplicity table is in thewildcards[funnycount]
01362       The number of arguments for each is in thewildcards[]
01363 */
01364       p = pattern+FUNHEAD;
01365       for ( i = funnycount; i < ntwa; i++ ) thewildcards[i] = -1;
01366       multiplicity = thewildcards + funnycount;
01367       renum = multiplicity + funnycount;
01368       j = 0;
01369       while ( p < pstop ) {
01370           if ( *p != -ARGWILD ) { p++; continue; }
01371           p++;
01372           if ( renum[*p] < 0 ) {
01373               renum[*p] = j;
01374               multiplicity[j] = 1;
01375               j++;
```

```
01376         }
01377         else multiplicity[renum[*p]]++;
01378         p++;
01379     }
01380 /*
01381     Strategy: First 'declared' has a tendency to be smaller
01382 */
01383     for ( i = 1; i < AN.NumTotWildArgs; i++ ) {
01384         if ( renum[i] < 0 ) continue;
01385         for ( j = i+1; j <= AN.NumTotWildArgs; j++ ) {
01386             if ( renum[j] < 0 ) continue;
01387             if ( renum[i] < renum[j] ) continue;
01388             k = multiplicity[renum[i]];
01389             multiplicity[renum[i]] = multiplicity[renum[j]];
01390             multiplicity[renum[j]] = k;
01391             k = renum[i]; renum[i] = renum[j]; renum[j] = k;
01392         }
01393     }
01394     for ( i = 0; i < funnycount; i++ ) thewildcards[i] = 0;
01395     iraise = funnycount-1;
01396     for ( ;; ) {
01397         for ( i = 0, j = sumeat; i < iraise; i++ )
01398                 j -= thewildcards[i]*multiplicity[i];
01399         if ( j < 0 || j % multiplicity[iraise] != 0 ) {
01400             if ( j > 0 ) {
01401                 thewildcards[iraise-1]++;
01402                 continue;
01403             }
01404             itop = iraise-1;
01405             while ( itop > 0 && j < 0 ) {
01406                 j += thewildcards[itop]*multiplicity[itop];
01407                 thewildcards[itop] = 0;
01408                 itop--;
01409             }
01410             if ( itop <= 0 && j <= 0 ) break;
01411             thewildcards[itop]++;
01412             continue;
01413         }
01414         thewildcards[iraise] = j / multiplicity[iraise];
01415
01416         for ( k = 0; k <= type; k++ ) {
01417             if ( k == 0 ) {
01418                 a = AT.pWorkSpace+oww; t = fun + FUNHEAD;
01419                 while ( t < tstop ) { *a++ = t; NEXTARG(t); }
01420             }
01421             else {
01422                 a = AT.pWorkSpace+oww+tcount; t = fun + FUNHEAD;
01423                 while ( t < tstop ) { *--a = t; NEXTARG(t); }
01424             }
01425             for ( i = 0; i < tcount; i++ ) { /* The various cycles */
01426                 p = pattern + FUNHEAD;
01427                 a = AT.pWorkSpace+oww;
01428                 wc = 0;
01429                 for ( j = 0; j < tcount; j++, a++ ) { /* The arguments */
01430                     t = *a;
01431                     if ( *p == -ARGWILD ) {
01432                         wc = thewildcards[renum[p[1]]];
01433                         AN.argaddress = (WORD *)a;
01434                         if ( CheckWild(BHEAD p[1],ARLTOARL,wc,(WORD *)a) ) break;
01435                         AddWild(BHEAD p[1],ARLTOARL,wc);
01436                         j += wc-1; a += wc-1; wc = 1;
01437                     }
01438                     else if ( MatchArgument(BHEAD t,p) == 0 ) break;
01439                     NEXTARG(p);
01440                 }
01441                 if ( j >= tcount ) { /* Match! */
01442
01443                     /* Continue with other functions. Make sure of the funnies */
01444
01445                     AN.RepFunList[AN.RepFunNum++] = offset;
01446                     AN.RepFunList[AN.RepFunNum++] = 0;
01447                     newpat = pattern + pattern[1];
01448                     if ( newpat >= AN.patstop ) {
01449                         if ( AN.UseFindOnly == 0 ) {
01450                             if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
01451                                 AT.WorkPointer = oldworkpointer;
01452                                 AT.pWorkPointer = oww;
01453                                 AN.UsedOtherFind = 1;
01454                                 return(1);
01455                             }
01456                             j = 0;
01457                         }
01458                         else {
01459                             AT.WorkPointer = oldworkpointer;
01460                             AT.pWorkPointer = oww;
01461                             return(1);
01462                         }
```

```
01463                     }
01464                     else j = ScanFunctions(BHEAD newpat,inter,par);
01465                     if ( j ) {
01466                         AT.WorkPointer = oldworkpointer;
01467                         AT.pWorkPointer = oww;
01468                         return(j); /* Full match. Return our success */
01469                     }
01470                     AN.RepFunNum -= 2;
01471                 }
01472 
01473                 /* No (deeper) match. -> reset wildcards and continue */
01474 
01475                 if ( wc ) {
01476                     j = nwstore;
01477                     m = AN.WildValue;
01478                     t = thewildcards + ntwa; r = AT.WildMask;
01479                     if ( j > 0 ) {
01480                         do {
01481                             *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01482                         } while ( --j > 0 );
01483                     }
01484                     C->numrhs = *t++;
01485                     C->Pointer = C->Buffer + oldcpointer;
01486                 }
01487                 a = AT.pWorkSpace+oww;
01488                 t = *a;
01489                 for ( j = 1; j < tcount; j++ ) { *a = a[1]; a++; }
01490                 *a = t;
01491             }
01492         }
01493         (thewildcards[iraise-1])++;
01494     }
01495 /*
01496     #] Case 3:
01497 */
01498 NoSuccess:
01499     if ( oldwilval > 0 ) {
01500 nomatch:;
01501         j = nwstore;
01502         m = AN.WildValue;
01503         t = lowlevel; r = AT.WildMask;
01504         if ( j > 0 ) {
01505             do {
01506                 *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01507             } while ( --j > 0 );
01508         }
01509         C->numrhs = *t++;
01510         C->Pointer = C->Buffer + oldcpointer;
01511     }
01512     AT.WorkPointer = oldworkpointer;
01513     AT.pWorkPointer = oww;
01514     return(0);
01515 }
01516 
01517 /*
01518     #] FunMatchCy :
01519     #[ FunMatchSy :
01520 
01521         Matching of (anti)symmetric functions.
01522         Like MatchE, but now for general functions.
01523 */
01524 
01525 int FunMatchSy(PHEAD WORD *pattern, WORD *fun, WORD *inter, WORD par)
01526 {
01527     GETBIDENTITY
01528     WORD *t, *tstop, *p, *pstop, *m, *r, *oldworkpointer = AT.WorkPointer;
01529     WORD **a, *thewildcards, oldwilval = 0;
01530     WORD newvalue, *lowlevel = 0, num, assig;
01531     WORD *cycles;
01532     LONG oww = AT.pWorkPointer, lhpars, lhfunnies;
01533     int argcount = 0, funnycount = 0, tcount = 0, signs = 0, signfun = 0, signo;
01534     int type = 0, pnum, i, j, k, nwstore, iraise, cou2;
01535     CBUF *C = cbuf+AT.ebufnum;
01536     int ntwa = 3*AN.NumTotWildArgs+1;
01537     LONG oldcpointer = C->Pointer - C->Buffer;
01538     WORD offset = fun-AN.terstart, *newpat;
01539 
01540     if ( (functions[fun[0]-FUNCTION].symmetric & ~REVERSEORDER) == RCYCLESYMMETRIC ) type = 1;
01541     pnum = pattern[0];
01542     nwstore = (AN.WildValue[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
01543     if ( pnum > FUNCTION + WILDOFFSET ) {
01544         pnum -= WILDOFFSET;
01545         if ( CheckWild(BHEAD pnum,FUNTOFUN,fun[0],&newvalue) ) return(0);
01546         oldwilval = 1;
01547         t = lowlevel = oldworkpointer;
01548         m = AN.WildValue;
01549         i = nwstore;
```

```
01550         r = AT.WildMask;
01551         if ( i > 0 ) {
01552             do {
01553                 *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
01554             } while ( --i > 0 );
01555         }
01556         *t++ = C->numrhs;
01557         if ( t >= AT.WorkTop ) {
01558             MLOCK(ErrorMessageLock);
01559             MesWork();
01560             MUNLOCK(ErrorMessageLock);
01561             return(-1);
01562         }
01563         AT.WorkPointer = t;
01564         AddWild(BHEAD pnum,FUNTOFUN,newvalue);
01565     }
01566     if ( (functions[pnum-FUNCTION].symmetric & ~REVERSEORDER) == RCYCLESYMMETRIC ) type = 1;
01567 
01568     /* Try for a straight match. After all, both have been normalized */
01569 
01570     if ( fun[1] == pattern[1] ) {
01571         i = fun[1]-FUNHEAD; p = pattern+FUNHEAD; t = fun + FUNHEAD;
01572         while ( --i >= 0 ) { if ( *p++ != *t++ ) break; }
01573         if ( i < 0 ) goto quicky;
01574     }
01575 
01576     /* First we have to make an inventory. Are there -ARGWILD pointers? */
01577 
01578     p = pattern + FUNHEAD;
01579     pstop = pattern + pattern[1];
01580     while ( p < pstop ) {
01581         if ( *p == -ARGWILD ) { p += 2; funnycount++; }
01582         else { NEXTARG(p); argcount++; }
01583     }
01584     t = fun + FUNHEAD;
01585     tstop = fun + fun[1];
01586     while ( t < tstop ) { NEXTARG(t); tcount++; }
01587 
01588     if ( argcount > tcount ) return(0);
01589     if ( argcount < tcount && funnycount == 0 ) return(0);
01590     if ( argcount == 0 && tcount == 0 && funnycount == 0 ) {
01591 quicky:
01592         if ( AN.SignCheck && signs != AN.ExpectedSign ) goto NoSuccess;
01593         AN.RepFunList[AN.RepFunNum++] = offset;
01594         AN.RepFunList[AN.RepFunNum++] = signs;
01595         newpat = pattern + pattern[1];
01596         if ( newpat >= AN.patstop ) {
01597             if ( AN.UseFindOnly == 0 ) {
01598                 if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
01599                     AT.WorkPointer = oldworkpointer;
01600                     AN.UsedOtherFind = 1;
01601                     return(1);
01602                 }
01603                 j = 0;
01604             }
01605             else {
01606                 AT.WorkPointer = oldworkpointer;
01607                 return(1);
01608             }
01609         }
01610         else j = ScanFunctions(BHEAD newpat,inter,par);
01611         if ( j ) {
01612             AT.WorkPointer = oldworkpointer;
01613             return(j);
01614         }
01615         goto NoSuccess;
01616     }
01617 
01618     /* Store the wildcard assignments */
01619 
01620     WantAddPointers(tcount+argcount+funnycount);
01621     AT.pWorkPointer += tcount+argcount+funnycount;
01622     thewildcards = t = AT.WorkPointer;
01623     t += ntwa;
01624     if ( oldwilval ) lowlevel = oldworkpointer;
01625     else lowlevel = t;
01626     m = AN.WildValue;
01627     i = nwstore; assig = 0;
01628     if ( i > 0 ) {
01629         r = AT.WildMask;
01630         do {
01631             assig += *r;
01632             *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
01633         } while ( --i > 0 );
01634         *t++ = C->numrhs;
01635     }
01636     if ( t >= AT.WorkTop ) {
```

```
01637         MLOCK(ErrorMessageLock);
01638         MesWork();
01639         MUNLOCK(ErrorMessageLock);
01640         return(-1);
01641     }
01642     AT.WorkPointer = t;
01643
01644     /* Store pointers to the arguments */
01645
01646     t = fun + FUNHEAD; a = AT.pWorkSpace+oww;
01647     while ( t < tstop ) { *a++ = t; NEXTARG(t) }
01648     lhpars = a-AT.pWorkSpace;
01649     t = pattern + FUNHEAD;
01650     while ( t < pstop ) {
01651         if ( *t != -ARGWILD ) *a++ = t;
01652         NEXTARG(t)
01653     }
01654     lhfunnies = a-AT.pWorkSpace;
01655     t = pattern + FUNHEAD; cou2 = 0;
01656     while ( t < pstop ) {
01657         cou2++;
01658         if ( *t == -ARGWILD ) {
01659             *a++ = t;
01660 /*
01661     signfun: last ?a: tcount-argcount: number of arguments in ?a (assume one ?a)
01662         argcount+funnycount-cou2: arguments after ?a.
01663     Together tells whether moving ?a to end of list is even or odd
01664 */
01665             signfun = ((argcount+funnycount-cou2)*(tcount-argcount)) & 1;
01666         }
01667         NEXTARG(t)
01668     }
01669     signs += signfun;
01670     if ( funnycount > 0 ) {
01671         if ( ( (functions[fun[0]-FUNCTION].symmetric & ~REVERSEORDER) == SYMMETRIC )
01672             || ( (functions[fun[0]-FUNCTION].symmetric & ~REVERSEORDER) == ANTISYMMETRIC )
01673             || ( (functions[pnum-FUNCTION].symmetric & ~REVERSEORDER) == SYMMETRIC )
01674             || ( (functions[pnum-FUNCTION].symmetric & ~REVERSEORDER) == ANTISYMMETRIC ) ) {
01675                 AT.WorkPointer = oldworkpointer;
01676                 AT.pWorkPointer = oww;
01677                 MLOCK(ErrorMessageLock);
01678                 MesPrint("Sorry: no argument field wildcards yet in (anti)symmetric functions");
01679                 MUNLOCK(ErrorMessageLock);
01680                 Terminate(-1);
01681         }
01682     }
01683 /*
01684     Sort the regular arguments by
01685     1: no wildcards, fast.
01686     2: wildcards that have been assigned.
01687     3: general arguments.
01688     4: wildcards without an assignment.
01689 */
01690     iraise = argcount;
01691     for ( i = 0; i < iraise; i++ ) {
01692         t = AT.pWorkSpace[i+lhpars];
01693         if ( *t > 0 ) { /* Category 3: general argument */
01694             continue;
01695         }
01696         else if ( *t <= -FUNCTION ) {
01697             if ( *t > -FUNCTION - WILDOFFSET ) goto cat1;
01698             type = FUNTOFUN; num = -*t - WILDOFFSET;
01699         }
01700         else if ( *t == -SYMBOL ) {
01701             if ( t[1] < 2*MAXPOWER ) goto cat1;
01702             type = SYMTOSYM; num = t[1] - 2*MAXPOWER;
01703         }
01704         else if ( *t == -INDEX ) {
01705             if ( t[1] < AM.OffsetIndex + WILDOFFSET ) goto cat1;
01706             type = INDTOIND; num = t[1] - WILDOFFSET;
01707         }
01708         else if ( *t == -VECTOR || *t == -MINVECTOR ) {
01709             if ( t[1] < AM.OffsetVector + WILDOFFSET ) goto cat1;
01710             type = VECTOVEC; num = t[1] - WILDOFFSET;
01711         }
01712         else goto cat1; /* Things like -SNUMBER etc. */
01713 /*
01714         Now we have a wildcard and have to see whether it was assigned
01715 */
01716         m = AN.WildValue;
01717         j = nwstore;
01718         r = AT.WildMask;
01719         while ( --j >= 0 ) {
01720             if ( m[2] == num && *r ) {
01721                 if ( type == *m ) break;
01722                 if ( type == SYMTOSYM ) {
01723                     if ( *m == SYMTONUM || *m == SYMTOSUB ) break;
```

```
01724                 }
01725                 else if ( type == INDTOIND ) {
01726                     if ( *m == INDTOSUB ) break;
01727                 }
01728                 else if ( type == VECTOVEC ) {
01729                     if ( *m == VECTOMIN || *m == VECTOSUB ) break;
01730                 }
01731             }
01732             m += 4; r++;
01733         }
01734         if ( j < 0 ) {  /* Category 4: Wildcard that was not assigned */
01735             a = AT.pWorkSpace+lhpars;
01736             iraise--;
01737             if ( iraise != i ) signs++;
01738             m = a[iraise];
01739             a[iraise] = a[i];
01740             a[i] = m; i--;
01741         }
01742         else {  /* Category 2: Wildcard that was assigned */
01743             for ( j = 0; j < tcount; j++ ) {
01744                 if ( MatchArgument(BHEAD AT.pWorkSpace[oww+j],t) ) {
01745                     k = nwstore;
01746                     r = AT.WildMask;
01747                     num = 0;
01748                     while ( --k >= 0 ) num += *r++;
01749                     if ( num == assig ) { /* no wildcards were changed */
01750                         goto oneless;
01751                     }
01752                     break;
01753                 }
01754             }
01755             if ( j >= tcount ) goto NoSuccess;
01756             j = nwstore;
01757             m = AN.WildValue;
01758             t = thewildcards + ntwa; r = AT.WildMask;
01759             if ( j > 0 ) {
01760                 do {    /* undo assignment */
01761                     *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01762                 } while ( --j > 0 );
01763             }
01764             C->numrhs = *t++;
01765         }
01766         continue;
01767 cat1:
01768         for ( j = 0; j < tcount; j++ ) {
01769             m = AT.pWorkSpace[j+oww];
01770             if ( *t != *m ) continue;
01771             if ( *t < 0 ) {
01772                 if ( *t <= -FUNCTION ) break;
01773                 if ( t[1] == m[1] ) break;
01774             }
01775             else {
01776                 k = *t; r = t;
01777                 while ( --k >= 0 && *m++ == *r++ ) {}
01778                 if ( k < 0 ) break;
01779             }
01780         }
01781         if ( j >= tcount ) goto NoSuccess;  /* Even the fixed ones don't match */
01782 oneless:
01783         signs += j - i;
01784 /*
01785         The next statements replace the one that is commented out
01786 */
01787         tcount--;
01788         while ( j < tcount ) {
01789             AT.pWorkSpace[oww+j] = AT.pWorkSpace[oww+j+1]; j++;
01790         }
01791 /*
01792         AT.pWorkSpace[oww+j] = AT.pWorkSpace[oww+(--tcount)];
01793 */
01794         argcount--; j = i;
01795         while ( j < argcount ) {
01796             AT.pWorkSpace[lhpars+j] = AT.pWorkSpace[lhpars+j+1]; j++;
01797         }
01798         iraise--; i--;
01799     }
01800 /*
01801     Now we see whether there are any ARGWILD objects that have been
01802     assigned already. In that case the work simplifies considerably.
01803     Currently (12-nov-2001) only in (R)CYCLIC functions; hence we do not
01804     test the sign!
01805 */
01806     for ( i = 0; i < funnycount; i++ ) {
01807         k = AT.pWorkSpace[lhfunnies+i][1];
01808         m = AN.WildValue;
01809         j = nwstore;
01810         r = AT.WildMask;
```

```
01811         while ( --j >= 0 ) {
01812             if ( *m == ARGTOARG && m[2] == k ) break;
01813             m += 4; r++;
01814         }
01815         if ( *r == 0 ) continue; /* not assigned yet */
01816         m = cbuf[AT.ebufnum].rhs[m[3]];
01817         if ( *m > 0 ) {     /* Tensor arguments */
01818             j = *m;
01819             if ( j > tcount - argcount ) goto NoSuccess;
01820             while ( --j >= 0 ) {
01821                 m++;
01822                 if ( *m < 0 ) type = -VECTOR;
01823                 else if ( *m < AM.OffsetIndex ) type = -SNUMBER;
01824                 else type = -INDEX;
01825                 a = AT.pWorkSpace+oww;
01826                 for ( k = 0; k < tcount; k++ ) {
01827                     if ( a[k][0] != type || a[k][1] != *m ) continue;
01828                     a[k] = a[--tcount];
01829                     goto nextjarg;
01830                 }
01831                 goto NoSuccess;
01832 nextjarg:;
01833             }
01834         }
01835         else {
01836             m++;
01837             while ( *m ) {
01838                 for ( k = 0; k < tcount; k++ ) {
01839                     t = AT.pWorkSpace[oww+k];
01840                     if ( *t != *m ) continue;
01841                     r = m;
01842                     if ( *r < 0 ) {
01843                         if ( *r < -FUNCTION ) goto nextargw;
01844                         else if ( r[1] == t[1] ) goto nextargw;
01845                     }
01846                     else {
01847                         j = *r;
01848                         while ( --j >= 0 && *r++ == *t++ ) {}
01849                         if ( j < 0 ) goto nextargw;
01850                     }
01851                 }
01852                 goto NoSuccess;
01853 nextargw:;
01854                 AT.pWorkSpace[oww+k] = AT.pWorkSpace[oww+(--tcount)];
01855                 NEXTARG(m)
01856             }
01857         }
01858         AT.pWorkSpace[lhfunnies+i] = AT.pWorkSpace[lhfunnies+(--funnycount)];
01859     }
01860     if ( tcount == 0 ) {
01861         if ( argcount > 0 ) goto NoSuccess;
01862         for ( i = 0; i < funnycount; i++ ) {
01863             AddWild(BHEAD AT.pWorkSpace[lhfunnies+i][1],ARGTOARG,0);
01864         }
01865         goto quicky;
01866     }
01867 /*
01868     We have now in lhpars first iraise elements with a dubious nature.
01869     Then argcount-iraise wildcards that have not been assigned.
01870     In lhfunnies we have funnycount ARGTOARG objects. ( (R)CyCLIC only )
01871
01872     First work our way through the 'dubious' objects
01873     We check whether assig changes.
01874 */
01875     for ( i = 0; i < iraise; i++ ) {
01876         for ( j = 0; j < tcount; j++ ) {
01877             if ( MatchArgument(BHEAD AT.pWorkSpace[oww+j],AT.pWorkSpace[lhpars+i]) ) {
01878                 k = nwstore;
01879                 r = AT.WildMask;
01880                 num = 0;
01881                 while ( --k >= 0 ) num += *r++;
01882                 if ( num == assig ) { /* no wildcards were changed */
01883                     signs += j-i;
01884                     AT.pWorkSpace[oww+j] = AT.pWorkSpace[oww+(--tcount)];
01885                     if ( tcount > j ) signs += tcount-j-1;
01886                     argcount--;
01887                     a = AT.pWorkSpace + lhpars;
01888                     for ( j = i; j < argcount; j++ ) a[j] = a[j+1];
01889                     iraise--;
01890                     goto nextiraise;
01891                 }
01892                 else {  /* We cannot use this yet */
01893                     j = nwstore;
01894                     m = AN.WildValue;
01895                     t = thewildcards + ntwa; r = AT.WildMask;
01896                     if ( j > 0 ) {
01897                         do {    /* undo assignment */
```

```
01898                               *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01899                           } while ( --j > 0 );
01900                       }
01901                       C->numrhs = *t++;
01902                       C->Pointer = C->Buffer + oldcpointer;
01903                       goto nextiraise;
01904                   }
01905               }
01906           }
01907           goto NoSuccess;
01908   nextiraise:;
01909       }
01910   /*
01911       Now all leftover patterns have unassigned wildcards in them.
01912       From now on we are in potential factorial territory.
01913
01914       Strategy:
01915           1: cycle through the regular objects.
01916           2: save wildcard settings
01917           3: divide the ARGWILDs
01918           4: make permutations of leftover arguments
01919           5: try them all
01920   */
01921       cycles = AT.WorkPointer;
01922       for ( i = 0; i < tcount; i++ ) cycles[i] = tcount-i;
01923       AT.WorkPointer += tcount;
01924       signo = 0;
01925   /*MesPrint("<1> signs = %d",signs);*/
01926       for (;;) {
01927           WORD oRepFunNum = AN.RepFunNum;
01928           for ( j = 0; j < argcount; j++ ) {
01929               if ( MatchArgument(BHEAD AT.pWorkSpace[oww+j],AT.pWorkSpace[lhpars+j]) == 0 ) {
01930                   break;
01931               }
01932           }
01933           if ( j >= argcount ) {
01934   /*
01935               Thus far we have a match. Now the funnies
01936   */
01937               if ( funnycount ) {
01938                   AT.WorkPointer = oldworkpointer;
01939                   AT.pWorkPointer = oww;
01940                   MLOCK(ErrorMessageLock);
01941                   MesPrint("Sorry: no argument field wildcards yet in (anti)symmetric functions");
01942                   MUNLOCK(ErrorMessageLock);
01943   /*
01944                   Bugfix 31-oct-2001, reported by Kasper Peeters
01945                   We returned here with value -1 but that is not caught.
01946                   Extra note (12-nov-2001): the sign becomes a bit problematic
01947                   if we have funnies. No more than one allowed in antisymmetric
01948                   functions, or we have serious problems.
01949   */
01950                   Terminate(-1);
01951               }
01952
01953               AN.RepFunList[AN.RepFunNum++] = offset;
01954               if ( ( (functions[fun[0]-FUNCTION].symmetric & ~REVERSEORDER) == ANTISYMMETRIC )
01955                   || ( (functions[pnum-FUNCTION].symmetric & ~REVERSEORDER) == ANTISYMMETRIC ) ) {
01956                   AN.RepFunList[AN.RepFunNum++] = ( signs + signo ) & 1;
01957               }
01958               else {
01959                   AN.RepFunList[AN.RepFunNum++] = 0;
01960               }
01961               newpat = pattern + pattern[1];
01962               if ( newpat >= AN.patstop ) {
01963                   WORD countsgn, sgn = 0;
01964                   for ( countsgn = oRepFunNum+1; countsgn < AN.RepFunNum; countsgn += 2 ) {
01965                       if ( AN.RepFunList[countsgn] ) sgn ^= 1;
01966                   }
01967                   if ( AN.SignCheck == 0 || sgn == AN.ExpectedSign ) {
01968                       AT.WorkPointer = oldworkpointer;
01969                       AT.pWorkPointer = oww;
01970                       return(1);
01971                   }
01972                   if ( AN.UseFindOnly == 0 ) {
01973                       if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
01974                           AT.WorkPointer = oldworkpointer;
01975                           AT.pWorkPointer = oww;
01976                           AN.UsedOtherFind = 1;
01977                           return(1);
01978                       }
01979                   }
01980                   j = 0;
01981               }
01982               else j = ScanFunctions(BHEAD newpat,inter,par);
01983               if ( j ) {
01984                   WORD countsgn, sgn = 0;
```

```
01985                 for ( countsgn = oRepFunNum+1; countsgn < AN.RepFunNum; countsgn += 2 ) {
01986                     if ( AN.RepFunList[countsgn] ) sgn ^= 1;
01987                 }
01988                 if ( AN.SignCheck == 0 || sgn == AN.ExpectedSign ) {
01989                     AT.WorkPointer = oldworkpointer;
01990                     AT.pWorkPointer = oww;
01991                     return(j);
01992                 }
01993             }
01994             AN.RepFunNum = oRepFunNum;
01995             i = argcount - 1;
01996         }
01997         else i = j;
01998         j = nwstore;
01999         m = AN.WildValue;
02000         t = thewildcards + ntwa; r = AT.WildMask;
02001         if ( j > 0 ) {
02002             do {
02003                 *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
02004             } while ( --j > 0 );
02005         }
02006         C->numrhs = *t++;
02007         C->Pointer = C->Buffer + oldcpointer;
02008 /*
02009         On to the next cycle
02010 */
02011         a = AT.pWorkSpace + oww;
02012         for ( j = i+1, t = a[i]; j < tcount; j++ ) a[j-1] = a[j];
02013         a[tcount-1] = t; cycles[i]--;
02014         signo += tcount - i - 1;
02015         while ( cycles[i] <= 0 ) {
02016             cycles[i] = tcount - i;
02017             i--;
02018             if ( i < 0 ) goto NoSuccess;
02019 /*
02020             MLOCK(ErrorMessageLock);
02021             MesPrint("Cycle i = %d",i);
02022             MUNLOCK(ErrorMessageLock);
02023 */
02024             for ( j = i+1, t = a[i]; j < tcount; j++ ) a[j-1] = a[j];
02025             a[tcount-1] = t; cycles[i]--;
02026             signo += tcount - i - 1;
02027         }
02028     }
02029 NoSuccess:
02030     if ( oldwilval > 0 ) {
02031         j = nwstore;
02032         m = AN.WildValue;
02033         t = lowlevel; r = AT.WildMask;
02034         if ( j > 0 ) {
02035             do {
02036                 *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
02037             } while ( --j > 0 );
02038         }
02039         C->numrhs = *t++;
02040         C->Pointer = C->Buffer + oldcpointer;
02041     }
02042     AT.WorkPointer = oldworkpointer;
02043     AT.pWorkPointer = oww;
02044     return(0);
02045 }
02046
02047 /*
02048     #] FunMatchSy :
02049     #[ MatchArgument :
02050 */
02051
02052 int MatchArgument(PHEAD WORD *arg, WORD *pat)
02053 {
02054     GETBIDENTITY
02055     WORD *m = pat, *t = arg, i, j, newvalue;
02056     WORD *argmstop = pat, *argtstop = arg;
02057     WORD *cto, *cfrom, *csav, ci;
02058     WORD oRepFunNum, *oRepFunList;
02059     WORD *oterstart,*oterstop,*opatstop;
02060     WORD wildargs, wildeat;
02061     WORD *mtrmstop, *ttrmstop, *msubstop, msizcoef;
02062     WORD *wildargtaken;
02063     int wc = 1;
02064
02065     NEXTARG(argmstop);
02066     NEXTARG(argtstop);
02067 /*
02068     #[ Both fast :
02069 */
02070     if ( *m < 0 && *t < 0 ) {
02071         if ( *t <= -FUNCTION ) {
```

```
02072             if ( *t == *m ) {}
02073             else if ( *m <= -FUNCTION-WILDOFFSET
02074             && functions[-*t-FUNCTION].spec
02075             == functions[-*m-FUNCTION-WILDOFFSET].spec ) {
02076                 i = -*m - WILDOFFSET; wc = 2;
02077                 if ( CheckWild(BHEAD i,FUNTOFUN,-*t,&newvalue) ) {
02078                     return(0);
02079                 }
02080                 AddWild(BHEAD i,FUNTOFUN,newvalue);
02081             }
02082             else if ( *m == -SYMBOL && m[1] >= 2*MAXPOWER ) {
02083                 i = m[1] - 2*MAXPOWER;
02084                 AN.argaddress = AT.FunArg;
02085                 AT.FunArg[ARGHEAD+1] = -*t;
02086                 if ( CheckWild(BHEAD i,SYMTOSUB,1,AN.argaddress) ) return(0);
02087                 AddWild(BHEAD i,SYMTOSUB,0);
02088             }
02089             else return(0);
02090         }
02091         else if ( *t == *m ) {
02092             if ( t[1] == m[1] ) {}
02093             else if ( *t == -SYMBOL ) {
02094                 j = SYMTOSYM;
02095 SymAll:         if ( ( i = m[1] - 2*MAXPOWER ) < 0 ) return(0);
02096                 wc = 2;
02097                 if ( CheckWild(BHEAD i,j,t[1],&newvalue) ) return(0);
02098                 AddWild(BHEAD i,j,newvalue);
02099             }
02100             else if ( *t == -INDEX ) {
02101 IndAll:         i = m[1] - WILDOFFSET;
02102                 if ( i < AM.OffsetIndex || i >= WILDOFFSET+AM.OffsetIndex )
02103                                                     return(0);
02104                                 /* We kill the summed over indices here */
02105                 wc = 2;
02106                 if ( CheckWild(BHEAD i,INDTOIND,t[1],&newvalue) ) return(0);
02107                 AddWild(BHEAD i,INDTOIND,newvalue);
02108             }
02109             else if ( *t == -VECTOR || *t == -MINVECTOR ) {
02110                 i = m[1] - WILDOFFSET;
02111                 if ( i < AM.OffsetVector ) return(0);
02112                 wc = 2;
02113                 if ( CheckWild(BHEAD i,VECTOVEC,t[1],&newvalue) ) return(0);
02114                 AddWild(BHEAD i,VECTOVEC,newvalue);
02115             }
02116             else return(0);
02117         }
02118         else if ( *m == -INDEX && m[1] >= AM.OffsetIndex+WILDOFFSET
02119         && m[1] < AM.OffsetIndex+(WILDOFFSET<<1) ) {
02120             if ( *t == -VECTOR ) goto IndAll;
02121             if ( *t == -SNUMBER && t[1] >= 0 && t[1] < AM.OffsetIndex ) goto IndAll;
02122             if ( *t == -MINVECTOR ) {
02123                 i = m[1] - WILDOFFSET;
02124                 AN.argaddress = AT.MinVecArg;
02125                 AT.MinVecArg[ARGHEAD+3] = t[1];
02126                 wc = 2;
02127                 if ( CheckWild(BHEAD i,INDTOSUB,1,AN.argaddress) ) return(0);
02128                 AddWild(BHEAD i,INDTOSUB,(WORD)0);
02129             }
02130             else return(0);
02131         }
02132         else if ( *m == -SYMBOL && m[1] >= 2*MAXPOWER && *t == -SNUMBER ) {
02133             j = SYMTONUM;
02134             goto SymAll;
02135         }
02136         else if ( *m == -VECTOR && *t == -MINVECTOR &&
02137         ( i = m[1] - WILDOFFSET ) >= AM.OffsetVector ) {
02138             wc = 2;
02139 /*
02140             AN.argaddress = AT.MinVecArg;
02141             AT.MinVecArg[ARGHEAD+3] = t[1];
02142             if ( CheckWild(BHEAD i,VECTOSUB,1,AN.argaddress) ) return(0);
02143             AddWild(BHEAD i,VECTOSUB,(WORD)0);
02144 */
02145             if ( CheckWild(BHEAD i,VECTOMIN,t[1],&newvalue) ) return(0);
02146             AddWild(BHEAD i,VECTOMIN,newvalue);
02147 
02148         }
02149         else if ( *m == -MINVECTOR && *t == -VECTOR &&
02150         ( i = m[1] - WILDOFFSET ) >= AM.OffsetVector ) {
02151             wc = 2;
02152 /*
02153             AN.argaddress = AT.MinVecArg;
02154             AT.MinVecArg[ARGHEAD+3] = t[1];
02155             if ( CheckWild(BHEAD i,VECTOSUB,1,AN.argaddress) ) return(0);
02156             AddWild(BHEAD i,VECTOSUB,(WORD)0);
02157 */
02158             if ( CheckWild(BHEAD i,VECTOMIN,t[1],&newvalue) ) return(0);
```

```
02159             AddWild(BHEAD i,VECTOMIN,newvalue);
02160         }
02161         else return(0);
02162     }
02163 /*
02164     #] Both fast :
02165     #[ Fast arg :
02166 */
02167     else if ( *m > 0 && *t <= -FUNCTION ) {
02168         if ( ( m[ARGHEAD]+ARGHEAD == *m ) && m[*m-1] == 3
02169         && m[*m-2] == 1 && m[*m-3] == 1 && m[ARGHEAD+1] >= FUNCTION
02170         && m[ARGHEAD+2] == *m-ARGHEAD-4 ) { /* Check for f(?a) etc */
02171             WORD *mmmst, *mmm, mmmi;
02172             if ( m[ARGHEAD+1] >= FUNCTION+WILDOFFSET ) {
02173                 mmmi = *m - WILDOFFSET;
02174                 wc = 2;
02175                 if ( CheckWild(BHEAD mmmi,FUNTOFUN,-*t,&newvalue) ) return(0);
02176                 AddWild(BHEAD mmmi,FUNTOFUN,newvalue);
02177             }
02178             else if ( m[ARGHEAD+1] != -*t ) return(0);
02179 /*
02180                 Only arguments allowed are ?a etc.
02181 */
02182             mmmst = m+*m-3;
02183             mmm = m + ARGHEAD + FUNHEAD + 1;
02184             while ( mmm < mmmst ) {
02185                 if ( *mmm != -ARGWILD ) return(0);
02186                 mmmi = 0;
02187                 AN.argaddress = t; wc = 2;
02188                 if ( CheckWild(BHEAD mmm[1],ARGTOARG,mmmi,t) ) return(0);
02189                 AddWild(BHEAD mmm[1],ARGTOARG,mmmi);
02190                 mmm += 2;
02191             }
02192         }
02193         else return(0);
02194     }
02195 /*
02196     #] Fast arg :
02197     #[ Fast pat :
02198 */
02199     else if ( *m < 0 && *t > 0 ) {
02200         if ( *m == -SYMBOL ) {          /* SYMTOSUB */
02201             if ( m[1] < 2*MAXPOWER ) return(0);
02202             i = m[1] - 2*MAXPOWER;
02203             AN.argaddress = t; wc = 2;
02204             if ( CheckWild(BHEAD i,SYMTOSUB,1,AN.argaddress) ) return(0);
02205             AddWild(BHEAD i,SYMTOSUB,0);
02206         }
02207         else if ( *m == -VECTOR ) {
02208             if ( ( i = m[1] - WILDOFFSET ) < AM.OffsetVector ) return(0);
02209             AN.argaddress = t; wc = 2;
02210             if ( CheckWild(BHEAD i,VECTOSUB,1,t) ) return(0);
02211             AddWild(BHEAD i,VECTOSUB,(WORD)0);
02212         }
02213         else if ( *m == -INDEX ) {
02214             if ( ( i = m[1] - WILDOFFSET ) < AM.OffsetIndex ) return(0);
02215             if ( i >= AM.OffsetIndex + WILDOFFSET ) return(0);
02216             AN.argaddress = t; wc = 2;
02217             if ( CheckWild(BHEAD i,INDTOSUB,1,AN.argaddress) ) return(0);
02218             AddWild(BHEAD i,INDTOSUB,(WORD)0);
02219         }
02220         else return(0);
02221     }
02222 /*
02223     #] Fast pat :
02224     #[ Both general :
02225 */
02226     else if ( *m > 0 && *t > 0 ) {
02227         i = *m;
02228         do { if ( *m++ != *t++ ) break; } while ( --i > 0 );
02229         if ( i > 0 ) {
02230 /*
02231             Not an exact match here.
02232             We have to hope that the pattern contains a composite wildcard.
02233 */
02234             m = pat; t = arg;
02235             m += ARGHEAD; t += ARGHEAD;         /* Point at (first?) term */
02236             mtrmstop = m + *m;
02237             ttrmstop = t + *t;
02238             if ( mtrmstop < argmstop ) return(0);/* More than one term */
02239             msizcoef = mtrmstop[-1];
02240             if ( msizcoef < 0 ) msizcoef = -msizcoef;
02241             msubstop = mtrmstop - msizcoef;
02242             m++;
02243             if ( m >= msubstop ) return(0); /* Only coefficient */
02244 /*
02245             Here we have a composite term. It can match provided it
```

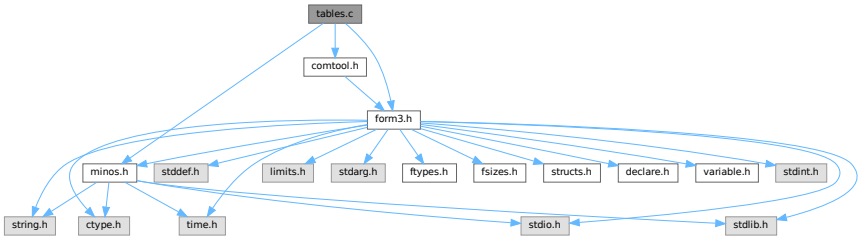
```
02246             matches the entire argument. This argument must be a
02247             single term also and the coefficients should match
02248             (more or less).
02249             The matching takes:
02250             1:  Match the functions etc. Nothing can be left.
02251             2:  Match dotproducts and symbols. ONLY must match
02252                 and nothing may be left.
02253             For safety it is best to take the term out and put it
02254             in workspace.
02255 */
02256             if ( argtstop > ttrmstop ) return(0);
02257             m--;
02258 
02259             oterstart = AN.terstart;
02260             oterstop = AN.terstop;
02261             opatstop = AN.patstop;
02262             oRepFunList = AN.RepFunList;
02263             oRepFunNum = AN.RepFunNum;
02264             AN.RepFunNum = 0;
02265             wildargtaken = AT.WorkPointer;
02266             AN.RepFunList = wildargtaken + AN.NumTotWildArgs;
02267             AT.WorkPointer = (WORD *)(((UBYTE *)(AN.RepFunList)) + AM.MaxTer/2);
02268             csav = cto = AT.WorkPointer;
02269             cfrom = t;
02270             ci = *t;
02271             while ( --ci >= 0 ) *cto++ = *cfrom++;
02272             AT.WorkPointer = cto;
02273             ci = msizcoef;
02274             cfrom = mtrmstop;
02275             while ( --ci >= 0 ) {
02276                 if ( *--cfrom != *--cto ) {
02277                     AT.WorkPointer = wildargtaken;
02278                     AN.RepFunList = oRepFunList;
02279                     AN.RepFunNum = oRepFunNum;
02280                     AN.terstart = oterstart;
02281                     AN.terstop = oterstop;
02282                     AN.patstop = opatstop;
02283                     return(0);
02284                 }
02285             }
02286             *m -= msizcoef;
02287             wildargs = AN.WildArgs;
02288             wildeat = AN.WildEat;
02289             for ( i = 0; i < wildargs; i++ ) wildargtaken[i] = AT.WildArgTaken[i];
02290             AN.ForFindOnly = 0; AN.UseFindOnly = 1;
02291             AN.nogroundlevel++;
02292             if ( FindRest(BHEAD csav,m) && ( AN.UsedOtherFind || FindOnly(BHEAD csav,m) ) ) { }
02293             else {
02294                 *m += msizcoef;
02295                 AT.WorkPointer = wildargtaken;
02296                 AN.RepFunList = oRepFunList;
02297                 AN.RepFunNum = oRepFunNum;
02298                 AN.terstart = oterstart;
02299                 AN.terstop = oterstop;
02300                 AN.patstop = opatstop;
02301                 AN.WildArgs = wildargs;
02302                 AN.WildEat = wildeat;
02303                 AN.nogroundlevel--;
02304                 for ( i = 0; i < wildargs; i++ ) AT.WildArgTaken[i] = wildargtaken[i];
02305                 return(0);
02306             }
02307             AN.nogroundlevel--;
02308             AN.WildArgs = wildargs;
02309             AN.WildEat = wildeat;
02310             for ( i = 0; i < wildargs; i++ ) AT.WildArgTaken[i] = wildargtaken[i];
02311             Substitute(BHEAD csav,m,1);
02312             cto = csav;
02313             cfrom = cto + *cto - msizcoef;
02314             cto++;
02315             *m += msizcoef;
02316             AT.WorkPointer = wildargtaken;
02317             AN.RepFunList = oRepFunList;
02318             AN.RepFunNum = oRepFunNum;
02319             AN.terstart = oterstart;
02320             AN.terstop = oterstop;
02321             AN.patstop = opatstop;
02322             if ( *cto != SUBEXPRESSION ) return(0);
02323             cto += cto[1];
02324             if ( cto < cfrom ) return(0);
02325         }
02326     }
02327 /*
02328     #] Both general :
02329 */
02330     else return(0);
02331 /*
02332     And now the success: (wc = 2 means that there was a wildcard involved)
```

```
02333 */
02334     return(wc);
02335 }
02336
02337 /*
02338     #] MatchArgument :
02339 */
```

4.143 tables.c File Reference

```
#include "form3.h"
#include "minos.h"
#include "comtool.h"
```

Include dependency graph for tables.c:



Functions

- void ClearTableTree (TABLES T)
- int InsTableTree (TABLES T, WORD *tp)
- void RedoTableTree (TABLES T, int newsize)
- int FindTableTree (TABLES T, WORD *tp, int inc)
- int DoTableExpansion (WORD *term, WORD level)
- int CoTableBase (UBYTE *s)
- int FlipTable (FUNCTIONS f, int type)
- int SpareTable (TABLES TT)
- DBASE * FindTB (UBYTE *name)
- int CoTBcreate (UBYTE *s)
- int CoTBopen (UBYTE *s)
- int CoTBaddto (UBYTE *s)
- int CoTBenter (UBYTE *s)
- int CoTestUse (UBYTE *s)
- int CheckTableDeclarations (DBASE *d)
- int CoTBload (UBYTE **ss)
- int TestUse (WORD *term, WORD level)
- int CoTBaudit (UBYTE *s)
- int CoTBon (UBYTE *s)
- int CoTBoff (UBYTE *s)
- int CoTBcleanup (UBYTE *s)
- int CoTBreplace (UBYTE *s)
- int CoTBuse (UBYTE *s)
- int CoApply (UBYTE *s)
- int CoTBhelp (UBYTE *s)
- void ReWorkT (WORD *term, WORD *funs, WORD numfuns)

- void Apply (WORD *term, WORD level)
- int ApplyExec (WORD *term, int maxtogo, WORD level)
- void ApplyReset (WORD level)
- void TableReset (void)
- int ReleaseTB (void)

Variables

- char * helptb [ ]

4.143.1 Detailed Description

Contains all functions that deal with the table bases on the 'FORM level' The low level database routines are in minos.c

Definition in file tables.c.

4.143.2 Function Documentation

4.143.2.1 ClearTableTree()

```
void ClearTableTree (
             TABLES T )
```

Definition at line 72 of file tables.c.

4.143.2.2 InsTableTree()

```
int InsTableTree (
             TABLES T,
             WORD * tp )
```

Definition at line 103 of file tables.c.

4.143.2.3 RedoTableTree()

```
void RedoTableTree (
             TABLES T,
             int newsize )
```

Definition at line 268 of file tables.c.

4.143.2.4 FindTableTree()

```
int FindTableTree (
             TABLES T,
             WORD * tp,
             int inc )
```

Definition at line 293 of file tables.c.

4.143.2.5 DoTableExpansion()

```
int DoTableExpansion (
            WORD * term,
            WORD level )
```

Definition at line 338 of file tables.c.

4.143.2.6 CoTableBase()

```
int CoTableBase (
            UBYTE * s )
```

Definition at line 633 of file tables.c.

4.143.2.7 FlipTable()

```
int FlipTable (
            FUNCTIONS f,
            int type )
```

Definition at line 677 of file tables.c.

4.143.2.8 SpareTable()

```
int SpareTable (
            TABLES TT )
```

Definition at line 700 of file tables.c.

4.143.2.9 FindTB()

```
DBASE * FindTB (
            UBYTE * name )
```

Definition at line 740 of file tables.c.

4.143.2.10 CoTBcreate()

```
int CoTBcreate (
            UBYTE * s )
```

Definition at line 762 of file tables.c.

4.143.2.11 CoTBopen()

```
int CoTBopen (
            UBYTE * s )
```

Definition at line 778 of file tables.c.

4.143.2.12 CoTBaddto()

```
int CoTBaddto (
            UBYTE * s )
```

Definition at line 807 of file tables.c.

4.143.2.13 CoTBenter()

```
int CoTBenter (
            UBYTE * s )
```

Definition at line 980 of file tables.c.

4.143.2.14 CoTestUse()

```
int CoTestUse (
            UBYTE * s )
```

Definition at line 1139 of file tables.c.

4.143.2.15 CheckTableDeclarations()

```
int CheckTableDeclarations (
            DBASE * d )
```

Definition at line 1184 of file tables.c.

4.143.2.16 CoTBload()

```
int CoTBload (
            UBYTE * ss )
```

Definition at line 1258 of file tables.c.

4.143.2.17 TestUse()

```
int TestUse (
            WORD * term,
            WORD level )
```

Definition at line 1420 of file tables.c.

4.143.2.18 CoTBaudit()

```
int CoTBaudit (
            UBYTE * s )
```

Definition at line 1482 of file tables.c.

4.143.2.19 CoTBon()

```
int CoTBon (
            UBYTE * s )
```

Definition at line 1522 of file tables.c.

4.143.2.20 CoTBoff()

```
int CoTBoff (
            UBYTE * s )
```

Definition at line 1553 of file tables.c.

4.143.2.21 CoTBcleanup()

```
int CoTBcleanup (
            UBYTE * s )
```

Definition at line 1584 of file tables.c.

4.143.2.22 CoTBreplace()

```
int CoTBreplace (
            UBYTE * s )
```

Definition at line 1596 of file tables.c.

4.143.2.23 CoTBuse()

```
int CoTBuse (
            UBYTE * s )
```

Definition at line 1611 of file tables.c.

4.143.2.24 CoApply()

```
int CoApply (
            UBYTE * s )
```

Definition at line 1809 of file tables.c.

4.143.2.25 CoTBhelp()

```
int CoTBhelp (
            UBYTE * s )
```

Definition at line 1896 of file tables.c.

4.143.2.26 ReWorkT()

```
void ReWorkT (
             WORD * term,
             WORD * funs,
             WORD numfuns )
```

Definition at line 1912 of file tables.c.

4.143.2.27 Apply()

```
void Apply (
             WORD * term,
             WORD level )
```

Definition at line 1993 of file tables.c.

4.143.2.28 ApplyExec()

```
int ApplyExec (
             WORD * term,
             int maxtogo,
             WORD level )
```

Definition at line 2051 of file tables.c.

4.143.2.29 ApplyReset()

```
void ApplyReset (
             WORD level )
```

Definition at line 2268 of file tables.c.

4.143.2.30 TableReset()

```
void TableReset (
             void  )
```

Definition at line 2303 of file tables.c.

4.143.2.31 ReleaseTB()

```
int ReleaseTB (
             void  )
```

Definition at line 2329 of file tables.c.

4.143.3 Variable Documentation

4.143.3.1 helptb

```
char* helptb[]
```

Definition at line 1864 of file tables.c.

4.144 tables.c

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032 /*
00033     #[ Includes :
00034
00035     File contains the routines for the tree structure of sparse tables
00036     We insert elements by
00037     InsTableTree(T,tp) with T the TABLES element and tp the pointer
00038         to the indices.
00039     We look for elements with
00040     FindTableTree(T,tp,inc) with T the TABLES element, tp the pointer to the
00041         indices or the function arguments and inc tells which of these options.
00042     The tree is cleared with ClearTableTree(T) and we rebuild the tree
00043     after a .store in which we lost a part of the table with
00044     RedoTableTree(T,newsize)
00045
00046     In T->tablepointers we have the lists of indices for each element.
00047     Additionally for each element there is an extension. There are
00048     TABLEEXTENSION WORDs reserved for that. The old system had two words
00049     One for the element in the rhs of the compile buffer and one for
00050     an additional rhs in case the original would be overwritten by a new
00051     definition, but the old was fixed by .global and hence it should be possible
00052     to restore it.
00053     New use (new = 24-sep-2001)
00054         rhs1,numCompBuffer1,rhs2,numCompBuffer2,usage
00055     Hence TABLEEXTENSION will be 5. Note that for 64 bits the use of the
00056     compiler buffer is overdoing it a bit, but it would be too complicated
00057     to try to give it special code.
00058 */
00059
00060 #include "form3.h"
00061 #include "minos.h"
00062 #include "comtool.h"
00063
00064 /* static UBYTE *sparse = (UBYTE *)"sparse"; */
00065 static UBYTE *tablebase = (UBYTE *)"tablebase";
00066
00067 /*
00068     #] Includes :
00069     #[ ClearTableTree :
00070 */
```

```
00071
00072 void ClearTableTree(TABLES T)
00073 {
00074     COMPTREE *root;
00075     if ( T->boomlijst == 0 ) {
00076         T->MaxTreeSize = 125;
00077         T->boomlijst = (COMPTREE *)Malloc1(T->MaxTreeSize*sizeof(COMPTREE),
00078                 "ClearTableTree");
00079     }
00080     root = T->boomlijst;
00081     T->numtree = 0;
00082     T->rootnum = 0;
00083     root->left = -1;
00084     root->right = -1;
00085     root->parent = -1;
00086     root->blnce = 0;
00087     root->value = -1;
00088     root->usage = 0;
00089 }
00090
00091 /*
00092     #] ClearTableTree :
00093     #[ InsTableTree :
00094
00095     int InsTableTree(TABLES T,WORD *,arglist)
00096         Searches for the element specified by the list of arguments.
00097         If found, it returns -(the offset in T->tablepointers)
00098         If not found, it will allocate a new element, balance the tree if
00099         necessary and return the number of the element in the boomlijst
00100         This number is always > 0, because we start from 1.
00101 */
00102
00103 int InsTableTree(TABLES T, WORD *tp)
00104 {
00105     COMPTREE *boomlijst, *q, *p, *s;
00106     WORD *v1, *v2, *v3, xstop;
00107     int ip, iq, is;
00108     if ( T->numtree + 1 >= T->MaxTreeSize ) {
00109         if ( T->MaxTreeSize == 0 ) ClearTableTree(T);
00110         else {
00111             is = T->MaxTreeSize * 2;
00112             s  = (COMPTREE *)Malloc1(is*sizeof(COMPTREE),"InsTableTree");
00113             for ( ip = 0; ip < T->MaxTreeSize; ip++ ) { s[ip] = T->boomlijst[ip]; }
00114             if ( T->boomlijst ) M_free(T->boomlijst,"InsTableTree");
00115             T->boomlijst = s;
00116             T->MaxTreeSize = is;
00117         }
00118     }
00119     boomlijst = T->boomlijst;
00120     q = boomlijst + T->rootnum;
00121     if ( q->right == -1 ) { /* First element */
00122         T->numtree++;
00123         s = boomlijst+T->numtree;
00124         q->right = T->numtree;
00125         s->parent = T->rootnum;
00126         s->left = s->right = -1;
00127         s->blnce = 0;
00128         s->value = tp - T->tablepointers;
00129         s->usage = 0;
00130         return(T->numtree);
00131     }
00132     ip = q->right;
00133     if ( T->numind >= 0 ) xstop = T->numind;
00134     else xstop = *tp + 1;
00135     while ( ip >= 0 ) {
00136         p = boomlijst + ip;
00137         v1 = T->tablepointers + p->value;
00138         v2 = tp; v3 = tp + xstop;
00139         while ( *v1 == *v2 && v2 < v3 ) { v1++; v2++; }
00140         if ( v2 >= v3 ) return(-p->value);
00141         if ( *v1 > *v2 ) {
00142             iq = p->right;
00143             if ( iq >= 0 ) { ip = iq; }
00144             else {
00145                 T->numtree++;
00146                 is = T->numtree;
00147                 p->right = is;
00148                 s = boomlijst + is;
00149                 s->parent = ip; s->left = s->right = -1;
00150                 s->blnce = 0;   s->value = tp - T->tablepointers;
00151                 s->usage = 0;
00152                 p->blnce++;
00153                 if ( p->blnce == 0 ) return(T->numtree);
00154                 goto balance;
00155             }
00156         }
00157         else if ( *v1 < *v2 ) {
```

```
00158             iq = p->left;
00159             if ( iq >= 0 ) { ip = iq; }
00160             else {
00161                 T->numtree++;
00162                 is = T->numtree;
00163                 s = boomlijst+is;
00164                 p->left = is;
00165                 s->parent = ip; s->left = s->right = -1;
00166                 s->blnce = 0;   s->value = tp - T->tablepointers;
00167                 s->usage = 0;
00168                 p->blnce--;
00169                 if ( p->blnce == 0 ) return(T->numtree);
00170                 goto balance;
00171             }
00172         }
00173     }
00174 /* INTERNAL_ERROR_EXCL_START */
00175     MesPrint("!>Serious problems in InsTableTree!\n");
00176     Terminate(-1);
00177     return(0);
00178 /* INTERNAL_ERROR_EXCL_STOP */
00179 balance:;
00180     for (;;) {
00181         p = boomlijst + ip;
00182         iq = p->parent;
00183         if ( iq == T->rootnum ) break;
00184         q = boomlijst + iq;
00185         if ( ip == q->left ) q->blnce--;
00186         else                 q->blnce++;
00187         if ( q->blnce == 0 ) break;
00188         if ( q->blnce == -2 ) {
00189             if ( p->blnce == -1 ) { /* single rotation */
00190                 q->left = p->right;
00191                 p->right = iq;
00192                 p->parent = q->parent;
00193                 q->parent = ip;
00194                 if ( boomlijst[p->parent].left == iq ) boomlijst[p->parent].left = ip;
00195                 else                                   boomlijst[p->parent].right = ip;
00196                 if ( q->left >= 0 ) boomlijst[q->left].parent = iq;
00197                 q->blnce = p->blnce = 0;
00198             }
00199             else {  /* double rotation */
00200                 s = boomlijst + is;
00201                 q->left = s->right;
00202                 p->right = s->left;
00203                 s->right = iq;
00204                 s->left = ip;
00205                 if ( p->right >= 0 ) boomlijst[p->right].parent = ip;
00206                 if ( q->left >= 0 ) boomlijst[q->left].parent = iq;
00207                 s->parent = q->parent;
00208                 q->parent = is;
00209                 p->parent = is;
00210                 if ( boomlijst[s->parent].left == iq )
00211                        boomlijst[s->parent].left = is;
00212                 else   boomlijst[s->parent].right = is;
00213                 if ( s->blnce > 0 ) { q->blnce = s->blnce = 0; p->blnce = -1; }
00214                 else if ( s->blnce < 0 ) { p->blnce = s->blnce = 0; q->blnce = 1; }
00215                 else { p->blnce = s->blnce = q->blnce = 0; }
00216             }
00217             break;
00218         }
00219         else if ( q->blnce == 2 ) {
00220             if ( p->blnce == 1 ) {  /* single rotation */
00221                 q->right = p->left;
00222                 p->left = iq;
00223                 p->parent = q->parent;
00224                 q->parent = ip;
00225                 if ( boomlijst[p->parent].left == iq ) boomlijst[p->parent].left = ip;
00226                 else                                   boomlijst[p->parent].right = ip;
00227                 if ( q->right >= 0 ) boomlijst[q->right].parent = iq;
00228                 q->blnce = p->blnce = 0;
00229             }
00230             else {  /* double rotation */
00231                 s = boomlijst + is;
00232                 q->right = s->left;
00233                 p->left = s->right;
00234                 s->left = iq;
00235                 s->right = ip;
00236                 if ( p->left >= 0 ) boomlijst[p->left].parent = ip;
00237                 if ( q->right >= 0 ) boomlijst[q->right].parent = iq;
00238                 s->parent = q->parent;
00239                 q->parent = is;
00240                 p->parent = is;
00241                 if ( boomlijst[s->parent].left == iq ) boomlijst[s->parent].left = is;
00242                 else                                   boomlijst[s->parent].right = is;
00243                 if ( s->blnce < 0 ) { q->blnce = s->blnce = 0; p->blnce = 1; }
00244                 else if ( s->blnce > 0 ) { p->blnce = s->blnce = 0; q->blnce = -1; }
```

```
00245                 else { p->blnce = s->blnce = q->blnce = 0; }
00246             }
00247             break;
00248         }
00249         is = ip; ip = iq;
00250     }
00251     return(T->numtree);
00252 }
00253
00254 /*
00255     #] InsTableTree :
00256     #[ RedoTableTree :
00257
00258     To be used when a sparse table is trimmed due to a .store
00259     We rebuild the tree. In the future one could try to become faster
00260     at the cost of quite some complexity.
00261     We need to keep the first 'size' elements in the boomlijst.
00262     Kill all others and reconstruct the tree with the original ordering.
00263     This is very complicated! Because .store will either keep the whole
00264     table or remove the whole table we should not come here often.
00265     Hence we choose the slow solution for now.
00266 */
00267
00268 void RedoTableTree(TABLES T, int newsize)
00269 {
00270     WORD *tp;
00271     int i;
00272     ClearTableTree(T);
00273     for ( i = 0, tp = T->tablepointers; i < newsize; i++ ) {
00274         InsTableTree(T,tp);
00275         tp += ABS(T->numind)+TABLEEXTENSION;
00276     }
00277 }
00278
00279 /*
00280     #] RedoTableTree :
00281     #[ FindTableTree :
00282
00283     int FindTableTree(TABLES T,WORD *,arglist,int,inc)
00284         Searches for the element specified by the list of arguments.
00285         If found, it returns the offset in T->tablepointers
00286         If not found, it will return -1
00287         The list here is from the list of function arguments. Hence it
00288         has pairs of numbers -SNUMBER,index
00289         Actually inc says how many numbers there are and the above case is
00290         for inc = 2. For inc = 1 we have just a list of indices.
00291 */
00292
00293 int FindTableTree(TABLES T, WORD *tp, int inc)
00294 {
00295     COMPTREE *boomlijst = T->boomlijst, *q = boomlijst + T->rootnum, *p;
00296     WORD *v1, *v2, *v3, xstop;
00297     int ip, iq;
00298     if ( q->right == -1 ) return(-1);
00299     ip = q->right;
00300     if ( inc > 1 ) tp += inc-1;
00301     if ( T->numind >= 0 ) xstop = T->numind;
00302     else {  /* We have to read the number of arguments first */
00303         if ( *tp <= 0 ) return(-1); /* Cannot be! */
00304         xstop = *tp+1;
00305     }
00306     while ( ip >= 0 ) {
00307         p = boomlijst + ip;
00308         v1 = T->tablepointers + p->value;
00309         v2 = tp; v3 = v1 + xstop;
00310         while ( *v1 == *v2 && v1 < v3 ) { v1++; v2 += inc; }
00311         if ( v1 == v3 ) {
00312             p->usage++;
00313             return(p->value);
00314         }
00315         if ( *v1 > *v2 ) {
00316             iq = p->right;
00317             if ( iq >= 0 ) { ip = iq; }
00318             else return(-1);
00319         }
00320         else if ( *v1 < *v2 ) {
00321             iq = p->left;
00322             if ( iq >= 0 ) { ip = iq; }
00323             else return(-1);
00324         }
00325     }
00326 /* INTERNAL_ERROR_EXCL_START */
00327     MesPrint("!>Serious problems in FindTableTree\n");
00328     Terminate(-1);
00329     return(-1);
00330 /* INTERNAL_ERROR_EXCL_STOP */
00331 }
```

```
00332
00333 /*
00334     #] FindTableTree :
00335     #[ DoTableExpansion :
00336 */
00337
00338 int DoTableExpansion(WORD *term, WORD level)
00339 {
00340     GETIDENTITY
00341     WORD *t, *tstop, *stopper, *termout, *m, *mm, *tp, *r, xx;
00342     WORD numsubexp, numbuf;
00343     TABLES T = 0;
00344     int i, j, num;
00345     AN.TeInFun = AR.TePos = 0;
00346     tstop = term + *term;
00347     stopper = tstop - ABS(tstop[-1]);
00348     t = term+1;
00349     while ( t < stopper ) {
00350         if ( *t != TABLEFUNCTION ) { t += t[1]; continue; }
00351         if ( t[FUNHEAD] > -FUNCTION ) { t += t[1]; continue; }
00352         T = functions[-t[FUNHEAD]-FUNCTION].tabl;
00353         if ( T == 0 ) { t += t[1]; continue; }
00354         if ( T->spare ) T = T->spare;
00355         if ( t[1] == FUNHEAD+2 && t[FUNHEAD+1] <= -FUNCTION ) break;
00356         if ( t[1] < FUNHEAD+1+2*ABS(T->numind) ) { t += t[1]; continue; }
00357         for ( i = 0; i < ABS(T->numind); i++ ) {
00358             if ( t[FUNHEAD+1+2*i] != -SYMBOL ) break;
00359         }
00360         if ( i >= ABS(T->numind) ) break;
00361         t += t[1];
00362     }
00363     if ( t >= stopper ) {
00364 /* INTERNAL_ERROR_EXCL_START */
00365         MesPrint("!>Internal error: Missing table_ function");
00366         Terminate(-1);
00367 /* INTERNAL_ERROR_EXCL_STOP */
00368     }
00369 /*
00370     Table in T. Now collect the numbers of the symbols;
00371 */
00372     termout = AT.WorkPointer;
00373     if ( T->sparse ) {
00374         for ( i = 0; i < T->totind; i++ ) {
00375 /*
00376             Loop over all table elements
00377 */
00378             m = termout + 1; mm = term + 1;
00379             while ( mm < t ) *m++ = *mm++;
00380             r = m;
00381             if ( t[1] == FUNHEAD+2 && t[FUNHEAD+1] <= -FUNCTION ) {
00382                 *m++ = -t[FUNHEAD+1];
00383                 tp = T->tablepointers + (ABS(T->numind)+TABLEEXTENSION)*i;
00384                 if ( T->numind < 0 ) {
00385                     xx = tp[0]+1;
00386                     *m++ = FUNHEAD+xx*2;
00387                     for ( j = 2; j < FUNHEAD; j++ ) *m++ = 0;
00388                     for ( j = 0; j < xx; j++ ) {
00389                         *m++ = -SNUMBER; *m++ = *tp++;
00390                     }
00391                 }
00392                 else {
00393                     *m++ = FUNHEAD+T->numind*2;
00394                     for ( j = 2; j < FUNHEAD; j++ ) *m++ = 0;
00395                     for ( j = 0; j < T->numind; j++ ) {
00396                         *m++ = -SNUMBER; *m++ = *tp++;
00397                     }
00398                 }
00399             }
00400             else if ( T->numind < 0 ) {
00401                 tp = T->tablepointers + (ABS(T->numind)+TABLEEXTENSION)*i;
00402                 xx = tp[0]+1;
00403                 *m++ = SYMBOL; *m++ = 2+xx*2; mm = t + FUNHEAD+1;
00404                 for ( j = 0; j < xx; j++, mm += 2, tp++ ) {
00405                     if ( *tp != 0 ) { *m++ = mm[1]; *m++ = *tp; }
00406                 }
00407                 r[1] = m-r;
00408                 if ( r[1] == 2 ) m = r;
00409             }
00410             else {
00411                 *m++ = SYMBOL; *m++ = 2+T->numind*2; mm = t + FUNHEAD+1;
00412                 tp = T->tablepointers + (T->numind+TABLEEXTENSION)*i;
00413                 for ( j = 0; j < T->numind; j++, mm += 2, tp++ ) {
00414                     if ( *tp != 0 ) { *m++ = mm[1]; *m++ = *tp; }
00415                 }
00416                 r[1] = m-r;
00417                 if ( r[1] == 2 ) m = r;
00418             }
```

```
00419 /*
00420             The next code replaces this old code
00421
00422             *m++ = SUBEXPRESSION;
00423             *m++ = SUBEXPSIZE;
00424             *m++ = *tp;
00425             *m++ = 1;
00426             *m++ = T->bufnum;
00427             FILLSUB(m);
00428             mm = t + t[1];
00429
00430             We had forgotten to take the parameters into account.
00431             Hence the subexpression prototype for wildcards was missed
00432             Now we slow things down a little bit, but we do not run
00433             any risks. There is still one problem. We have not checked
00434             that the prototype matches.
00435 */
00436             tp = T->tablepointers + (ABS(T->numind)+TABLEEXTENSION)*i
00437                     +ABS(T->numind);
00438             numsubexp = tp[0]; numbuf = tp[1];
00439             r = m;
00440 #ifdef WITHPTHREADS
00441             tp = T->prototype[identity];
00442 #else
00443             tp = T->prototype;
00444 #endif
00445             for ( j = 0; j < tp[1]; j++ ) *m++ = tp[j];
00446             r[2] = numsubexp; r[4] = numbuf;
00447 /*
00448             r = m;
00449             tp = T->tablepointers + (ABS(T->numind)+TABLEEXTENSION)*i;
00450             *m++ = -t[FUNHEAD];
00451             if ( T->numind < 0 ) {
00452                 xx = tp[0]+1;
00453                 *m++ = t[1] - xx - T->numind - 1;
00454             }
00455             else {
00456                 xx = T->numind;
00457                 *m++ = t[1] - 1;
00458             }
00459             for ( j = 2; j < FUNHEAD; j++ ) *m++ = t[j];
00460             for ( j = 0; j < xx; j++ ) {
00461                 *m++ = -SNUMBER; *m++ = *tp++;
00462             }
00463             tp = t + FUNHEAD + 1 + 2*T->numind;
00464             mm = t + t[1];
00465             while ( tp < mm ) *m++ = *tp++;
00466             r[1] = m-r;
00467 */
00468 /*
00469             From now on is old code
00470 */
00471             mm = t + t[1];
00472             while ( mm < tstop ) *m++ = *mm++;
00473             *termout = m - termout;
00474             AT.WorkPointer = m;
00475             if ( Generator(BHEAD termout,level) ) {
00476                 MesCall("DoTableExpand");
00477                 return(-1);
00478             }
00479             AT.WorkPointer = termout;
00480         }
00481     }
00482     else {
00483         for ( i = 0; i < T->totind; i++ ) {
00484 #if TABLEEXTENSION == 2
00485             if ( T->tablepointers[i] < 0 ) continue;
00486 #else
00487             if ( T->tablepointers[TABLEEXTENSION*i] < 0 ) continue;
00488 #endif
00489             m = termout + 1; mm = term + 1;
00490             while ( mm < t ) *m++ = *mm++;
00491             r = m;
00492             if ( t[1] == FUNHEAD+2 && t[FUNHEAD+1] <= -FUNCTION ) {
00493                 *m++ = -t[FUNHEAD+1];
00494                 *m++ = FUNHEAD+T->numind*2;
00495                 for ( j = 2; j < FUNHEAD; j++ ) *m++ = 0;
00496                 tp = T->tablepointers + (T->numind+TABLEEXTENSION)*i;
00497                 for ( j = 0; j < T->numind; j++ ) {
00498                     if ( j > 0 ) {
00499                         num = T->mm[j].mini + ( i % T->mm[j-1].size ) / T->mm[j].size;
00500                     }
00501                     else {
00502                         num = T->mm[j].mini + i / T->mm[j].size;
00503                     }
00504                     *m++ = -SNUMBER; *m++ = num;
00505                 }
```

```
00506             }
00507             else {
00508                 *m++ = SYMBOL; *m++ = 2+T->numind*2; mm = t + FUNHEAD+1;
00509                 for ( j = 0; j < T->numind; j++, mm += 2 ) {
00510                     if ( j > 0 ) {
00511                         num = T->mm[j].mini + ( i % T->mm[j-1].size ) / T->mm[j].size;
00512                     }
00513                     else {
00514                         num = T->mm[j].mini + i / T->mm[j].size;
00515                     }
00516                     if ( num != 0 ) { *m++ = mm[1]; *m++ = num; }
00517                 }
00518                 r[1] = m-r;
00519                 if ( r[1] == 2 ) m = r;
00520             }
00521 /*
00522             The next code replaces this old code
00523
00524             *m++ = SUBEXPRESSION;
00525             *m++ = SUBEXPSIZE;
00526             *m++ = *tp;
00527             *m++ = 1;
00528             *m++ = T->bufnum;
00529             FILLSUB(m);
00530             mm = t + t[1];
00531
00532             We had forgotten to take the parameters into account.
00533             Hence the subexpression prototype for wildcards was missed
00534             Now we slow things down a little bit, but we do not run
00535             any risks. There is still one problem. We have not checked
00536             that the prototype matches.
00537 */
00538             r = m;
00539             *m++ = -t[FUNHEAD];
00540             *m++ = t[1] - 1;
00541             for ( j = 2; j < FUNHEAD; j++ ) *m++ = t[j];
00542             for ( j = 0; j < T->numind; j++ ) {
00543                 if ( j > 0 ) {
00544                     num = T->mm[j].mini + ( i % T->mm[j-1].size ) / T->mm[j].size;
00545                 }
00546                 else {
00547                     num = T->mm[j].mini + i / T->mm[j].size;
00548                 }
00549                 *m++ = -SNUMBER; *m++ = num;
00550             }
00551             tp = t + FUNHEAD + 1 + 2*T->numind;
00552             mm = t + t[1];
00553             while ( tp < mm ) *m++ = *tp++;
00554             r[1] = m - r;
00555 /*
00556             From now on is old code
00557 */
00558             while ( mm < tstop ) *m++ = *mm++;
00559             *termout = m - termout;
00560             AT.WorkPointer = m;
00561             if ( Generator(BHEAD termout,level) ) {
00562                 MesCall("DoTableExpand");
00563                 return(-1);
00564             }
00565         }
00566     }
00567     return(0);
00568 }
00569
00570 /*
00571     #] DoTableExpansion :
00572     #[ TableBase :
00573
00574     File with all the database related things.
00575     We have the routines for the generic database command
00576     TableBase,options;
00577     TB,options;
00578     Options are:
00579         Open "File.tbl";                    Open for R/W
00580         Open "File.tbl", readonly;          Open for R
00581         Create "File.tbl";                  Create for write
00582         Load "File.tbl", tablename;         Loads stubs of table
00583         Load "File.tbl";                    Loads stubs of all tables
00584         Enter "File.tbl", tablename;        Loads whole table
00585         Enter "File.tbl";                   Loads all tables
00586         Audit "File.tbl", options;          Print list of contents
00587         Replace "File.tbl", tablename;      Saves a table (with overwrite)
00588         Replace "File.tbl", table element; Saves a table element   „
00589         Cleanup "File.tbl";                 Makes tables contingent
00590         AddTo "File.tbl" tablename;         Add if not yet there.
00591         AddTo "File.tbl" table element;     Add if not yet there.
00592         Delete "File.tbl" tablename;
```

```
00593         Delete "File.tbl" table element;
00594
00595         On/Off substitute;
00596         On/Off compress "File.tbl";
00597     id tbl_(f?,?a) = f(?a);
00598     When a tbl_ is used, automatically the corresponding element is compiled
00599     at the start of the next module.
00600     if TB,On,substitute [tablename], use of table RHS (if loaded)
00601     if TB,Off,substitute [tablename], use of tbl_(table,...);
00602
00603
00604     Still needed: Something like OverLoad to allow loading parts of a table
00605     from more than one file. Date stamps needed? In that case we need a touch
00606     command as well.
00607
00608     If we put all our diagrams inside, we have to go outside the concept
00609     of tables.
00610
00611     #] TableBase :
00612     #[ CoTableBase :
00613
00614     To be followed by ,subkey
00615 */
00616 static KEYWORD tboptions[] = {
00617      {"addto",           (TFUN)CoTBaddto,      0,    PARTEST}
00618     ,{"audit",           (TFUN)CoTBaudit,      0,    PARTEST}
00619     ,{"cleanup",         (TFUN)CoTBcleanup,    0,    PARTEST}
00620     ,{"create",          (TFUN)CoTBcreate,     0,    PARTEST}
00621     ,{"enter",           (TFUN)CoTBenter,      0,    PARTEST}
00622     ,{"help",            (TFUN)CoTBhelp,       0,    PARTEST}
00623     ,{"load",            (TFUN)CoTBload,       0,    PARTEST}
00624     ,{"off",             (TFUN)CoTBoff,        0,    PARTEST}
00625     ,{"on",              (TFUN)CoTBon,         0,    PARTEST}
00626     ,{"open",            (TFUN)CoTBopen,       0,    PARTEST}
00627     ,{"replace",         (TFUN)CoTBreplace,    0,    PARTEST}
00628     ,{"use",             (TFUN)CoTBuse,        0,    PARTEST}
00629 };
00630
00631 static UBYTE *tablebasename = 0;
00632
00633 int CoTableBase(UBYTE *s)
00634 {
00635     UBYTE *option, c, *t;
00636     int i,optlistsize = sizeof(tboptions)/sizeof(KEYWORD), error = 0;
00637     while ( *s == ' ' ) s++;
00638     if ( *s != '"' ) {
00639         if ( ( tolower(*s) == 'h' ) && ( tolower(s[1]) == 'e' )
00640          && ( tolower(s[2]) == 'l' ) && ( tolower(s[3]) == 'p' )
00641          && ( FG.cTable[s[4]] > 1 ) ) {
00642            CoTBhelp(s);
00643            return(0);
00644         }
00645 proper:;
00646         MesPrint("&Proper syntax: TableBase \"filename\" options");
00647         return(1);
00648     }
00649     s++; tablebasename = s;
00650     while ( *s && *s != '"' ) s++;
00651     if ( *s != '"' ) goto proper;
00652     t = s; s++; *t = 0;
00653     while ( *s == ' ' || *s == '\t' || *s == ',' ) s++;
00654     option = s;
00655     while ( FG.cTable[*s] == 0 ) s++;
00656     c = *s; *s = 0;
00657     for ( i = 0; i < optlistsize; i++ ) {
00658         if ( StrICmp(option,(UBYTE *)(tboptions[i].name)) == 0 ) {
00659             *s = c;
00660             while ( *s == ',' ) s++;
00661             error = (tboptions[i].func)(s);
00662             *t = '"';
00663             return(error);
00664         }
00665     }
00666     MesPrint("&Unrecognized option %s in TableBase statement",option);
00667     return(1);
00668 }
00669
00670 /*
00671     #] CoTableBase :
00672     #[ FlipTable :
00673
00674     Flips the table between use as 'stub' and regular use
00675 */
00676
00677 int FlipTable(FUNCTIONS f, int type)
00678 {
00679     TABLES T, TT;
```

```
00680     T = f->tabl;
00681     if ( ( TT = T->spare ) == 0 ) {
00682         MesPrint("Error: trying to change mode on a table that has no tablebase");
00683         return(-1);
00684     }
00685     if ( TT->mode == type ) f->tabl = TT;
00686     return(0);
00687 }
00688
00689 /*
00690     #] FlipTable :
00691     #[ SpareTable :
00692
00693     Creates a spare element for a table. This is used in the table bases.
00694     It is a (thus far) empty copy of the TT table.
00695     By using FlipTable we can switch between them and alter which version of
00696     a table we will be using. Note that this also causes some extra work in the
00697     ResetVariables and the Globalize routines.
00698 */
00699
00700 int SpareTable(TABLES TT)
00701 {
00702     TABLES T;
00703     T = (TABLES)Malloc1(sizeof(struct TaBlEs),"table");
00704     T->defined = T->mdefined = 0; T->sparse = TT->sparse; T->mm = 0; T->flags = 0;
00705     T->numtree = 0; T->rootnum = 0; T->MaxTreeSize = 0;
00706     T->boomlijst = 0;
00707     T->strict = TT->strict;
00708     T->bounds = TT->bounds;
00709     T->bufnum = inicbufs();
00710     T->argtail = TT->argtail;
00711     T->spare = TT;
00712     T->bufferssize = 8;
00713     T->buffers = (WORD *)Malloc1(sizeof(WORD)*T->bufferssize,"SpareTable buffers");
00714     T->buffersfill = 0;
00715     T->buffers[T->buffersfill++] = T->bufnum;
00716     T->mode = 0;
00717     T->numind = TT->numind;
00718     T->totind = 0;
00719     T->prototype = TT->prototype;
00720     T->pattern = TT->pattern;
00721     T->tablepointers = 0;
00722     T->reserved = 0;
00723     T->tablenum = 0;
00724     T->numdummies = 0;
00725     T->mm = (MINMAX *)Malloc1(ABS(T->numind)*sizeof(MINMAX),"table dimensions");
00726     T->flags = (WORD *)Malloc1(ABS(T->numind)*sizeof(WORD),"table flags");
00727     ClearTableTree(T);
00728     TT->spare = T;
00729     TT->mode = 1;
00730     return(0);
00731 }
00732
00733 /*
00734     #] SpareTable :
00735     #[ FindTB :
00736
00737     Looks for a tablebase with the given name in the active tablebases.
00738 */
00739
00740 DBASE *FindTB(UBYTE *name)
00741 {
00742     DBASE *d;
00743     int i;
00744     for ( i = 0; i < NumTableBases; i++ ) {
00745         d = tablebases+i;
00746         if ( d->name && ( StrCmp(name,(UBYTE *)(d->name)) == 0 ) ) { return(d); }
00747     }
00748     return(0);
00749 }
00750
00751 /*
00752     #] FindTB :
00753     #[ CoTBcreate :
00754
00755     Creates a new tablebase.
00756     Error is when there is already an active tablebase by this name.
00757     If a file with the given name exists already, but it does not correspond
00758     to an active table base, its contents will be lost.
00759     Note that tablebasename is a static variable, defined in CoTableBase
00760 */
00761
00762 int CoTBcreate(UBYTE *s)
00763 {
00764     DUMMYUSE(s);
00765     if ( FindTB(tablebasename) != 0 ) {
00766         MesPrint("&There is already an open TableBase with the name %s",tablebasename);
```

```
00767         return(-1);
00768     }
00769     NewDbase((char *)tablebasename,0);
00770     return(0);
00771 }
00772
00773 /*
00774     #] CoTBcreate :
00775     #[ CoTBopen :
00776 */
00777
00778 int CoTBopen(UBYTE *s)
00779 {
00780     DBASE *d;
00781     MLONG rw = 1;
00782
00783     SkipSpaces(&s);
00784
00785     if ( *s ) {
00786         if ( ConsumeOption(&s,"readonly") != 0 ) {
00787             rw = 0;
00788         } else {
00789             MesPrint("&Invalid option for TableBase open: %s, ignoring", s);
00790         }
00791     }
00792
00793     if ( ( d = FindTB(tablebasename) ) != 0 ) {
00794         MesPrint("&There is already an open TableBase with the name %s",tablebasename);
00795         return(-1);
00796     }
00797     d = GetDbase((char *)tablebasename, rw);
00798     if ( CheckTableDeclarations(d) ) return(-1);
00799     return(0);
00800 }
00801
00802 /*
00803     #] CoTBopen :
00804     #[ CoTBaddto :
00805 */
00806
00807 int CoTBaddto(UBYTE *s)
00808 {
00809     GETIDENTITY
00810     DBASE *d;
00811     UBYTE *tablename, c, *t, elementstring[ELEMENTSIZE+20], *ss, *es;
00812     WORD type, funnum, lbrac, first, num, *expr, *w;
00813     TABLES T = 0;
00814     MLONG basenumber;
00815     LONG x;
00816     int i, j, error = 0, sum;
00817     if ( ( d = FindTB(tablebasename) ) == 0 ) {
00818         MesPrint("&No open tablebase with the name %s",tablebasename);
00819         return(-1);
00820     }
00821
00822     if ( ( d->rwmode ) == 0 ) {
00823         MesPrint("&Tablebase with the name %s opened in read only mode",tablebasename);
00824         return(-1);
00825     }
00826     AO.DollarOutSizeBuffer = 32;
00827     AO.DollarOutBuffer = (UBYTE *)Malloc1(AO.DollarOutSizeBuffer,
00828                             "TableOutBuffer");
00829 /*
00830     Now loop through the names and start adding
00831 */
00832     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
00833     while ( *s ) {
00834         tablename = s;
00835         if ( ( s = SkipAName(s) ) == 0 ) goto tableabort;
00836         c = *s; *s = 0;
00837         if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
00838             || ( T = functions[funnum].tabl ) == 0 ) {
00839             MesPrint("&%s should be a previously declared table",tablename);
00840             *s = c; goto tableabort;
00841         }
00842         if ( T->sparse == 0 ) {
00843             MesPrint("&%s should be a sparse table",tablename);
00844             *s = c; goto tableabort;
00845         }
00846         basenumber = AddTableName(d,(char *)tablename,T);
00847         if ( T->spare && ( T->mode ==  1 ) ) T = T->spare;
00848         if ( basenumber < 0 ) basenumber = -basenumber;
00849         else if ( basenumber == 0 ) { *s = c; goto tableabort; }
00850         *s = c;
00851         if ( *s == '(' ) { /* Addition of single element */
00852             s++; es = s;
00853             for ( i = 0, w = AT.WorkPointer; i < ABS(T->numind); i++ ) {
```

```
00854               ParseSignedNumber(x,s);
00855               if ( FG.cTable[s[-1]] != 1 || ( *s != ',' && *s != ')' ) ) {
00856                 MesPrint("&Table arguments in TableBase addto statement should be numbers");
00857                 return(1);
00858               }
00859               *w++ = x;
00860               if ( *s == ')' ) break;
00861               s++;
00862             }
00863             if ( *s != ')' || i < ( ABS(T->numind) - 1 ) ) {
00864               MesPrint("&Incorrect number of table arguments in TableBase addto statement. Should be
      %d"
00865                 ,ABS(T->numind));
00866               error = 1;
00867             }
00868             c = *s; *s = 0;
00869             i = FindTableTree(T,AT.WorkPointer,1);
00870             if ( i < 0 ) {
00871                 MesPrint("&Element %s has not been defined",es);
00872                 error = 1;
00873                 *s++ = c;
00874             }
00875             else if ( ExistsObject(d,basenumber,(char *)es) ) {}
00876             else {
00877               int dict = AO.CurrentDictionary;
00878               AO.CurrentDictionary = 0;
00879               sum = i + ABS(T->numind);
00880 /*
00881               See also commentary below
00882 */
00883               AO.DollarInOutBuffer = 1;
00884               AO.PrintType = 1;
00885               ss = AO.DollarOutBuffer;
00886               *ss = 0;
00887               AO.OutInBuffer = 1;
00888 #if ( TABLEEXTENSION == 2 )
00889               expr = cbuf[T->bufnum].rhs[T->tablepointers[sum]];
00890 #else
00891               expr = cbuf[T->tablepointers[sum+1]].rhs[T->tablepointers[sum]];
00892 #endif
00893               lbrac = 0; first = 0;
00894               while ( *expr ) {
00895                 if ( WriteTerm(expr,&lbrac,first,PRINTON,0) ) {
00896                         error = 1; break;
00897                 }
00898                 expr += *expr;
00899               }
00900               AO.OutInBuffer = 0;
00901               AddObject(d,basenumber,(char *)es,(char *)(AO.DollarOutBuffer));
00902               *s++ = c;
00903               AO.CurrentDictionary = dict;
00904             }
00905         }
00906         else {
00907 /*
00908         Now we have to start looping through all defined elements of this table.
00909         We have to construct the arguments in text format.
00910 */
00911           for ( i = 0; i < T->totind; i++ ) {
00912 #if ( TABLEEXTENSION == 2 )
00913             if ( !T->sparse && T->tablepointers[i] < 0 ) continue;
00914 #else
00915             if ( !T->sparse && T->tablepointers[TABLEEXTENSION*i] < 0 ) continue;
00916 #endif
00917             sum = i * ( ABS(T->numind) + TABLEEXTENSION );
00918             t = elementstring;
00919             for ( j = 0; j < ABS(T->numind); j++, sum++ ) {
00920                 if ( j > 0 ) *t++ = ',';
00921                 num = T->tablepointers[sum];
00922                 t = NumCopy(num,t);
00923                 if ( ( t - elementstring ) >= ELEMENTSIZE ) {
00924                     MesPrint("&Table element specification takes more than %ld characters and cannot
      be handled",
00925                         (MLONG)ELEMENTSIZE);
00926                     goto tableabort;
00927                 }
00928             }
00929             if ( ExistsObject(d,basenumber,(char *)elementstring) ) { continue; }
00930 /*
00931             We have the number in basenumber and the element in elementstring.
00932             Now we need the rhs. We can use the code from WriteDollarToBuffer.
00933             Main complication: in the table compiler buffer there can be
00934             brackets. The dollars do not have those......
00935 */
00936             AO.DollarInOutBuffer = 1;
00937             AO.PrintType = 1;
00938             ss = AO.DollarOutBuffer;
```

```
00939             *ss = 0;
00940             AO.OutInBuffer = 1;
00941 #if ( TABLEEXTENSION == 2 )
00942             expr = cbuf[T->bufnum].rhs[T->tablepointers[sum]];
00943 #else
00944             expr = cbuf[T->tablepointers[sum+1]].rhs[T->tablepointers[sum]];
00945 #endif
00946             lbrac = 0; first = 0;
00947             while ( *expr ) {
00948                 if ( WriteTerm(expr,&lbrac,first,PRINTON,0) ) {
00949                     error = 1; break;
00950                 }
00951                 expr += *expr;
00952             }
00953             AO.OutInBuffer = 0;
00954             AddObject(d,basenumber,(char *)elementstring,(char *)(AO.DollarOutBuffer));
00955           }
00956         }
00957         while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
00958     }
00959     if ( WriteIniInfo(d) ) goto tableabort;
00960     M_free(AO.DollarOutBuffer,"DollarOutBuffer");
00961     AO.DollarOutBuffer = 0;
00962     AO.DollarOutSizeBuffer = 0;
00963     return(error);
00964 tableabort:;
00965     M_free(AO.DollarOutBuffer,"DollarOutBuffer");
00966     AO.DollarOutBuffer = 0;
00967     AO.DollarOutSizeBuffer = 0;
00968     AO.OutInBuffer = 0;
00969     return(1);
00970 }
00971
00972 /*
00973     #] CoTBaddto :
00974     #[ CoTBenter :
00975
00976     Loads the elements of the tables specified into memory and sends them
00977     one by one to the compiler as Fill statements.
00978 */
00979
00980 int CoTBenter(UBYTE *s)
00981 {
00982     DBASE *d;
00983     MLONG basenumber;
00984     UBYTE *arguments, *rhs, *buffer, *t, *u, c, *tablename;
00985     LONG size;
00986     int i, j, error = 0, error1 = 0, printall = 0;
00987     TABLES T = 0;
00988     WORD type, funnum;
00989     int dict = AO.CurrentDictionary;
00990     AO.CurrentDictionary = 0;
00991     if ( ( d = FindTB(tablebasename) ) == 0 ) {
00992         MesPrint("&No open tablebase with the name %s, check for existence of file or try readonly
    mode when opening.",tablebasename);
00993         error = -1;
00994         goto Endofall;
00995     }
00996     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
00997     if ( *s == '!' ) { printall = 1; s++; }
00998     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
00999     if ( *s ) {
01000       while ( *s ) {
01001         tablename = s;
01002         if ( ( s = SkipAName(s) ) == 0 ) { error = 1; goto Endofall; }
01003         c = *s; *s = 0;
01004         if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
01005             || ( T = functions[funnum].tabl ) == 0 ) {
01006             MesPrint("&%s should be a previously declared table",tablename);
01007             basenumber = 0;
01008         }
01009         else if ( T->sparse == 0 ) {
01010             MesPrint("&%s should be a sparse table",tablename);
01011             basenumber = 0;
01012         }
01013         else { basenumber = GetTableName(d,(char *)tablename); }
01014         if ( T->spare == 0 ) { SpareTable(T); }
01015         if ( basenumber > 0 ) {
01016             for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01017                 for ( j = 0; j < NUMOBJECTS; j++ ) {
01018                     if ( basenumber != d->iblocks[i]->objects[j].tablenumber )
01019                         continue;
01020                     arguments = (UBYTE *)(d->iblocks[i]->objects[j].element);
01021                     rhs = (UBYTE *)ReadObject(d,basenumber,(char *)arguments);
01022                     if ( printall ) {
01023                         if ( rhs ) {
01024                             MesPrint("%s(%s) = %s",tablename,arguments,rhs);
```

```
01025                       }
01026                       else {
01027                           MesPrint("%s(%s) = 0",tablename,arguments);
01028                       }
01029                   }
01030                   if ( rhs ) {
01031                       u = rhs; while ( *u ) u++;
01032                       size = u-rhs;
01033                       u = arguments; while ( *u ) u++;
01034                       size += u-arguments;
01035                       u = tablename; while ( *u ) u++;
01036                       size += u-tablename;
01037                       size += 6;
01038                       buffer  = (UBYTE *)Malloc1(size,"TableBase copy");
01039                       t = tablename; u = buffer;
01040                       while ( *t ) *u++ = *t++;
01041                       *u++ = '(';
01042                       t = arguments;
01043                       while ( *t ) *u++ = *t++;
01044                       *u++ = ')'; *u++ = '=';
01045                       t = rhs;
01046                       while ( *t ) *u++ = *t++;
01047                       if ( t == rhs ) *u++ = '0';
01048                       *u++ = 0; *u = 0;
01049                       M_free(rhs,"rhs in TBenter");
01050
01051                       error1 = CoFill(buffer);
01052
01053                       if ( error1 < 0 ) goto Endofall;
01054                       if ( error1 != 0 ) error = error1;
01055                       M_free(buffer,"TableBase copy");
01056                   }
01057               }
01058           }
01059       }
01060       *s = c;
01061       while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
01062     }
01063   }
01064   else {
01065     s = (UBYTE *)(d->tablenames); basenumber = 0;
01066     while ( *s ) {
01067       basenumber++;
01068       tablename = s; while ( *s ) s++; s++;
01069       while ( *s ) s++;
01070       s++;
01071       if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
01072           || ( T = functions[funnum].tabl ) == 0 ) {
01073           MesPrint("&%s should be a previously declared table",tablename);
01074       }
01075       else if ( T->sparse == 0 ) {
01076           MesPrint("&%s should be a sparse table",tablename);
01077       }
01078       if ( T->spare == 0 ) { SpareTable(T); }
01079       for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01080           for ( j = 0; j < NUMOBJECTS; j++ ) {
01081               if ( d->iblocks[i]->objects[j].tablenumber == basenumber ) {
01082                   arguments = (UBYTE *)(d->iblocks[i]->objects[j].element);
01083                   rhs = (UBYTE *)ReadObject(d,basenumber,(char *)arguments);
01084                   if ( printall ) {
01085                       if ( rhs ) {
01086                           MesPrint("%s%s = %s",tablename,arguments,rhs);
01087                       }
01088                       else {
01089                           MesPrint("%s%s = 0",tablename,arguments);
01090                       }
01091                   }
01092                   if ( rhs ) {
01093                       u = rhs; while ( *u ) u++;
01094                       size = u-rhs;
01095                       u = arguments; while ( *u ) u++;
01096                       size += u-arguments;
01097                       u = tablename; while ( *u ) u++;
01098                       size += u-tablename;
01099                       size += 6;
01100                       buffer  = (UBYTE *)Malloc1(size,"TableBase copy");
01101                       t = tablename; u = buffer;
01102                       while ( *t ) *u++ = *t++;
01103                       *u++ = '(';
01104                       t = arguments;
01105                       while ( *t ) *u++ = *t++;
01106                       *u++ = ')'; *u++ = '=';
01107                       t = rhs;
01108                       while ( *t ) *u++ = *t++;
01109                       if ( t == rhs ) *u++ = '0';
01110                       *u++ = 0; *u = 0;
01111                       M_free(rhs,"rhs in TBenter");
```

```
01112
01113                           error1 = CoFill(buffer);
01114
01115                           if ( error1 < 0 ) goto Endofall;
01116                           if ( error1 != 0 ) error = error1;
01117                           M_free(buffer,"TableBase copy");
01118                       }
01119                   }
01120               }
01121           }
01122         }
01123     }
01124 Endofall:;
01125     AO.CurrentDictionary = dict;
01126     return(error);
01127 }
01128
01129 /*
01130     #] CoTBenter :
01131     #[ CoTestUse :
01132
01133     Possibly to be followed by names of tables.
01134     We make an array of TABLES structs to be tested in AC.usedtables.
01135     Note: only sparse tables are allowed.
01136     No arguments means all tables.
01137 */
01138
01139 int CoTestUse(UBYTE *s)
01140 {
01141     GETIDENTITY
01142     UBYTE *tablename, c;
01143     WORD type, funnum, *w;
01144     TABLES T;
01145     int error = 0;
01146     w = AT.WorkPointer;
01147     *w++ = TYPETESTUSE; *w++ = 2;
01148     while ( *s ) {
01149         tablename = s;
01150         if ( ( s = SkipAName(s) ) == 0 ) return(1);
01151         c = *s; *s = 0;
01152         if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
01153             || ( T = functions[funnum].tabl ) == 0 ) {
01154             MesPrint("&%s should be a previously declared table",tablename);
01155             error = 1;
01156         }
01157         else if ( T->sparse == 0 ) {
01158             MesPrint("&%s should be a sparse table",tablename);
01159             error = 1;
01160         }
01161         *w++ = funnum + FUNCTION;
01162         *s = c;
01163         while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
01164     }
01165     AT.WorkPointer[1] = w - AT.WorkPointer;
01166 /*
01167     if ( AT.WorkPointer[1] > 2 ) {
01168         AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
01169     }
01170 */
01171     AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
01172     return(error);
01173 }
01174
01175 /*
01176     #] CoTestUse :
01177     #[ CheckTableDeclarations :
01178
01179         Checks that all tables in a tablebase  have identical properties to
01180         possible previous declarations. If they have not been declared
01181         before, they are declared here.
01182 */
01183
01184 int CheckTableDeclarations(DBASE *d)
01185 {
01186     WORD type, funnum;
01187     UBYTE *s, *ss, *t, *command = 0;
01188     int k, error = 0, error1, i;
01189     TABLES T;
01190     LONG commandsize = 0;
01191
01192     s = (UBYTE *)(d->tablenames);
01193     for ( k = 0; k < d->topnumber; k++ ) {
01194         if ( GetVar(s,&type,&funnum,ANYTYPE,NOAUTO) == NAMENOTFOUND ) {
01195 /*
01196             We have to declare the table
01197 */
01198             ss = s; i = 0; while ( *ss ) { ss++; i++; } /* name */
```

```
01199             ss++; while ( *ss ) { ss++; i++; } /* tail */
01200             if ( commandsize == 0 ) {
01201                 commandsize = i + 15;
01202                 if ( commandsize < 100 ) commandsize = 100;
01203             }
01204             if ( (i+11) > commandsize ) {
01205                 if ( command ) { M_free(command,"table command"); command = 0; }
01206                 commandsize = i+10;
01207             }
01208             if ( command == 0 ) {
01209                 command = (UBYTE *)Malloc1(commandsize,"table command");
01210             }
01211             t = command; ss = tablebase; while ( *ss ) *t++ = *ss++;
01212             *t++ = ','; while ( *s ) *t++ = *s++;
01213             s++; while ( *s ) *t++ = *s++;
01214             *t++ = ')'; *t = 0; s++;
01215             error1 = DoTable(command,1);
01216             if ( error1 ) error = error1;
01217         }
01218         else if ( ( type != CFUNCTION )
01219              || ( ( T = functions[funnum].tabl ) == 0 )
01220              || ( T->sparse == 0 ) ) {
01221             MesPrint("&%s has been declared previously, but not as a sparse table.",s);
01222             error = 1;
01223             while ( *s ) s++;
01224             s++;
01225             while ( *s ) s++;
01226             s++;
01227         }
01228         else {
01229 /*
01230             Test dimension and argtail. There should be an exact match.
01231             We are not going to rename arguments when reading the elements.
01232 */
01233             ss = s;
01234             while ( *s ) s++;
01235             s++;
01236             if ( StrCmp(s,T->argtail) ) {
01237                 MesPrint("&Declaration of table %s in %s different from previous
      declaration",ss,d->name);
01238                 error = 1;
01239             }
01240             while ( *s ) s++;
01241             s++;
01242         }
01243     }
01244     if ( command ) { M_free(command,"table command"); }
01245     return(error);
01246 }
01247
01248 /*
01249     #] CheckTableDeclarations :
01250     #[ CoTBload :
01251
01252         Loads the table stubbs of the specified tables in the indicated
01253         tablebase. Syntax:
01254         TableBase "tablebasename.tbl" load [tablename(s)];
01255         If no tables are specified all tables are taken.
01256 */
01257
01258 int CoTBload(UBYTE *ss)
01259 {
01260     DBASE *d;
01261     UBYTE *s, *name, *t, *r, *command, *arguments, *tail;
01262     LONG commandsize;
01263     int num, cs, es, ns, ts, i, j, error = 0, error1;
01264     if ( ( d = FindTB(tablebasename) ) == 0 ) {
01265         MesPrint("&No open tablebase with the name %s",tablebasename);
01266         return(-1);
01267     }
01268     commandsize = 120;
01269     command = (UBYTE *)Malloc1(commandsize,"Fill command");
01270     AC.vetofilling = 1;
01271     if ( *ss ) {
01272       while ( *ss == ',' || *ss == ' ' || *ss == '\t' ) ss++;
01273       while ( *ss ) {
01274         name = ss; ss = SkipAName(ss); *ss = 0;
01275         s = (UBYTE *)(d->tablenames);
01276         num = 0; ns = 0;
01277         while ( *s ) {
01278             num++;
01279             if ( StrCmp(s,name) ) {
01280                 while ( *s ) s++;
01281                 s++;
01282                 while ( *s ) s++;
01283                 s++;
01284                 num++;
```

```
01285                 continue;
01286             }
01287             name = s; while ( *s ) s++; ns = s-name; s++;
01288             tail = s; while ( *s ) s++; ts = s-tail; s++;
01289             tail++; while ( FG.cTable[*tail] == 1 ) tail++;
01290 /*
01291             Go through all elements
01292 */
01293             for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01294                 for ( j = 0; j < NUMOBJECTS; j++ ) {
01295                     if ( d->iblocks[i]->objects[j].tablenumber == num ) {
01296                         t = arguments = (UBYTE *)(d->iblocks[i]->objects[j].element);
01297                         while ( *t ) t++;
01298                         es = t - arguments;
01299                         cs = 2*es + 2*ns + ts + 10;
01300                         if ( cs > commandsize ) {
01301                             commandsize = 2*cs;
01302                             if ( command ) M_free(command,"Fill command");
01303                             command = (UBYTE *)Malloc1(commandsize,"Fill command");
01304                         }
01305                         r = command; t = name; while ( *t ) *r++ = *t++;
01306                         *r++ = '('; t = arguments; while ( *t ) *r++ = *t++;
01307                         *r++ = ')'; *r++ = '='; *r++ = 't'; *r++ = 'b'; *r++ = 'l';
01308                         *r++ = '_'; *r++ = '('; t = name; while ( *t ) *r++ = *t++;
01309                         *r++ = ','; t = arguments; while ( *t ) *r++ = *t++;
01310                         t = tail; while ( *t ) {
01311                             if ( *t == '?' && r[-1] != ',' ) {
01312                                 t++;
01313                                 if ( FG.cTable[*t] == 0 || *t == '$' || *t == '[' ) {
01314                                     t = SkipAName(t);
01315                                     if ( *t == '[' ) {
01316                                         SKIPBRA1(t);
01317                                     }
01318                                 }
01319                                 else if ( *t == '{' ) {
01320                                     SKIPBRA2(t);
01321                                 }
01322                                 else if ( *t ) { *r++ = *t++; continue; }
01323                             }
01324                             else *r++ = *t++;
01325                         }
01326                         *r++ = ')'; *r = 0;
01327 /*
01328                         Still to do: replacemode or no replacemode?
01329 */
01330                         AC.vetotablebasefill = 1;
01331                         error1 = CoFill(command);
01332                         AC.vetotablebasefill = 0;
01333                         if ( error1 < 0 ) goto finishup;
01334                         if ( error1 != 0 ) error = error1;
01335                     }
01336                 }
01337             }
01338             break;
01339         }
01340         while ( *ss == ',' || *ss == ' ' || *ss == '\t' ) ss++;
01341       }
01342     }
01343     else {  /* do all of them */
01344         s = (UBYTE *)(d->tablenames);
01345         num = 0; ns = 0;
01346         while ( *s ) {
01347             num++;
01348             name = s; while ( *s ) s++; ns = s-name; s++;
01349             tail = s; while ( *s ) s++; ts = s-tail; s++;
01350             tail++; while ( FG.cTable[*tail] == 1 ) tail++;
01351 /*
01352             Go through all elements
01353 */
01354             for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01355                 for ( j = 0; j < NUMOBJECTS; j++ ) {
01356                     if ( d->iblocks[i]->objects[j].tablenumber == num ) {
01357                         t = arguments = (UBYTE *)(d->iblocks[i]->objects[j].element);
01358                         while ( *t ) t++;
01359                         es = t - arguments;
01360                         cs = 2*es + 2*ns + ts + 10;
01361                         if ( cs > commandsize ) {
01362                             commandsize = 2*cs;
01363                             if ( command ) M_free(command,"Fill command");
01364                             command = (UBYTE *)Malloc1(commandsize,"Fill command");
01365                         }
01366                         r = command; t = name; while ( *t ) *r++ = *t++;
01367                         *r++ = '('; t = arguments; while ( *t ) *r++ = *t++;
01368                         *r++ = ')'; *r++ = '='; *r++ = 't'; *r++ = 'b'; *r++ = 'l';
01369                         *r++ = '_'; *r++ = '('; t = name; while ( *t ) *r++ = *t++;
01370                         *r++ = ','; t = arguments; while ( *t ) *r++ = *t++;
01371                         t = tail; while ( *t ) {
```

```
01372                               if ( *t == '?' && r[-1] != ',' ) {
01373                                   t++;
01374                                   if ( FG.cTable[*t] == 0 || *t == '$' || *t == '[' ) {
01375                                       t = SkipAName(t);
01376                                       if ( *t == '[' ) {
01377                                           SKIPBRA1(t);
01378                                       }
01379                                   }
01380                                   else if ( *t == '{' ) {
01381                                       SKIPBRA2(t);
01382                                   }
01383                                   else if ( *t ) { *r++ = *t++; continue; }
01384                               }
01385                               else *r++ = *t++;
01386                           }
01387                           *r++ = ')'; *r = 0;
01388 /*
01389                           Still to do: replacemode or no replacemode?
01390 */
01391                           AC.vetotablebasefill = 1;
01392                           error1 = CoFill(command);
01393                           AC.vetotablebasefill = 0;
01394                           if ( error1 < 0 ) goto finishup;
01395                           if ( error1 != 0 ) error = error1;
01396                       }
01397                   }
01398               }
01399           }
01400       }
01401 finishup:;
01402       AC.vetofilling = 0;
01403       if ( command ) M_free(command,"Fill command");
01404       return(error);
01405 }
01406
01407 /*
01408       #] CoTBload :
01409       #[ TestUse :
01410
01411       Look for tbl_(tablename,arguments)
01412       if tablename is encountered, check first whether the element is in
01413       use already. If not, check in the tables in AC.usedtables.
01414       If the element is not there, add it to AC.usedtables.
01415
01416
01417       We need the arguments of TestUse to see for which tables it is to be done
01418 */
01419
01420 int TestUse(WORD *term, WORD level)
01421 {
01422       WORD *tstop, *t, *m, *tstart, tabnum;
01423       WORD *funs, numfuns;
01424       int error = 0;
01425       TABLES T;
01426       LONG i;
01427       CBUF *C = cbuf+AM.rbufnum;
01428       int isp;
01429
01430       numfuns = C->lhs[level][1] - 2;
01431       funs = C->lhs[level] + 2;
01432       GETSTOP(term,tstop);
01433       t = term+1;
01434       while ( t < tstop ) {
01435           if ( *t != TABLESTUB ) { t += t[1]; continue; }
01436           tstart = t;
01437           m = t + FUNHEAD;
01438           t += t[1];
01439           if ( *m >= -FUNCTION ) continue;
01440           tabnum = -*m;
01441           if ( ( T = functions[tabnum-FUNCTION].tabl ) == 0 ) continue;
01442           if ( T->sparse == 0 ) continue;
01443 /*
01444           Check whether we have to test this one
01445 */
01446           if ( numfuns > 0 ) {
01447               for ( i = 0; i < numfuns; i++ ) {
01448                   if ( tabnum == funs[i] ) break;
01449               }
01450               if ( i >= numfuns && numfuns > 0 ) continue;
01451           }
01452 /*
01453           Test whether the element has been defined already.
01454               If not, mark it as used.
01455           Note: we only allow sparse tables (for now)
01456 */
01457           m++;
01458           for ( i = 0; i < ABS(T->numind); i++, m += 2 ) {
```

```
01459             if ( m >= t || *m != -SNUMBER ) break;
01460         }
01461         if ( ( i == ABS(T->numind) ) &&
01462          ( ( isp = FindTableTree(T,tstart+FUNHEAD+1,2) ) >= 0 ) ) {
01463             if ( ( T->tablepointers[isp+ABS(T->numind)+4] & ELEMENTLOADED ) == 0 ) {
01464                     T->tablepointers[isp+ABS(T->numind)+4] |= ELEMENTUSED;
01465             }
01466         }
01467         else {
01468 /* INTERNAL_ERROR_EXCL_START */
01469             MesPrint("!>TestUse: Encountered a table element inside tbl_ that does not correspond to a
      tablebase element");
01470             error = -1;
01471 /* INTERNAL_ERROR_EXCL_STOP */
01472         }
01473     }
01474     return(error);
01475 }
01476
01477 /*
01478     #] TestUse :
01479     #[ CoTBaudit :
01480 */
01481
01482 int CoTBaudit(UBYTE *s)
01483 {
01484     DBASE *d;
01485     UBYTE *name, *tail;
01486     int i, j, error = 0, num;
01487
01488     if ( ( d = FindTB(tablebasename) ) == 0 ) {
01489         MesPrint("&No open tablebase with the name %s",tablebasename);
01490         return(-1);
01491     }
01492     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
01493     while ( *s ) {
01494 /*
01495         Get the options here
01496         They will mainly involve the sorting of the output.
01497 */
01498         s++;
01499     }
01500     s = (UBYTE *)(d->tablenames); num = 0;
01501     while ( *s ) {
01502         num++;
01503         name = s; while ( *s ) s++; s++;
01504         tail = s; while ( *s ) s++; s++;
01505         MesPrint("Table,sparse,%s%s)",name,tail);
01506         for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01507             for ( j = 0; j < NUMOBJECTS; j++ ) {
01508                 if ( d->iblocks[i]->objects[j].tablenumber == num ) {
01509                     MesPrint("    %s(%s)",name,d->iblocks[i]->objects[j].element);
01510                 }
01511             }
01512         }
01513     }
01514     return(error);
01515 }
01516
01517 /*
01518     #] CoTBaudit :
01519     #[ CoTBon :
01520 */
01521
01522 int CoTBon(UBYTE *s)
01523 {
01524     DBASE *d;
01525     UBYTE *ss, c;
01526     int error = 0;
01527     if ( ( d = FindTB(tablebasename) ) == 0 ) {
01528         MesPrint("&No open tablebase with the name %s",tablebasename);
01529         return(-1);
01530     }
01531     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
01532     while ( *s ) {
01533         ss = SkipAName(s);
01534         c = *ss; *ss = 0;
01535         if ( StrICmp(s,(UBYTE *)("compress")) == 0 ) {
01536             d->mode &= ~NOCOMPRESS;
01537         }
01538         else {
01539             MesPrint("&subkey %s not defined in TableBase On statement");
01540             error = 1;
01541         }
01542         *ss = c; s = ss;
01543         while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
01544     }
```

```
01545     return(error);
01546 }
01547
01548 /*
01549     #] CoTBon :
01550     #[ CoTBoff :
01551 */
01552
01553 int CoTBoff(UBYTE *s)
01554 {
01555     DBASE *d;
01556     UBYTE *ss, c;
01557     int error = 0;
01558     if ( ( d = FindTB(tablebasename) ) == 0 ) {
01559         MesPrint("&No open tablebase with the name %s",tablebasename);
01560         return(-1);
01561     }
01562     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
01563     while ( *s ) {
01564         ss = SkipAName(s);
01565         c = *ss; *ss = 0;
01566         if ( StrICmp(s,(UBYTE *)("compress")) == 0 ) {
01567             d->mode |= NOCOMPRESS;
01568         }
01569         else {
01570             MesPrint("&subkey %s not defined in TableBase Off statement");
01571             error = 1;
01572         }
01573         *ss = c; s = ss;
01574         while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
01575     }
01576     return(error);
01577 }
01578
01579 /*
01580     #] CoTBoff :
01581     #[ CoTBcleanup :
01582 */
01583
01584 int CoTBcleanup(UBYTE *s)
01585 {
01586     DUMMYUSE(s);
01587     MesPrint("&TableBase Cleanup statement not yet implemented");
01588     return(1);
01589 }
01590
01591 /*
01592     #] CoTBcleanup :
01593     #[ CoTBreplace :
01594 */
01595
01596 int CoTBreplace(UBYTE *s)
01597 {
01598     DUMMYUSE(s);
01599     MesPrint("&TableBase Replace statement not yet implemented");
01600     return(1);
01601 }
01602
01603 /*
01604     #] CoTBreplace :
01605     #[ CoTBuse :
01606
01607     Here the actual table use as determined in TestUse causes the needed
01608     table elements to be loaded
01609 */
01610
01611 int CoTBuse(UBYTE *s)
01612 {
01613     GETIDENTITY
01614     DBASE *d;
01615     MLONG basenumber;
01616     UBYTE *arguments, *rhs, *buffer, *t, *u, c, *tablename, *p;
01617     LONG size, sum, x;
01618     int i, j, error = 0, error1 = 0, k;
01619     TABLES T = 0;
01620     WORD type, funnum, mode, *w;
01621     if ( ( d = FindTB(tablebasename) ) == 0 ) {
01622         MesPrint("&No open tablebase with the name %s",tablebasename);
01623         return(-1);
01624     }
01625     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
01626     if ( *s ) {
01627       while ( *s ) {
01628         tablename = s;
01629         if ( ( s = SkipAName(s) ) == 0 ) return(1);
01630         c = *s; *s = 0;
01631         if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
```

```
01632             || ( T = functions[funnum].tabl ) == 0 ) {
01633             MesPrint("&%s should be a previously declared table",tablename);
01634             basenumber = 0;
01635         }
01636         else if ( T->sparse == 0 ) {
01637             MesPrint("&%s should be a sparse table",tablename);
01638             basenumber = 0;
01639         }
01640         else { basenumber = GetTableName(d,(char *)tablename); }
01641 /*      if ( T->spare == 0 ) { SpareTable(T); } */
01642         if ( basenumber > 0 ) {
01643             for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01644                 for ( j = 0; j < NUMOBJECTS; j++ ) {
01645                     if ( d->iblocks[i]->objects[j].tablenumber != basenumber ) continue;
01646                     arguments = p = (UBYTE *)(d->iblocks[i]->objects[j].element);
01647 /*
01648                     Now translate the arguments and see whether we need
01649                     this one....
01650 */
01651                     for ( k = 0, w = AT.WorkPointer; k < ABS(T->numind); k++ ) {
01652                         ParseSignedNumber(x,p);
01653                         *w++ = x; p++;
01654                     }
01655                     sum = FindTableTree(T,AT.WorkPointer,1);
01656                     if ( sum < 0 ) {
01657                         MesPrint("Table %s in tablebase %s has not been loaded properly"
01658                                 ,tablename,tablebasename);
01659                         error = 1;
01660                         continue;
01661                     }
01662                     sum += ABS(T->numind) + 4;
01663                     mode = T->tablepointers[sum];
01664                     if ( ( mode & ELEMENTLOADED ) == ELEMENTLOADED ) {
01665                         T->tablepointers[sum] &= ~ELEMENTUSED;
01666                         continue;
01667                     }
01668                     if ( ( mode & ELEMENTUSED ) == 0 ) continue;
01669 /*
01670                     We need this one!
01671 */
01672                     rhs = (UBYTE *)ReadijObject(d,i,j,(char *)arguments);
01673                     if ( rhs ) {
01674                         u = rhs; while ( *u ) u++;
01675                         size = u-rhs;
01676                         u = arguments; while ( *u ) u++;
01677                         size += u-arguments;
01678                         u = tablename; while ( *u ) u++;
01679                         size += u-tablename;
01680                         size += 6;
01681                         buffer  = (UBYTE *)Malloc1(size,"TableBase copy");
01682                         t = tablename; u = buffer;
01683                         while ( *t ) *u++ = *t++;
01684                         *u++ = '(';
01685                         t = arguments;
01686                         while ( *t ) *u++ = *t++;
01687                         *u++ = ')'; *u++ = '=';
01688                         t = rhs;
01689                         while ( *t ) *u++ = *t++;
01690                         if ( t == rhs ) { *u++ = '0'; }
01691                         *u++ = 0; *u = 0;
01692                         M_free(rhs,"rhs in TBuse xxx");
01693
01694                         error1 = CoFill(buffer);
01695
01696                         if ( error1 < 0 ) { return(error); }
01697                         if ( error1 != 0 ) error = error1;
01698                         M_free(buffer,"TableBase copy");
01699                     }
01700                     T->tablepointers[sum] &= ~ELEMENTUSED;
01701                     T->tablepointers[sum] |= ELEMENTLOADED;
01702                 }
01703             }
01704         }
01705         *s = c;
01706         while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
01707       }
01708     }
01709     else {
01710       s = (UBYTE *)(d->tablenames); basenumber = 0;
01711       while ( *s ) {
01712         basenumber++;
01713         tablename = s;
01714         while ( *s ) s++;
01715         s++;
01716         while ( *s ) s++;
01717         s++;
01718         if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
```

```
01719             || ( T = functions[funnum].tabl ) == 0 ) {
01720             MesPrint("&%s should be a previously declared table",tablename);
01721         }
01722         else if ( T->sparse == 0 ) {
01723             MesPrint("&%s should be a sparse table",tablename);
01724         }
01725     if ( T->spare && T->mode == 0 ) {
01726 /* INTERNAL_ERROR_EXCL_START */
01727         MesPrint("!>In table %s we have a problem with stubb orders in CoTBuse",tablename);
01728         error = -1;
01729 /* INTERNAL_ERROR_EXCL_STOP */
01730     }
01731 /*      if ( T->spare == 0 ) { SpareTable(T); } */
01732         for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01733             for ( j = 0; j < NUMOBJECTS; j++ ) {
01734                 if ( d->iblocks[i]->objects[j].tablenumber == basenumber ) {
01735                     arguments = p = (UBYTE *)(d->iblocks[i]->objects[j].element);
01736 /*
01737                     Now translate the arguments and see whether we need
01738                     this one....
01739 */
01740                     for ( k = 0, w = AT.WorkPointer; k < ABS(T->numind); k++ ) {
01741                         ParseSignedNumber(x,p);
01742                         *w++ = x; p++;
01743                     }
01744                     sum = FindTableTree(T,AT.WorkPointer,1);
01745                     if ( sum < 0 ) {
01746 /* INTERNAL_ERROR_EXCL_START */
01747                         MesPrint("!>Table %s in tablebase %s has not been loaded properly"
01748                                 ,tablename,tablebasename);
01749                         error = 1;
01750                         continue;
01751 /* INTERNAL_ERROR_EXCL_STOP */
01752                     }
01753                     sum += ABS(T->numind) + 4;
01754                     mode = T->tablepointers[sum];
01755                     if ( ( mode & ELEMENTLOADED ) == ELEMENTLOADED ) {
01756                         T->tablepointers[sum] &= ~ELEMENTUSED;
01757                         continue;
01758                     }
01759                     if ( ( mode & ELEMENTUSED ) == 0 ) continue;
01760 /*
01761                     We need this one!
01762 */
01763                     rhs = (UBYTE *)ReadijObject(d,i,j,(char *)arguments);
01764                     if ( rhs ) {
01765                         u = rhs; while ( *u ) u++;
01766                         size = u-rhs;
01767                         u = arguments; while ( *u ) u++;
01768                         size += u-arguments;
01769                         u = tablename; while ( *u ) u++;
01770                         size += u-tablename;
01771                         size += 6;
01772                         buffer  = (UBYTE *)Malloc1(size,"TableBase copy");
01773                         t = tablename; u = buffer;
01774                         while ( *t ) *u++ = *t++;
01775                         *u++ = '(';
01776                         t = arguments;
01777                         while ( *t ) *u++ = *t++;
01778                         *u++ = ')'; *u++ = '=';
01779
01780                         t = rhs;
01781                         while ( *t ) *u++ = *t++;
01782                         if ( t == rhs ) { *u++ = '0'; }
01783                         *u++ = 0; *u = 0;
01784                         M_free(rhs,"rhs in TBuse");
01785
01786                         error1 = CoFill(buffer);
01787
01788                         if ( error1 < 0 ) { return(error); }
01789                         if ( error1 != 0 ) error = error1;
01790                         M_free(buffer,"TableBase copy");
01791                     }
01792                     T->tablepointers[sum] &= ~ELEMENTUSED;
01793                     T->tablepointers[sum] |= ELEMENTLOADED;
01794                 }
01795             }
01796         }
01797       }
01798     }
01799     return(error);
01800 }
01801
01802 /*
01803     #] CoTBuse :
01804     #[ CoApply :
01805
```

```
01806     Possibly to be followed by names of tables.
01807 */
01808
01809 int CoApply(UBYTE *s)
01810 {
01811     GETIDENTITY
01812     UBYTE *tablename, c;
01813     WORD type, funnum, *w;
01814     TABLES T;
01815     LONG maxtogo = MAXPOSITIVE;
01816     int error = 0;
01817     w = AT.WorkPointer;
01818     if ( FG.cTable[*s] == 1 ) {
01819         maxtogo = 0;
01820         while ( FG.cTable[*s] == 1 ) {
01821             maxtogo = maxtogo*10 + (*s-'0');
01822             s++;
01823         }
01824         while ( *s == ',' ) s++;
01825         if ( maxtogo > MAXPOSITIVE || maxtogo < 0 ) maxtogo = MAXPOSITIVE;
01826     }
01827     *w++ = TYPEAPPLY; *w++ = 3; *w++ = maxtogo;
01828     while ( *s ) {
01829         tablename = s;
01830         if ( ( s = SkipAName(s) ) == 0 ) return(1);
01831         c = *s; *s = 0;
01832         if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
01833             || ( T = functions[funnum].tabl ) == 0 ) {
01834             MesPrint("&%s should be a previously declared table",tablename);
01835             error = 1;
01836         }
01837         else if ( T->sparse == 0 ) {
01838             MesPrint("&%s should be a sparse table",tablename);
01839             error = 1;
01840         }
01841         *w++ = funnum + FUNCTION;
01842         *s = c;
01843         while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
01844     }
01845     AT.WorkPointer[1] = w - AT.WorkPointer;
01846 /*
01847     if ( AT.WorkPointer[1] > 2 ) {
01848         AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
01849     }
01850 */
01851     AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
01852 /*
01853     AT.WorkPointer[0] = TYPEAPPLYRESET;
01854     AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
01855 */
01856     return(error);
01857 }
01858
01859 /*
01860     #] CoApply :
01861     #[ CoTBhelp :
01862 */
01863
01864 char *helptb[] = {
01865      "The TableBase statement is used as follows:"
01866     ,"TableBase \"file.tbl\" keyword subkey(s)"
01867     ,"    in which we have"
01868     ,"Keyword   Subkey(s)   Action"
01869     ,"open                  Opens file.tbl for R/W"
01870     ,"create                Creates file.tbl for R/W. Old contents are lost"
01871     ,"load                  Loads all stubs of all tables"
01872     ,"load    tablename(s) Loads all stubs the tables mentioned"
01873     ,"enter                 Loads all stubs and rhs of all tables"
01874     ,"enter   tablename(s) Loads all stubs and rhs of the tables mentioned"
01875     ,"audit                 Prints list of contents"
01876 /*  ,"replace tablename     saves a table (with overwrite)" */
01877 /*  ,"replace tableelement saves a table element (with overwrite)" */
01878 /*  ,"cleanup               makes tables contingent" */
01879     ,"addto   tablename     adds all elements if not yet there"
01880     ,"addto   tableelement adds element if not yet there"
01881 /*  ,"delete  tablename     removes table from tablebase" */
01882 /*  ,"delete  tableelement removes element from tablebase" */
01883     ,"on      compress      elements are stored in gzip format (default)"
01884     ,"off     compress      elements are stored in uncompressed format"
01885     ,"use                   compiles all needed elements"
01886     ,"use     tablename(s) compiles all needed elements of these tables"
01887     ,""
01888     ,"Related commands are:"
01889     ,"testuse               marks which tbl_ elements occur for all tables"
01890     ,"testuse tablename(s) marks which tbl_ elements occur for given tables"
01891     ,"apply                 replaces tbl_ if rhs available"
01892     ,"apply   tablename(s) replaces tbl_ for given tables if rhs available"
```

```
01893     ,""
01894         };
01895
01896 int CoTBhelp(UBYTE *s)
01897 {
01898     int i, ii = sizeof(helptb)/sizeof(char *);
01899     DUMMYUSE(s);
01900     for ( i = 0; i < ii; i++ ) MesPrint("%s",helptb[i]);
01901     return(0);
01902 }
01903
01904 /*
01905     #] CoTBhelp :
01906     #[ ReWorkT :
01907
01908     Replaces the STUBBS of the functions in the list.
01909     This gains one space. Hence we have to be very careful
01910 */
01911
01912 void ReWorkT(WORD *term, WORD *funs, WORD numfuns)
01913 {
01914     WORD *tstop, *tend, *m, *t, *tt, *mm, *mmm, *r, *rr;
01915     int i, j;
01916     tend = term + *term; tstop = tend - ABS(tend[-1]);
01917     m = t = term+1;
01918     while ( t < tstop ) {
01919         if ( *t == TABLESTUB ) {
01920             for ( i = 0; i < numfuns; i++ ) {
01921                 if ( -t[FUNHEAD] == funs[i] ) break;
01922             }
01923             if ( numfuns == 0 || i < numfuns ) { /* Hit */
01924                 i = t[1] - 1;
01925                 *m++ = -t[FUNHEAD]; *m++ = i; t += 2; i -= FUNHEAD;
01926                 if ( m < t ) { for ( j = 0; j < FUNHEAD-2; j++ ) *m++ = *t++; }
01927                 else { m += FUNHEAD-2; t += FUNHEAD-2; }
01928                 t++;
01929                 while ( i-- > 0 ) { *m++ = *t++; }
01930                 tt = t; mm = m;
01931                 if ( mm < tt ) {
01932                     while ( tt < tend ) *mm++ = *tt++;
01933                     *term = mm - term;
01934                     tend = term + *term; tstop = tend - ABS(tend[-1]);
01935                     t = m;
01936                 }
01937             }
01938             else { goto inc; }
01939         }
01940         else if ( *t >= FUNCTION ) {
01941             tt = t + t[1];
01942             mm = m;
01943             for ( j = 0; j < FUNHEAD; j++ ) {
01944                 if ( m == t ) { m++; t++; }
01945                 else *m++ = *t++;
01946             }
01947             while ( t < tt ) {
01948                 if ( *t <= -FUNCTION ) {
01949                     if ( m == t ) { m++; t++; }
01950                     else *m++ = *t++;
01951                 }
01952                 else if ( *t < 0 ) {
01953                     if ( m == t ) { m += 2; t += 2; }
01954                     else { *m++ = *t++; *m++ = *t++; }
01955                 }
01956                 else {
01957                     rr = t + *t; mmm = m;
01958                     for ( j = 0; j < ARGHEAD; j++ ) {
01959                         if ( m == t ) { m++; t++; }
01960                         else *m++ = *t++;
01961                     }
01962                     while ( t < rr ) {
01963                         r = t + *t;
01964                         ReWorkT(t,funs,numfuns);
01965                         j = *t;
01966                         if ( m == t ) { m += j; t += j; }
01967                         else { while ( j-- >= 0 ) *m++ = *t++; }
01968                         t = r;
01969                     }
01970                     *mmm = m-mmm;
01971                 }
01972             }
01973             mm[1] = m - mm;
01974             t = tt;
01975         }
01976         else {
01977 inc:        j = t[1];
01978             if ( m < t ) { while ( j-- >= 0 ) *m++ = *t++; }
01979             else { m += j; t += j; }
```

```
01980         }
01981     }
01982     if ( m < t ) {
01983         while ( t < tend ) *m++ = *t++;
01984         *term = m - term;
01985     }
01986 }
01987
01988 /*
01989     #] ReWorkT :
01990     #[ Apply :
01991 */
01992
01993 void Apply(WORD *term, WORD level)
01994 {
01995     WORD *funs, numfuns;
01996     TABLES T;
01997     int i, j;
01998     CBUF *C = cbuf+AM.rbufnum;
01999 /*
02000     Point the tables in the proper direction
02001 */
02002     numfuns = C->lhs[level][1] - 2;
02003     funs = C->lhs[level] + 2;
02004     if ( numfuns > 0 ) {
02005         for ( i = MAXBUILTINFUNCTION-FUNCTION; i < AC.FunctionList.num; i++ ) {
02006             if ( ( T = functions[i].tabl ) != 0 ) {
02007                 for ( j = 0; j < numfuns; j++ ) {
02008                     if ( i == (funs[j]-FUNCTION) && T->spare ) {
02009                         FlipTable(&(functions[i]),0);
02010                         break;
02011                     }
02012                 }
02013             }
02014         }
02015     }
02016     else {
02017         for ( i = MAXBUILTINFUNCTION-FUNCTION; i < AC.FunctionList.num; i++ ) {
02018             if ( ( T = functions[i].tabl ) != 0 ) {
02019                 if ( T->spare ) FlipTable(&(functions[i]),0);
02020             }
02021         }
02022     }
02023 /*
02024     Now the replacements everywhere of
02025         id tbl_(table,?a) = table(?a);
02026     Actually, this has to be done recursively.
02027     Note that we actually gain one space.
02028 */
02029     ReWorkT(term,funs,numfuns);
02030 }
02031
02032 /*
02033     #] Apply :
02034     #[ ApplyExec :
02035
02036     Replaces occurrences of tbl_(table,indices,pattern) by the proper
02037     rhs of table(indices,pattern). It does this up to maxtogo times
02038     in the given term. It starts with the occurrences inside the
02039     arguments of functions. If necessary it finishes at groundlevel.
02040     An infinite number of tries is indicated by maxtogo = 2^15-1 or 2^31-1.
02041     The occurrences are replaced by subexpressions. This allows TestSub
02042     to finish the job properly.
02043
02044     The main trick here is T = T->spare which turns to the proper rhs.
02045
02046     The return value is the number of substitutions that can still be made
02047     based on maxtogo. Hence, if the returnvalue is different from maxtogo
02048     there was a substitution.
02049 */
02050
02051 int ApplyExec(WORD *term, int maxtogo, WORD level)
02052 {
02053     GETIDENTITY
02054     WORD rhsnumber, *Tpattern, *funs, numfuns, funnum;
02055     WORD ii, *t, *t1, *w, *p, *m, *m1, *u, *r, tbufnum, csize, wilds;
02056     NESTING NN;
02057     int i, j, isp, stilltogo;
02058     CBUF *C;
02059     TABLES T;
02060 /*
02061     Startup. We need NestPoin for when we have to replace something deep down.
02062 */
02063     t = term;
02064     m = t + *t;
02065     csize = ABS(m[-1]);
02066     m -= csize;
```

```
02067     AT.NestPoin->termsize = t;
02068     if ( AT.NestPoin == AT.Nest ) AN.EndNest = t + *t;
02069     t++;
02070 /*
02071     First we look inside function arguments. Also when clean!
02072 */
02073     while ( t < m ) {
02074         if ( *t < FUNCTION ) { t += t[1]; continue; }
02075         if ( functions[*t-FUNCTION].spec > 0 ) { t += t[1]; continue; }
02076         AT.NestPoin->funsize = t;
02077         r = t + t[1];
02078         t += FUNHEAD;
02079         while ( t < r ) {
02080             if ( *t < 0 ) { NEXTARG(t); continue; }
02081             AT.NestPoin->argsize = t1 = t;
02082             u = t + *t;
02083             t += ARGHEAD;
02084             AT.NestPoin++;
02085             while ( t < u ) {
02086 /*
02087                 Now we loop over the terms inside a function argument
02088                 This defines a recursion and we have to call ApplyExec again.
02089                 The real problem is when we catch something and we have
02090                 to insert a subexpression pointer. This may use more or
02091                 less space and the whole term has to be readjusted.
02092                 This is why we have the NestPoin variables. They tell us
02093                 where the sizes of the term, the function and the arguments
02094                 are sitting, and also where the dirty flags are.
02095                 This readjusting is of course done in the groundlevel code.
02096                 Here we worry abound the maxtogo count.
02097 */
02098                 stilltogo = ApplyExec(t,maxtogo,level);
02099                 if ( stilltogo != maxtogo ) {
02100                     if ( stilltogo <= 0 ) {
02101                         AT.NestPoin--;
02102                         return(stilltogo);
02103                     }
02104                     maxtogo = stilltogo;
02105                     u = t1 + *t1;
02106                     m = term + *term - csize;
02107                 }
02108                 t += *t;
02109             }
02110             AT.NestPoin--;
02111         }
02112     }
02113 /*
02114     Now we look at the ground level
02115 */
02116     C = cbuf+AM.rbufnum;
02117     t = term + 1;
02118     while ( t < m ) {
02119         if ( *t != TABLESTUB ) { t += t[1]; continue; }
02120         funnum = -t[FUNHEAD];
02121         if ( ( funnum < FUNCTION )
02122           || ( funnum >= FUNCTION+WILDOFFSET )
02123           || ( ( T = functions[funnum-FUNCTION].tabl ) == 0 )
02124           || ( T->sparse == 0 )
02125           || ( T->spare == 0 ) ) { t += t[1]; continue; }
02126         numfuns = C->lhs[level][1] - 3;
02127         funs = C->lhs[level] + 3;
02128         if ( numfuns > 0 ) {
02129             for ( i = 0; i < numfuns; i++ ) {
02130                 if ( funs[i] == funnum ) break;
02131             }
02132             if ( i >= numfuns ) { t += t[1]; continue; }
02133         }
02134         r = t + t[1];
02135         AT.NestPoin->funsize = t + 1;
02136         t1 = t;
02137         t += FUNHEAD + 1;
02138 /*
02139         Test whether the table catches
02140         Test 1: index arguments and range. isp will be the number
02141             of the element in the table.
02142 */
02143         T = T->spare;
02144 #ifdef WITHPTHREADS
02145         Tpattern = T->pattern[identity];
02146 #else
02147         Tpattern = T->pattern;
02148 #endif
02149         p = Tpattern+FUNHEAD+1;
02150         for ( i = 0; i < ABS(T->numind); i++, t += 2 ) {
02151             if ( *t != -SNUMBER ) break;
02152         }
02153         if ( i < ABS(T->numind) ) { t = r; continue; }
```

```
02154         isp = FindTableTree(T,t1+FUNHEAD+1,2);
02155         if ( isp < 0 ) { t = r; continue; }
02156         rhsnumber = T->tablepointers[isp+ABS(T->numind)];
02157 #if ( TABLEEXTENSION == 2 )
02158         tbufnum = T->bufnum;
02159 #else
02160         tbufnum = T->tablepointers[isp+ABS(T->numind)+1];
02161 #endif
02162         t = t1+FUNHEAD+2;
02163         ii = ABS(T->numind);
02164         while ( --ii >= 0 ) {
02165             *p = *t; t += 2; p += 2;
02166         }
02167 /*
02168         If there are more arguments we have to do some
02169         pattern matching. This should be easy. We adapted the
02170         pattern, so that the array indices match already.
02171 */
02172 #ifdef WITHPTHREADS
02173         AN.FullProto = T->prototype[identity];
02174 #else
02175         AN.FullProto = T->prototype;
02176 #endif
02177         AN.WildValue = AN.FullProto + SUBEXPSIZE;
02178         AN.WildStop = AN.FullProto+AN.FullProto[1];
02179         ClearWild(BHEAD0);
02180         AN.RepFunNum = 0;
02181         AN.RepFunList = AN.EndNest;
02182         AT.WorkPointer = (WORD *)(((UBYTE *)(AN.EndNest)) + AM.MaxTer/2);
02183 /*
02184         The RepFunList is after the term but not very relevant.
02185         We need because MatchFunction uses it
02186 */
02187         if ( AT.WorkPointer + t1[1] >= AT.WorkTop ) { MesWork(); }
02188         wilds = 0;
02189         w = AT.WorkPointer;
02190         *w++ = -t1[FUNHEAD];
02191         *w++ = t1[1] - 1;
02192         for ( i = 2; i < FUNHEAD; i++ ) *w++ = t1[i];
02193         t = t1 + FUNHEAD+1;
02194         while ( t < r ) *w++ = *t++;
02195         t = AT.WorkPointer;
02196         AT.WorkPointer = w;
02197         if ( MatchFunction(BHEAD Tpattern,t,&wilds) > 0 ) {
02198 /*
02199             Here we caught one. Now we should worry about:
02200             1: inserting the subexpression pointer with its wildcards
02201             2: NestPoin because we may not be at the lowest level
02202             The function starts at t1.
02203 */
02204 #ifdef WITHPTHREADS
02205             m1 = T->prototype[identity];
02206 #else
02207             m1 = T->prototype;
02208 #endif
02209             m1[2] = rhsnumber;
02210             m1[4] = tbufnum;
02211             t = t1;
02212             j = t[1];
02213             i = m1[1];
02214             if ( j > i ) {
02215                 j = i - j;
02216                 NCOPY(t,m1,i);
02217                 m1 = AN.EndNest;
02218                 while ( r < m1 ) *t++ = *r++;
02219                 AN.EndNest = t;
02220                 *term += j;
02221                 NN = AT.NestPoin;
02222                 while ( NN > AT.Nest ) {
02223                     NN--;
02224                     NN->termsize[0] += j;
02225                     NN->funsize[1] += j;
02226                     NN->argsize[0] += j;
02227                     NN->funsize[2] |= DIRTYFLAG;
02228                     NN->argsize[1] |= DIRTYFLAG;
02229                 }
02230                 m += j;
02231             }
02232             else if ( j < i ) {
02233                 j = i-j;
02234                 t = AN.EndNest;
02235                 while ( t >= r ) { t[j] = *t; t--; }
02236                 t = t1;
02237                 NCOPY(t,m1,i);
02238                 AN.EndNest += j;
02239                 *term += j;
02240                 NN = AT.NestPoin;
```

```
02241                 while ( NN > AT.Nest ) {
02242                     NN--;
02243                     NN->termsize[0] += j;
02244                     NN->funsize[1] += j;
02245                     NN->argsize[0] += j;
02246                     NN->funsize[2] |= DIRTYFLAG;
02247                     NN->argsize[1] |= DIRTYFLAG;
02248                 }
02249                 m += j;
02250             }
02251             else {
02252                 NCOPY(t,m1,j);
02253             }
02254             r = t1 + t1[1];
02255             maxtogo--;
02256             if ( maxtogo <= 0 ) return(maxtogo);
02257         }
02258         t = r;
02259     }
02260     return(maxtogo);
02261 }
02262
02263 /*
02264     #] ApplyExec :
02265     #[ ApplyReset :
02266 */
02267
02268 void ApplyReset(WORD level)
02269 {
02270     WORD *funs, numfuns;
02271     TABLES T;
02272     int i, j;
02273     CBUF *C = cbuf+AM.rbufnum;
02274
02275     numfuns = C->lhs[level][1] - 2;
02276     funs = C->lhs[level] + 2;
02277     if ( numfuns > 0 ) {
02278         for ( i = MAXBUILTINFUNCTION-FUNCTION; i < AC.FunctionList.num; i++ ) {
02279             if ( ( T = functions[i].tabl ) != 0 ) {
02280                 for ( j = 0; j < numfuns; j++ ) {
02281                     if ( i == (funs[j]-FUNCTION) && T->spare ) {
02282                         FlipTable(&(functions[i]),1);
02283                         break;
02284                     }
02285                 }
02286             }
02287         }
02288     }
02289     else {
02290         for ( i = MAXBUILTINFUNCTION-FUNCTION; i < AC.FunctionList.num; i++ ) {
02291             if ( ( T = functions[i].tabl ) != 0 ) {
02292                 if ( T->spare ) FlipTable(&(functions[i]),1);
02293             }
02294         }
02295     }
02296 }
02297
02298 /*
02299     #] ApplyReset :
02300     #[ TableReset :
02301 */
02302
02303 void TableReset(void)
02304 {
02305     TABLES T;
02306     int i;
02307
02308     for ( i = MAXBUILTINFUNCTION-FUNCTION; i < AC.FunctionList.num; i++ ) {
02309         if ( ( T = functions[i].tabl ) != 0 && T->spare && T->mode == 0 ) {
02310             functions[i].tabl = T->spare;
02311         }
02312     }
02313 }
02314
02315 /*
02316     #] TableReset :
02317     #[ LoadTableElement :
02318 ?????
02319 int LoadTableElement(DBASE *d, TABLE *T, WORD num)
02320 {
02321 }
02322
02323     #] LoadTableElement :
02324     #[ ReleaseTB :
02325
02326     Releases all TableBases
02327 */
```

```
02328
02329 int ReleaseTB(void)
02330 {
02331     DBASE *d;
02332     int i;
02333     for ( i = NumTableBases - 1; i >= 0; i-- ) {
02334         d = tablebases+i;
02335         fclose(d->handle);
02336         FreeTableBase(d);
02337     }
02338     return(0);
02339 }
02340
02341 /*
02342     #] ReleaseTB :
02343 */
```

4.145 threads.c File Reference

4.145.1 Detailed Description

Routines for the interface of FORM with the pthreads library

This is the main part of the parallelization of TFORM. It is important to also look in the files structs.h and variable.h because the treatment of the A and B structs is essential (these structs are used by means of the macros AM, AP, AC, AS, AR, AT, AN, AO and AX). Also the definitions and use of the macros BHEAD and PHEAD should be looked up.

The sources are set up in such a way that if WITHPTHREADS isn't defined there is no trace of pthread parallelization. The reason is that TFORM is far more memory hungry than sequential FORM.

Special attention should also go to the locks. The proper use of the locks is essential and determines whether TFORM can work at all. We use the LOCK/UNLOCK macros which are empty in the case of sequential FORM These locks are at many places in the source files when workers can interfere with each others data or with the data of the master.

Definition in file threads.c.

4.146 threads.c

Go to the documentation of this file.

```
00001
00022 /* #[ License : */
00023 /*
00024  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00025  *   When using this file you are requested to refer to the publication
00026  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00027  *   This is considered a matter of courtesy as the development was paid
00028  *   for by FOM the Dutch physics granting agency and we would like to
00029  *   be able to track its scientific use to convince FOM of its value
00030  *   for the community.
00031  *
00032  *   This file is part of FORM.
00033  *
00034  *   FORM is free software: you can redistribute it and/or modify it under the
00035  *   terms of the GNU General Public License as published by the Free Software
00036  *   Foundation, either version 3 of the License, or (at your option) any later
00037  *   version.
00038  *
00039  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00040  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00041  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00042  *   details.
00043  *
00044  *   You should have received a copy of the GNU General Public License along
00045  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
```

```
00046  */
00047 /* #] License : */
00048
00049 #ifdef WITHPTHREADS
00050
00051 #define WHOLEBRACKETS
00052 /*
00053     #[ Variables :
00054
00055     The sortbot additions are from 17-may-2007 and after. They constitute
00056     an attempt to make the final merge sorting faster for the master.
00057     This way the master has only one compare per term.
00058     It does add some complexity, but the final merge routine (MasterMerge)
00059     is much simpler for the sortbots. On the other hand the original merging is
00060     for a large part a copy of the MergePatches routine in sort.c and hence
00061     even though complex the bad part has been thoroughly debugged.
00062 */
00063
00064 #include "form3.h"
00065 #include <math.h>
00066
00067 #ifdef WITH_ALARM
00068 // This is only required if we are blocking SIG_ALRM in the worker threads.
00069 #include <signal.h>
00070 #endif
00071
00072 #ifdef WITHFLOAT
00073 #include <gmp.h>
00074
00075 int PackFloat(WORD *,mpf_t);
00076 int UnpackFloat(mpf_t, WORD *);
00077 void RatToFloat(mpf_t result, UWORD *formrat, int ratsize);
00078 #endif
00079
00080 static int numberofthreads;
00081 static int numberofworkers;
00082 static int identityofthreads = 0;
00083 static int *listofavailables;
00084 static int topofavailables = 0;
00085 static pthread_key_t identitykey;
00086 static INILOCK(numberofthreadslock)
00087 static INILOCK(availabilitylock)
00088 static pthread_t *threadpointers = 0;
00089 static pthread_mutex_t *wakeuplocks;
00090 static pthread_mutex_t *wakeupmasterthreadlocks;
00091 static pthread_cond_t *wakeupconditions;
00092 static pthread_condattr_t *wakeupconditionattributes;
00093 static int *wakeup;
00094 static int *wakeupmasterthread;
00095 static INILOCK(wakeupmasterlock)
00096 static pthread_cond_t wakeupmasterconditions = PTHREAD_COND_INITIALIZER;
00097 static pthread_cond_t *wakeupmasterthreadconditions;
00098 static int wakeupmaster = 0;
00099 static int identityretval;
00100 /* static INILOCK(clearclocklock) */
00101 static LONG *timerinfo;
00102 static LONG *sumtimerinfo;
00103 static int numberclaimed;
00104
00105 static THREADBUCKET **threadbuckets, **freebuckets;
00106 static int numthreadbuckets;
00107 static int numberoffullbuckets;
00108
00109 /* static int numberbusy = 0; */
00110
00111 INILOCK(dummylock)
00112 INIRWLOCK(dummyrwlock)
00113 static pthread_cond_t dummywakeupcondition = PTHREAD_COND_INITIALIZER;
00114
00115 #ifdef WITHSORTBOTS
00116 static POSITION SortBotPosition;
00117 static int numberofsortbots;
00118 static INILOCK(wakeupsortbotlock)
00119 static pthread_cond_t wakeupsortbotconditions = PTHREAD_COND_INITIALIZER;
00120 static int topsortbotavailables = 0;
00121 static LONG numberofterms;
00122 #endif
00123
00124 /*
00125     #] Variables :
00126     #[ Identity :
00127         #[ StartIdentity :
00128 */
00134 void StartIdentity(void)
00135 {
00136     pthread_key_create(&identitykey,FinishIdentity);
00137 }
```

```
00138
00139 /*
00140         #] StartIdentity :
00141         #[ FinishIdentity :
00142 */
00147 void FinishIdentity(void *keyp)
00148 {
00149     DUMMYUSE(keyp);
00150 /*  free(keyp); */
00151 }
00152
00153 /*
00154         #] FinishIdentity :
00155         #[ SetIdentity :
00156 */
00161 int SetIdentity(int *identityretval)
00162 {
00163 /*
00164 #ifdef _MSC_VER
00165     printf("addr %d\n",&numberofthreadslock);
00166     printf("size %d\n",sizeof(numberofthreadslock));
00167 #endif
00168 */
00169     LOCK(numberofthreadslock);
00170     *identityretval = identityofthreads++;
00171     UNLOCK(numberofthreadslock);
00172     pthread_setspecific(identitykey,(void *)identityretval);
00173     return(*identityretval);
00174 }
00175
00176 /*
00177         #] SetIdentity :
00178         #[ WhoAmI :
00179 */
00180
00191 int WhoAmI(void)
00192 {
00193     int *identity;
00194 /*
00195     First a fast exit for when there is at most one thread
00196 */
00197     if ( identityofthreads <= 1 ) return(0);
00198 /*
00199     Now the reading of the key.
00200     According to the book the statement should read:
00201
00202     pthread_getspecific(identitykey,(void **)(&identity));
00203
00204     but according to the information in pthread.h it is:
00205 */
00206     identity = (int *)pthread_getspecific(identitykey);
00207     return(*identity);
00208 }
00209
00210 /*
00211         #] WhoAmI :
00212         #[ BeginIdentities :
00213 */
00219 void BeginIdentities(void)
00220 {
00221     StartIdentity();
00222     SetIdentity(&identityretval);
00223 }
00224
00225 /*
00226         #] BeginIdentities :
00227     #] Identity :
00228     #[ StartHandleLock :
00229 */
00236 void StartHandleLock(void)
00237 {
00238     AM.handlelock = dummyrwlock;
00239 }
00240
00241 /*
00242     #] StartHandleLock :
00243     #[ StartAllThreads :
00244 */
00262 int StartAllThreads(int number)
00263 {
00264     int identity, j, dummy, mul;
00265     ALLPRIVATES *B;
00266     pthread_t thethread;
00267     identity = WhoAmI();
00268
00269 #ifdef WITHSORTBOTS
00270     timerinfo = (LONG *)Malloc1(sizeof(LONG)*number*2,"timerinfo");
```

```
00271     sumtimerinfo = (LONG *)Malloc1(sizeof(LONG)*number*2,"sumtimerinfo");
00272     for ( j = 0; j < number*2; j++ ) { timerinfo[j] = 0; sumtimerinfo[j] = 0; }
00273     mul = 2;
00274 #else
00275     timerinfo = (LONG *)Malloc1(sizeof(LONG)*number,"timerinfo");
00276     sumtimerinfo = (LONG *)Malloc1(sizeof(LONG)*number,"sumtimerinfo");
00277     for ( j = 0; j < number; j++ ) { timerinfo[j] = 0; sumtimerinfo[j] = 0; }
00278     mul = 1;
00279 #endif
00280
00281     listofavailables = (int *)Malloc1(sizeof(int)*(number+1),"listofavailables");
00282     threadpointers = (pthread_t *)Malloc1(sizeof(pthread_t)*number*mul,"threadpointers");
00283     AB = (ALLPRIVATES **)Malloc1(sizeof(ALLPRIVATES *)*number*mul,"Private structs");
00284
00285     wakeup = (int *)Malloc1(sizeof(int)*number*mul,"wakeup");
00286     wakeuplocks = (pthread_mutex_t *)Malloc1(sizeof(pthread_mutex_t)*number*mul,"wakeuplocks");
00287     wakeupconditions = (pthread_cond_t
      *)Malloc1(sizeof(pthread_cond_t)*number*mul,"wakeupconditions");
00288     wakeupconditionattributes = (pthread_condattr_t *)
00289             Malloc1(sizeof(pthread_condattr_t)*number*mul,"wakeupconditionattributes");
00290
00291     wakeupmasterthread = (int *)Malloc1(sizeof(int)*number*mul,"wakeupmasterthread");
00292     wakeupmasterthreadlocks = (pthread_mutex_t
      *)Malloc1(sizeof(pthread_mutex_t)*number*mul,"wakeupmasterthreadlocks");
00293     wakeupmasterthreadconditions = (pthread_cond_t
      *)Malloc1(sizeof(pthread_cond_t)*number*mul,"wakeupmasterthread");
00294
00295     numberofthreads = number;
00296     numberofworkers = number - 1;
00297     threadpointers[identity] = pthread_self();
00298     topofavailables = 0;
00299
00300 #ifdef WITH_ALARM
00301     /* During thread creation, we block SIGALRM on the main thread. The created
00302        threads will inherit this. This is required for #timeout to work properly
00303        in TFORM: only the main thread should recieve SIGALRM. */
00304     sigset_t sig_set;
00305     sigemptyset(&sig_set);
00306     sigaddset(&sig_set, SIGALRM);
00307     pthread_sigmask(SIG_BLOCK, &sig_set, NULL);
00308 #endif
00309
00310     for ( j = 1; j < number; j++ ) {
00311         if ( pthread_create(&thethread,NULL,RunThread,(void *)(&dummy)) )
00312             goto failure;
00313     }
00314 /*
00315     Now we initialize the master at the same time that the workers are doing so.
00316 */
00317     B = InitializeOneThread(identity);
00318     AR.infile = &(AR.Fscr[0]);
00319     AR.outfile = &(AR.Fscr[1]);
00320     AR.hidefile = &(AR.Fscr[2]);
00321     AM.sbuflock = dummylock;
00322     AS.inputslock = dummylock;
00323     AS.outputslock = dummylock;
00324     AS.MaxExprSizeLock = dummylock;
00325     AP.PreVarLock = dummylock;
00326     AC.halfmodlock = dummylock;
00327     MakeThreadBuckets(number,0);
00328 /*
00329     Now we wait for the workers to finish their startup.
00330     We don't want to initialize the sortbots yet and run the risk that
00331     some of them may end up with a lower number than one of the workers.
00332 */
00333     MasterWaitAll();
00334 #ifdef WITHSORTBOTS
00335     if ( numberofworkers > 2 ) {
00336         numberofsortbots = numberofworkers-2;
00337         for ( j = numberofworkers+1; j < 2*numberofworkers-1; j++ ) {
00338             if ( pthread_create(&thethread,NULL,RunSortBot,(void *)(&dummy)) )
00339                 goto failure;
00340         }
00341     }
00342     else {
00343         numberofsortbots = 0;
00344     }
00345     MasterWaitAllSortBots();
00346     DefineSortBotTree();
00347 #endif
00348     IniSortBlocks(number-1);
00349     AS.MasterSort = 0;
00350     AM.storefilelock = dummylock;
00351
00352 #ifdef WITH_ALARM
00353     /* Now we allow the main thread to recieve SIGALRM again. */
00354     pthread_sigmask(SIG_UNBLOCK, &sig_set, NULL);
```

```
00355 #endif
00356
00357 /*
00358 MesPrint("AB = %x %x %x  %d",AB[0],AB[1],AB[2], identityofthreads);
00359 */
00360     return(0);
00361 /* INTERNAL_ERROR_EXCL_START */
00362 failure:
00363     MesPrint("!>Cannot start %d threads",number);
00364     Terminate(-1);
00365     return(-1);
00366 /* INTERNAL_ERROR_EXCL_STOP */
00367 }
00368
00369 /*
00370     #] StartAllThreads :
00371     #[ InitializeOneThread :
00372 */
00376 UBYTE *scratchname[] = { (UBYTE *)"scratchsize",
00377                          (UBYTE *)"scratchsize",
00378                          (UBYTE *)"hidesize" };
00405 ALLPRIVATES *InitializeOneThread(int identity)
00406 {
00407     WORD *t, *ScratchBuf;
00408     int i, j, bsize, *bp;
00409     LONG ScratchSize[3], IOsize;
00410     ALLPRIVATES *B;
00411     UBYTE *s;
00412
00413     wakeup[identity] = 0;
00414     wakeuplocks[identity] = dummylock;
00415     pthread_condattr_init(&(wakeupconditionattributes[identity]));
00416     pthread_condattr_setpshared(&(wakeupconditionattributes[identity]),PTHREAD_PROCESS_PRIVATE);
00417     wakeupconditions[identity] = dummywakeupcondition;
00418     pthread_cond_init(&(wakeupconditions[identity]),&(wakeupconditionattributes[identity]));
00419     wakeupmasterthread[identity] = 0;
00420     wakeupmasterthreadlocks[identity] = dummylock;
00421     wakeupmasterthreadconditions[identity] = dummywakeupcondition;
00422
00423     bsize = sizeof(ALLPRIVATES);
00424     bsize = (bsize+sizeof(int)-1)/sizeof(int);
00425     B = (ALLPRIVATES *)Malloc1(sizeof(int)*bsize,"B struct");
00426     for ( bp = (int *)B, j = 0; j < bsize; j++ ) *bp++ = 0;
00427
00428     AB[identity] = B;
00429 /*
00430             12-jun-2007 JV:
00431     For the timing one has to know a few things:
00432     The POSIX standard is that there is only a single process ID and that
00433     getrusage returns the time of all the threads together.
00434     Under Linux there are two methods though: The older LinuxThreads and NPTL.
00435     LinuxThreads gives each thread its own process id. This makes that we
00436     can time the threads with getrusage, and hence this was done. Under NPTL
00437     this has been 'corrected' and suddenly getruage doesn't work anymore the
00438     way it used to. Now we need
00439         clock_gettime(CLOCK_THREAD_CPUTIME_ID,&timing)
00440     which is declared in <time.h> and we need -lrt extra in the link statement.
00441     (this is at least the case on blade02 at DESY-Zeuthen).
00442     See also the code in tools.c at the routine Timer.
00443     We may still have to include more stuff there.
00444 */
00445     if ( identity > 0 ) TimeCPU(0);
00446
00447 #ifdef WITHSORTBOTS
00448
00449     if ( identity > numberofworkers ) {
00450 /*
00451         Some workspace is needed when we have a PolyFun and we have to add
00452         two terms and the new result is going to be longer than the old result.
00453 */
00454         LONG length = AM.WorkSize*sizeof(WORD)/8+AM.MaxTer*2;
00455         AT.WorkSpace = (WORD *)Malloc1(length,"WorkSpace");
00456         AT.WorkTop = AT.WorkSpace + length/sizeof(WORD);
00457         AT.WorkPointer = AT.WorkSpace;
00458         AT.identity = identity;
00459 /*
00460         The SB struct gets treated in IniSortBlocks.
00461         The SortBotIn variables will be defined DefineSortBotTree.
00462 */
00463         if ( AN.SoScratC == 0 ) {
00464             AN.SoScratC = (UWORD *)Malloc1(2*(AM.MaxTal+2)*sizeof(UWORD),"Scratch in SortBot");
00465         }
00466         AT.SS = (SORTING *)Malloc1(sizeof(SORTING),"dummy sort buffer");
00467         AT.SS->PolyFlag = 0;
00468
00469         AT.comsym[0] = 8;
00470         AT.comsym[1] = SYMBOL;
```

```
00471         AT.comsym[2] = 4;
00472         AT.comsym[3] = 0;
00473         AT.comsym[4] = 1;
00474         AT.comsym[5] = 1;
00475         AT.comsym[6] = 1;
00476         AT.comsym[7] = 3;
00477         AT.comnum[0] = 4;
00478         AT.comnum[1] = 1;
00479         AT.comnum[2] = 1;
00480         AT.comnum[3] = 3;
00481         AT.comfun[0] = FUNHEAD+4;
00482         AT.comfun[1] = FUNCTION;
00483         AT.comfun[2] = FUNHEAD;
00484         AT.comfun[3] = 0;
00485 #if FUNHEAD > 3
00486         for ( i = 4; i <= FUNHEAD; i++ )
00487             AT.comfun[i] = 0;
00488 #endif
00489         AT.comfun[FUNHEAD+1] = 1;
00490         AT.comfun[FUNHEAD+2] = 1;
00491         AT.comfun[FUNHEAD+3] = 3;
00492         AT.comind[0] = 7;
00493         AT.comind[1] = INDEX;
00494         AT.comind[2] = 3;
00495         AT.comind[3] = 0;
00496         AT.comind[4] = 1;
00497         AT.comind[5] = 1;
00498         AT.comind[6] = 3;
00499 
00500         AT.inprimelist = -1;
00501         AT.sizeprimelist = 0;
00502         AT.primelist = 0;
00503         AT.bracketinfo = 0;
00504 
00505         AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
00506 
00507         AR.sLevel = 0;
00508         AR.wranfia = 0;
00509         AR.wranfcall = 0;
00510         AR.wranfnpair1 = NPAIR1;
00511         AR.wranfnpair2 = NPAIR2;
00512         AN.NumFunSorts = 5;
00513         AN.MaxFunSorts = 5;
00514         AN.SplitScratch = 0;
00515         AN.SplitScratchSize = AN.InScratch = 0;
00516         AN.SplitScratch1 = 0;
00517         AN.SplitScratchSize1 = AN.InScratch1 = 0;
00518 
00519         AN.FunSorts = (SORTING **)Malloc1((AN.NumFunSorts+1)*sizeof(SORTING *),"FunSort pointers");
00520         for ( i = 0; i <= AN.NumFunSorts; i++ ) AN.FunSorts[i] = 0;
00521         AN.FunSorts[0] = AT.S0 = AT.SS;
00522         AN.idfunctionflag = 0;
00523         AN.tryterm = 0;
00524 
00525         return(B);
00526     }
00527     if ( identity == 0 && AN.SoScratC == 0 ) {
00528         AN.SoScratC = (UWORD *)Malloc1(2*(AM.MaxTal+2)*sizeof(UWORD),"Scratch in SortBot");
00529     }
00530 #endif
00531     AR.CurDum = AM.IndDum;
00532     for ( j = 0; j < 3; j++ ) {
00533         if ( identity == 0 ) {
00534             if ( j == 2 ) {
00535                 ScratchSize[j] = AM.HideSize;
00536             }
00537             else {
00538                 ScratchSize[j] = AM.ScratSize;
00539             }
00540             if ( ScratchSize[j] < 10*AM.MaxTer ) ScratchSize[j] = 10 * AM.MaxTer;
00541         }
00542         else {
00543 /*
00544             ScratchSize[j] = AM.ScratSize / (numberofthreads-1);
00545             ScratchSize[j] = ScratchSize[j] / 20;
00546             if ( ScratchSize[j] < 10*AM.MaxTer ) ScratchSize[j] = 10 * AM.MaxTer;
00547 */
00548             if ( j == 1 ) ScratchSize[j] = AM.ThreadScratOutSize;
00549             else          ScratchSize[j] = AM.ThreadScratSize;
00550             if ( ScratchSize[j] < 4*AM.MaxTer ) ScratchSize[j] = 4 * AM.MaxTer;
00551             AR.Fscr[j].name = 0;
00552         }
00553         ScratchSize[j] = ( ScratchSize[j] + 255 ) / 256;
00554         ScratchSize[j] = ScratchSize[j] * 256;
00555         ScratchBuf = (WORD *)Malloc1(ScratchSize[j]*sizeof(WORD),(char *)(scratchname[j]));
00556         AR.Fscr[j].POsize = ScratchSize[j] * sizeof(WORD);
00557         AR.Fscr[j].POfull = AR.Fscr[j].POfill = AR.Fscr[j].PObuffer = ScratchBuf;
```

```
00558         AR.Fscr[j].POstop = AR.Fscr[j].PObuffer + ScratchSize[j];
00559         PUTZERO(AR.Fscr[j].POposition);
00560         AR.Fscr[j].pthreadslock = dummylock;
00561         AR.Fscr[j].wPOsize = AR.Fscr[j].POsize;
00562         AR.Fscr[j].wPObuffer = AR.Fscr[j].PObuffer;
00563         AR.Fscr[j].wPOfill = AR.Fscr[j].POfill;
00564         AR.Fscr[j].wPOfull = AR.Fscr[j].POfull;
00565         AR.Fscr[j].wPOstop = AR.Fscr[j].POstop;
00566     }
00567     AR.InInBuf = 0;
00568     AR.InHiBuf = 0;
00569     AR.Fscr[0].handle = -1;
00570     AR.Fscr[1].handle = -1;
00571     AR.Fscr[2].handle = -1;
00572     AR.FoStage4[0].handle = -1;
00573     AR.FoStage4[1].handle = -1;
00574     IOsize = AM.S0->file.POsize;
00575 #ifdef WITHZLIB
00576     AR.FoStage4[0].ziosize = IOsize;
00577     AR.FoStage4[1].ziosize = IOsize;
00578     AR.FoStage4[0].ziobuffer = 0;
00579     AR.FoStage4[1].ziobuffer = 0;
00580 #endif
00581     AR.FoStage4[0].POsize  = ((IOsize+sizeof(WORD)-1)/sizeof(WORD))*sizeof(WORD);
00582     AR.FoStage4[1].POsize  = ((IOsize+sizeof(WORD)-1)/sizeof(WORD))*sizeof(WORD);
00583
00584     AR.hidefile = &(AR.Fscr[2]);
00585     AR.StoreData.Handle = -1;
00586     AR.SortType = AC.SortType;
00587
00588     AN.IndDum = AM.IndDum;
00589
00590     if ( identity > 0 ) {
00591         s = (UBYTE *)(FG.fname); i = 0;
00592         while ( *s ) { s++; i++; }
00593         s = (UBYTE *)Malloc1(sizeof(char)*(i+12),"name for Fscr[0] file");
00594         snprintf((char *)s,i+12,"%s.%d",FG.fname,identity);
00595         s[i-3] = 's'; s[i-2] = 'c'; s[i-1] = '0';
00596         AR.Fscr[0].name = (char *)s;
00597         s = (UBYTE *)(FG.fname); i = 0;
00598         while ( *s ) { s++; i++; }
00599         s = (UBYTE *)Malloc1(sizeof(char)*(i+12),"name for Fscr[1] file");
00600         snprintf((char *)s,i+12,"%s.%d",FG.fname,identity);
00601         s[i-3] = 's'; s[i-2] = 'c'; s[i-1] = '1';
00602         AR.Fscr[1].name = (char *)s;
00603     }
00604
00605     AR.CompressBuffer = (WORD *)Malloc1((AM.CompressSize+10)*sizeof(WORD),"compresssize");
00606     AR.ComprTop = AR.CompressBuffer + AM.CompressSize;
00607     AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
00608 /*
00609     Here we make all allocations for the struct AT
00610     (which is AB[identity].T or B->T with B = AB+identity).
00611 */
00612     AT.WorkSpace = (WORD *)Malloc1(AM.WorkSize*sizeof(WORD),"WorkSpace");
00613     AT.WorkTop = AT.WorkSpace + AM.WorkSize;
00614     AT.WorkPointer = AT.WorkSpace;
00615
00616     AT.Nest = (NESTING)Malloc1((LONG)sizeof(struct NeStInG)*AM.maxFlevels,"functionlevels");
00617     AT.NestStop = AT.Nest + AM.maxFlevels;
00618     AT.NestPoin = AT.Nest;
00619
00620     AT.WildMask = (WORD *)Malloc1((LONG)AM.MaxWildcards*sizeof(WORD),"maxwildcards");
00621
00622     LOCK(availabilitylock);
00623     AT.ebufnum = inicbufs();        /* Buffer for extras during execution */
00624     AT.fbufnum = inicbufs();        /* Buffer for caching in factorization */
00625     AT.allbufnum = inicbufs();      /* Buffer for id,all */
00626     AT.aebufnum = inicbufs();       /* Buffer for id,all */
00627     UNLOCK(availabilitylock);
00628
00629     AT.RepCount = (int *)Malloc1((LONG)((AM.RepMax+3)*sizeof(int)),"repeat buffers");
00630     AN.RepPoint = AT.RepCount;
00631     AN.polysortflag = 0;
00632     AN.subsubveto = 0;
00633     AN.tryterm = 0;
00634     AT.RepTop = AT.RepCount + AM.RepMax;
00635
00636     AT.WildArgTaken = (WORD *)Malloc1((LONG)AC.WildcardBufferSize*sizeof(WORD)/2
00637                 ,"argument list names");
00638     AT.WildcardBufferSize = AC.WildcardBufferSize;
00639     AT.previousEfactor = 0;
00640
00641     AT.identity = identity;
00642     AT.LoadBalancing = 0;
00643 /*
00644     Still to do: the SS stuff.
```

```
00645                 the Fscr[3]
00646                 the FoStage4[2]
00647 */
00648     if ( AT.WorkSpace == 0 ||
00649          AT.Nest == 0 ||
00650          AT.WildMask == 0 ||
00651          AT.RepCount == 0 ||
00652          AT.WildArgTaken == 0 ) goto OnError;
00653 /*
00654     And initializations
00655 */
00656     AT.comsym[0] = 8;
00657     AT.comsym[1] = SYMBOL;
00658     AT.comsym[2] = 4;
00659     AT.comsym[3] = 0;
00660     AT.comsym[4] = 1;
00661     AT.comsym[5] = 1;
00662     AT.comsym[6] = 1;
00663     AT.comsym[7] = 3;
00664     AT.comnum[0] = 4;
00665     AT.comnum[1] = 1;
00666     AT.comnum[2] = 1;
00667     AT.comnum[3] = 3;
00668     AT.comfun[0] = FUNHEAD+4;
00669     AT.comfun[1] = FUNCTION;
00670     AT.comfun[2] = FUNHEAD;
00671     AT.comfun[3] = 0;
00672 #if FUNHEAD > 3
00673     for ( i = 4; i <= FUNHEAD; i++ )
00674         AT.comfun[i] = 0;
00675 #endif
00676     AT.comfun[FUNHEAD+1] = 1;
00677     AT.comfun[FUNHEAD+2] = 1;
00678     AT.comfun[FUNHEAD+3] = 3;
00679     AT.comind[0] = 7;
00680     AT.comind[1] = INDEX;
00681     AT.comind[2] = 3;
00682     AT.comind[3] = 0;
00683     AT.comind[4] = 1;
00684     AT.comind[5] = 1;
00685     AT.comind[6] = 3;
00686     AT.locwildvalue[0] = SUBEXPRESSION;
00687     AT.locwildvalue[1] = SUBEXPSIZE;
00688     for ( i = 2; i < SUBEXPSIZE; i++ ) AT.locwildvalue[i] = 0;
00689     AT.mulpat[0] = TYPEMULT;
00690     AT.mulpat[1] = SUBEXPSIZE+3;
00691     AT.mulpat[2] = 0;
00692     AT.mulpat[3] = SUBEXPRESSION;
00693     AT.mulpat[4] = SUBEXPSIZE;
00694     AT.mulpat[5] = 0;
00695     AT.mulpat[6] = 1;
00696     for ( i = 7; i < SUBEXPSIZE+5; i++ ) AT.mulpat[i] = 0;
00697     AT.proexp[0] = SUBEXPSIZE+4;
00698     AT.proexp[1] = EXPRESSION;
00699     AT.proexp[2] = SUBEXPSIZE;
00700     AT.proexp[3] = -1;
00701     AT.proexp[4] = 1;
00702     for ( i = 5; i < SUBEXPSIZE+1; i++ ) AT.proexp[i] = 0;
00703     AT.proexp[SUBEXPSIZE+1] = 1;
00704     AT.proexp[SUBEXPSIZE+2] = 1;
00705     AT.proexp[SUBEXPSIZE+3] = 3;
00706     AT.proexp[SUBEXPSIZE+4] = 0;
00707     AT.dummysubexp[0] = SUBEXPRESSION;
00708     AT.dummysubexp[1] = SUBEXPSIZE+4;
00709     for ( i = 2; i < SUBEXPSIZE; i++ ) AT.dummysubexp[i] = 0;
00710     AT.dummysubexp[SUBEXPSIZE] = WILDDUMMY;
00711     AT.dummysubexp[SUBEXPSIZE+1] = 4;
00712     AT.dummysubexp[SUBEXPSIZE+2] = 0;
00713     AT.dummysubexp[SUBEXPSIZE+3] = 0;
00714
00715     AT.MinVecArg[0] = 7+ARGHEAD;
00716     AT.MinVecArg[ARGHEAD] = 7;
00717     AT.MinVecArg[1+ARGHEAD] = INDEX;
00718     AT.MinVecArg[2+ARGHEAD] = 3;
00719     AT.MinVecArg[3+ARGHEAD] = 0;
00720     AT.MinVecArg[4+ARGHEAD] = 1;
00721     AT.MinVecArg[5+ARGHEAD] = 1;
00722     AT.MinVecArg[6+ARGHEAD] = -3;
00723     t = AT.FunArg;
00724     *t++ = 4+ARGHEAD+FUNHEAD;
00725     for ( i = 1; i < ARGHEAD; i++ ) *t++ = 0;
00726     *t++ = 4+FUNHEAD;
00727     *t++ = 0;
00728     *t++ = FUNHEAD;
00729     for ( i = 2; i < FUNHEAD; i++ ) *t++ = 0;
00730     *t++ = 1; *t++ = 1; *t++ = 3;
00731
```

```
00732     AT.inprimelist = -1;
00733     AT.sizeprimelist = 0;
00734     AT.primelist = 0;
00735     AT.nfac = AT.nBer = 0;
00736     AT.factorials = 0;
00737     AT.bernoullis = 0;
00738     AR.wranfia = 0;
00739     AR.wranfcall = 0;
00740     AR.wranfnpair1 = NPAIR1;
00741     AR.wranfnpair2 = NPAIR2;
00742     AR.wranfseed = 0;
00743
00744     AT.NormData = Malloc1(sizeof(*(AT.NormData)), "NormData thread pointers");
00745     AT.NormDataSize = 1;
00746     AT.NormData[0] = AllocNormData();
00747     AT.NormDepth = 0;
00748
00749     AN.SplitScratch = 0;
00750     AN.SplitScratchSize = AN.InScratch = 0;
00751     AN.SplitScratch1 = 0;
00752     AN.SplitScratchSize1 = AN.InScratch1 = 0;
00753 /*
00754     Now the sort buffers. They depend on which thread. The master
00755     inherits the sortbuffer from AM.S0
00756 */
00757     if ( identity == 0 ) {
00758         AT.S0 = AM.S0;
00759     }
00760     else {
00761 /*
00762         For the moment we don't have special settings.
00763         They may become costly in virtual memory.
00764 */
00765         AT.S0 = AllocSort(AM.S0->LargeSize*sizeof(WORD)/numberofworkers
00766                         ,AM.S0->SmallSize*sizeof(WORD)/numberofworkers
00767                         ,AM.S0->SmallEsize*sizeof(WORD)/numberofworkers
00768                         ,AM.S0->TermsInSmall
00769                         ,AM.S0->MaxPatches
00770 /*                      ,AM.S0->MaxPatches/numberofworkers  */
00771                         ,AM.S0->MaxFpatches/numberofworkers
00772                         ,AM.S0->file.POsize
00773                         ,0);
00774     }
00775     AR.CompressPointer = AR.CompressBuffer;
00776 /*
00777     Install the store caches (15-aug-2006 JV)
00778 */
00779     AT.StoreCache = AT.StoreCacheAlloc = 0;
00780     if ( AM.NumStoreCaches > 0 ) {
00781         STORECACHE sa, sb;
00782         LONG size;
00783         size = sizeof(struct StOrEcAcHe)+AM.SizeStoreCache;
00784         size = ((size-1)/sizeof(size_t)+1)*sizeof(size_t);
00785         AT.StoreCacheAlloc = (STORECACHE)Malloc1(size*AM.NumStoreCaches,"StoreCaches");
00786         sa = AT.StoreCache = AT.StoreCacheAlloc;
00787         for ( i = 0; i < AM.NumStoreCaches; i++ ) {
00788             sb = (STORECACHE)(void *)((UBYTE *)sa+size);
00789             if ( i == AM.NumStoreCaches-1 ) {
00790                 sa->next = 0;
00791             }
00792             else {
00793                 sa->next = sb;
00794             }
00795             SETBASEPOSITION(sa->position,-1);
00796             SETBASEPOSITION(sa->toppos,-1);
00797             sa = sb;
00798         }
00799     }
00800
00801     ReserveTempFiles(2);
00802     return(B);
00803 /* INTERNAL_ERROR_EXCL_START */
00804 OnError:;
00805     MLOCK(ErrorMessageLock);
00806     MesPrint("!>Error initializing thread %d",identity);
00807     MUNLOCK(ErrorMessageLock);
00808     Terminate(-1);
00809     return(B);
00810 /* INTERNAL_ERROR_EXCL_STOP */
00811 }
00812
00813 /*
00814     #] InitializeOneThread :
00815     #[ FinalizeOneThread :
00816 */
00827 void FinalizeOneThread(int identity)
00828 {
```

```
00829     timerinfo[identity] = TimeCPU(1);
00830 }
00831
00832 /*
00833     #] FinalizeOneThread :
00834     #[ ClearAllThreads :
00835 */
00842 void ClearAllThreads(void)
00843 {
00844     int i;
00845     MasterWaitAll();
00846     for ( i = 1; i <= numberofworkers; i++ ) {
00847         WakeupThread(i,CLEARCLOCK);
00848     }
00849 #ifdef WITHSORTBOTS
00850     for ( i = numberofworkers+1; i <= numberofworkers+numberofsortbots; i++ ) {
00851         WakeupThread(i,CLEARCLOCK);
00852     }
00853 #endif
00854 }
00855
00856 /*
00857     #] ClearAllThreads :
00858     #[ TerminateAllThreads :
00859 */
00866 void TerminateAllThreads(void)
00867 {
00868     int i;
00869     for ( i = 1; i <= numberofworkers; i++ ) {
00870         GetThread(i);
00871         WakeupThread(i,TERMINATETHREAD);
00872     }
00873 #ifdef WITHSORTBOTS
00874     for ( i = numberofworkers+1; i <= numberofworkers+numberofsortbots; i++ ) {
00875         WakeupThread(i,TERMINATETHREAD);
00876     }
00877 #endif
00878     for ( i = 1; i <= numberofworkers; i++ ) {
00879         pthread_join(threadpointers[i],NULL);
00880     }
00881 #ifdef WITHSORTBOTS
00882     for ( i = numberofworkers+1; i <= numberofworkers+numberofsortbots; i++ ) {
00883         pthread_join(threadpointers[i],NULL);
00884     }
00885 #endif
00886 }
00887
00888 /*
00889     #] TerminateAllThreads :
00890     #[ MakeThreadBuckets :
00891 */
00917 int MakeThreadBuckets(int number, int par)
00918 {
00919     int i;
00920     LONG sizethreadbuckets;
00921     THREADBUCKET *thr;
00922
00923     // Here we divide by 4, which has been the behaviour with the default
00924     // AC.ThreadBucketSize for a long time.
00925     // MAXTER is the default value of AM.MaxTer, in units of sizeof(WORD).
00926     sizethreadbuckets = (AC.ThreadBucketSize*MAXTER)/4;
00927     // Now we scale up the buffer logarithmically with the user AM.MaxTer.
00928     // If we scale linearly, we end up with really enormous buffers here
00929     // when AM.MaxTer is of the order of millions of WORDs.
00930     float scale = 1.0;
00931     if ( AM.MaxTer/sizeof(WORD) > MAXTER ) {
00932         scale += log(((float)AM.MaxTer/sizeof(WORD))/MAXTER);
00933     }
00934     sizethreadbuckets = (LONG)((float)sizethreadbuckets*scale);
00935     // Nonetheless, we must fit at least BUCKETMINTERMS terms in each bucket!
00936     // So the buffer will eventually scale linearly with MaxTer anyway, but
00937     // much less aggressively than the old code.
00938     sizethreadbuckets = MaX((ULONG)sizethreadbuckets, BUCKETMINTERMS*AM.MaxTer/sizeof(WORD));
00939
00940     if ( par == 0 ) {
00941         numthreadbuckets = 2*(number-1);
00942         threadbuckets = (THREADBUCKET **)Malloc1(numthreadbuckets*sizeof(THREADBUCKET
    *),"threadbuckets");
00943         freebuckets   = (THREADBUCKET **)Malloc1(numthreadbuckets*sizeof(THREADBUCKET
    *),"freebuckets");
00944     }
00945     if ( par > 0 ) {
00946         if ( sizethreadbuckets <= threadbuckets[0]->threadbuffersize ) return(0);
00947         for ( i = 0; i < numthreadbuckets; i++ ) {
00948             thr = threadbuckets[i];
00949             M_free(thr->deferbuffer,"deferbuffer");
00950             M_free(thr->threadbuffer,"threadbuffer");
```

```
00951             M_free(thr->compressbuffer,"compressbuffer");
00952         }
00953     }
00954     else {
00955         for ( i = 0; i < numthreadbuckets; i++ ) {
00956             threadbuckets[i] = (THREADBUCKET *)Malloc1(sizeof(THREADBUCKET),"threadbuckets");
00957             threadbuckets[i]->lock = dummylock;
00958         }
00959     }
00960     for ( i = 0; i < numthreadbuckets; i++ ) {
00961         thr = threadbuckets[i];
00962         thr->threadbuffersize = sizethreadbuckets;
00963         // This buffer does not need to be so large, start it at MaxTer.
00964         // We'll double it if necessary.
00965         thr->compressbuffersize = AM.MaxTer/sizeof(WORD);
00966         thr->free = BUCKETFREE;
00967         thr->deferbuffer = (POSITION
      *)Malloc1((AC.ThreadBucketSize+1)*sizeof(POSITION),"deferbuffer");
00968         thr->threadbuffer   = (WORD *)Malloc1(sizethreadbuckets*sizeof(WORD),"threadbuffer");
00969         thr->compressbuffer = (WORD *)Malloc1(thr->compressbuffersize*sizeof(WORD),"compressbuffer");
00970         thr->busy = BUCKETPREPARINGTERM;
00971         thr->usenum = thr->totnum = 0;
00972         thr->type = BUCKETDOINGTERMS;
00973     }
00974     return(0);
00975 }
00976
00977 /*
00978     #] MakeThreadBuckets :
00979     #[ GetTimerInfo :
00980 */
00981
00987 /* UNFINISHED_FEATURE_EXCL_START */
00988 int GetTimerInfo(LONG** ti,LONG** sti)
00989 {
00990     *ti = timerinfo;
00991     *sti = sumtimerinfo;
00992 #ifdef WITHSORTBOTS
00993     return AM.totalnumberofthreads*2;
00994 #else
00995     return AM.totalnumberofthreads;
00996 #endif
00997 }
00998 /* UNFINISHED_FEATURE_EXCL_STOP */
00999 /*
01000     #] GetTimerInfo :
01001     #[ WriteTimerInfo :
01002 */
01003
01008 /* UNFINISHED_FEATURE_EXCL_START */
01009 void WriteTimerInfo(LONG* ti,LONG* sti)
01010 {
01011     int i;
01012 #ifdef WITHSORTBOTS
01013     int max = AM.totalnumberofthreads*2;
01014 #else
01015     int max = AM.totalnumberofthreads;
01016 #endif
01017     for ( i=0; i<max; ++i ) {
01018         timerinfo[i] = ti[i];
01019         sumtimerinfo[i] = sti[i];
01020     }
01021 }
01022 /* UNFINISHED_FEATURE_EXCL_STOP */
01023 /*
01024     #] WriteTimerInfo :
01025     #[ GetWorkerTimes :
01026 */
01032 LONG GetWorkerTimes(void)
01033 {
01034     LONG retval = 0;
01035     int i;
01036     for ( i = 1; i <= numberofworkers; i++ ) retval += timerinfo[i] + sumtimerinfo[i];
01037 #ifdef WITHSORTBOTS
01038     for ( i = numberofworkers+1; i <= numberofworkers+numberofsortbots; i++ )
01039         retval += timerinfo[i] + sumtimerinfo[i];
01040 #endif
01041     return(retval);
01042 }
01043
01044 /*
01045     #] GetWorkerTimes :
01046     #[ UpdateOneThread :
01047 */
01054 int UpdateOneThread(int identity)
01055 {
01056     ALLPRIVATES *B = AB[identity], *B0 = AB[0];
```

```
01057     AR.GetFile = AR0.GetFile;
01058     AR.KeptInHold = AR0.KeptInHold;
01059     AR.CurExpr = AR0.CurExpr;
01060     AR.SortType = AC.SortType;
01061     if ( AT.WildcardBufferSize < AC.WildcardBufferSize ) {
01062         M_free(AT.WildArgTaken,"argument list names");
01063         AT.WildcardBufferSize = AC.WildcardBufferSize;
01064         AT.WildArgTaken = (WORD *)Malloc1((LONG)AC.WildcardBufferSize*sizeof(WORD)/2
01065                 ,"argument list names");
01066         if ( AT.WildArgTaken == 0 ) return(-1);
01067     }
01068     return(0);
01069 }
01070
01071 /*
01072     #] UpdateOneThread :
01073     #[ LoadOneThread :
01074 */
01088 int LoadOneThread(int from, int identity, THREADBUCKET *thr, int par)
01089 {
01090     WORD *t1, *t2;
01091     ALLPRIVATES *B = AB[identity], *B0 = AB[from];
01092
01093     AR.DefPosition = AR0.DefPosition;
01094     AR.NoCompress = AR0.NoCompress;
01095     AR.gzipCompress = AR0.gzipCompress;
01096     AR.BracketOn = AR0.BracketOn;
01097     AR.CurDum = AR0.CurDum;
01098     AR.DeferFlag = AR0.DeferFlag;
01099     AR.TePos = 0;
01100     AR.sLevel = AR0.sLevel;
01101     AR.Stage4Name = AR0.Stage4Name;
01102     AR.GetOneFile = AR0.GetOneFile;
01103     AR.PolyFun = AR0.PolyFun;
01104     AR.PolyFunInv = AR0.PolyFunInv;
01105     AR.PolyFunType = AR0.PolyFunType;
01106     AR.PolyFunExp = AR0.PolyFunExp;
01107     AR.PolyFunVar = AR0.PolyFunVar;
01108     AR.PolyFunPow = AR0.PolyFunPow;
01109     AR.Eside = AR0.Eside;
01110     AR.Cnumlhs = AR0.Cnumlhs;
01111 /*
01112     AR.MaxBracket = AR0.MaxBracket;
01113
01114     The compressbuffer contents are mainly relevant for keep brackets
01115     We should do this only if there is a keep brackets statement
01116     We may however still need the compressbuffer for expressions in the rhs.
01117 */
01118     if ( par >= 1 ) {
01119 /*
01120         We may not need this %%%%% 7-apr-2006
01121 */
01122         t1 = AR.CompressBuffer; t2 = AR0.CompressBuffer;
01123         while ( t2 < AR0.CompressPointer ) *t1++ = *t2++;
01124         AR.CompressPointer = t1;
01125
01126     }
01127     else {
01128         AR.CompressPointer = AR.CompressBuffer;
01129     }
01130     if ( AR.DeferFlag ) {
01131         if ( AR.infile->handle < 0 ) {
01132             AR.infile->POfill = AR0.infile->POfill;
01133         }
01134         else {
01135 /*
01136             We have to set the value of POposition to something that will
01137             force a read in the first try.
01138 */
01139             AR.infile->POfull = AR.infile->POfill = AR.infile->PObuffer;
01140         }
01141     }
01142     if ( par == 0 ) {
01143         AN.threadbuck = thr;
01144         AN.ninterms = thr->firstterm;
01145     }
01146     else if ( par == 1 ) {
01147         WORD *tstop;
01148         t1 = thr->threadbuffer; tstop = t1 + *t1;
01149         t2 = AT.WorkPointer;
01150         while ( t1 < tstop ) *t2++ = *t1++;
01151         AN.ninterms = thr->firstterm;
01152     }
01153     AN.TeInFun = 0;
01154     AN.ncmod = AC.ncmod;
01155     AT.BrackBuf = AT0.BrackBuf;
01156     AT.bracketindexflag = AT0.bracketindexflag;
```

```
01157     AN.PolyFunTodo = 0;
01158 /*
01159     The relevant variables and the term are in their place.
01160     There is nothing more to do.
01161 */
01162     return(0);
01163 }
01164
01165 /*
01166     #] LoadOneThread :
01167     #[ BalanceRunThread :
01168 */
01183 /* UNFINISHED_FEATURE_EXCL_START */
01184 int BalanceRunThread(PHEAD int identity, WORD *term, WORD level)
01185 {
01186     GETBIDENTITY
01187     ALLPRIVATES *BB;
01188     WORD *t, *tt;
01189     int i, *ti, *tti;
01190
01191     LoadOneThread(AT.identity,identity,0,2);
01192 /*
01193     Extra loading if needed. Quantities changed in Generator.
01194     Like the level that has to be passed.
01195 */
01196     BB = AB[identity];
01197     BB->R.level = level;
01198     BB->T.TMbuff = AT.TMbuff;
01199     ti = AT.RepCount; tti = BB->T.RepCount;
01200     i = AN.RepPoint - AT.RepCount;
01201     BB->N.RepPoint = BB->T.RepCount + i;
01202     for ( ; i >= 0; i-- ) tti[i] = ti[i];
01203
01204     t = term; i = *term;
01205     tt = BB->T.WorkSpace;
01206     NCOPY(tt,t,i);
01207     BB->T.WorkPointer = tt;
01208
01209     WakeupThread(identity,HIGHERLEVELGENERATION);
01210
01211     return(0);
01212 }
01213 /* UNFINISHED_FEATURE_EXCL_STOP */
01214 /*
01215     #] BalanceRunThread :
01216     #[ SetWorkerFiles :
01217 */
01222 void SetWorkerFiles(void)
01223 {
01224     int id;
01225     ALLPRIVATES *B, *B0 = AB[0];
01226     for ( id = 1; id < AM.totalnumberofthreads; id++ ) {
01227         B = AB[id];
01228         AR.infile = &(AR.Fscr[0]);
01229         AR.outfile = &(AR.Fscr[1]);
01230         AR.hidefile = &(AR.Fscr[2]);
01231         AR.infile->handle = AR0.infile->handle;
01232         AR.hidefile->handle = AR0.hidefile->handle;
01233         if ( AR.infile->handle < 0 ) {
01234             AR.infile->PObuffer = AR0.infile->PObuffer;
01235             AR.infile->POstop = AR0.infile->POstop;
01236             AR.infile->POfill = AR0.infile->POfill;
01237             AR.infile->POfull = AR0.infile->POfull;
01238             AR.infile->POsize = AR0.infile->POsize;
01239             AR.InInBuf = AR0.InInBuf;
01240             AR.infile->POposition = AR0.infile->POposition;
01241             AR.infile->filesize = AR0.infile->filesize;
01242         }
01243         else {
01244             AR.infile->PObuffer = AR.infile->wPObuffer;
01245             AR.infile->POstop = AR.infile->wPOstop;
01246             AR.infile->POfill = AR.infile->wPOfill;
01247             AR.infile->POfull = AR.infile->wPOfull;
01248             AR.infile->POsize = AR.infile->wPOsize;
01249             AR.InInBuf = 0;
01250             PUTZERO(AR.infile->POposition);
01251         }
01252 /*
01253         If there is some writing, it betters happens to ones own outfile.
01254         Currently this is to be done only for InParallel.
01255         Merging of the outputs is then done by the CopyExpression routine.
01256 */
01257         {
01258             AR.outfile->PObuffer = AR.outfile->wPObuffer;
01259             AR.outfile->POstop = AR.outfile->wPOstop;
01260             AR.outfile->POfill = AR.outfile->wPOfill;
01261             AR.outfile->POfull = AR.outfile->wPOfull;
```

```
01262             AR.outfile->POsize = AR.outfile->wPOsize;
01263             PUTZERO(AR.outfile->POposition);
01264         }
01265         if ( AR.hidefile->handle < 0 ) {
01266             AR.hidefile->PObuffer = AR0.hidefile->PObuffer;
01267             AR.hidefile->POstop = AR0.hidefile->POstop;
01268             AR.hidefile->POfill = AR0.hidefile->POfill;
01269             AR.hidefile->POfull = AR0.hidefile->POfull;
01270             AR.hidefile->POsize = AR0.hidefile->POsize;
01271             AR.InHiBuf = AR0.InHiBuf;
01272             AR.hidefile->POposition = AR0.hidefile->POposition;
01273             AR.hidefile->filesize = AR0.hidefile->filesize;
01274         }
01275         else {
01276             AR.hidefile->PObuffer = AR.hidefile->wPObuffer;
01277             AR.hidefile->POstop = AR.hidefile->wPOstop;
01278             AR.hidefile->POfill = AR.hidefile->wPOfill;
01279             AR.hidefile->POfull = AR.hidefile->wPOfull;
01280             AR.hidefile->POsize = AR.hidefile->wPOsize;
01281             AR.InHiBuf = 0;
01282             PUTZERO(AR.hidefile->POposition);
01283         }
01284     }
01285     if ( AR0.StoreData.dirtyflag ) {
01286         for ( id = 1; id < AM.totalnumberofthreads; id++ ) {
01287             B = AB[id];
01288             AR.StoreData = AR0.StoreData;
01289         }
01290     }
01291 }
01292
01293 /*
01294     #] SetWorkerFiles :
01295     #[ RunThread :
01296 */
01304 void *RunThread(void *dummy)
01305 {
01306     WORD *term, *ttin, *tt, *ttco, *oldwork;
01307     int identity, wakeupsignal, identityretv, i, tobereleased, errorcode;
01308     ALLPRIVATES *B;
01309     THREADBUCKET *thr;
01310     POSITION *ppdef;
01311     EXPRESSIONS e;
01312     DUMMYUSE(dummy);
01313     identity = SetIdentity(&identityretv);
01314     threadpointers[identity] = pthread_self();
01315     B = InitializeOneThread(identity);
01316     while ( ( wakeupsignal = ThreadWait(identity) ) > 0 ) {
01317         switch ( wakeupsignal ) {
01318 /*
01319             #[ STARTNEWEXPRESSION :
01320 */
01321             case STARTNEWEXPRESSION:
01322 /*
01323                 Set up the sort routines etc.
01324                 Start with getting some buffers synchronized with the compiler
01325 */
01326                 if ( UpdateOneThread(identity) ) {
01327 /* INTERNAL_ERROR_EXCL_START */
01328                     MLOCK(ErrorMessageLock);
01329                     MesPrint("!>Update error in starting expression in thread %d in module
      %d",identity,AC.CModule);
01330                     MUNLOCK(ErrorMessageLock);
01331                     Terminate(-1);
01332 /* INTERNAL_ERROR_EXCL_STOP */
01333                 }
01334                 AR.DeferFlag = AC.ComDefer;
01335                 AR.sLevel = AS.sLevel;
01336                 AR.MaxDum = AM.IndDum;
01337                 AR.expchanged = AB[0]->R.expchanged;
01338                 AR.expflags = AB[0]->R.expflags;
01339                 AR.PolyFun = AB[0]->R.PolyFun;
01340                 AR.PolyFunInv = AB[0]->R.PolyFunInv;
01341                 AR.PolyFunType = AB[0]->R.PolyFunType;
01342                 AR.PolyFunExp = AB[0]->R.PolyFunExp;
01343                 AR.PolyFunVar = AB[0]->R.PolyFunVar;
01344                 AR.PolyFunPow = AB[0]->R.PolyFunPow;
01345 /*
01346                 Now fire up the sort buffer.
01347 */
01348                 NewSort(BHEAD0);
01349                 break;
01350 /*
01351             #] STARTNEWEXPRESSION :
01352             #[ LOWESTLEVELGENERATION :
01353 */
01354             case LOWESTLEVELGENERATION:
```

```
01355 #ifdef INNERTEST
01356                 if ( AC.InnerTest ) {
01357                     if ( StrCmp(AC.TestValue,(UBYTE *)INNERTEST) == 0 ) {
01358                         MesPrint("Testing(Worker%d): value = %s",AT.identity,AC.TestValue);
01359                     }
01360                 }
01361 #endif
01362                 e = Expressions + AR.CurExpr;
01363                 thr = AN.threadbuck;
01364                 ppdef = thr->deferbuffer;
01365                 ttin = thr->threadbuffer;
01366                 ttco = thr->compressbuffer;
01367                 term = AT.WorkPointer;
01368                 thr->usenum = 0;
01369                 tobereleased = 0;
01370                 AN.inputnumber = thr->firstterm;
01371                 AN.ninterms = thr->firstterm;
01372                 do {
01373                   thr->usenum++;    /* For if the master wants to steal the bucket */
01374                   tt = term; i = *ttin;
01375                   NCOPY(tt,ttin,i);
01376                   AT.WorkPointer = tt;
01377                   if ( AR.DeferFlag ) {
01378                     tt = AR.CompressBuffer; i = *ttco;
01379                     NCOPY(tt,ttco,i);
01380                     AR.CompressPointer = tt;
01381                     AR.DefPosition = ppdef[0]; ppdef++;
01382                   }
01383                   if ( thr->free == BUCKETTERMINATED ) {
01384 /*
01385                     The next statement allows the master to steal the bucket
01386                     for load balancing purposes. We do still execute the current
01387                     term, but afterwards we drop out.
01388                     Once we have written the release code, we cannot use this
01389                     bucket anymore. Hence the exit to the label bucketstolen.
01390 */
01391                     if ( thr->usenum == thr->totnum ) {
01392                         thr->free = BUCKETCOMINGFREE;
01393                     }
01394                     else {
01395                         thr->free = BUCKETRELEASED;
01396                         tobereleased = 1;
01397                     }
01398                   }
01399 /*
01400                     What if we want to steal and we set thr->free while
01401                     the thread is inside the next code for a long time?
01402                   if ( AT.LoadBalancing ) {
01403 */
01404                     LOCK(thr->lock);
01405                     thr->busy = BUCKETDOINGTERM;
01406                     UNLOCK(thr->lock);
01407 /*
01408                   }
01409                   else {
01410                     thr->busy = BUCKETDOINGTERM;
01411                   }
01412 */
01413                   AN.RepPoint = AT.RepCount + 1;
01414
01415                   if ( ( e->vflags & ISFACTORIZED ) != 0 && term[1] == HAAKJE ) {
01416                     StoreTerm(BHEAD term);
01417                   }
01418                   else {
01419                   if ( AR.DeferFlag ) {
01420                     AR.CurDum = AN.IndDum = Expressions[AR.CurExpr].numdummies + AM.IndDum;
01421                   }
01422                   else {
01423                     AN.IndDum = AM.IndDum;
01424                     AR.CurDum = ReNumber(BHEAD term);
01425                   }
01426                   if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
01427                   if ( AN.ncmod ) {
01428                     if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
01429                     else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
01430                   }
01431                   else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
01432                   if ( ( AP.PreDebug & THREADSDEBUG ) != 0 ) {
01433                     MLOCK(ErrorMessageLock);
01434                     MesPrint("Thread %w executing term:");
01435                     PrintTerm(term,"LLG");
01436                     MUNLOCK(ErrorMessageLock);
01437                   }
01438                   if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
01439                         && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
01440                         PolyFunClean(BHEAD term);
01441                   }
```

```
01442                   if ( Generator(BHEAD term,0) ) {
01443 /* INTERNAL_ERROR_EXCL_START */
01444                     LowerSortLevel();
01445                     MLOCK(ErrorMessageLock);
01446                     MesPrint("!>Error in processing one term in thread %d in module
      %d",identity,AC.CModule);
01447                     MUNLOCK(ErrorMessageLock);
01448                     Terminate(-1);
01449 /* INTERNAL_ERROR_EXCL_STOP */
01450                   }
01451                   AN.ninterms++;
01452                   }
01453 /*                if ( AT.LoadBalancing ) { */
01454                     LOCK(thr->lock);
01455                     thr->busy = BUCKETPREPARINGTERM;
01456                     UNLOCK(thr->lock);
01457 /*
01458                   }
01459                   else {
01460                     thr->busy = BUCKETPREPARINGTERM;
01461                   }
01462 */
01463                   if ( thr->free == BUCKETTERMINATED ) {
01464                     if ( thr->usenum == thr->totnum ) {
01465                         thr->free = BUCKETCOMINGFREE;
01466                     }
01467                     else {
01468                         thr->free = BUCKETRELEASED;
01469                         tobereleased = 1;
01470                     }
01471                   }
01472                   if ( tobereleased ) goto bucketstolen;
01473                 } while ( *ttin );
01474                 thr->free = BUCKETCOMINGFREE;
01475 bucketstolen:;
01476 /*              if ( AT.LoadBalancing ) { */
01477                     LOCK(thr->lock);
01478                     thr->busy = BUCKETTOBERELEASED;
01479                     UNLOCK(thr->lock);
01480 /*              }
01481                 else {
01482                     thr->busy = BUCKETTOBERELEASED;
01483                 }
01484 */
01485                 AT.WorkPointer = term;
01486                 break;
01487 /*
01488             #] LOWESTLEVELGENERATION :
01489             #[ FINISHEXPRESSION :
01490 */
01491 #ifdef WITHSORTBOTS
01492             case CLAIMOUTPUT:
01493                 LOCK(AT.SB.MasterBlockLock[1]);
01494                 break;
01495 #endif
01496             case FINISHEXPRESSION:
01497 /*
01498                 Finish the sort
01499
01500                 Start with claiming the first block
01501                 Once we have claimed it we can let the master know that
01502                 everything is all right.
01503 */
01504                 LOCK(AT.SB.MasterBlockLock[1]);
01505                 ThreadClaimedBlock(identity);
01506 /*
01507                 Entry for when we work with sortbots
01508 */
01509 #ifdef WITHSORTBOTS
01510                 /* fall through */
01511             case FINISHEXPRESSION2:
01512 #endif
01513 /*
01514                 Now we may need here an fsync on the sort file
01515 */
01516                 if ( AC.ThreadSortFileSynch ) {
01517                   if ( AT.S0->file.handle >= 0 ) {
01518                     SynchFile(AT.S0->file.handle);
01519                   }
01520                 }
01521                 AT.SB.FillBlock = 1;
01522                 AT.SB.MasterFill[1] = AT.SB.MasterStart[1];
01523                 errorcode = EndSort(BHEAD AT.S0->sBuffer,0);
01524                 UNLOCK(AT.SB.MasterBlockLock[AT.SB.FillBlock]);
01525                 UpdateMaxSize();
01526                 if ( errorcode ) {
01527                     MLOCK(ErrorMessageLock);
```

```
01528                     MesPrint("Error terminating sort in thread %d in module %d",identity,AC.CModule);
01529                     MUNLOCK(ErrorMessageLock);
01530                     Terminate(-1);
01531                 }
01532                 break;
01533 /*
01534             #] FINISHEXPRESSION :
01535             #[ CLEANUPEXPRESSION :
01536 */
01537             case CLEANUPEXPRESSION:
01538 /*
01539                 Cleanup everything and wait for the next expression
01540 */
01541                 if ( AR.outfile->handle >= 0 ) {
01542                     CloseFile(AR.outfile->handle);
01543                     AR.outfile->handle = -1;
01544                     remove(AR.outfile->name);
01545                     AR.outfile->POfill = AR.outfile->POfull = AR.outfile->PObuffer;
01546                     PUTZERO(AR.outfile->POposition);
01547                     PUTZERO(AR.outfile->filesize);
01548                 }
01549                 else {
01550                     AR.outfile->POfill = AR.outfile->POfull = AR.outfile->PObuffer;
01551                     PUTZERO(AR.outfile->POposition);
01552                     PUTZERO(AR.outfile->filesize);
01553                 }
01554                 {
01555                     CBUF *C = cbuf+AT.ebufnum;
01556                     WORD **w, ii;
01557                     if ( C->numrhs > 0 || C->numlhs > 0 ) {
01558                         if ( C->rhs ) {
01559                             w = C->rhs; ii = C->numrhs;
01560                             do { *w++ = 0; } while ( --ii > 0 );
01561                         }
01562                         if ( C->lhs ) {
01563                             w = C->lhs; ii = C->numlhs;
01564                             do { *w++ = 0; } while ( --ii > 0 );
01565                         }
01566                         C->numlhs = C->numrhs = 0;
01567                         ClearTree(AT.ebufnum);
01568                         C->Pointer = C->Buffer;
01569                     }
01570                 }
01571                 break;
01572 /*
01573             #] CLEANUPEXPRESSION :
01574             #[ HIGHERLEVELGENERATION :
01575 */
01576             case HIGHERLEVELGENERATION:
01577 /*
01578                 When foliating halfway the tree.
01579                 This should only be needed in a second level load balancing
01580 */
01581                 term = AT.WorkSpace; AT.WorkPointer = term + *term;
01582                 if ( Generator(BHEAD term,AR.level) ) {
01583                     LowerSortLevel();
01584                     MLOCK(ErrorMessageLock);
01585                     MesPrint("Error in load balancing one term at level %d in thread %d in module
    %d",AR.level,AT.identity,AC.CModule);
01586                     MUNLOCK(ErrorMessageLock);
01587                     Terminate(-1);
01588                 }
01589                 AT.WorkPointer = term;
01590                 break;
01591 /*
01592             #] HIGHERLEVELGENERATION :
01593             #[ STARTNEWMODULE :
01594 */
01595             case STARTNEWMODULE:
01596 /*
01597                 For resetting variables.
01598 */
01599                 SpecialCleanup(B);
01600                 break;
01601 /*
01602             #] STARTNEWMODULE :
01603             #[ TERMINATETHREAD :
01604 */
01605             case TERMINATETHREAD:
01606                 goto EndOfThread;
01607 /*
01608             #] TERMINATETHREAD :
01609             #[ DOONEEXPRESSION :
01610
01611                 When a thread has to do a complete (not too big) expression.
01612                 The number of the expression to be done is in AR.exprtodo.
01613                 The code is mostly taken from Processor. The only difference
```

```
01614                 is with what to do with the output.
01615                 The output should go to the scratch buffer of the worker
01616                 (which is free at the right moment). If this buffer is too
01617                 small we have a problem. We could write to file or give the
01618                 master what we have and from now on the master has to collect
01619                 pieces until things are complete.
01620                 Note: this assumes that the expressions don't keep their order.
01621                 If they have to keep their order, don't use this feature.
01622 */
01623             case DOONEEXPRESSION: {
01624
01625                 POSITION position, outposition;
01626                 FILEHANDLE *fi, *fout, *oldoutfile;
01627                 LONG dd = 0;
01628                 WORD oldBracketOn = AR.BracketOn;
01629                 WORD *oldBrackBuf = AT.BrackBuf;
01630                 WORD oldbracketindexflag = AT.bracketindexflag;
01631                 WORD fromspectator = 0;
01632                 e = Expressions + AR.exprtodo;
01633                 i = AR.exprtodo;
01634                 AR.CurExpr = i;
01635                 AR.SortType = AC.SortType;
01636                 AR.expchanged = 0;
01637                 if ( ( e->vflags & ISFACTORIZED ) != 0 ) {
01638                     AR.BracketOn = 1;
01639                     AT.BrackBuf = AM.BracketFactors;
01640                     AT.bracketindexflag = 1;
01641                 }
01642
01643                 position = AS.OldOnFile[i];
01644                 if ( e->status == HIDDENLEXPRESSION || e->status == HIDDENGEXPRESSION
01645                     || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION ) {
01646                     AR.GetFile = 2; fi = AR.hidefile;
01647                 }
01648                 else {
01649                     AR.GetFile = 0; fi = AR.infile;
01650                 }
01651 /*
01652                 PUTZERO(fi->POposition);
01653                 if ( fi->handle >= 0 ) {
01654                     fi->POfill = fi->POfull = fi->PObuffer;
01655                 }
01656 */
01657                 SetScratch(fi,&position);
01658                 term = oldwork = AT.WorkPointer;
01659                 AR.CompressPointer = AR.CompressBuffer;
01660                 AR.CompressPointer[0] = 0;
01661                 AR.KeptInHold = 0;
01662                 if ( GetTerm(BHEAD term) <= 0 ) {
01663                     MLOCK(ErrorMessageLock);
01664                     MesPrint("Expression %d has problems in scratchfile (t)",i);
01665                     MUNLOCK(ErrorMessageLock);
01666                     Terminate(-1);
01667                 }
01668                 if ( AT.bracketindexflag > 0 ) OpenBracketIndex(i);
01669                 term[3] = i;
01670                 if ( term[5] < 0 ) {
01671                     fromspectator = -term[5];
01672                     PUTZERO(AM.SpectatorFiles[fromspectator-1].readpos);
01673                     term[5] = AC.cbufnum;
01674                 }
01675                 PUTZERO(outposition);
01676                 fout = AR.outfile;
01677                 fout->POfill = fout->POfull = fout->PObuffer;
01678                 fout->POposition = outposition;
01679                 if ( fout->handle >= 0 ) {
01680                     fout->POposition = outposition;
01681                 }
01682 /*
01683                 The next statement is needed because we need the system
01684                 to believe that the expression is at position zero for
01685                 the moment. In this worker, with no memory of other expressions,
01686                 it is. This is needed for when a bracket index is made
01687                 because there e->onfile is an offset. Afterwards, when the
01688                 expression is written to its final location in the masters
01689                 output e->onfile will get its real value.
01690 */
01691                 PUTZERO(e->onfile);
01692                 if ( PutOut(BHEAD term,&outposition,fout,0) < 0 ) goto ProcErr;
01693
01694                 AR.DeferFlag = AC.ComDefer;
01695
01696                 AR.sLevel = AB[0]->R.sLevel;
01697                 term = AT.WorkPointer;
01698                 NewSort(BHEAD0);
01699                 AR.MaxDum = AM.IndDum;
01700                 AN.ninterms = 0;
```

```
01701                 if ( fromspectator ) {
01702                  while ( GetFromSpectator(term,fromspectator-1) ) {
01703                   AT.WorkPointer = term + *term;
01704                   AN.RepPoint = AT.RepCount + 1;
01705                   AN.IndDum = AM.IndDum;
01706                   AR.CurDum = ReNumber(BHEAD term);
01707                   if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
01708                   if ( AN.ncmod ) {
01709                     if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
01710                     else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
01711                   }
01712                   else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
01713                   if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
01714                         && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
01715                         PolyFunClean(BHEAD term);
01716                   }
01717                   if ( Generator(BHEAD term,0) ) {
01718                     LowerSortLevel(); goto ProcErr;
01719                   }
01720                  }
01721                 }
01722                 else {
01723                  while ( GetTerm(BHEAD term) ) {
01724                   SeekScratch(fi,&position);
01725                   AN.ninterms++; dd = AN.deferskipped;
01726                   if ( ( e->vflags & ISFACTORIZED ) != 0 && term[1] == HAAKJE ) {
01727                       StoreTerm(BHEAD term);
01728                   }
01729                   else {
01730                   if ( AC.CollectFun && *term <= (AM.MaxTer/(2*(LONG)sizeof(WORD))) ) {
01731                     if ( GetMoreTerms(term) < 0 ) {
01732                       LowerSortLevel(); goto ProcErr;
01733                     }
01734                     SeekScratch(fi,&position);
01735                   }
01736                   AT.WorkPointer = term + *term;
01737                   AN.RepPoint = AT.RepCount + 1;
01738                   if ( AR.DeferFlag ) {
01739                     AR.CurDum = AN.IndDum = Expressions[AR.exprtodo].numdummies;
01740                   }
01741                   else {
01742                     AN.IndDum = AM.IndDum;
01743                     AR.CurDum = ReNumber(BHEAD term);
01744                   }
01745                   if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
01746                   if ( AN.ncmod ) {
01747                     if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
01748                     else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
01749                   }
01750                   else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
01751                   if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
01752                         && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
01753                         PolyFunClean(BHEAD term);
01754                   }
01755                   if ( Generator(BHEAD term,0) ) {
01756                     LowerSortLevel(); goto ProcErr;
01757                   }
01758                   AN.ninterms += dd;
01759                   }
01760                   SetScratch(fi,&position);
01761                   if ( fi == AR.hidefile ) {
01762                     AR.InHiBuf = (fi->POfull-fi->PObuffer)
01763                         -DIFBASE(position,fi->POposition)/sizeof(WORD);
01764                   }
01765                   else {
01766                     AR.InInBuf = (fi->POfull-fi->PObuffer)
01767                         -DIFBASE(position,fi->POposition)/sizeof(WORD);
01768                   }
01769                  }
01770                 }
01771                 AN.ninterms += dd;
01772                 if ( EndSort(BHEAD AT.S0->sBuffer,0) < 0 ) goto ProcErr;
01773                 e->numdummies = AR.MaxDum - AM.IndDum;
01774                 AR.BracketOn = oldBracketOn;
01775                 AT.BrackBuf = oldBrackBuf;
01776                 if ( ( e->vflags & TOBEFACTORED ) != 0 )
01777                         poly_factorize_expression(e);
01778                 else if ( ( ( e->vflags & TOBEUNFACTORED ) != 0 )
01779                  && ( ( e->vflags & ISFACTORIZED ) != 0 ) )
01780                         poly_unfactorize_expression(e);
01781                 if ( AT.S0->TermsLeft )   e->vflags &= ~ISZERO;
01782                 else                      e->vflags |= ISZERO;
01783                 if ( AR.expchanged == 0 ) e->vflags |= ISUNMODIFIED;
01784                 if ( AT.S0->TermsLeft ) AR.expflags |= ISZERO;
01785                 if ( AR.expchanged )    AR.expflags |= ISUNMODIFIED;
01786                 AR.GetFile = 0;
01787                 AT.bracketindexflag = oldbracketindexflag;
```

```
01788 /*
01789                 Now copy the whole thing from fout to AR0.outfile
01790                 Do this in one go to keep the lock occupied as short as possible
01791 */
01792                 SeekScratch(fout,&outposition);
01793                 LOCK(AS.outputslock);
01794                 oldoutfile = AB[0]->R.outfile;
01795                 if ( e->status == INTOHIDELEXPRESSION || e->status == INTOHIDEGEXPRESSION ) {
01796                     AB[0]->R.outfile = AB[0]->R.hidefile;
01797                 }
01798                 SeekScratch(AB[0]->R.outfile,&position);
01799                 e->onfile = position;
01800                 if ( CopyExpression(fout,AB[0]->R.outfile) < 0 ) {
01801                     AB[0]->R.outfile = oldoutfile;
01802                     UNLOCK(AS.outputslock);
01803                     MLOCK(ErrorMessageLock);
01804                     MesPrint("Error copying output of 'InParallel' expression to master. Thread:
      %d",identity);
01805                     MUNLOCK(ErrorMessageLock);
01806                     goto ProcErr;
01807                 }
01808                 AB[0]->R.outfile = oldoutfile;
01809                 AB[0]->R.expflags = AR.expflags;
01810                 UNLOCK(AS.outputslock);
01811
01812                 if ( fout->handle >= 0 ) {    /* Now get rid of the file */
01813                     CloseFile(fout->handle);
01814                     fout->handle = -1;
01815                     remove(fout->name);
01816                     PUTZERO(fout->POposition);
01817                     PUTZERO(fout->filesize);
01818                     fout->POfill = fout->POfull = fout->PObuffer;
01819                 }
01820                 UpdateMaxSize();
01821
01822                 AT.WorkPointer = oldwork;
01823
01824                 } break;
01825 /*
01826             #] DOONEEXPRESSION :
01827             #[ DOBRACKETS :
01828
01829                 In case we have a bracket index we can have the worker treat
01830                 one or more of the entries in the bracket index.
01831                 The advantage is that identical terms will meet each other
01832                 sooner in the sorting and hence fewer compares will be needed.
01833                 Also this way the master doesn't need to fill the buckets.
01834                 The main problem is the load balancing which can become very
01835                 bad when there is a long tail without things outside the bracket.
01836
01837                 We get sent:
01838                 1: The number of the first bracket to be done
01839                 2: The number of the last bracket to be done
01840 */
01841             case DOBRACKETS: {
01842                 BRACKETINFO *binfo;
01843                 BRACKETINDEX *bi;
01844                 FILEHANDLE *fi;
01845                 POSITION stoppos,where;
01846                 e = Expressions + AR.CurExpr;
01847                 binfo = e->bracketinfo;
01848                 thr = AN.threadbuck;
01849                 bi = &(binfo->indexbuffer[thr->firstbracket]);
01850                 if ( AR.GetFile == 2 ) fi = AR.hidefile;
01851                 else                   fi = AR.infile;
01852                 where = bi->start;
01853                 ADD2POS(where,AS.OldOnFile[AR.CurExpr]);
01854                 SetScratch(fi,&(where));
01855                 stoppos = binfo->indexbuffer[thr->lastbracket].next;
01856                 ADD2POS(stoppos,AS.OldOnFile[AR.CurExpr]);
01857                 AN.ninterms = thr->firstterm;
01858 /*
01859                 Now we have to put the 'value' of the bracket in the
01860                 Compress buffer.
01861 */
01862                 ttco = AR.CompressBuffer;
01863                 tt = binfo->bracketbuffer + bi->bracket;
01864                 i = *tt;
01865                 NCOPY(ttco,tt,i)
01866                 AR.CompressPointer = ttco;
01867                 term = AT.WorkPointer;
01868                 while ( GetTerm(BHEAD term) ) {
01869                     SeekScratch(fi,&where);
01870                     AT.WorkPointer = term + *term;
01871                     AN.IndDum = AM.IndDum;
01872                     AR.CurDum = ReNumber(BHEAD term);
01873                     if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
```

```
01874                     if ( AN.ncmod ) {
01875                         if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
01876                         else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
01877                     }
01878                     else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
01879                     if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
01880                         && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
01881                         PolyFunClean(BHEAD term);
01882                     }
01883                     if ( ( AP.PreDebug & THREADSDEBUG ) != 0 ) {
01884                         MLOCK(ErrorMessageLock);
01885                         MesPrint("Thread %w executing term:");
01886                         PrintTerm(term,"DoBrackets");
01887                         MUNLOCK(ErrorMessageLock);
01888                     }
01889                     AT.WorkPointer = term + *term;
01890                     if ( Generator(BHEAD term,0) ) {
01891                         LowerSortLevel();
01892                         MLOCK(ErrorMessageLock);
01893                         MesPrint("Error in processing one term in thread %d in module
      %d",identity,AC.CModule);
01894                         MUNLOCK(ErrorMessageLock);
01895                         Terminate(-1);
01896                     }
01897                     AN.ninterms++;
01898                     SetScratch(fi,&(where));
01899                     if ( ISGEPOS(where,stoppos) ) break;
01900                 }
01901                 AT.WorkPointer = term;
01902                 thr->free = BUCKETCOMINGFREE;
01903                 break;
01904             }
01905 /*
01906             #] DOBRACKETS :
01907             #[ CLEARCLOCK :
01908
01909             The program only comes here after a .clear
01910 */
01911             case CLEARCLOCK:
01912 /*              LOCK(clearclocklock); */
01913                 sumtimerinfo[identity] += TimeCPU(1);
01914                 timerinfo[identity] = TimeCPU(0);
01915 /*              UNLOCK(clearclocklock); */
01916                 break;
01917 /*
01918             #] CLEARCLOCK :
01919             #[ MCTSEXPANDTREE :
01920 */
01921             case MCTSEXPANDTREE:
01922                 AT.optimtimes = AB[0]->T.optimtimes;
01923                 find_Horner_MCTS_expand_tree();
01924                 break;
01925 /*
01926             #] MCTSEXPANDTREE :
01927             #[ OPTIMIZEEXPRESSION :
01928 */
01929             case OPTIMIZEEXPRESSION:
01930                 optimize_expression_given_Horner();
01931                 break;
01932 /*
01933             #] OPTIMIZEEXPRESSION :
01934 */
01935             default:
01936 /* INTERNAL_ERROR_EXCL_START */
01937                 MLOCK(ErrorMessageLock);
01938                 MesPrint("!>Illegal wakeup signal %d for thread %d",wakeupsignal,identity);
01939                 MUNLOCK(ErrorMessageLock);
01940                 Terminate(-1);
01941                 break;
01942 /* INTERNAL_ERROR_EXCL_STOP */
01943         }
01944         /* we need the following update in case we are using checkpoints. then we
01945            need to readjust the clocks when recovering using this information */
01946         timerinfo[identity] = TimeCPU(1);
01947     }
01948 EndOfThread:;
01949 /*
01950     This is the end of the thread. We cleanup and exit.
01951     If we are using flint, call the per-thread cleanup function. This keep valgrind happy.
01952 */
01953 #ifdef WITHFLINT
01954     flint_final_cleanup_thread();
01955 #endif
01956     FinalizeOneThread(identity);
01957     return(0);
01958 ProcErr:
01959     Terminate(-1);
```

```
01960     return(0);
01961 }
01962
01963 /*
01964     #] RunThread :
01965     #[ RunSortBot :
01966 */
01974 #ifdef WITHSORTBOTS
01975
01976 void *RunSortBot(void *dummy)
01977 {
01978     int identity, wakeupsignal, identityretv;
01979     ALLPRIVATES *B, *BB;
01980     DUMMYUSE(dummy);
01981     identity = SetIdentity(&identityretv);
01982     threadpointers[identity] = pthread_self();
01983     B = InitializeOneThread(identity);
01984     while ( ( wakeupsignal = SortBotWait(identity) ) > 0 ) {
01985         switch ( wakeupsignal ) {
01986 /*
01987             #[ INISORTBOT :
01988 */
01989             case INISORTBOT:
01990                 AR.CompressBuffer = AB[0]->R.CompressBuffer;
01991                 AR.ComprTop = AB[0]->R.ComprTop;
01992                 AR.CompressPointer = AB[0]->R.CompressPointer;
01993                 AR.CurExpr = AB[0]->R.CurExpr;
01994                 AR.PolyFun = AB[0]->R.PolyFun;
01995                 AR.PolyFunInv = AB[0]->R.PolyFunInv;
01996                 AR.PolyFunType = AB[0]->R.PolyFunType;
01997                 AR.PolyFunExp = AB[0]->R.PolyFunExp;
01998                 AR.PolyFunVar = AB[0]->R.PolyFunVar;
01999                 AR.PolyFunPow = AB[0]->R.PolyFunPow;
02000                 AR.SortType = AC.SortType;
02001                 if ( AR.PolyFun == 0 ) { AT.SS->PolyFlag = 0; }
02002                 else if ( AR.PolyFunType == 1 ) { AT.SS->PolyFlag = 1; }
02003                 else if ( AR.PolyFunType == 2 ) {
02004                     if ( AR.PolyFunExp == 2
02005                       || AR.PolyFunExp == 3 ) AT.SS->PolyFlag = 1;
02006                     else                      AT.SS->PolyFlag = 2;
02007                 }
02008                 AT.SS->PolyWise = 0;
02009                 AN.ncmod = AC.ncmod;
02010                 LOCK(AT.SB.MasterBlockLock[1]);
02011                 BB = AB[AT.SortBotIn1];
02012                 LOCK(BB->T.SB.MasterBlockLock[BB->T.SB.MasterNumBlocks]);
02013                 BB = AB[AT.SortBotIn2];
02014                 LOCK(BB->T.SB.MasterBlockLock[BB->T.SB.MasterNumBlocks]);
02015                 AT.SB.FillBlock = 1;
02016                 AT.SB.MasterFill[1] = AT.SB.MasterStart[1];
02017                 SETBASEPOSITION(AN.theposition,0);
02018                 // Reset the sortbot comparison count
02019                 AT.SS->verbComparisons = 0;
02020                 // Reset the maximal term size count
02021                 AT.SS->verbMaxTermSize = 0;
02022                 break;
02023 /*
02024             #] INISORTBOT :
02025             #[ RUNSORTBOT :
02026 */
02027             case RUNSORTBOT:
02028                 SortBotMerge(B);
02029                 break;
02030 /*
02031             #] RUNSORTBOT :
02032             #[ TERMINATETHREAD :
02033 */
02034             case TERMINATETHREAD:
02035                 goto EndOfThread;
02036 /*
02037             #] TERMINATETHREAD :
02038             #[ CLEARCLOCK :
02039
02040             The program only comes here after a .clear
02041 */
02042             case CLEARCLOCK:
02043 /*              LOCK(clearclocklock); */
02044                 sumtimerinfo[identity] += TimeCPU(1);
02045                 timerinfo[identity] = TimeCPU(0);
02046 /*              UNLOCK(clearclocklock); */
02047                 break;
02048 /*
02049             #] CLEARCLOCK :
02050 */
02051             default:
02052 /* INTERNAL_ERROR_EXCL_START */
02053                 MLOCK(ErrorMessageLock);
```

```
02054                 MesPrint("!>Illegal wakeup signal %d for thread %d",wakeupsignal,identity);
02055                 MUNLOCK(ErrorMessageLock);
02056                 Terminate(-1);
02057                 break;
02058 /* INTERNAL_ERROR_EXCL_STOP */
02059         }
02060     }
02061 EndOfThread:;
02062 /*
02063     This is the end of the thread. We cleanup and exit.
02064     If we are using flint, call the per-thread cleanup function. This keep valgrind happy.
02065 */
02066 #ifdef WITHFLINT
02067     flint_final_cleanup_thread();
02068 #endif
02069     FinalizeOneThread(identity);
02070     return(0);
02071 }
02072
02073 #endif
02074
02075 /*
02076     #] RunSortBot :
02077     #[ IAmAvailable :
02078 */
02090 void IAmAvailable(int identity)
02091 {
02092     int top;
02093     LOCK(availabilitylock);
02094     top = topofavailables;
02095     listofavailables[topofavailables++] = identity;
02096     if ( top == 0 ) {
02097         UNLOCK(availabilitylock);
02098         LOCK(wakeupmasterlock);
02099         wakeupmaster = identity;
02100         pthread_cond_signal(&wakeupmasterconditions);
02101         UNLOCK(wakeupmasterlock);
02102     }
02103     else {
02104         UNLOCK(availabilitylock);
02105     }
02106 }
02107
02108 /*
02109     #] IAmAvailable :
02110     #[ GetAvailableThread :
02111 */
02120 int GetAvailableThread(void)
02121 {
02122     int retval = -1;
02123     LOCK(availabilitylock);
02124     if ( topofavailables > 0 ) retval = listofavailables[--topofavailables];
02125     UNLOCK(availabilitylock);
02126     if ( retval >= 0 ) {
02127 /*
02128         Make sure the thread is indeed waiting and not between
02129         saying that it is available and starting to wait.
02130 */
02131         LOCK(wakeuplocks[retval]);
02132         UNLOCK(wakeuplocks[retval]);
02133     }
02134     return(retval);
02135 }
02136
02137 /*
02138     #] GetAvailableThread :
02139     #[ ConditionalGetAvailableThread :
02140 */
02148 int ConditionalGetAvailableThread(void)
02149 {
02150     int retval = -1;
02151     if ( topofavailables > 0 ) {
02152         LOCK(availabilitylock);
02153         if ( topofavailables > 0 ) {
02154             retval = listofavailables[--topofavailables];
02155         }
02156         UNLOCK(availabilitylock);
02157         if ( retval >= 0 ) {
02158 /*
02159             Make sure the thread is indeed waiting and not between
02160             saying that it is available and starting to wait.
02161 */
02162             LOCK(wakeuplocks[retval]);
02163             UNLOCK(wakeuplocks[retval]);
02164         }
02165     }
02166     return(retval);
```

```
02167 }
02168
02169 /*
02170     #] ConditionalGetAvailableThread :
02171     #[ GetThread :
02172 */
02182 int GetThread(int identity)
02183 {
02184     int retval = -1, j;
02185     LOCK(availabilitylock);
02186     for ( j = 0; j < topofavailables; j++ ) {
02187         if ( identity == listofavailables[j] ) break;
02188     }
02189     if ( j < topofavailables ) {
02190         --topofavailables;
02191         for ( ; j < topofavailables; j++ ) {
02192             listofavailables[j] = listofavailables[j+1];
02193         }
02194         retval = identity;
02195     }
02196     UNLOCK(availabilitylock);
02197     return(retval);
02198 }
02199
02200 /*
02201     #] GetThread :
02202     #[ ThreadWait :
02203 */
02213 int ThreadWait(int identity)
02214 {
02215     int retval, top, j;
02216     LOCK(wakeuplocks[identity]);
02217     LOCK(availabilitylock);
02218     top = topofavailables;
02219     for ( j = topofavailables; j > 0; j-- )
02220         listofavailables[j] = listofavailables[j-1];
02221     listofavailables[0] = identity;
02222     topofavailables++;
02223     if ( top == 0 || topofavailables == numberofworkers ) {
02224         UNLOCK(availabilitylock);
02225         LOCK(wakeupmasterlock);
02226         wakeupmaster = identity;
02227         pthread_cond_signal(&wakeupmasterconditions);
02228         UNLOCK(wakeupmasterlock);
02229     }
02230     else {
02231         UNLOCK(availabilitylock);
02232     }
02233     while ( wakeup[identity] == 0 ) {
02234         pthread_cond_wait(&(wakeupconditions[identity]),&(wakeuplocks[identity]));
02235     }
02236     retval = wakeup[identity];
02237     wakeup[identity] = 0;
02238     UNLOCK(wakeuplocks[identity]);
02239     return(retval);
02240 }
02241
02242 /*
02243     #] ThreadWait :
02244     #[ SortBotWait :
02245 */
02246
02247 #ifdef WITHSORTBOTS
02257 int SortBotWait(int identity)
02258 {
02259     int retval;
02260     LOCK(wakeuplocks[identity]);
02261     LOCK(availabilitylock);
02262     topsortbotavailables++;
02263     if ( topsortbotavailables >= numberofsortbots ) {
02264         UNLOCK(availabilitylock);
02265         LOCK(wakeupsortbotlock);
02266         wakeupmaster = identity;
02267         pthread_cond_signal(&wakeupsortbotconditions);
02268         UNLOCK(wakeupsortbotlock);
02269     }
02270     else {
02271         UNLOCK(availabilitylock);
02272     }
02273     while ( wakeup[identity] == 0 ) {
02274         pthread_cond_wait(&(wakeupconditions[identity]),&(wakeuplocks[identity]));
02275     }
02276     retval = wakeup[identity];
02277     wakeup[identity] = 0;
02278     UNLOCK(wakeuplocks[identity]);
02279     return(retval);
02280 }
```

```
02281
02282 #endif
02283
02284 /*
02285     #] SortBotWait :
02286     #[ ThreadClaimedBlock :
02287 */
02298 int ThreadClaimedBlock(int identity)
02299 {
02300     LOCK(availabilitylock);
02301     numberclaimed++;
02302     if ( numberclaimed >= numberofworkers ) {
02303         UNLOCK(availabilitylock);
02304         LOCK(wakeupmasterlock);
02305         wakeupmaster = identity;
02306         pthread_cond_signal(&wakeupmasterconditions);
02307         UNLOCK(wakeupmasterlock);
02308     }
02309     else {
02310         UNLOCK(availabilitylock);
02311     }
02312     return(0);
02313 }
02314
02315 /*
02316     #] ThreadClaimedBlock :
02317     #[ MasterWait :
02318 */
02326 int MasterWait(void)
02327 {
02328     int retval;
02329     LOCK(wakeupmasterlock);
02330     while ( wakeupmaster == 0 ) {
02331         pthread_cond_wait(&wakeupmasterconditions,&wakeupmasterlock);
02332     }
02333     retval = wakeupmaster;
02334     wakeupmaster = 0;
02335     UNLOCK(wakeupmasterlock);
02336     return(retval);
02337 }
02338
02339 /*
02340     #] MasterWait :
02341     #[ MasterWaitThread :
02342 */
02349 int MasterWaitThread(int identity)
02350 {
02351     int retval;
02352     LOCK(wakeupmasterthreadlocks[identity]);
02353     while ( wakeupmasterthread[identity] == 0 ) {
02354         pthread_cond_wait(&(wakeupmasterthreadconditions[identity])
02355                 ,&(wakeupmasterthreadlocks[identity]));
02356     }
02357     retval = wakeupmasterthread[identity];
02358     wakeupmasterthread[identity] = 0;
02359     UNLOCK(wakeupmasterthreadlocks[identity]);
02360     return(retval);
02361 }
02362
02363 /*
02364     #] MasterWaitThread :
02365     #[ MasterWaitAll :
02366 */
02373 void MasterWaitAll(void)
02374 {
02375     LOCK(wakeupmasterlock);
02376     while ( topofavailables < numberofworkers ) {
02377         pthread_cond_wait(&wakeupmasterconditions,&wakeupmasterlock);
02378     }
02379     UNLOCK(wakeupmasterlock);
02380     return;
02381 }
02382
02383 /*
02384     #] MasterWaitAll :
02385     #[ MasterWaitAllSortBots :
02386 */
02387
02388 #ifdef WITHSORTBOTS
02389
02395 void MasterWaitAllSortBots(void)
02396 {
02397     LOCK(wakeupsortbotlock);
02398     while ( topsortbotavailables < numberofsortbots ) {
02399         pthread_cond_wait(&wakeupsortbotconditions,&wakeupsortbotlock);
02400     }
02401     UNLOCK(wakeupsortbotlock);
```

```
02402     return;
02403 }
02404
02405 #endif
02406
02407 /*
02408     #] MasterWaitAllSortBots :
02409     #[ MasterWaitAllBlocks :
02410 */
02417 void MasterWaitAllBlocks(void)
02418 {
02419     LOCK(wakeupmasterlock);
02420     while ( numberclaimed < numberofworkers ) {
02421         pthread_cond_wait(&wakeupmasterconditions,&wakeupmasterlock);
02422     }
02423     UNLOCK(wakeupmasterlock);
02424     return;
02425 }
02426
02427 /*
02428     #] MasterWaitAllBlocks :
02429     #[ WakeupThread :
02430 */
02439 void WakeupThread(int identity, int signalnumber)
02440 {
02441     if ( signalnumber == 0 ) {
02442 /* INTERNAL_ERROR_EXCL_START */
02443         MLOCK(ErrorMessageLock);
02444         MesPrint("!>Illegal wakeup signal for thread %d",identity);
02445         MUNLOCK(ErrorMessageLock);
02446         Terminate(-1);
02447 /* INTERNAL_ERROR_EXCL_STOP */
02448     }
02449     LOCK(wakeuplocks[identity]);
02450     wakeup[identity] = signalnumber;
02451     pthread_cond_signal(&(wakeupconditions[identity]));
02452     UNLOCK(wakeuplocks[identity]);
02453 }
02454
02455 /*
02456     #] WakeupThread :
02457     #[ WakeupMasterFromThread :
02458 */
02467 void WakeupMasterFromThread(int identity, int signalnumber)
02468 {
02469     if ( signalnumber == 0 ) {
02470 /* INTERNAL_ERROR_EXCL_START */
02471         MLOCK(ErrorMessageLock);
02472         MesPrint("!>Illegal wakeup signal for master %d",identity);
02473         MUNLOCK(ErrorMessageLock);
02474         Terminate(-1);
02475 /* INTERNAL_ERROR_EXCL_STOP */
02476     }
02477     LOCK(wakeupmasterthreadlocks[identity]);
02478     wakeupmasterthread[identity] = signalnumber;
02479     pthread_cond_signal(&(wakeupmasterthreadconditions[identity]));
02480     UNLOCK(wakeupmasterthreadlocks[identity]);
02481 }
02482
02483 /*
02484     #] WakeupMasterFromThread :
02485     #[ SendOneBucket :
02486 */
02492 int SendOneBucket(int type)
02493 {
02494     ALLPRIVATES *B0 = AB[0];
02495     THREADBUCKET *thr = 0;
02496     int j, k, id;
02497     for ( j = 0; j < numthreadbuckets; j++ ) {
02498         if ( threadbuckets[j]->free == BUCKETFILLED ) {
02499             thr = threadbuckets[j];
02500             for ( k = j+1; k < numthreadbuckets; k++ )
02501                 threadbuckets[k-1] = threadbuckets[k];
02502             threadbuckets[numthreadbuckets-1] = thr;
02503             break;
02504         }
02505     }
02506     AN0.ninterms++;
02507     while ( ( id = GetAvailableThread() ) < 0 ) { MasterWait(); }
02508 /*
02509     Prepare the thread. Give it the term and variables.
02510 */
02511     LoadOneThread(0,id,thr,0);
02512     thr->busy = BUCKETASSIGNED;
02513     thr->free = BUCKETINUSE;
02514     numberoffullbuckets--;
02515 /*
```

```
02516     And signal the thread to run.
02517     Form now on we may only interfere with this bucket
02518     1: after it has been marked BUCKETCOMINGFREE
02519     2: when thr->busy == BUCKETDOINGTERM and then only when protected by
02520        thr->lock. This would be for load balancing.
02521 */
02522     WakeupThread(id,type);
02523 /*  AN0.ninterms += thr->ddterms; */
02524     return(0);
02525 }
02526
02527 /*
02528     #] SendOneBucket :
02529     #[ InParallelProcessor :
02530 */
02550 int InParallelProcessor(void)
02551 {
02552     GETIDENTITY
02553     int i, id, retval = 0, num = 0;
02554     EXPRESSIONS e;
02555     if ( numberofworkers >= 2 ) {
02556         SetWorkerFiles();
02557         for ( i = 0; i < NumExpressions; i++ ) {
02558             e = Expressions+i;
02559             if ( e->partodo <= 0 ) continue;
02560             if ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION
02561             || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION
02562             || e->status == INTOHIDELEXPRESSION || e->status == INTOHIDEGEXPRESSION ) {
02563             }
02564             else {
02565                 e->partodo = 0;
02566                 continue;
02567             }
02568             if ( e->counter == 0 ) { /* Expression with zero terms */
02569                 e->partodo = 0;
02570                 continue;
02571             }
02572 /*
02573             This expression should go to an idle worker
02574 */
02575             while ( ( id = GetAvailableThread() ) < 0 ) { MasterWait(); }
02576             LoadOneThread(0,id,0,-1);
02577             AB[id]->R.exprtodo = i;
02578             WakeupThread(id,DOONEEXPRESSION);
02579             num++;
02580         }
02581 /*
02582         Now we have to wait for all workers to finish
02583 */
02584         if ( num > 0 ) MasterWaitAll();
02585
02586         if ( AC.CollectFun ) AR.DeferFlag = 0;
02587     }
02588     else {
02589         for ( i = 0; i < NumExpressions; i++ ) {
02590             Expressions[i].partodo = 0;
02591         }
02592     }
02593     return(retval);
02594 }
02595
02596 /*
02597     #] InParallelProcessor :
02598     #[ ThreadsProcessor :
02599 */
02624 int ThreadsProcessor(EXPRESSIONS e, WORD LastExpression, WORD fromspectator)
02625 {
02626     ALLPRIVATES *B0 = AB[0], *B = B0;
02627     int id, oldgzipCompress, endofinput = 0, j, still, k, defcount = 0, bra = 0, first = 1;
02628     LONG dd = 0, ddd, thrbufsiz, thrbufsiz0, thrbufsiz2, numbucket = 0, numpasses;
02629     LONG num, i;
02630     WORD *oldworkpointer = AT0.WorkPointer, *tt, *ttco = 0, *t1 = 0, ter, *tstop = 0, *t2;
02631     THREADBUCKET *thr = 0;
02632     FILEHANDLE *oldoutfile = AR0.outfile;
02633     GETTERM GetTermP = &GetTerm;
02634     POSITION eonfile = AS.OldOnFile[e-Expressions];
02635     numberoffullbuckets = 0;
02636 /*
02637     Start up all threads. The lock needs to be around the whole loop
02638     to keep processes from terminating quickly and putting themselves
02639     in the list of available threads again.
02640 */
02641     AM.tracebackflag = 1;
02642
02643     AS.sLevel = AR0.sLevel;
02644     LOCK(availabilitylock);
02645     topofavailables = 0;
```

```
02646     for ( id = 1; id <= numberofworkers; id++ ) {
02647         WakeupThread(id,STARTNEWEXPRESSION);
02648     }
02649     UNLOCK(availabilitylock);
02650     NewSort(BHEAD0);
02651     AN0.ninterms = 1;
02652 /*
02653     Now for redefine
02654 */
02655     if ( AC.numpfirstnum > 0 ) {
02656         for ( j = 0; j < AC.numpfirstnum; j++ ) {
02657             AC.inputnumbers[j] = -1;
02658         }
02659     }
02660     MasterWaitAll();
02661 /*
02662     Determine a reasonable bucketsize.
02663     This is based on the value of AC.ThreadBucketSize and the number
02664     of terms. We want at least 5 buckets per worker at the moment.
02665     Some research should show whether this is reasonable.
02666
02667     The number of terms in the expression is in e->counter
02668 */
02669     thrbufsiz2 = thrbufsiz = AC.ThreadBucketSize-1;
02670     if ( ( e->counter / ( numberofworkers * 5 ) ) < thrbufsiz ) {
02671         thrbufsiz = e->counter / ( numberofworkers * 5 ) - 1;
02672         if ( thrbufsiz < 0 ) thrbufsiz = 0;
02673     }
02674     thrbufsiz0 = thrbufsiz;
02675     numpasses = 5; /* this is just for trying */
02676     thrbufsiz = thrbufsiz0 / (2 « numpasses);
02677 /*
02678     Mark all buckets as free and take the first.
02679 */
02680     for ( j = 0; j < numthreadbuckets; j++ )
02681         threadbuckets[j]->free = BUCKETFREE;
02682     thr = threadbuckets[0];
02683 /*
02684     #[ Whole brackets :
02685
02686     First we look whether we have to work with entire brackets
02687     This is the case when there is a non-NULL address in e->bracketinfo.
02688     Of course we shouldn't have interference from a collect or keep statement.
02689 */
02690 #ifdef WHOLEBRACKETS
02691     if ( e->bracketinfo && AC.CollectFun == 0 && AR0.DeferFlag == 0 ) {
02692         FILEHANDLE *curfile;
02693         int didone = 0;
02694         LONG num, n;
02695         AN0.expr = e;
02696         for ( n = 0; n < e->bracketinfo->indexfill; n++ ) {
02697             num = TreatIndexEntry(B0,n);
02698             if ( num > 0 ) {
02699                 didone = 1;
02700 /*
02701                 This bracket can be sent off.
02702                 1: Look for an empty bucket
02703 */
02704 ReTry:;
02705                 for ( j = 0; j < numthreadbuckets; j++ ) {
02706                     switch ( threadbuckets[j]->free ) {
02707                         case BUCKETFREE:
02708                             thr = threadbuckets[j];
02709                             goto Found1;
02710                         case BUCKETCOMINGFREE:
02711                             thr = threadbuckets[j];
02712                             thr->free = BUCKETFREE;
02713                             for ( k = j+1; k < numthreadbuckets; k++ )
02714                                 threadbuckets[k-1] = threadbuckets[k];
02715                             threadbuckets[numthreadbuckets-1] = thr;
02716                             j--;
02717                             break;
02718                         default:
02719                             break;
02720                     }
02721                 }
02722 Found1:;
02723                 if ( j < numthreadbuckets ) {
02724 /*
02725                     Found an empty bucket. Fill it.
02726 */
02727                     thr->firstbracket = n;
02728                     thr->lastbracket = n + num - 1;
02729                     thr->type = BUCKETDOINGBRACKET;
02730                     thr->free = BUCKETFILLED;
02731                     thr->firstterm = AN0.ninterms;
02732                     for ( j = n; j < n+num; j++ ) {
```

```
02733                       AN0.ninterms += e->bracketinfo->indexbuffer[j].termsinbracket;
02734                   }
02735                   n += num-1;
02736                   numberoffullbuckets++;
02737                   if ( topofavailables > 0 ) {
02738                       SendOneBucket(DOBRACKETS);
02739                   }
02740               }
02741 /*
02742                   All buckets are in use.
02743                   Look/wait for an idle worker. Give it a bucket.
02744                   After that, retry for a bucket
02745 */
02746               else {
02747                   while ( topofavailables <= 0 ) {
02748                       MasterWait();
02749                   }
02750                   SendOneBucket(DOBRACKETS);
02751                   goto ReTry;
02752               }
02753           }
02754       }
02755       if ( didone ) {
02756 /*
02757           And now put the input back in the original position.
02758 */
02759           switch ( e->status ) {
02760               case UNHIDELEXPRESSION:
02761               case UNHIDEGEXPRESSION:
02762               case DROPHLEXPRESSION:
02763               case DROPHGEXPRESSION:
02764               case HIDDENLEXPRESSION:
02765               case HIDDENGEXPRESSION:
02766                   curfile = AR0.hidefile;
02767                   break;
02768               default:
02769                   curfile = AR0.infile;
02770                   break;
02771           }
02772           SetScratch(curfile,&eonfile);
02773           GetTerm(B0,AT0.WorkPointer);
02774 /*
02775           Now we point the GetTerm that is used to the one that is selective
02776 */
02777           GetTermP = &GetTerm2;
02778 /*
02779           Next wait till there is a bucket available and initialize thr to it.
02780 */
02781           for(;;) {
02782               for ( j = 0; j < numthreadbuckets; j++ ) {
02783                   switch ( threadbuckets[j]->free ) {
02784                       case BUCKETFREE:
02785                           thr = threadbuckets[j];
02786                           goto Found2;
02787                       case BUCKETCOMINGFREE:
02788                           thr = threadbuckets[j];
02789                           thr->free = BUCKETFREE;
02790                           for ( k = j+1; k < numthreadbuckets; k++ )
02791                               threadbuckets[k-1] = threadbuckets[k];
02792                           threadbuckets[numthreadbuckets-1] = thr;
02793                           j--;
02794                           break;
02795                       default:
02796                           break;
02797                   }
02798               }
02799               while ( topofavailables <= 0 ) {
02800                   MasterWait();
02801               }
02802               while ( topofavailables > 0 && numberoffullbuckets > 0 ) {
02803                   SendOneBucket(DOBRACKETS);
02804               }
02805           }
02806 Found2:;
02807           while ( numberoffullbuckets > 0 ) {
02808               while ( topofavailables <= 0 ) {
02809                   MasterWait();
02810               }
02811               while ( topofavailables > 0 && numberoffullbuckets > 0 ) {
02812                   SendOneBucket(DOBRACKETS);
02813               }
02814           }
02815 /*
02816           Disable the 'warming up' with smaller buckets.
02817
02818           numpasses = 0;
02819           thrbufsiz = thrbufsiz0;
```

```
02820 */
02821             AN0.lastinindex = -1;
02822         }
02823         MasterWaitAll();
02824     }
02825 #endif
02826 /*
02827     #] Whole brackets :
02828
02829     Now the loop to start a bucket
02830 */
02831     for(;;) {
02832         if ( fromspectator ) {
02833             ter = GetFromSpectator(thr->threadbuffer,fromspectator-1);
02834             if ( ter == 0 ) fromspectator = 0;
02835         }
02836         else {
02837             ter = GetTermP(B0,thr->threadbuffer);
02838 /*
02839             At this point we could check whether the input term is
02840             just an expression that resides in a scratch file.
02841             If this is the case we should store the current input info
02842             (file and buffer content) and redirect the input.
02843             At the end we can go back to where we were.
02844             There are two possibilities: we are in the same scratchfile
02845             as the main input and the other expression is still in the
02846             input buffer, or we have to do some reading. The reading is
02847             of course done in GetTerm.
02848
02849             How to set this up can also be studied in TestSub (in file proces.c)
02850             where it checks for EXPRESSION.
02851 */
02852         }
02853         if ( ter < 0 ) break;
02854         if ( ter == 0 ) { endofinput = 1; goto Finalize; }
02855         dd = AN0.deferskipped;
02856         if ( AR0.DeferFlag ) {
02857             defcount = 0;
02858             thr->deferbuffer[defcount++] = AR0.DefPosition;
02859             ttco = thr->compressbuffer; t1 = AR0.CompressBuffer; j = *t1;
02860             while ( thr->compressbuffersize <= j ) {
02861                 // the compressbuffer is not large enough!
02862                 WORD *top = thr->compressbuffer+thr->compressbuffersize;
02863                 DoubleBuffer((void**)&(thr->compressbuffer),(void**)&(top),
02864                     sizeof(*(thr->compressbuffer)), "double compressbuffer");
02865                 ttco = thr->compressbuffer;
02866                 thr->compressbuffersize *= 2;
02867             }
02868             NCOPY(ttco,t1,j);
02869         }
02870         else if ( first && ( AC.CollectFun == 0 ) ) { /* Brackets ? */
02871             first = 0;
02872             t1 = tstop = thr->threadbuffer;
02873             tstop += *tstop; tstop -= ABS(tstop[-1]);
02874             t1++;
02875             while ( t1 < tstop ) {
02876                 if ( t1[0] == HAAKJE ) { bra = 1; break; }
02877                 t1 += t1[1];
02878             }
02879             t1 = thr->threadbuffer;
02880         }
02881 /*
02882         Check whether we have a collect,function going. If so execute it.
02883 */
02884         if ( AC.CollectFun && *(thr->threadbuffer) < (AM.MaxTer/((LONG)sizeof(WORD))-10) ) {
02885             if ( ( dd = GetMoreTerms(thr->threadbuffer) ) < 0 ) {
02886                 LowerSortLevel(); goto ProcErr;
02887             }
02888         }
02889 /*
02890         Check whether we have a priority task:
02891 */
02892         if ( topofavailables > 0 && numberoffullbuckets > 0 ) SendOneBucket(LOWESTLEVELGENERATION);
02893 /*
02894         Now put more terms in the bucket. Position tt after the first term
02895 */
02896         tt = thr->threadbuffer; tt += *tt;
02897         thr->totnum = 1;
02898         thr->usenum = 0;
02899 /*
02900         Next we worry about the 'slow startup' in which we make the initial
02901         buckets smaller, so that we get all threads busy as soon as possible.
02902 */
02903         if ( numpasses > 0 ) {
02904             numbucket++;
02905             if ( numbucket >= numberofworkers ) {
02906                 numbucket = 0;
```

```
02907                 numpasses--;
02908                 if ( numpasses == 0 ) thrbufsiz = thrbufsiz0;
02909                 else                  thrbufsiz = thrbufsiz0 / (2 « numpasses);
02910             }
02911             thrbufsiz2 = thrbufsiz + thrbufsiz/5; /* for completing brackets */
02912         }
02913 /*
02914         we have already 1+dd terms
02915 */
02916         while ( ( dd < thrbufsiz ) &&
02917             ( tt - thr->threadbuffer ) < ( thr->threadbuffersize - AM.MaxTer/((LONG)sizeof(WORD)) - 2
      ) ) {
02918 /*
02919             First check:
02920 */
02921             if ( topofavailables > 0 && numberoffullbuckets > 0 )
      SendOneBucket(LOWESTLEVELGENERATION);
02922 /*
02923             There is room in the bucket. Fill yet another term.
02924 */
02925             if ( GetTermP(B0,tt) == 0 ) { endofinput = 1; break; }
02926             dd++;
02927             thr->totnum++;
02928             dd += AN0.deferskipped;
02929             if ( AR0.DeferFlag ) {
02930                 thr->deferbuffer[defcount++] = AR0.DefPosition;
02931                 t1 = AR0.CompressBuffer; j = *t1;
02932                 while ( thr->compressbuffer+thr->compressbuffersize-ttco <= j ) {
02933                     // the compressbuffer is not large enough!
02934                     const ptrdiff_t oldoffset = ttco - thr->compressbuffer;
02935                     WORD *top = thr->compressbuffer+thr->compressbuffersize;
02936                     DoubleBuffer((void**)&(thr->compressbuffer),(void**)&(top),
02937                         sizeof(*(thr->compressbuffer)), "double compressbuffer");
02938                     ttco = thr->compressbuffer + oldoffset;
02939                     thr->compressbuffersize *= 2;
02940                 }
02941                 NCOPY(ttco,t1,j);
02942             }
02943             if ( AC.CollectFun && *tt < (AM.MaxTer/((LONG)sizeof(WORD))-10) ) {
02944                 if ( ( ddd = GetMoreTerms(tt) ) < 0 ) {
02945                     LowerSortLevel(); goto ProcErr;
02946                 }
02947                 dd += ddd;
02948             }
02949             t1 = tt;
02950             tt += *tt;
02951         }
02952 /*
02953         Check whether there are regular brackets and if we have no DeferFlag
02954         and no collect, we try to add more terms till we finish the current
02955         bracket. We should however not overdo it. Let us say: up to 20%
02956         more terms are allowed.
02957 */
02958         if ( bra ) {
02959             tstop = t1 + *t1; tstop -= ABS(tstop[-1]);
02960             t2 = t1+1;
02961             while ( t2 < tstop ) {
02962                 if ( t2[0] == HAAKJE ) { break; }
02963                 t2 += t2[1];
02964             }
02965             if ( t2[0] == HAAKJE ) {
02966               t2 += t2[1]; num = t2 - t1;
02967               while ( ( dd < thrbufsiz2 ) &&
02968                 ( tt - thr->threadbuffer ) < ( thr->threadbuffersize - AM.MaxTer - 2 ) ) {
02969 /*
02970                 First check:
02971 */
02972                 if ( topofavailables > 0 && numberoffullbuckets > 0 )
      SendOneBucket(LOWESTLEVELGENERATION);
02973 /*
02974                 There is room in the bucket. Fill yet another term.
02975 */
02976                 if ( GetTermP(B0,tt) == 0 ) { endofinput = 1; break; }
02977 /*
02978                 Same bracket?
02979 */
02980                 tstop = tt + *tt; tstop -= ABS(tstop[-1]);
02981                 if ( tstop-tt < num ) { /* Different: abort */
02982                     AR0.KeptInHold = 1;
02983                     break;
02984                 }
02985                 for ( i = 1; i < num; i++ ) {
02986                     if ( t1[i] != tt[i] ) break;
02987                 }
02988                 if ( i < num ) { /* Different: abort */
02989                     AR0.KeptInHold = 1;
02990                     break;
```

```
02991                 }
02992 /*
02993                 Same bracket. We need this term.
02994 */
02995                 dd++;
02996                 thr->totnum++;
02997                 tt += *tt;
02998               }
02999             }
03000         }
03001         thr->ddterms = dd; /* total number of terms including keep brackets */
03002         thr->firstterm = AN0.ninterms;
03003         AN0.ninterms += dd;
03004         *tt = 0;            /* mark end of bucket */
03005         thr->free = BUCKETFILLED;
03006         thr->type = BUCKETDOINGTERMS;
03007         numberoffullbuckets++;
03008         if ( topofavailables <= 0 && endofinput == 0 ) {
03009 /*
03010             Problem: topofavailables may already be > 0, but the
03011             thread has not yet gone into waiting. Can the signal get lost?
03012             How can we tell that a thread is waiting for a signal?
03013
03014             All threads are busy. Try to load up another bucket.
03015             In the future we could be more sophisticated.
03016             At the moment we load a complete bucket which could be
03017             1000 terms or even more.
03018             In principle it is better to keep a full bucket ready
03019             and check after each term we put in the next bucket. That
03020             way we don't waste time of the workers.
03021 */
03022             for ( j = 0; j < numthreadbuckets; j++ ) {
03023                 switch ( threadbuckets[j]->free ) {
03024                     case BUCKETFREE:
03025                         thr = threadbuckets[j];
03026                         if ( !endofinput ) goto NextBucket;
03027 /*
03028                         If we are at the end of the input we mark
03029                         the free buckets in a special way. That way
03030                         we don't keep running into them.
03031 */
03032                         thr->free = BUCKETATEND;
03033                         break;
03034                     case BUCKETCOMINGFREE:
03035                         thr = threadbuckets[j];
03036                         thr->free = BUCKETFREE;
03037 /*
03038                         Bucket has just been finished.
03039                         Put at the end of the list. We don't want
03040                         an early bucket to wait to be treated last.
03041 */
03042                         for ( k = j+1; k < numthreadbuckets; k++ )
03043                             threadbuckets[k-1] = threadbuckets[k];
03044                         threadbuckets[numthreadbuckets-1] = thr;
03045                         j--;   /* we have to redo the same number j. */
03046                         break;
03047                     default:
03048                         break;
03049                 }
03050             }
03051 /*
03052             We have no free bucket or we are at the end.
03053             The only thing we can do now is wait for a worker to come free,
03054             provided there are still buckets to send.
03055 */
03056         }
03057 /*
03058         Look for the next bucket to send. There is at least one full bucket!
03059 */
03060         for ( j = 0; j < numthreadbuckets; j++ ) {
03061             if ( threadbuckets[j]->free == BUCKETFILLED ) {
03062                 thr = threadbuckets[j];
03063                 for ( k = j+1; k < numthreadbuckets; k++ )
03064                     threadbuckets[k-1] = threadbuckets[k];
03065                 threadbuckets[numthreadbuckets-1] = thr;
03066                 break;
03067             }
03068         }
03069 /*
03070         Wait for a thread to become available
03071         The bucket we are going to use is in thr.
03072 */
03073 DoBucket:;
03074         AN0.ninterms++;
03075         while ( ( id = GetAvailableThread() ) < 0 ) { MasterWait(); }
03076 /*
03077         Prepare the thread. Give it the term and variables.
```

```
03078 */
03079         LoadOneThread(0,id,thr,0);
03080         LOCK(thr->lock);
03081         thr->busy = BUCKETASSIGNED;
03082         UNLOCK(thr->lock);
03083         thr->free = BUCKETINUSE;
03084         numberoffullbuckets--;
03085 /*
03086         And signal the thread to run.
03087         Form now on we may only interfere with this bucket
03088         1: after it has been marked BUCKETCOMINGFREE
03089         2: when thr->busy == BUCKETDOINGTERM and then only when protected by
03090            thr->lock. This would be for load balancing.
03091 */
03092         WakeupThread(id,LOWESTLEVELGENERATION);
03093 /*      AN0.ninterms += thr->ddterms; */
03094 /*
03095         Now look whether there is another bucket filled and a worker available
03096 */
03097         if ( topofavailables > 0 ) {  /* there is a worker */
03098             for ( j = 0; j < numthreadbuckets; j++ ) {
03099                 if ( threadbuckets[j]->free == BUCKETFILLED ) {
03100                     thr = threadbuckets[j];
03101                     for ( k = j+1; k < numthreadbuckets; k++ )
03102                         threadbuckets[k-1] = threadbuckets[k];
03103                     threadbuckets[numthreadbuckets-1] = thr;
03104                     goto DoBucket; /* and we found a bucket */
03105                 }
03106             }
03107 /*
03108             no bucket is loaded but there is a thread available
03109             find a bucket to load. If there is none (all are USED or ATEND)
03110             we jump out of the loop.
03111 */
03112             for ( j = 0; j < numthreadbuckets; j++ ) {
03113                 switch ( threadbuckets[j]->free ) {
03114                     case BUCKETFREE:
03115                         thr = threadbuckets[j];
03116                         if ( !endofinput ) goto NextBucket;
03117                         thr->free = BUCKETATEND;
03118                         break;
03119                     case BUCKETCOMINGFREE:
03120                         thr = threadbuckets[j];
03121                         if ( endofinput ) {
03122                             thr->free = BUCKETATEND;
03123                         }
03124                         else {
03125                             thr->free = BUCKETFREE;
03126                             for ( k = j+1; k < numthreadbuckets; k++ )
03127                                 threadbuckets[k-1] = threadbuckets[k];
03128                             threadbuckets[numthreadbuckets-1] = thr;
03129                             j--;
03130                         }
03131                         break;
03132                     default:
03133                         break;
03134                 }
03135             }
03136             if ( j >= numthreadbuckets ) break;
03137         }
03138         else {
03139 /*
03140             No worker available.
03141             Look for a bucket to load.
03142             Its number will be in "still"
03143 */
03144 Finalize:;
03145             still = -1;
03146             for ( j = 0; j < numthreadbuckets; j++ ) {
03147                 switch ( threadbuckets[j]->free ) {
03148                     case BUCKETFREE:
03149                         thr = threadbuckets[j];
03150                         if ( !endofinput ) goto NextBucket;
03151                         thr->free = BUCKETATEND;
03152                         break;
03153                     case BUCKETCOMINGFREE:
03154                         thr = threadbuckets[j];
03155                         if ( endofinput ) thr->free = BUCKETATEND;
03156                         else {
03157                             thr->free = BUCKETFREE;
03158                             for ( k = j+1; k < numthreadbuckets; k++ )
03159                                 threadbuckets[k-1] = threadbuckets[k];
03160                             threadbuckets[numthreadbuckets-1] = thr;
03161                             j--;
03162                         }
03163                         break;
03164                     case BUCKETFILLED:
```

```
03165                         if ( still < 0 ) still = j;
03166                         break;
03167                     default:
03168                         break;
03169                 }
03170             }
03171             if ( still < 0 ) {
03172 /*
03173                 No buckets to be executed and no buckets FREE.
03174                 We must be at the end. Break out of the loop.
03175 */
03176                 break;
03177             }
03178             thr = threadbuckets[still];
03179             for ( k = still+1; k < numthreadbuckets; k++ )
03180                 threadbuckets[k-1] = threadbuckets[k];
03181             threadbuckets[numthreadbuckets-1] = thr;
03182             goto DoBucket;
03183         }
03184 NextBucket:;
03185     }
03186 /*
03187     Now the stage one load balancing.
03188     If the load has been readjusted we have again filled buckets.
03189     In that case we jump back in the loop.
03190
03191     Tricky point: when do the workers see the new value of AT.LoadBalancing?
03192     It should activate the locks on thr->busy
03193 */
03194     if ( AC.ThreadBalancing ) {
03195         for ( id = 1; id <= numberofworkers; id++ ) {
03196             AB[id]->T.LoadBalancing = 1;
03197         }
03198         if ( LoadReadjusted() ) goto Finalize;
03199         for ( id = 1; id <= numberofworkers; id++ ) {
03200             AB[id]->T.LoadBalancing = 0;
03201         }
03202     }
03203     if ( AC.ThreadBalancing ) {
03204 /*
03205         The AS.Balancing flag should have Generator look for
03206         free workers and apply the "buro" method.
03207
03208         There is still a serious problem.
03209         When for instance a sum_, there may be space created in a local
03210         compiler buffer for a wildcard substitution or whatever.
03211         Compiler buffer execution scribble space.....
03212         This isn't copied along?
03213         Look up ebufnum. There are 12 places with AddRHS!
03214         Problem: one process allocates in ebuf. Then term is given to
03215         other process. It would like to use from this ebuf, but the sender
03216         finishes first and removes the ebuf (and/or overwrites it).
03217
03218         Other problem: local $ variables aren't copied along.
03219 */
03220         AS.Balancing = 0;
03221     }
03222     MasterWaitAll();
03223     AS.Balancing = 0;
03224 /*
03225     When we deal with the last expression we can now remove the input
03226     scratch file. This saves potentially much disk space (up to 1/3)
03227 */
03228     if ( LastExpression ) {
03229         UpdateMaxSize();
03230         if ( AR0.infile->handle >= 0 ) {
03231             CloseFile(AR0.infile->handle);
03232             AR0.infile->handle = -1;
03233             remove(AR0.infile->name);
03234             PUTZERO(AR0.infile->POposition);
03235             AR0.infile->POfill = AR0.infile->POfull = AR0.infile->PObuffer;
03236         }
03237     }
03238 /*
03239     We order the threads to finish in the MasterMerge routine
03240     It will start with waiting for all threads to finish.
03241     One could make an administration in which threads that have
03242     finished can start already with the final sort but
03243     1: The load balancing should not make this super urgent
03244     2: It would definitely not be very compatible with the second
03245        stage load balancing.
03246 */
03247     oldgzipCompress = AR0.gzipCompress;
03248     AR0.gzipCompress = 0;
03249     if ( AR0.outtohide ) AR0.outfile = AR0.hidefile;
03250     if ( MasterMerge() < 0 ) {
03251         if ( AR0.outtohide ) AR0.outfile = oldoutfile;
```

```
03252         AR0.gzipCompress = oldgzipCompress;
03253         goto ProcErr;
03254     }
03255     if ( AR0.outtohide ) AR0.outfile = oldoutfile;
03256     AR0.gzipCompress = oldgzipCompress;
03257 /*
03258     Now wait for all threads to be ready to give them the cleaning up signal.
03259     With the new MasterMerge routine we can do the cleanup already automatically
03260     avoiding having to send these signals.
03261 */
03262     MasterWaitAll();
03263     AR0.sLevel--;
03264     for ( id = 1; id < AM.totalnumberofthreads; id++ ) {
03265         if ( GetThread(id) > 0 ) WakeupThread(id,CLEANUPEXPRESSION);
03266     }
03267     e->numdummies = 0;
03268     for ( id = 1; id < AM.totalnumberofthreads; id++ ) {
03269         if ( AB[id]->R.MaxDum - AM.IndDum > e->numdummies )
03270             e->numdummies = AB[id]->R.MaxDum - AM.IndDum;
03271         AR0.expchanged |= AB[id]->R.expchanged;
03272     }
03273 /*
03274     And wait for all to be clean.
03275 */
03276     MasterWaitAll();
03277     AT0.WorkPointer = oldworkpointer;
03278     return(0);
03279 ProcErr:;
03280     return(-1);
03281 }
03282
03283 /*
03284     #] ThreadsProcessor :
03285     #[ LoadReadjusted :
03286 */
03305 int LoadReadjusted(void)
03306 {
03307     ALLPRIVATES *B0 = AB[0];
03308     THREADBUCKET *thr = 0, *thrtogo = 0;
03309     int numtogo, numfree, numbusy, n, nperbucket, extra, i, j, u, bus;
03310     LONG numinput;
03311     WORD *t1, *c1, *t2, *c2, *t3;
03312 /*
03313     Start with waiting for at least one free processor.
03314     We don't want the master competing for time when all are busy.
03315 */
03316     while ( topofavailables <= 0 ) MasterWait();
03317 /*
03318     Now look for the fullest bucket and make a list of free buckets
03319     The bad part is that most numbers can change at any moment.
03320 */
03321 restart:;
03322     numtogo = 0;
03323     numfree = 0;
03324     numbusy = 0;
03325     for ( j = 0; j < numthreadbuckets; j++ ) {
03326         thr = threadbuckets[j];
03327         if ( thr->free == BUCKETFREE || thr->free == BUCKETATEND
03328         || thr->free == BUCKETCOMINGFREE ) {
03329             freebuckets[numfree++] = thr;
03330         }
03331         else if ( thr->type != BUCKETDOINGTERMS ) {}
03332         else if ( thr->totnum > 1 ) { /* never steal from a bucket with one term */
03333             LOCK(thr->lock);
03334             bus = thr->busy;
03335             UNLOCK(thr->lock);
03336             if ( thr->free == BUCKETINUSE ) {
03337                 n = thr->totnum-thr->usenum;
03338                 if ( bus == BUCKETASSIGNED ) numbusy++;
03339                 else if ( ( bus != BUCKETASSIGNED )
03340                         && ( n > numtogo ) ) {
03341                     numtogo = n;
03342                     thrtogo = thr;
03343                 }
03344             }
03345             else if ( bus == BUCKETTOBERELEASED
03346              && thr->free == BUCKETRELEASED ) {
03347                 freebuckets[numfree++] = thr;
03348                 thr->free = BUCKETATEND;
03349                 LOCK(thr->lock);
03350                 thr->busy = BUCKETPREPARINGTERM;
03351                 UNLOCK(thr->lock);
03352             }
03353         }
03354     }
03355     if ( numfree == 0 ) return(0); /* serious problem */
03356     if ( numtogo > 0 ) {   /* provisionally there is something to be stolen */
```

```
03357         thr = thrtogo;
03358 /*
03359         If the number has changed there is good progress.
03360         Maybe there is another thread that needs assistance.
03361         We start all over.
03362 */
03363         if ( thr->totnum-thr->usenum < numtogo ) goto restart;
03364 /*
03365         If the thread is in the term loading phase
03366         (thr->busy == BUCKETPREPARINGTERM) we better stay away from it.
03367         We wait now for the thread to be busy, and don't allow it
03368         now to drop out of this state till we are done here.
03369         This all depends on whether AT.LoadBalancing == 1 is seen by
03370         the thread.
03371 */
03372         LOCK(thr->lock);
03373         if ( thr->busy != BUCKETDOINGTERM ) {
03374             UNLOCK(thr->lock);
03375             goto restart;
03376         }
03377         if ( thr->totnum-thr->usenum < numtogo ) {
03378             UNLOCK(thr->lock);
03379             goto restart;
03380         }
03381         thr->free = BUCKETTERMINATED;
03382 /*
03383         The above will signal the thread we want to terminate.
03384         Next all effort goes into making sure the landing is soft.
03385         Unfortunately we don't want to wait for a signal, because the thread
03386         may be working for a long time on a single term.
03387 */
03388         if ( thr->usenum == thr->totnum ) {
03389 /*
03390             Terminated in the mean time or by now working on the
03391             last term. Try again.
03392 */
03393             thr->free = BUCKETATEND;
03394             UNLOCK(thr->lock);
03395             goto restart;
03396         }
03397         goto intercepted;
03398     }
03399 /* This has always been commented. Indeed no lock is held here. */
03400 /*  UNLOCK(thr->lock); */
03401     if ( numbusy > 0 ) {
03402         /* JD: this avoids large runtimes for tform tests under valgrind.
03403            What seems to happen is we return from here, goto Finalize, and
03404            end up in LoadReadjusted again without the threads having a
03405            chance to update their busy status. Then we end up here again.
03406            Sleep the thread for, say, 1us to allow threads to aquire the lock. */
03407         struct timespec sleeptime;
03408         sleeptime.tv_sec = 0;
03409         sleeptime.tv_nsec = 1000L;
03410         nanosleep(&sleeptime, NULL);
03411         return(1); /* Wait a bit.... */
03412     }
03413     return(0);
03414 intercepted:;
03415 /*
03416     We intercepted one successfully. Now it becomes interesting. Action:
03417     1: determine how many terms per free bucket.
03418     2: find the first untreated term.
03419     3: put the terms in the free buckets.
03420
03421     Remember: we still have the lock to avoid interference from the thread
03422     that is being robbed. We were holding it and then jumped here with
03423     goto intercepted.
03424 */
03425     numinput = thr->firstterm + thr->usenum;
03426     nperbucket = numtogo / numfree;
03427     extra = numtogo - nperbucket*numfree;
03428     if ( AR0.DeferFlag ) {
03429         t1 = thr->threadbuffer; c1 = thr->compressbuffer; u = thr->usenum;
03430         for ( n = 0; n < thr->usenum; n++ ) { t1 += *t1; c1 += *c1; }
03431         t3 = t1;
03432         if ( extra > 0 ) {
03433           for ( i = 0; i < extra; i++ ) {
03434             thrtogo = freebuckets[i];
03435             t2 = thrtogo->threadbuffer;
03436             c2 = thrtogo->compressbuffer;
03437             thrtogo->free = BUCKETFILLED;
03438             thrtogo->type = BUCKETDOINGTERMS;
03439             thrtogo->totnum = nperbucket+1;
03440             thrtogo->ddterms = 0;
03441             thrtogo->usenum = 0;
03442             thrtogo->busy = BUCKETASSIGNED;
03443             thrtogo->firstterm = numinput;
```

```
03444             numinput += nperbucket+1;
03445             for ( n = 0; n <= nperbucket; n++ ) {
03446                 j = *t1; NCOPY(t2,t1,j);
03447                 j = *c1; NCOPY(c2,c1,j);
03448                 thrtogo->deferbuffer[n] = thr->deferbuffer[u++];
03449             }
03450             *t2 = *c2 = 0;
03451           }
03452         }
03453         if ( nperbucket > 0 ) {
03454           for ( i = extra; i < numfree; i++ ) {
03455             thrtogo = freebuckets[i];
03456             t2 = thrtogo->threadbuffer;
03457             c2 = thrtogo->compressbuffer;
03458             thrtogo->free = BUCKETFILLED;
03459             thrtogo->type = BUCKETDOINGTERMS;
03460             thrtogo->totnum = nperbucket;
03461             thrtogo->ddterms = 0;
03462             thrtogo->usenum = 0;
03463             thrtogo->busy = BUCKETASSIGNED;
03464             thrtogo->firstterm = numinput;
03465             numinput += nperbucket;
03466             for ( n = 0; n < nperbucket; n++ ) {
03467                 j = *t1; NCOPY(t2,t1,j);
03468                 j = *c1; NCOPY(c2,c1,j);
03469                 thrtogo->deferbuffer[n] = thr->deferbuffer[u++];
03470             }
03471             *t2 = *c2 = 0;
03472           }
03473         }
03474     }
03475     else {
03476         t1 = thr->threadbuffer;
03477         for ( n = 0; n < thr->usenum; n++ ) { t1 += *t1; }
03478         t3 = t1;
03479         if ( extra > 0 ) {
03480           for ( i = 0; i < extra; i++ ) {
03481             thrtogo = freebuckets[i];
03482             t2 = thrtogo->threadbuffer;
03483             thrtogo->free = BUCKETFILLED;
03484             thrtogo->type = BUCKETDOINGTERMS;
03485             thrtogo->totnum = nperbucket+1;
03486             thrtogo->ddterms = 0;
03487             thrtogo->usenum = 0;
03488             thrtogo->busy = BUCKETASSIGNED;
03489             thrtogo->firstterm = numinput;
03490             numinput += nperbucket+1;
03491             for ( n = 0; n <= nperbucket; n++ ) {
03492                 j = *t1; NCOPY(t2,t1,j);
03493             }
03494             *t2 = 0;
03495           }
03496         }
03497         if ( nperbucket > 0 ) {
03498           for ( i = extra; i < numfree; i++ ) {
03499             thrtogo = freebuckets[i];
03500             t2 = thrtogo->threadbuffer;
03501             thrtogo->free = BUCKETFILLED;
03502             thrtogo->type = BUCKETDOINGTERMS;
03503             thrtogo->totnum = nperbucket;
03504             thrtogo->ddterms = 0;
03505             thrtogo->usenum = 0;
03506             thrtogo->busy = BUCKETASSIGNED;
03507             thrtogo->firstterm = numinput;
03508             numinput += nperbucket;
03509             for ( n = 0; n < nperbucket; n++ ) {
03510                 j = *t1; NCOPY(t2,t1,j);
03511             }
03512             *t2 = 0;
03513           }
03514         }
03515     }
03516     *t3 = 0;   /* This is some form of extra insurance */
03517     if ( thr->free == BUCKETRELEASED && thr->busy == BUCKETTOBERELEASED ) {
03518         thr->free = BUCKETATEND; thr->busy = BUCKETPREPARINGTERM;
03519     }
03520     UNLOCK(thr->lock);
03521     return(1);
03522 }
03523
03524 /*
03525     #] LoadReadjusted :
03526     #[ SortStrategy :
03527 */
03560 /*
03561     #] SortStrategy :
03562     #[ PutToMaster :
```

```
03563 */
03586 int PutToMaster(PHEAD WORD *term)
03587 {
03588     int i,j,nexti,ret = 0;
03589     int urgent = 0;
03590     WORD *t, *fill, *top, zero = 0;
03591     if ( term == 0 ) { /* Mark the end of the expression */
03592         t = &zero; j = 1;
03593     }
03594     else {
03595         t = term; ret = j = *term;
03596         if ( j == 0 ) { j = 1; } /* Just in case there is a spurious end */
03597     }
03598     i = AT.SB.FillBlock;          /* The block we are working at */
03599     fill = AT.SB.MasterFill[i];     /* Where we are filling */
03600     top = AT.SB.MasterStop[i];      /* End of the block */
03601
03602     // If there is space in the block, and we have already written MINWRITENUMBEROFTERMS,
03603     // determine if the reading thread is waiting for us by trying to lock the previous
03604     // block. If we manage to lock it, then we still have time to continue filling this
03605     // block. If we can't lock it, the reading thread is waiting for us and we should
03606     // move to the next block ASAP.
03607     if ( j < top - fill && AT.SB.BlockTerms[i] > MINWRITENUMBEROFTERMS ) {
03608         const int prev = ( i == 1 ? AT.SB.MasterNumBlocks : i-1 );
03609         if ( ! pthread_mutex_trylock(&(AT.SB.MasterBlockLock[prev])) ) {
03610             UNLOCK(AT.SB.MasterBlockLock[prev]);
03611         }
03612         else {
03613             urgent = 1;
03614         }
03615     }
03616
03617     // If the term doesn't fit in the current block, or a thread is waiting for us
03618     // (and we've already written at least MINWRITENUMBEROFTERMS), move to the next:
03619     if ( ( j >= top - fill ) || urgent ) {
03620         nexti = i+1;
03621         if ( nexti > AT.SB.MasterNumBlocks ) {
03622             nexti = 1;
03623         }
03624         LOCK(AT.SB.MasterBlockLock[nexti]);
03625         UNLOCK(AT.SB.MasterBlockLock[i]);
03626         AT.SB.MasterFill[i] = AT.SB.MasterStart[i];
03627         AT.SB.FillBlock = i = nexti;
03628         fill = AT.SB.MasterStart[i];
03629         top = AT.SB.MasterStop[i];
03630         if ( AT.SB.BlockTerms[i] != 0 ) {
03631             // In this case, there has been an accounting error in a previous use
03632             // of this block. Blocks that have been read from and unlocked, should
03633             // have BlockTerms == 0.
03634 /* INTERNAL_ERROR_EXCL_START */
03635             MLOCK(ErrorMessageLock);
03636             MesPrint("!>Error in PutToMaster, starting a block with BlockTerms != 0");
03637             MUNLOCK(ErrorMessageLock);
03638             Terminate(-1);
03639 /* INTERNAL_ERROR_EXCL_STOP */
03640         }
03641     }
03642
03643     NCOPY(fill, t, j);
03644     AT.SB.BlockTerms[i]++;
03645     AT.SB.MasterFill[i] = fill;
03646     return(ret);
03647 }
03648
03649 /*
03650     #] PutToMaster :
03651     #[ SortBotOut :
03652 */
03653
03654 #ifdef WITHSORTBOTS
03655
03664 int
03665 SortBotOut(PHEAD WORD *term)
03666 {
03667     WORD im;
03668
03669     if ( AT.identity != 0 ) return(PutToMaster(BHEAD term));
03670
03671     if ( term == 0 ) {
03672         if ( FlushOut(&SortBotPosition,AR.outfile,1) ) return(-1);
03673         ADDPOS(AT.SS->SizeInFile[0],1);
03674         return(0);
03675     }
03676     else {
03677         numberofterms++;
03678         if ( ( im = PutOut(BHEAD term,&SortBotPosition,AR.outfile,1) ) < 0 ) {
03679 /* INTERNAL_ERROR_EXCL_START */
```

```
03680             MLOCK(ErrorMessageLock);
03681             MesPrint("!>Called from MasterMerge/SortBotOut");
03682             MUNLOCK(ErrorMessageLock);
03683             return(-1);
03684 /* INTERNAL_ERROR_EXCL_STOP */
03685         }
03686         ADDPOS(AT.SS->SizeInFile[0],im);
03687         return(im);
03688     }
03689 }
03690
03691 #endif
03692
03693 /*
03694     #] SortBotOut :
03695     #[ MasterMerge :
03696 */
03714 int MasterMerge(void)
03715 {
03716     ALLPRIVATES *B0 = AB[0], *B = 0;
03717     SORTING *S = AT0.SS;
03718     WORD **poin, **poin2, ul, k, i, im, *m1, j;
03719     WORD lpat, mpat, level, l1, l2, r1, r2, r3, c;
03720     WORD *m2, *m3, r31, r33, ki, *rr;
03721     UWORD *coef;
03722     POSITION position;
03723     FILEHANDLE *fin, *fout;
03724 #ifdef WITHSORTBOTS
03725     if ( numberofworkers > 2 ) return(SortBotMasterMerge());
03726 #endif
03727     fin = &S->file;
03728     if ( AR0.PolyFun == 0 ) { S->PolyFlag = 0; }
03729     else if ( AR0.PolyFunType == 1 ) { S->PolyFlag = 1; }
03730     else if ( AR0.PolyFunType == 2 ) {
03731         if ( AR0.PolyFunExp == 2
03732           || AR0.PolyFunExp == 3 ) S->PolyFlag = 1;
03733         else                       S->PolyFlag = 2;
03734     }
03735     S->TermsLeft = 0;
03736     coef = AN0.SoScratC;
03737     poin = S->poina; poin2 = S->poin2a;
03738     rr = AR0.CompressPointer;
03739     *rr = 0;
03740 /*
03741         #[ Setup :
03742 */
03743     S->inNum = numberofthreads;
03744     fout = AR0.outfile;
03745 /*
03746     Load the patches. The threads have to finish their sort first.
03747 */
03748     S->lPatch = S->inNum - 1;
03749 /*
03750     Claim all zero blocks. We need them anyway.
03751     In principle the workers should never get into these.
03752     We also claim all last blocks. This is a safety procedure that
03753     should prevent the workers from working their way around the clock
03754     before the master gets started again.
03755 */
03756     AS.MasterSort = 1;
03757     numberclaimed = 0;
03758     for ( i = 1; i <= S->lPatch; i++ ) {
03759         B = AB[i];
03760         LOCK(AT.SB.MasterBlockLock[0]);
03761         LOCK(AT.SB.MasterBlockLock[AT.SB.MasterNumBlocks]);
03762     }
03763 /*
03764     Now wake up the threads and have them start their final sorting.
03765     They should start with claiming their block and the master is
03766     not allowed to continue until that has been done.
03767     This waiting of the master will be done below in MasterWaitAllBlocks
03768 */
03769     for ( i = 0; i < S->lPatch; i++ ) {
03770         GetThread(i+1);
03771         WakeupThread(i+1,FINISHEXPRESSION);
03772     }
03773 /*
03774     Prepare the output file.
03775 */
03776     if ( fout->handle >= 0 ) {
03777         PUTZERO(position);
03778         SeekFile(fout->handle,&position,SEEK_END);
03779         ADDPOS(position,((fout->POfill-fout->PObuffer)*sizeof(WORD)));
03780     }
03781     else {
03782         SETBASEPOSITION(position,(fout->POfill-fout->PObuffer)*sizeof(WORD));
03783     }
```

```
03784 /*
03785     Wait for all threads to finish loading their first block.
03786 */
03787     MasterWaitAllBlocks();
03788 /*
03789     Claim all first blocks.
03790     We don't release the last blocks.
03791     The strategy is that we always keep the previous block.
03792     In principle it looks like it isn't needed for the last block but
03793     actually it is to keep the front from overrunning the tail when writing.
03794 */
03795     for ( i = 1; i <= S->lPatch; i++ ) {
03796         B = AB[i];
03797         LOCK(AT.SB.MasterBlockLock[1]);
03798         AT.SB.MasterBlock = 1;
03799     }
03800 /*
03801         #] Setup :
03802
03803     Now construct the tree:
03804 */
03805     lpat = 1;
03806     do { lpat «= 1; } while ( lpat < S->lPatch );
03807     mpat = ( lpat » 1 ) - 1;
03808     k = lpat - S->lPatch;
03809 /*
03810     k is the number of empty places in the tree. they will
03811     be at the even positions from 2 to 2*k
03812 */
03813     for ( i = 1; i < lpat; i++ ) { S->tree[i] = -1; }
03814     for ( i = 1; i <= k; i++ ) {
03815         im = ( i * 2 ) - 1;
03816         poin[im] = AB[i]->T.SB.MasterStart[AB[i]->T.SB.MasterBlock];
03817         poin2[im] = poin[im] + *(poin[im]);
03818         S->used[i] = im;
03819         S->ktoi[im] = i-1;
03820         S->tree[mpat+i] = 0;
03821         poin[im-1] = poin2[im-1] = 0;
03822     }
03823     for ( i = (k*2)+1; i <= lpat; i++ ) {
03824         S->used[i-k] = i;
03825         S->ktoi[i] = i-k-1;
03826         poin[i] = AB[i-k]->T.SB.MasterStart[AB[i-k]->T.SB.MasterBlock];
03827         poin2[i] = poin[i] + *(poin[i]);
03828     }
03829 /*
03830     the array poin tells the position of the i-th element of the S->tree
03831     'S->used' is a stack with the S->tree elements that need to be entered
03832     into the S->tree. at the beginning this is S->lPatch. during the
03833     sort there will be only very few elements.
03834     poin2 is the next value of poin. it has to be determined
03835     before the comparisons as the position or the size of the
03836     term indicated by poin may change.
03837     S->ktoi translates a S->tree element back to its stream number.
03838
03839     start the sort
03840 */
03841     level = S->lPatch;
03842 /*
03843     introduce one term
03844 */
03845 OneTerm:
03846     k = S->used[level];
03847     i = k + lpat - 1;
03848     if ( !*(poin[k]) ) {
03849         // Stream k has hit the end-of-stream "0". We still need to decrement
03850         // BlockTerms, which includes the marker in the count.
03851         ki = S->ktoi[k];
03852         AB[ki+1]->T.SB.BlockTerms[AB[ki+1]->T.SB.MasterBlock]--;
03853         do {
03854             if ( !( i »= 1 ) ) {
03855                 goto EndOfMerge;
03856             }
03857         } while ( !S->tree[i] );
03858         if ( S->tree[i] == -1 ) {
03859             S->tree[i] = 0;
03860             level--;
03861             goto OneTerm;
03862         }
03863         k = S->tree[i];
03864         S->used[level] = k;
03865         S->tree[i] = 0;
03866     }
03867 /*
03868     move terms down the tree
03869 */
03870     while ( i »= 1 ) {
```

```
03871         if ( S->tree[i] > 0 ) {
03872             if ( ( c = CompareTerms(B0, poin[S->tree[i]],poin[k],(WORD)0) ) > 0 ) {
03873 /*
03874                 S->tree[i] is the smaller. Exchange and go on.
03875 */
03876                 S->used[level] = S->tree[i];
03877                 S->tree[i] = k;
03878                 k = S->used[level];
03879             }
03880             else if ( !c ) {    /* Terms are equal */
03881 /*
03882                 S->TermsLeft--;
03883                     Here the terms are equal and their coefficients
03884                     have to be added.
03885 */
03886                 l1 = *( m1 = poin[S->tree[i]] );
03887                 l2 = *( m2 = poin[k] );
03888                 if ( S->PolyWise ) {  /* Here we work with PolyFun */
03889                     WORD *tt1, *w;
03890                     tt1 = m1;
03891                     m1 += S->PolyWise;
03892                     m2 += S->PolyWise;
03893                     if ( S->PolyFlag == 2 ) {
03894                         w = poly_ratfun_add(B0,m1,m2);
03895                         if ( *tt1 + w[1] - m1[1] > AM.MaxTer/((LONG)sizeof(WORD)) ) {
03896                             MLOCK(ErrorMessageLock);
03897                             MesPrint("Term too complex in PolyRatFun addition. MaxTermSize of %10l is
    too small",AM.MaxTer);
03898                             MUNLOCK(ErrorMessageLock);
03899                             Terminate(-1);
03900                         }
03901                         AT0.WorkPointer = w;
03902                         if ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 && w[1] > FUNHEAD ) {
03903                             goto cancelled;
03904                         }
03905                     }
03906                     else {
03907                         w = AT0.WorkPointer;
03908                         if ( w + m1[1] + m2[1] > AT0.WorkTop ) {
03909                             MLOCK(ErrorMessageLock);
03910                             MesPrint("MasterMerge: A WorkSpace of %10l is too small",AM.WorkSize);
03911                             MUNLOCK(ErrorMessageLock);
03912                             Terminate(-1);
03913                         }
03914                         AddArgs(B0,m1,m2,w);
03915                     }
03916                     r1 = w[1];
03917                     if ( r1 <= FUNHEAD
03918                         || ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 ) )
03919                             { goto cancelled; }
03920                     if ( r1 == m1[1] ) {
03921                         NCOPY(m1,w,r1);
03922                     }
03923                     else if ( r1 < m1[1] ) {
03924                         r2 = m1[1] - r1;
03925                         m2 = w + r1;
03926                         m1 += m1[1];
03927                         while ( --r1 >= 0 ) *--m1 = *--m2;
03928                         m2 = m1 - r2;
03929                         r1 = S->PolyWise;
03930                         while ( --r1 >= 0 ) *--m1 = *--m2;
03931                         *m1 -= r2;
03932                         poin[S->tree[i]] = m1;
03933                     }
03934                     else {
03935                         // Here we are writing the new merged term *before* the original start of
    term1.
03936                         // We can always do this, since before term1 there is previous term data of
    this
03937                         // block, or the previous block, for which we are holding a lock. This
    requires
03938                         // the existence of "block 0", if term1 is the first term of block 1!
03939                         // It also requires the blocks to be contiguous in memory; we can't allocate
03940                         // separate memory regions for each block without larger-scale changes.
03941                         r2 = r1 - m1[1];
03942                         m2 = tt1 - r2;
03943                         r1 = S->PolyWise;
03944                         m1 = tt1;
03945                         *m1 += r2;
03946                         poin[S->tree[i]] = m2;
03947                         NCOPY(m2,m1,r1);
03948                         r1 = w[1];
03949                         NCOPY(m2,w,r1);
03950                     }
03951                 }
03952 #ifdef WITHFLOAT
03953                 else if ( AT.SortFloatMode ) {
```

```
03954                     WORD *term1, *term2;
03955                     term1 = poin[S->tree[i]];
03956                     term2 = poin[k];
03957                     if ( MergeWithFloat(B0, &term1,&term2) == 0 )
03958                         goto cancelled;
03959                     poin[S->tree[i]] = term1;
03960                 }
03961 #endif
03962                 else {
03963                     r1 = *( m1 += l1 - 1 );
03964                     m1 -= ABS(r1) - 1;
03965                     r1 = ( ( r1 > 0 ) ? (r1-1) : (r1+1) ) >> 1;
03966                     r2 = *( m2 += l2 - 1 );
03967                     m2 -= ABS(r2) - 1;
03968                     r2 = ( ( r2 > 0 ) ? (r2-1) : (r2+1) ) >> 1;
03969
03970                     if ( AddRat(B0,(UWORD *)m1,r1,(UWORD *)m2,r2,coef,&r3) ) {
03971                         MLOCK(ErrorMessageLock);
03972                         MesCall("MasterMerge");
03973                         MUNLOCK(ErrorMessageLock);
03974                         SETERROR(-1)
03975                     }
03976
03977                     if ( AN.ncmod != 0 ) {
03978                         if ( ( AC.modmode & POSNEG ) != 0 ) {
03979                             NormalModulus(coef,&r3);
03980                         }
03981                         else if ( BigLong(coef,r3,(UWORD *)AC.cmod,ABS(AN.ncmod)) >= 0 ) {
03982                             WORD ii;
03983                             SubPLon(coef,r3,(UWORD *)AC.cmod,ABS(AN.ncmod),coef,&r3);
03984                             coef[r3] = 1;
03985                             for ( ii = 1; ii < r3; ii++ ) coef[r3+ii] = 0;
03986                         }
03987                     }
03988                     r3 *= 2;
03989                     r33 = ( r3 > 0 ) ? ( r3 + 1 ) : ( r3 - 1 );
03990                     if ( r3 < 0 ) r3 = -r3;
03991                     if ( r1 < 0 ) r1 = -r1;
03992                     r1 *= 2;
03993                     r31 = r3 - r1;
03994                     if ( !r3 ) {        /* Terms cancel */
03995 cancelled:
03996                         ul = S->used[level] = S->tree[i];
03997                         S->tree[i] = -1;
03998 /*
03999                         We skip to the next term in stream ul
04000 */
04001                         im = *poin2[ul];
04002                         poin[ul] = poin2[ul];
04003                         ki = S->ktoi[ul];
04004                         AB[ki+1]->T.SB.BlockTerms[AB[ki+1]->T.SB.MasterBlock]--;
04005                         if ( AB[ki+1]->T.SB.BlockTerms[AB[ki+1]->T.SB.MasterBlock] == 0 ) {
04006 /*
04007                             We made it to the end of the block. We have to
04008                             release the previous block and claim the next.
04009 */
04010                             B = AB[ki+1];
04011                             i = AT.SB.MasterBlock;
04012                             if ( i == 1 ) {
04013                                 UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterNumBlocks]);
04014                             }
04015                             else {
04016                                 UNLOCK(AT.SB.MasterBlockLock[i-1]);
04017                             }
04018                             if ( i == AT.SB.MasterNumBlocks ) {
04019                                 i = 1;
04020                             }
04021                             else { i++; }
04022                             LOCK(AT.SB.MasterBlockLock[i]);
04023                             AT.SB.MasterBlock = i;
04024                             poin[ul] = AT.SB.MasterStart[i];
04025                             im = *poin[ul];
04026                             poin2[ul] = poin[ul] + im;
04027                         }
04028                         else {
04029                             poin2[ul] += im;
04030                         }
04031                         S->used[++level] = k;
04032                     }
04033                     else if ( !r31 ) {      /* copy coef into term1 */
04034                         goto CopCof2;
04035                     }
04036                     else if ( r31 < 0 ) {       /* copy coef into term1
04037                                                 and adjust the length of term1 */
04038                         goto CopCoef;
04039                     }
04040                     else {
```

```
04041 /*
04042                             this is the dreaded calamity.
04043                             is there enough space?
04044 */
04045                         if( (poin[S->tree[i]]+l1+r31) >= poin2[S->tree[i]] ) {
04046 /*
04047                                 no space! now the special trick for which
04048                                 we left 2*maxlng spaces open at the beginning
04049                                 of each patch.
04050 */
04051                             if ( (l1 + r31)*((LONG)sizeof(WORD)) >= AM.MaxTer ) {
04052                                 MLOCK(ErrorMessageLock);
04053                                 MesPrint("MasterMerge: Coefficient overflow during sort");
04054                                 MUNLOCK(ErrorMessageLock);
04055                                 goto ReturnError;
04056                             }
04057                             m2 = poin[S->tree[i]];
04058                             m3 = ( poin[S->tree[i]] -= r31 );
04059                             do { *m3++ = *m2++; } while ( m2 < m1 );
04060                             m1 = m3;
04061                         }
04062 CopCoef:
04063                         *(poin[S->tree[i]]) += r31;
04064 CopCof2:
04065                         m2 = (WORD *)coef; im = r3;
04066                         NCOPY(m1,m2,im);
04067                         *m1 = r33;
04068                     }
04069                 }
04070 /*
04071                 Now skip to the next term in stream k.
04072 */
04073 NextTerm:
04074                 im = poin2[k][0];
04075                 poin[k] = poin2[k];
04076                 ki = S->ktoi[k];
04077                 AB[ki+1]->T.SB.BlockTerms[AB[ki+1]->T.SB.MasterBlock]--;
04078                 if ( AB[ki+1]->T.SB.BlockTerms[AB[ki+1]->T.SB.MasterBlock] == 0 ) {
04079 /*
04080                 We made it to the end of the block. We have to
04081                 release the previous block and claim the next.
04082 */
04083                     B = AB[ki+1];
04084                     i = AT.SB.MasterBlock;
04085                     if ( i == 1 ) {
04086                         UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterNumBlocks]);
04087                     }
04088                     else {
04089                         UNLOCK(AT.SB.MasterBlockLock[i-1]);
04090                     }
04091                     if ( i == AT.SB.MasterNumBlocks ) {
04092                         i = 1;
04093                     }
04094                     else { i++; }
04095                     LOCK(AT.SB.MasterBlockLock[i]);
04096                     AT.SB.MasterBlock = i;
04097                     poin[k] = AT.SB.MasterStart[i];
04098                     im = *poin[k];
04099                     poin2[k] = poin[k] + im;
04100                 }
04101                 else {
04102                     poin2[k] += im;
04103                 }
04104                 goto OneTerm;
04105             }
04106         }
04107         else if ( S->tree[i] < 0 ) {
04108             S->tree[i] = k;
04109             level--;
04110             goto OneTerm;
04111         }
04112     }
04113 /*
04114     found the smallest in the set. indicated by k.
04115     write to its destination.
04116 */
04117     S->TermsLeft++;
04118     if ( ( im = PutOut(B0,poin[k],&position,fout,1) ) < 0 ) {
04119 /* INTERNAL_ERROR_EXCL_START */
04120         MLOCK(ErrorMessageLock);
04121         MesPrint("!>Called from MasterMerge with k = %d (stream %d)",k,S->ktoi[k]);
04122         MUNLOCK(ErrorMessageLock);
04123         goto ReturnError;
04124 /* INTERNAL_ERROR_EXCL_STOP */
04125     }
04126     ADDPOS(S->SizeInFile[0],im);
04127     goto NextTerm;
```

```
04128 EndOfMerge:
04129     if ( FlushOut(&position,fout,1) ) goto ReturnError;
04130     ADDPOS(S->SizeInFile[0],1);
04131     CloseFile(fin->handle);
04132     remove(fin->name);
04133     fin->handle = -1;
04134     position = S->SizeInFile[0];
04135     MULPOS(position,sizeof(WORD));
04136 
04137     // Collect global sort statistics information from the threads.
04138     // The total GenTerms is the sum of the thread GenTerms.
04139     // The total small/large buffer sort info is the sum of the thread info.
04140     // The total comparison count is the sum of the thread counts.
04141     // The total unsorted size is the sum of the total generated terms sizes
04142     // The total maximal term size is the maximum of all maximal term sizes
04143     S->GenTerms = 0;
04144     for ( j = 1; j <= numberofworkers; j++ ) {
04145         S->GenTerms += AB[j]->T.SS->GenTerms;
04146         S->verbComparisons += AB[j]->T.SS->verbComparisons;
04147         if ( S->verbMaxTermSize < AB[j]->T.SS->verbMaxTermSize )
04148             S->verbMaxTermSize = AB[j]->T.SS->verbMaxTermSize;
04149         S->verbSBsortTerms += AB[j]->T.SS->verbSBsortTerms;
04150         S->verbSBsortCap += AB[j]->T.SS->verbSBsortCap;
04151         S->verbLBsortPatches += AB[j]->T.SS->verbLBsortPatches;
04152         S->verbLBsortCap += AB[j]->T.SS->verbLBsortCap;
04153         S->verbUnsortedSize += AB[j]->T.SS->verbUnsortedSize;
04154     }
04155 
04156     WriteStats(&position,STATSPOSTSORT,NOCHECKLOGTYPE);
04157     Expressions[AR0.CurExpr].counter = S->TermsLeft;
04158     Expressions[AR0.CurExpr].size = position;
04159 /*
04160     Release all locks
04161 */
04162     for ( i = 1; i <= S->lPatch; i++ ) {
04163         B = AB[i];
04164         UNLOCK(AT.SB.MasterBlockLock[0]);
04165         if ( AT.SB.MasterBlock == 1 ) {
04166             UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterNumBlocks]);
04167         }
04168         else {
04169             UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterBlock-1]);
04170         }
04171         UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterBlock]);
04172     }
04173     AS.MasterSort = 0;
04174     return(0);
04175 ReturnError:
04176     for ( i = 1; i <= S->lPatch; i++ ) {
04177         B = AB[i];
04178         UNLOCK(AT.SB.MasterBlockLock[0]);
04179         if ( AT.SB.MasterBlock == 1 ) {
04180             UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterNumBlocks]);
04181         }
04182         else {
04183             UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterBlock-1]);
04184         }
04185         UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterBlock]);
04186     }
04187     AS.MasterSort = 0;
04188     return(-1);
04189 }
04190 
04191 /*
04192     #] MasterMerge :
04193     #[ SortBotMasterMerge :
04194 */
04195 
04196 #ifdef WITHSORTBOTS
04197 
04213 int SortBotMasterMerge(void)
04214 {
04215     FILEHANDLE *fin, *fout;
04216     ALLPRIVATES *B = AB[0], *BB;
04217     POSITION position;
04218     SORTING *S = AT.SS;
04219     int i, j;
04220 /*
04221     Get the sortbots get to claim their writing blocks.
04222     We have to wait till all have been claimed because they also have to
04223     claim the last writing blocks of the workers to prevent the head of
04224     the circular buffer to overrun the tail.
04225 
04226     Before waiting we can do some needed initializations.
04227     Also the master has to claim the last writing blocks of its input.
04228 */
04229     topsortbotavailables = 0;
```

```
04230     for ( i = numberofworkers+1; i <= numberofworkers+numberofsortbots; i++ ) {
04231         WakeupThread(i,INISORTBOT);
04232     }
04233 
04234     AS.MasterSort = 1;
04235     fout = AR.outfile;
04236     numberofterms = 0;
04237     AR.CompressPointer[0] = 0;
04238     numberclaimed = 0;
04239     BB = AB[AT.SortBotIn1];
04240     LOCK(BB->T.SB.MasterBlockLock[BB->T.SB.MasterNumBlocks]);
04241     BB = AB[AT.SortBotIn2];
04242     LOCK(BB->T.SB.MasterBlockLock[BB->T.SB.MasterNumBlocks]);
04243 
04244     MasterWaitAllSortBots();
04245 /*
04246     Now we can start up the workers. They will claim their writing blocks.
04247     Here the master will wait till all writing blocks have been claimed.
04248 */
04249     for ( i = 1; i <= numberofworkers; i++ ) {
04250         j = GetThread(i);
04251         WakeupThread(i,FINISHEXPRESSION);
04252     }
04253 /*
04254     Prepare the output file in the mean time.
04255 */
04256     if ( fout->handle >= 0 ) {
04257         PUTZERO(SortBotPosition);
04258         SeekFile(fout->handle,&SortBotPosition,SEEK_END);
04259         ADDPOS(SortBotPosition,((fout->POfill-fout->PObuffer)*sizeof(WORD)));
04260     }
04261     else {
04262         SETBASEPOSITION(SortBotPosition,(fout->POfill-fout->PObuffer)*sizeof(WORD));
04263     }
04264     MasterWaitAllBlocks();
04265 /*
04266     Now we can start the sortbots after which the master goes in
04267     sortbot mode to do its part of the job (the very final merge and
04268     the writing to output file).
04269 */
04270     topsortbotavailables = 0;
04271     for ( i = numberofworkers+1; i <= numberofworkers+numberofsortbots; i++ ) {
04272         WakeupThread(i,RUNSORTBOT);
04273     }
04274     if ( SortBotMerge(BHEAD0) ) {
04275 /* INTERNAL_ERROR_EXCL_START */
04276         MLOCK(ErrorMessageLock);
04277         MesPrint("!>Called from SortBotMasterMerge");
04278         MUNLOCK(ErrorMessageLock);
04279         AS.MasterSort = 0;
04280         return(-1);
04281 /* INTERNAL_ERROR_EXCL_STOP */
04282     }
04283 /*
04284     And next the cleanup
04285 */
04286     if ( S->file.handle >= 0 )
04287     {
04288         fin = &S->file;
04289         CloseFile(fin->handle);
04290         remove(fin->name);
04291         fin->handle = -1;
04292     }
04293     position = S->SizeInFile[0];
04294     MULPOS(position,sizeof(WORD));
04295 
04296     // Collect global sort statistics information from the threads.
04297     // The total GenTerms is the sum of the thread GenTerms.
04298     // The total small/large buffer sort info is the sum of the thread info.
04299     // The total comparison count is the sum of the thread and sortbot counts.
04300     // The total unsorted size is the sum of the total generated terms sizes
04301     // The total maximal term size is the maximum of all maximal term sizes
04302     S->GenTerms = 0;
04303     for ( j = 1; j <= numberofworkers; j++ ) {
04304         S->GenTerms += AB[j]->T.SS->GenTerms;
04305         S->verbComparisons += AB[j]->T.SS->verbComparisons;
04306         if ( S->verbMaxTermSize < AB[j]->T.SS->verbMaxTermSize )
04307             S->verbMaxTermSize = AB[j]->T.SS->verbMaxTermSize;
04308         S->verbSBsortTerms += AB[j]->T.SS->verbSBsortTerms;
04309         S->verbSBsortCap += AB[j]->T.SS->verbSBsortCap;
04310         S->verbLBsortPatches += AB[j]->T.SS->verbLBsortPatches;
04311         S->verbLBsortCap += AB[j]->T.SS->verbLBsortCap;
04312         S->verbUnsortedSize += AB[j]->T.SS->verbUnsortedSize;
04313     }
04314     for ( j = numberofworkers+1; j <= numberofworkers+numberofsortbots; j++ ) {
04315         S->verbComparisons += AB[j]->T.SS->verbComparisons;
04316         if ( S->verbMaxTermSize < AB[j]->T.SS->verbMaxTermSize )
```

```
04317             S->verbMaxTermSize = AB[j]->T.SS->verbMaxTermSize;
04318     }
04319
04320     S->TermsLeft = numberofterms;
04321     WriteStats(&position,STATSPOSTSORT,NOCHECKLOGTYPE);
04322     Expressions[AR.CurExpr].counter = S->TermsLeft;
04323     Expressions[AR.CurExpr].size = position;
04324     AS.MasterSort = 0;
04325 /*
04326     The next statement is to prevent one of the sortbots not having
04327     completely cleaned up before the next module starts.
04328     If this statement is omitted every once in a while one of the sortbots
04329     is still running when the next expression starts and misses its
04330     initialization. The result is usually disastrous.
04331 */
04332     MasterWaitAllSortBots();
04333
04334     return(0);
04335 }
04336
04337 #endif
04338
04339 /*
04340     #] SortBotMasterMerge :
04341     #[ SortBotMerge :
04342 */
04343
04344 #ifdef WITHSORTBOTS
04345
04351 int SortBotMerge(PHEAD0)
04352 {
04353     GETBIDENTITY
04354     ALLPRIVATES *Bin1 = AB[AT.SortBotIn1],*Bin2 = AB[AT.SortBotIn2];
04355     WORD *term1, *term2, *wp;
04356     int blin1, blin2;   /* Current block numbers */
04357     int error = 0;
04358     WORD l1, l2, *m1, *m2, *w, r1, r2, r3, r33, r31, *tt1, ii;
04359     WORD *to, *from, im, c;
04360     UWORD *coef;
04361     SORTING *S = AT.SS;
04362 #ifdef WITHFLOAT
04363     WORD *fun1, *fun2, *fun3, *tstop1, *tstop2, size1, size2, size3, l3, jj, *m3;
04364 #endif
04365 /*
04366     Set the pointers to the input terms and the output space
04367 */
04368     coef = AN.SoScratC;
04369     blin1 = 1;
04370     blin2 = 1;
04371     if ( AT.identity == 0 ) {
04372         wp = AT.WorkPointer;
04373     }
04374     else {
04375         wp = AT.WorkPointer = AT.WorkSpace;
04376     }
04377 /*
04378     Get the locks for reading the input
04379     This means that we can start once these locks have been cleared
04380     which means that there will be input.
04381 */
04382     LOCK(Bin1->T.SB.MasterBlockLock[blin1]);
04383     LOCK(Bin2->T.SB.MasterBlockLock[blin2]);
04384
04385     term1 = Bin1->T.SB.MasterStart[blin1];
04386     term2 = Bin2->T.SB.MasterStart[blin2];
04387     AT.SB.FillBlock = 1;
04388 /*
04389     Now the main loop. Keep going until one of the two hits the end.
04390 */
04391     while ( *term1 && *term2 ) {
04392         if ( ( c = CompareTerms(BHEAD term1,term2,(WORD)0) ) > 0 ) {
04393 /*
04394             #[ One is smallest :
04395 */
04396             Bin1->T.SB.BlockTerms[blin1]--;
04397             if ( SortBotOut(BHEAD term1) < 0 ) {
04398 /* INTERNAL_ERROR_EXCL_START */
04399                 MLOCK(ErrorMessageLock);
04400                 MesPrint("!>Called from SortBotMerge with thread = %d",AT.identity);
04401                 MUNLOCK(ErrorMessageLock);
04402                 error = -1;
04403                 goto ReturnError;
04404 /* INTERNAL_ERROR_EXCL_STOP */
04405             }
04406             term1 += *term1;
04407             if ( Bin1->T.SB.BlockTerms[blin1] == 0 ) {
04408                 if ( blin1 == 1 ) {
```

```
04409                     UNLOCK(Bin1->T.SB.MasterBlockLock[Bin1->T.SB.MasterNumBlocks]);
04410                 }
04411                 else {
04412                     UNLOCK(Bin1->T.SB.MasterBlockLock[blin1-1]);
04413                 }
04414                 if ( blin1 == Bin1->T.SB.MasterNumBlocks ) {
04415                     blin1 = 1;
04416                 }
04417                 else {
04418                     blin1++;
04419                 }
04420                 LOCK(Bin1->T.SB.MasterBlockLock[blin1]);
04421                 Bin1->T.SB.MasterBlock = blin1;
04422                 term1 = Bin1->T.SB.MasterStart[blin1];
04423             }
04424 /*
04425             #] One is smallest :
04426 */
04427         }
04428         else if ( c < 0 ) {
04429 /*
04430             #[ Two is smallest :
04431 */
04432             Bin2->T.SB.BlockTerms[blin2]--;
04433             if ( SortBotOut(BHEAD term2) < 0 ) {
04434 /* INTERNAL_ERROR_EXCL_START */
04435                 MLOCK(ErrorMessageLock);
04436                 MesPrint("!>Called from SortBotMerge with thread = %d",AT.identity);
04437                 MUNLOCK(ErrorMessageLock);
04438                 error = -1;
04439                 goto ReturnError;
04440 /* INTERNAL_ERROR_EXCL_STOP */
04441             }
04442 next2:
04443             term2 += *term2;
04444             if ( Bin2->T.SB.BlockTerms[blin2] == 0 ) {
04445                 if ( blin2 == 1 ) {
04446                     UNLOCK(Bin2->T.SB.MasterBlockLock[Bin2->T.SB.MasterNumBlocks]);
04447                 }
04448                 else {
04449                     UNLOCK(Bin2->T.SB.MasterBlockLock[blin2-1]);
04450                 }
04451                 if ( blin2 == Bin2->T.SB.MasterNumBlocks ) {
04452                     blin2 = 1;
04453                 }
04454                 else {
04455                     blin2++;
04456                 }
04457                 LOCK(Bin2->T.SB.MasterBlockLock[blin2]);
04458                 Bin2->T.SB.MasterBlock = blin2;
04459                 term2 = Bin2->T.SB.MasterStart[blin2];
04460             }
04461 /*
04462             #] Two is smallest :
04463 */
04464         }
04465         else {
04466 /*
04467             #[ Equal :
04468 */
04469             Bin1->T.SB.BlockTerms[blin1]--;
04470             Bin2->T.SB.BlockTerms[blin2]--;
04471             l1 = *( m1 = term1 );
04472             l2 = *( m2 = term2 );
04473             if ( S->PolyWise ) {  /* Here we work with PolyFun */
04474                 tt1 = m1;
04475                 m1 += S->PolyWise;
04476                 m2 += S->PolyWise;
04477                 if ( S->PolyFlag == 2 ) {
04478                     AT.WorkPointer = wp;
04479                     w = poly_ratfun_add(BHEAD m1,m2);
04480                     if ( *tt1 + w[1] - m1[1] > AM.MaxTer/((LONG)sizeof(WORD)) ) {
04481                         MLOCK(ErrorMessageLock);
04482                         MesPrint("Term too complex in PolyRatFun addition. MaxTermSize of %10l is too
      small",AM.MaxTer);
04483                         MUNLOCK(ErrorMessageLock);
04484                         Terminate(-1);
04485                     }
04486                     AT.WorkPointer = wp;
04487                     if ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 && w[1] > FUNHEAD ) {
04488                         goto cancelled;
04489                     }
04490                 }
04491                 else {
04492                     w = wp;
04493                     if ( w + m1[1] + m2[1] > AT.WorkTop ) {
04494                         MLOCK(ErrorMessageLock);
```

```
04495                         MesPrint("SortBotMerge(%d): A Maxtermsize of %10l is too small",
04496                                 AT.identity,AM.MaxTer/sizeof(WORD));
04497                         MesPrint("m1[1] = %d, m2[1] = %d, Space =
      %l",m1[1],m2[1],(LONG)(AT.WorkTop-wp));
04498                         PrintTerm(term1,"term1");
04499                         PrintTerm(term2,"term2");
04500                         MesPrint("PolyWise = %d",S->PolyWise);
04501                         MUNLOCK(ErrorMessageLock);
04502                         Terminate(-1);
04503                     }
04504                     AddArgs(BHEAD m1,m2,w);
04505                 }
04506                 r1 = w[1];
04507                 if ( r1 <= FUNHEAD
04508                     || ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 ) )
04509                          { goto cancelled; }
04510                 if ( r1 == m1[1] ) {
04511                     NCOPY(m1,w,r1);
04512                 }
04513                 else if ( r1 < m1[1] ) {
04514                     r2 = m1[1] - r1;
04515                     m2 = w + r1;
04516                     m1 += m1[1];
04517                     while ( --r1 >= 0 ) *--m1 = *--m2;
04518                     m2 = m1 - r2;
04519                     r1 = S->PolyWise;
04520                     while ( --r1 >= 0 ) *--m1 = *--m2;
04521                     *m1 -= r2;
04522                     term1 = m1;
04523                 }
04524                 else {
04525                     // Here we are writing the new merged term *before* the original start of term1.
04526                     // We can always do this, since before term1 there is previous term data of this
04527                     // block, or the previous block, for which we are holding a lock. This requires
04528                     // the existence of "block 0", if term1 is the first term of block 1!
04529                     // It also requires the blocks to be contiguous in memory; we can't allocate
04530                     // separate memory regions for each block without larger-scale changes.
04531                     r2 = r1 - m1[1];
04532                     m2 = tt1 - r2;
04533                     r1 = S->PolyWise;
04534                     m1 = tt1;
04535                     *m1 += r2;
04536                     term1 = m2;
04537                     NCOPY(m2,m1,r1);
04538                     r1 = w[1];
04539                     NCOPY(m2,w,r1);
04540                 }
04541             }
04542 #ifdef WITHFLOAT
04543             else if ( AT.SortFloatMode ) {
04544 /*
04545                 The terms are in m1/term1 and m2/term2 and their length in l1 and l2.
04546                 We have to locate the floats which have already been
04547                 verified in the compare routine. Once we have the new
04548                 term we can jump to the code that writes away the
04549                 result after adding the rationals.
04550 */
04551                 tstop1 = m1+l1; size1 = tstop1[-1]; tstop1 -= ABS(size1);
04552                 tstop2 = m2+l2; size2 = tstop2[-1]; tstop2 -= ABS(size2);
04553                 if ( AT.SortFloatMode == 3 ) {
04554                     fun1 = m1+1;
04555                     while ( fun1[0] != FLOATFUN && fun1+fun1[1] < tstop1 ) {
04556                         fun1 += fun1[1];
04557                     }
04558                     if ( size1 < 0 ) fun1[FUNHEAD+3] = -fun1[FUNHEAD+3];
04559                     UnpackFloat(aux1,fun1);
04560                     fun2 = m2+1;
04561                     while ( fun2[0] != FLOATFUN && fun2+fun2[1] < tstop2 ) {
04562                         fun2 += fun2[1];
04563                     }
04564                     if ( size2 < 0 ) fun2[FUNHEAD+3] = -fun2[FUNHEAD+3];
04565                     UnpackFloat(aux2,fun2);
04566                 }
04567                 else if ( AT.SortFloatMode == 1 ) {
04568                     fun1 = m1+1;
04569                     while ( fun1[0] != FLOATFUN && fun1+fun1[1] < tstop1 ) {
04570                         fun1 += fun1[1];
04571                     }
04572                     if ( size1 < 0 ) fun1[FUNHEAD+3] = -fun1[FUNHEAD+3];
04573                     UnpackFloat(aux1,fun1);
04574                     fun2 = tstop2;
04575                     RatToFloat(aux2,(UWORD *)fun2,size2);
04576                 }
04577                 else if ( AT.SortFloatMode == 2 ) {
04578                     fun1 = tstop1;
04579                     RatToFloat(aux1,(UWORD *)fun1,size1);
04580                     fun2 = m2+1;
```

```
04581                     while ( fun2[0] != FLOATFUN && fun2+fun2[1] < tstop2 ) {
04582                         fun2 += fun2[1];
04583                     }
04584                     if ( size2 < 0 ) fun2[FUNHEAD+3] = -fun2[FUNHEAD+3];
04585                     UnpackFloat(aux2,fun2);
04586                 }
04587                 else {
04588 /* INTERNAL_ERROR_EXCL_START */
04589                     MLOCK(ErrorMessageLock);
04590                     MesPrint("!>Illegal value %d for AT.SortFloatMode in
      SortBotMerge.",AT.SortFloatMode);
04591                     MUNLOCK(ErrorMessageLock);
04592                     Terminate(-1);
04593                     return(0);
04594 /* INTERNAL_ERROR_EXCL_STOP */
04595                 }
04596                 mpf_add(aux3,aux1,aux2);
04597                 size3 = mpf_sgn(aux3);
04598                 if ( size3 == 0 ) { /* Cancelling! Rare! */
04599                     goto cancelled;
04600                 }
04601                 else if ( size3 < 0 ) mpf_neg(aux3,aux3);
04602                 fun3 = TermMalloc("MasterMerge");
04603                 PackFloat(fun3,aux3);
04604                 l3 = fun3[1]+(fun1-m1)+3; /* new size */
04605 
04606                 if ( l3 != l1 ) {
04607 /*
04608                     Copy it all to wp
04609 */
04610                     if ( (l3)*((LONG)sizeof(WORD)) >= AM.MaxTer ) {
04611                         MLOCK(ErrorMessageLock);
04612                         MesPrint("MasterMerge: Coefficient overflow during sort");
04613                         MUNLOCK(ErrorMessageLock);
04614                         goto ReturnError;
04615                     }
04616                     m3 = wp; m2 = term1;
04617                     while ( m2 < fun1 ) *m3++ = *m2++;
04618                     for ( jj = 0; jj < fun3[1]; jj++ ) *m3++ = fun3[jj];
04619                     *m3++ = 1; *m3++ = 1;
04620                     *m3++ = size3 < 0 ? -3: 3;
04621                     *wp = m3-wp;
04622                     TermFree(fun3,"MasterMerge");
04623                     goto PutOutwp;
04624                 }
04625                 for ( jj = 0; jj < fun3[1]; jj++ ) fun1[jj] = fun3[jj];
04626                 fun1 += fun3[1];
04627                 *fun1++ = 1; *fun1++ = 1;
04628                 *fun1++ = size3 < 0 ? -3: 3;
04629                 *term1 = fun1-term1;
04630                 TermFree(fun3,"MasterMerge");
04631             }
04632 #endif
04633             else {
04634                 r1 = *( m1 += l1 - 1 );
04635                 m1 -= ABS(r1) - 1;
04636                 r1 = ( ( r1 > 0 ) ? (r1-1) : (r1+1) ) >> 1;
04637                 r2 = *( m2 += l2 - 1 );
04638                 m2 -= ABS(r2) - 1;
04639                 r2 = ( ( r2 > 0 ) ? (r2-1) : (r2+1) ) >> 1;
04640 
04641                 if ( AddRat(BHEAD (UWORD *)m1,r1,(UWORD *)m2,r2,coef,&r3) ) {
04642                     MLOCK(ErrorMessageLock);
04643                     MesCall("SortBotMerge");
04644                     MUNLOCK(ErrorMessageLock);
04645                     SETERROR(-1)
04646                 }
04647 
04648                 if ( AN.ncmod != 0 ) {
04649                     if ( ( AC.modmode & POSNEG ) != 0 ) {
04650                         NormalModulus(coef,&r3);
04651                     }
04652                     else if ( BigLong(coef,r3,(UWORD *)AC.cmod,ABS(AN.ncmod)) >= 0 ) {
04653                         SubPLon(coef,r3,(UWORD *)AC.cmod,ABS(AN.ncmod),coef,&r3);
04654                         coef[r3] = 1;
04655                         for ( ii = 1; ii < r3; ii++ ) coef[r3+ii] = 0;
04656                     }
04657                 }
04658                 if ( !r3 ) { goto cancelled; }
04659                 r3 *= 2;
04660                 r33 = ( r3 > 0 ) ? ( r3 + 1 ) : ( r3 - 1 );
04661                 if ( r3 < 0 ) r3 = -r3;
04662                 if ( r1 < 0 ) r1 = -r1;
04663                 r1 *= 2;
04664                 r31 = r3 - r1;
04665                 if ( !r31 ) {       /* copy coef into term1 */
04666                     m2 = (WORD *)coef; im = r3;
```

```
04667                     NCOPY(m1,m2,im);
04668                     *m1 = r33;
04669                 }
04670                 else {
04671                     to = wp; from = term1;
04672                     while ( from < m1 ) *to++ = *from++;
04673                     from = (WORD *)coef; im = r3;
04674                     NCOPY(to,from,im);
04675                     *to++ = r33;
04676                     wp[0] = to - wp;
04677 PutOutwp:
04678                     if ( SortBotOut(BHEAD wp) < 0 ) {
04679 /* INTERNAL_ERROR_EXCL_START */
04680                         MLOCK(ErrorMessageLock);
04681                         MesPrint("!>Called from SortBotMerge with thread = %d",AT.identity);
04682                         MUNLOCK(ErrorMessageLock);
04683                         error = -1;
04684                         goto ReturnError;
04685 /* INTERNAL_ERROR_EXCL_STOP */
04686                     }
04687                     goto cancelled;
04688                 }
04689             }
04690             if ( SortBotOut(BHEAD term1) < 0 ) {
04691 /* INTERNAL_ERROR_EXCL_START */
04692                 MLOCK(ErrorMessageLock);
04693                 MesPrint("!>Called from SortBotMerge with thread = %d",AT.identity);
04694                 MUNLOCK(ErrorMessageLock);
04695                 error = -1;
04696                 goto ReturnError;
04697 /* INTERNAL_ERROR_EXCL_STOP */
04698             }
04699 cancelled:;     /* Now we need two new terms */
04700             term1 += *term1;
04701             if ( Bin1->T.SB.BlockTerms[blin1] == 0 ) {
04702 
04703                 if ( blin1 == 1 ) {
04704                     UNLOCK(Bin1->T.SB.MasterBlockLock[Bin1->T.SB.MasterNumBlocks]);
04705                 }
04706                 else {
04707                     UNLOCK(Bin1->T.SB.MasterBlockLock[blin1-1]);
04708                 }
04709                 if ( blin1 == Bin1->T.SB.MasterNumBlocks ) {
04710                     blin1 = 1;
04711                 }
04712                 else {
04713                     blin1++;
04714                 }
04715                 LOCK(Bin1->T.SB.MasterBlockLock[blin1]);
04716                 Bin1->T.SB.MasterBlock = blin1;
04717                 term1 = Bin1->T.SB.MasterStart[blin1];
04718             }
04719             goto next2;
04720 /*
04721             #] Equal :
04722 */
04723         }
04724     }
04725 /*
04726     Copy the tail
04727 */
04728     if ( *term1 ) {
04729 /*
04730             #[ Tail in one :
04731 */
04732         while ( *term1 ) {
04733             Bin1->T.SB.BlockTerms[blin1]--;
04734             if ( SortBotOut(BHEAD term1) < 0 ) {
04735 /* INTERNAL_ERROR_EXCL_START */
04736                 MLOCK(ErrorMessageLock);
04737                 MesPrint("!>Called from SortBotMerge with thread = %d",AT.identity);
04738                 MUNLOCK(ErrorMessageLock);
04739                 error = -1;
04740                 goto ReturnError;
04741 /* INTERNAL_ERROR_EXCL_STOP */
04742             }
04743             if ( Bin1->T.SB.BlockTerms[blin1] == 0 ) {
04744 
04745                 if ( blin1 == 1 ) {
04746                     UNLOCK(Bin1->T.SB.MasterBlockLock[Bin1->T.SB.MasterNumBlocks]);
04747                 }
04748                 else {
04749                     UNLOCK(Bin1->T.SB.MasterBlockLock[blin1-1]);
04750                 }
04751                 if ( blin1 == Bin1->T.SB.MasterNumBlocks ) {
04752                     blin1 = 1;
04753                 }
```

```
04754                 else {
04755                     blin1++;
04756                 }
04757                 LOCK(Bin1->T.SB.MasterBlockLock[blin1]);
04758                 Bin1->T.SB.MasterBlock = blin1;
04759                 term1 = Bin1->T.SB.MasterStart[blin1];
04760             }
04761             else {
04762                 term1 += *term1;
04763             }
04764         }
04765 /*
04766             #] Tail in one :
04767 */
04768     }
04769     else if ( *term2 ) {
04770 /*
04771             #[ Tail in two :
04772 */
04773         while ( *term2 ) {
04774             Bin2->T.SB.BlockTerms[blin2]--;
04775             if ( SortBotOut(BHEAD term2) < 0 ) {
04776 /* INTERNAL_ERROR_EXCL_START */
04777                 MLOCK(ErrorMessageLock);
04778                 MesPrint("!>Called from SortBotMerge with thread = %d",AT.identity);
04779                 MUNLOCK(ErrorMessageLock);
04780                 error = -1;
04781                 goto ReturnError;
04782 /* INTERNAL_ERROR_EXCL_STOP */
04783             }
04784             if ( Bin2->T.SB.BlockTerms[blin2] == 0 ) {
04785 
04786                 if ( blin2 == 1 ) {
04787                     UNLOCK(Bin2->T.SB.MasterBlockLock[Bin2->T.SB.MasterNumBlocks]);
04788                 }
04789                 else {
04790                     UNLOCK(Bin2->T.SB.MasterBlockLock[blin2-1]);
04791                 }
04792                 if ( blin2 == Bin2->T.SB.MasterNumBlocks ) {
04793                     blin2 = 1;
04794                 }
04795                 else {
04796                     blin2++;
04797                 }
04798                 LOCK(Bin2->T.SB.MasterBlockLock[blin2]);
04799                 Bin2->T.SB.MasterBlock = blin2;
04800                 term2 = Bin2->T.SB.MasterStart[blin2];
04801             }
04802             else {
04803                 term2 += *term2;
04804             }
04805         }
04806 /*
04807             #] Tail in two :
04808 */
04809     }
04810 
04811     // Both streams have hit the end-of-stream marker "0". We still need to
04812     // decrement the BlockTerms counters a final time, the marker is included
04813     // in the count.
04814     Bin1->T.SB.BlockTerms[blin1]--;
04815     Bin2->T.SB.BlockTerms[blin2]--;
04816 
04817     SortBotOut(BHEAD 0);
04818 ReturnError:;
04819 /*
04820     Release all locks.
04821 */
04822     UNLOCK(Bin1->T.SB.MasterBlockLock[blin1]);
04823     if ( blin1 > 1 ) {
04824         UNLOCK(Bin1->T.SB.MasterBlockLock[blin1-1]);
04825     }
04826     else {
04827         UNLOCK(Bin1->T.SB.MasterBlockLock[Bin1->T.SB.MasterNumBlocks]);
04828     }
04829     UNLOCK(Bin2->T.SB.MasterBlockLock[blin2]);
04830     if ( blin2 > 1 ) {
04831         UNLOCK(Bin2->T.SB.MasterBlockLock[blin2-1]);
04832     }
04833     else {
04834         UNLOCK(Bin2->T.SB.MasterBlockLock[Bin2->T.SB.MasterNumBlocks]);
04835     }
04836     if ( AT.identity > 0 ) {
04837         UNLOCK(AT.SB.MasterBlockLock[AT.SB.FillBlock]);
04838     }
04839 /*
04840     And that was all folks
```

```
04841 */
04842     return(error);
04843 }
04844
04845 #endif
04846
04847 /*
04848     #] SortBotMerge :
04849     #[ IniSortBlocks :
04850 */
04851
04852 static int SortBlocksInitialized = 0;
04853
04860 int IniSortBlocks(int numworkers)
04861 {
04862     ALLPRIVATES *B;
04863     SORTING *S;
04864     LONG totalsize, workersize, blocksize, numberofterms;
04865     int maxter, id, j;
04866     int numberofblocks = NUMBEROFBLOCKSINSORT, numparts;
04867     WORD *w;
04868
04869     if ( SortBlocksInitialized ) return(0);
04870     SortBlocksInitialized = 1;
04871     if ( numworkers == 0 ) return(0);
04872
04873 #ifdef WITHSORTBOTS
04874     if ( numworkers > 2 ) {
04875         numparts = 2*numworkers - 2;
04876         numberofblocks = numberofblocks/2;
04877     }
04878     else {
04879         numparts = numworkers;
04880     }
04881 #else
04882     numparts = numworkers;
04883 #endif
04884     S = AM.S0;
04885     totalsize = S->LargeSize + S->SmallEsize;
04886     workersize = totalsize / numparts;
04887     maxter = AM.MaxTer/sizeof(WORD);
04888     blocksize = ( workersize - maxter )/numberofblocks;
04889     numberofterms = blocksize / maxter;
04890     if ( numberofterms < MINIMUMNUMBEROFTERMS ) {
04891 /*
04892         This should have been taken care of in RecalcSetups.
04893 */
04894 /* INTERNAL_ERROR_EXCL_START */
04895         MesPrint("!>We have a problem with the size of the blocks in IniSortBlocks");
04896         Terminate(-1);
04897 /* INTERNAL_ERROR_EXCL_STOP */
04898     }
04899 /*
04900     Layout:  For each worker
04901                 block 0: size is maxter WORDS
04902                 numberofblocks blocks of size blocksize WORDS
04903 */
04904     w = S->lBuffer;
04905     if ( w == 0 ) w = S->sBuffer;
04906     for ( id = 1; id <= numparts; id++ ) {
04907         B = AB[id];
04908         AT.SB.MasterBlockLock = (pthread_mutex_t *)Malloc1(
04909             sizeof(pthread_mutex_t)*(numberofblocks+1),"MasterBlockLock");
04910         AT.SB.MasterStart = (WORD **)Malloc1(sizeof(WORD *)*(numberofblocks+1)*3,"MasterBlock");
04911         AT.SB.MasterFill = AT.SB.MasterStart + (numberofblocks+1);
04912         AT.SB.MasterStop = AT.SB.MasterFill  + (numberofblocks+1);
04913         AT.SB.MasterNumBlocks = numberofblocks;
04914         AT.SB.BlockTerms = (LONG*)Malloc1(sizeof(LONG)*(numberofblocks+1),"BlockTerms");
04915         AT.SB.MasterBlock = 0;
04916         AT.SB.FillBlock = 0;
04917         AT.SB.MasterFill[0] = AT.SB.MasterStart[0] = w;
04918         AT.SB.BlockTerms[0] = 0;
04919         w += maxter;
04920         AT.SB.MasterStop[0] = w;
04921         AT.SB.MasterBlockLock[0] = dummylock;
04922         for ( j = 1; j <= numberofblocks; j++ ) {
04923             AT.SB.MasterFill[j] = AT.SB.MasterStart[j] = w;
04924             AT.SB.BlockTerms[j] = 0;
04925             w += blocksize;
04926             AT.SB.MasterStop[j] = w;
04927             AT.SB.MasterBlockLock[j] = dummylock;
04928         }
04929     }
04930     if ( w > S->sTop2 ) {
04931 /* INTERNAL_ERROR_EXCL_START */
04932         MesPrint("!>Counting problem in IniSortBlocks");
04933         Terminate(-1);
```

```
04934 /* INTERNAL_ERROR_EXCL_STOP */
04935     }
04936     return(0);
04937 }
04938
04939 /*
04940     #] IniSortBlocks :
04941     #[ UpdateSortBlocks :
04942 */
04943
04948 int UpdateSortBlocks(int numworkers)
04949 {
04950     ALLPRIVATES *B;
04951     SORTING *S;
04952     LONG totalsize, workersize, blocksize, numberofterms;
04953     int maxter, id, j;
04954     int numberofblocks = NUMBEROFBLOCKSINSORT, numparts;
04955     WORD *w;
04956
04957     if ( numworkers == 0 ) return(0);
04958
04959 #ifdef WITHSORTBOTS
04960     if ( numworkers > 2 ) {
04961         numparts = 2*numworkers - 2;
04962         numberofblocks = numberofblocks/2;
04963     }
04964     else {
04965         numparts = numworkers;
04966     }
04967 #else
04968     numparts = numworkers;
04969 #endif
04970     S = AM.S0;
04971     totalsize = S->LargeSize + S->SmallEsize;
04972     workersize = totalsize / numparts;
04973     maxter = AM.MaxTer/sizeof(WORD);
04974     blocksize = ( workersize - maxter )/numberofblocks;
04975     numberofterms = blocksize / maxter;
04976     if ( numberofterms < MINIMUMNUMBEROFTERMS ) {
04977 /*
04978         This should have been taken care of in RecalcSetups.
04979 */
04980 /* INTERNAL_ERROR_EXCL_START */
04981         MesPrint("!>We have a problem with the size of the blocks in UpdateSortBlocks");
04982         Terminate(-1);
04983 /* INTERNAL_ERROR_EXCL_STOP */
04984     }
04985 /*
04986     Layout:  For each worker
04987                 block 0: size is maxter WORDS
04988                 numberofblocks blocks of size blocksize WORDS
04989 */
04990     w = S->lBuffer;
04991     if ( w == 0 ) w = S->sBuffer;
04992     for ( id = 1; id <= numparts; id++ ) {
04993         B = AB[id];
04994         AT.SB.MasterFill[0] = AT.SB.MasterStart[0] = w;
04995         w += maxter;
04996         AT.SB.MasterStop[0] = w;
04997         for ( j = 1; j <= numberofblocks; j++ ) {
04998             AT.SB.MasterFill[j] = AT.SB.MasterStart[j] = w;
04999             w += blocksize;
05000             AT.SB.MasterStop[j] = w;
05001         }
05002     }
05003     if ( w > S->sTop2 ) {
05004 /* INTERNAL_ERROR_EXCL_START */
05005         MesPrint("!>Counting problem in UpdateSortBlocks");
05006         Terminate(-1);
05007 /* INTERNAL_ERROR_EXCL_STOP */
05008     }
05009     return(0);
05010 }
05011
05012 /*
05013     #] UpdateSortBlocks :
05014     #[ DefineSortBotTree :
05015 */
05016
05017 #ifdef WITHSORTBOTS
05018
05024 void DefineSortBotTree(void)
05025 {
05026     ALLPRIVATES *B;
05027     int n, i, from;
05028     if ( numberofworkers <= 2 ) return;
05029     n = numberofworkers*2-2;
```

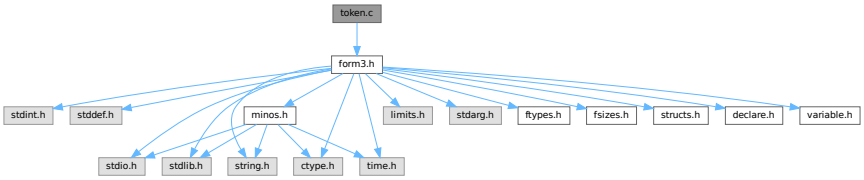
```
05030     for ( i = numberofworkers+1, from = 1; i <= n; i++ ) {
05031         B = AB[i];
05032         AT.SortBotIn1 = from++;
05033         AT.SortBotIn2 = from++;
05034     }
05035     B = AB[0];
05036     AT.SortBotIn1 = from++;
05037     AT.SortBotIn2 = from++;
05038 }
05039
05040 #endif
05041
05042 /*
05043     #] DefineSortBotTree :
05044     #[ GetTerm2 :
05045
05046     Routine does a GetTerm but only when a bracket index is involved and
05047     only from brackets that have been judged not suitable for treatment
05048     as complete brackets by a single worker. Whether or not a bracket should
05049     be treated by a single worker is decided by TreatIndexEntry
05050 */
05051
05052 WORD GetTerm2(PHEAD WORD *term)
05053 {
05054     WORD *ttco, *tt, retval;
05055     LONG n,i;
05056     FILEHANDLE *fi;
05057     EXPRESSIONS e = AN.expr;
05058     BRACKETINFO *b  = e->bracketinfo;
05059     BRACKETINDEX *bi = b->indexbuffer;
05060     POSITION where, eonfile = AS.OldOnFile[e-Expressions], bstart, bnext;
05061 /*
05062     1: Get the current position.
05063 */
05064     switch ( e->status ) {
05065         case UNHIDELEXPRESSION:
05066         case UNHIDEGEXPRESSION:
05067         case DROPHLEXPRESSION:
05068         case DROPHGEXPRESSION:
05069         case HIDDENLEXPRESSION:
05070         case HIDDENGEXPRESSION:
05071             fi = AR.hidefile;
05072             break;
05073         default:
05074             fi = AR.infile;
05075             break;
05076     }
05077     if ( AR.KeptInHold ) {
05078         retval = GetTerm(BHEAD term);
05079         return(retval);
05080     }
05081     SeekScratch(fi,&where);
05082     if ( AN.lastinindex < 0 ) {
05083 /*
05084         We have to test whether we have to do the first bracket
05085 */
05086         if ( ( n = TreatIndexEntry(BHEAD 0) ) <= 0 ) {
05087             AN.lastinindex = n;
05088             where = bi[n].start;
05089             ADD2POS(where,eonfile);
05090             SetScratch(fi,&where);
05091 /*
05092             Put the bracket in the Compress buffer.
05093 */
05094             ttco = AR.CompressBuffer;
05095             tt = b->bracketbuffer + bi[0].bracket;
05096             i = *tt;
05097             NCOPY(ttco,tt,i)
05098             AR.CompressPointer = ttco;
05099             retval = GetTerm(BHEAD term);
05100             return(retval);
05101         }
05102         else AN.lastinindex = n-1;
05103     }
05104 /*
05105     2: Find the corresponding index number
05106        a: test whether it is in the current bracket
05107 */
05108     n = AN.lastinindex;
05109     bstart = bi[n].start;
05110     ADD2POS(bstart,eonfile);
05111     bnext = bi[n].next;
05112     ADD2POS(bnext,eonfile);
05113     if ( ISLESSPOS(bstart,where) && ISLESSPOS(where,bnext) ) {
05114         retval = GetTerm(BHEAD term);
05115         return(retval);
05116     }
```

```
05117     for ( n++ ; n < b->indexfill; n++ ) {
05118         i = TreatIndexEntry(BHEAD n);
05119         if ( i <= 0 ) {
05120 /*
05121             Put the bracket in the Compress buffer.
05122 */
05123             ttco = AR.CompressBuffer;
05124             tt = b->bracketbuffer + bi[n].bracket;
05125             i = *tt;
05126             NCOPY(ttco,tt,i)
05127             AR.CompressPointer = ttco;
05128             AN.lastinindex = n;
05129             where = bi[n].start;
05130             ADD2POS(where,eonfile);
05131             SetScratch(fi,&(where));
05132             retval = GetTerm(BHEAD term);
05133             return(retval);
05134         }
05135         else n += i - 1;
05136     }
05137     return(0);
05138 }
05139
05140 /*
05141     #] GetTerm2 :
05142     #[ TreatIndexEntry :
05143 */
05152 int TreatIndexEntry(PHEAD LONG n)
05153 {
05154     BRACKETINFO *b  = AN.expr->bracketinfo;
05155     LONG numbra = b->indexfill - 1, i;
05156     LONG totterms;
05157     BRACKETINDEX *bi;
05158     POSITION pos1, average;
05159 /*
05160     1: number of the bracket should be such that there is one bucket
05161        for each worker remaining.
05162 */
05163     if ( ( numbra - n ) <= numberofworkers ) return(0);
05164 /*
05165     2: size of the bracket contents should be less than what remains in
05166        the expression divided by the number of workers.
05167 */
05168     bi = b->indexbuffer;
05169     DIFPOS(pos1,bi[numbra].next,bi[n].next);  /* Size of what remains */
05170     BASEPOSITION(average) = DIVPOS(pos1,(3*numberofworkers));
05171     DIFPOS(pos1,bi[n].next,bi[n].start);      /* Size of the current bracket */
05172     if ( ISLESSPOS(average,pos1) ) return(0);
05173 /*
05174     It passes.
05175     Now check whether we can do more brackets
05176 */
05177     totterms = bi->termsinbracket;
05178     if ( totterms > 2*AC.ThreadBucketSize ) return(1);
05179     for ( i = 1; i < numbra-n; i++ ) {
05180         DIFPOS(pos1,bi[n+i].next,bi[n].start); /* Size of the combined brackets */
05181         if ( ISLESSPOS(average,pos1) ) return(i);
05182         totterms += bi->termsinbracket;
05183         if ( totterms > 2*AC.ThreadBucketSize ) return(i+1);
05184     }
05185 /*
05186     We have a problem at the end of the system. Just do one.
05187 */
05188     return(1);
05189 }
05190
05191 /*
05192     #] TreatIndexEntry :
05193     #[ SetHideFiles :
05194 */
05195
05196 void SetHideFiles(void) {
05197     int i;
05198     ALLPRIVATES *B, *B0 = AB[0];
05199     for ( i = 1; i <= numberofworkers; i++ ) {
05200         B = AB[i];
05201         AR.hidefile->handle = AR0.hidefile->handle;
05202         if ( AR.hidefile->handle < 0 ) {
05203             AR.hidefile->PObuffer = AR0.hidefile->PObuffer;
05204             AR.hidefile->POstop = AR0.hidefile->POstop;
05205             AR.hidefile->POfill = AR0.hidefile->POfill;
05206             AR.hidefile->POfull = AR0.hidefile->POfull;
05207             AR.hidefile->POsize = AR0.hidefile->POsize;
05208             AR.hidefile->POposition = AR0.hidefile->POposition;
05209             AR.hidefile->filesize = AR0.hidefile->filesize;
05210         }
05211         else {
```

```
05212             AR.hidefile->PObuffer = AR.hidefile->wPObuffer;
05213             AR.hidefile->POstop = AR.hidefile->wPOstop;
05214             AR.hidefile->POfill = AR.hidefile->wPOfill;
05215             AR.hidefile->POfull = AR.hidefile->wPOfull;
05216             AR.hidefile->POsize = AR.hidefile->wPOsize;
05217             PUTZERO(AR.hidefile->POposition);
05218         }
05219     }
05220 }
05221
05222 /*
05223     #] SetHideFiles :
05224     #[ IniFbufs :
05225 */
05226
05227 void IniFbufs(void)
05228 {
05229     int i;
05230     for ( i = 0; i < AM.totalnumberofthreads; i++ ) {
05231         IniFbuffer(AB[i]->T.fbufnum);
05232     }
05233 }
05234
05235 /*
05236     #] IniFbufs :
05237     #[ SetMods :
05238 */
05239
05240 void SetMods(void)
05241 {
05242     ALLPRIVATES *B;
05243     int i, n, j;
05244     for ( j = 0; j < AM.totalnumberofthreads; j++ ) {
05245         B = AB[j];
05246         AN.ncmod = AC.ncmod;
05247         if ( AN.cmod != 0 ) M_free(AN.cmod,"AN.cmod");
05248         n = ABS(AN.ncmod);
05249         AN.cmod = (UWORD *)Malloc1(sizeof(WORD)*n,"AN.cmod");
05250         for ( i = 0; i < n; i++ ) AN.cmod[i] = AC.cmod[i];
05251     }
05252 }
05253
05254 /*
05255     #] SetMods :
05256     #[ UnSetMods :
05257 */
05258
05259 void UnSetMods(void)
05260 {
05261     ALLPRIVATES *B;
05262     int j;
05263     for ( j = 0; j < AM.totalnumberofthreads; j++ ) {
05264         B = AB[j];
05265         if ( AN.cmod != 0 ) M_free(AN.cmod,"AN.cmod");
05266         AN.cmod = 0;
05267     }
05268 }
05269
05270 /*
05271     #] UnSetMods :
05272     #[ find_Horner_MCTS_expand_tree_threaded :
05273 */
05274
05275 void find_Horner_MCTS_expand_tree_threaded(void) {
05276     int id;
05277     while (( id = GetAvailableThread() ) < 0)
05278         MasterWait();
05279     WakeupThread(id,MCTSEXPANDTREE);
05280 }
05281
05282 /*
05283     #] find_Horner_MCTS_expand_tree_threaded :
05284     #[ optimize_expression_given_Horner_threaded :
05285 */
05286
05287 extern void optimize_expression_given_Horner_threaded(void) {
05288     int id;
05289     while (( id = GetAvailableThread() ) < 0)
05290         MasterWait();
05291     WakeupThread(id,OPTIMIZEEXPRESSION);
05292 }
05293
05294 /*
05295     #] optimize_expression_given_Horner_threaded :
05296 */
05297
05298 #endif
```

4.147 token.c File Reference

`#include "form3.h"`
Include dependency graph for token.c:



Macros

- #define CHECKPOLY {if(polyflag)MesPrint("&Illegal use of polynomial function"); polyflag = 0; }

Functions

- int tokenize (UBYTE ∗in, WORD leftright)
- void WriteTokens (SBYTE ∗in)
- int simp1token (SBYTE ∗s)
- int simpwtoken (SBYTE ∗s)
- int simp2token (SBYTE ∗s)
- int simp3atoken (SBYTE ∗s, int mode)
- int simp3btoken (SBYTE ∗s, int mode)
- int simp4token (SBYTE ∗s)
- int simp5token (SBYTE ∗s, int mode)
- int simp6token (SBYTE ∗tokens, int mode)

Variables

- char ∗ ttypes [ ]

4.147.1 Detailed Description

The tokenizer. This is a part of the compiler that does an intermediate type of translation. It does look up the names etc and can do a number of optimizations. The resulting output is a stream of bytes which can be processed by the code generator (in the file compiler.c)

Definition in file token.c.

4.147.2 Macro Definition Documentation

4.147.2.1 CHECKPOLY

```
#define CHECKPOLY {if(polyflag)MesPrint("&Illegal use of polynomial function"); polyflag = 0;
}
```

Definition at line 56 of file token.c.

4.147.3 Function Documentation

4.147.3.1 tokenize()

```
int tokenize (
            UBYTE * in,
            WORD leftright )
```

Definition at line 58 of file token.c.

4.147.3.2 WriteTokens()

```
void WriteTokens (
            SBYTE * in )
```

Definition at line 652 of file token.c.

4.147.3.3 simp1token()

```
int simp1token (
            SBYTE * s )
```

Definition at line 700 of file token.c.

4.147.3.4 simpwtoken()

```
int simpwtoken (
            SBYTE * s )
```

Definition at line 789 of file token.c.

4.147.3.5 simp2token()

```
int simp2token (
            SBYTE * s )
```

Definition at line 943 of file token.c.

4.147.3.6 simp3atoken()

```
int simp3atoken (
            SBYTE * s,
            int mode )
```

Definition at line 1191 of file token.c.

4.147.3.7 simp3btoken()

```
int simp3btoken (
            SBYTE * s,
            int mode )
```

Definition at line 1432 of file token.c.

4.147.3.8 simp4token()

```
int simp4token (
            SBYTE * s )
```

Definition at line 1843 of file token.c.

4.147.3.9 simp5token()

```
int simp5token (
            SBYTE * s,
            int mode )
```

Definition at line 2005 of file token.c.

4.147.3.10 simp6token()

```
int simp6token (
            SBYTE * tokens,
            int mode )
```

Definition at line 2051 of file token.c.

4.147.4 Variable Documentation

4.147.4.1 ttypes

```
char* ttypes[]
```

Initial value:

```
= { "\n", "S", "I", "V", "F", "set", "E", "dotp", "#",
   "sub", "d_", "$", "dub", "(", ")", "?", "??", ".", "[", "]",
   ",", "((", "))", "*", "/", "^", "+", "-", "!", "end", "{{", "}}",
   "N_?", "conj", "()", "#d", "^d", "_", "snum"  }
```

Definition at line 647 of file token.c.

4.148 token.c

Go to the documentation of this file.

```
00001
00008 /* #[ License : */
00009 /*
00010  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00011  *   When using this file you are requested to refer to the publication
00012  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013  *   This is considered a matter of courtesy as the development was paid
00014  *   for by FOM the Dutch physics granting agency and we would like to
00015  *   be able to track its scientific use to convince FOM of its value
00016  *   for the community.
00017  *
00018  *   This file is part of FORM.
00019  *
00020  *   FORM is free software: you can redistribute it and/or modify it under the
00021  *   terms of the GNU General Public License as published by the Free Software
00022  *   Foundation, either version 3 of the License, or (at your option) any later
00023  *   version.
00024  *
00025  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00027  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00028  *   details.
00029  *
00030  *   You should have received a copy of the GNU General Public License along
00031  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00032  */
00033 /* #] License : */
00034 /*
00035     #[ Includes :
00036 */
00037
00038 #include "form3.h"
00039
00040 /*
00041     #] Includes :
00042     #[ Compiler :
00043         #[ tokenize :
00044
00045         Takes the input in 'in' and translates it into tokens.
00046         The tokens are put in the token buffer which starts at 'AC.tokens'
00047         and runs till 'AC.toptokens'
00048         We may assume that the various types of brackets match properly.
00049         object = -1: after , or (
00050         object = 0: name/variable/number etc is allowed
00051         object = 1: variable.
00052         object = 2: number
00053         object = 3: ) after subexpression
00054 */
00055
00056 #define CHECKPOLY {if(polyflag)MesPrint("&Illegal use of polynomial function"); polyflag = 0; }
00057
00058 int tokenize(UBYTE *in, WORD leftright)
00059 {
00060     int error = 0, object, funlevel = 0, bracelevel = 0, explevel = 0, numexp;
00061     int polyflag = 0;
00062     WORD number, type;
00063     UBYTE *s = in, c;
00064     SBYTE *out, *outtop, num[MAXNUMSIZE], *t;
00065     LONG i;
00066     if ( AC.tokens == 0 ) {
00067         SBYTE **ppp = &(AC.tokens); /* to avoid a compiler warning */
00068         SBYTE **pppp = &(AC.toptokens);
00069         DoubleBuffer((void **)ppp,(void **)pppp,sizeof(SBYTE),"start tokens");
00070     }
00071     out = AC.tokens;
00072     outtop = AC.toptokens - MAXNUMSIZE;
00073     AC.dumnumflag = 0;
00074     object = 0;
00075     while ( *in ) {
00076         if ( out > outtop ) {
00077             LONG oldsize = (LONG)(out - AC.tokens);
00078             SBYTE **ppp = &(AC.tokens); /* to avoid a compiler warning */
00079             SBYTE **pppp = &(AC.toptokens);
00080             DoubleBuffer((void **)ppp,(void **)pppp,sizeof(SBYTE),"expand tokens");
00081             out = AC.tokens + oldsize;
00082             outtop = AC.toptokens - MAXNUMSIZE;
00083         }
00084         switch ( FG.cTable[*in] ) {
00085             case 0:     /* a-zA-Z */
00086                 CHECKPOLY
00087                 s = in++;
00088                 while ( FG.cTable[*in] == 0 || FG.cTable[*in] == 1
```

```
00089                 || *in == '_' ) in++;
00090 dovariable:     c = *in; *in = 0;
00091                 if ( object > 0 ) {
00092                     MesPrint("&Illegal position for %s",s);
00093                     if ( !error ) error = 1;
00094                 }
00095                 if ( out > AC.tokens && ( out[-1] == TWILDCARD || out[-1] == TNOT ) ) {
00096                     type = GetName(AC.varnames,s,&number,NOAUTO);
00097                 }
00098                 else {
00099                     type = GetName(AC.varnames,s,&number,WITHAUTO);
00100                 }
00101                 if ( type < 0 )
00102                     type = GetName(AC.exprnames,s,&number,NOAUTO);
00103                 switch ( type ) {
00104                     case CSYMBOL:       *out++ = TSYMBOL;     break;
00105                     case CINDEX:
00106                         if ( number >= (AM.IndDum-AM.OffsetIndex) ) {
00107                             if ( c != '?' ) {
00108                                 MesPrint("&Generated indices should be of the type Nnumber_?");
00109                                 error = 1;
00110                             }
00111                             else {
00112                                 *in++ = c; c = *in; *in = 0;
00113                                 AC.dumnumflag = 1;
00114                             }
00115                         }
00116                         *out++ = TINDEX;
00117                         break;
00118                     case CVECTOR:       *out++ = TVECTOR;     break;
00119                     case CFUNCTION:
00120 #ifdef WITHMPI
00121                         /*
00122                          * In the preprocessor, random functions in #$var=... and #inside
00123                          * may cause troubles, because the program flow on a slave may be
00124                          * different from those on others. We set AC.RhsExprInModuleFlag in order
00125                          * to make the change of $-variable be done on the master and thus keep the
00126                          * consistency among the master and all slave processes. The previous value
00127                          * of AC.RhsExprInModuleFlag will be restored after #$var=... and #inside.
00128                          */
00129                         if ( AP.PreAssignFlag || AP.PreInsideLevel ) {
00130                             switch ( number + FUNCTION ) {
00131                                 case RANDOMFUNCTION:
00132                                 case RANPERM:
00133                                     AC.RhsExprInModuleFlag = 1;
00134                             }
00135                         }
00136 #endif
00137                         *out++ = TFUNCTION;
00138                         break;
00139                     case CSET:          *out++ = TSET;        break;
00140                     case CEXPRESSION:   *out++ = TEXPRESSION;
00141                                         if ( leftright == LHSIDE ) {
00142                                             if ( !error ) error = 1;
00143                                             MesPrint("&Expression not allowed in LH-side of
      substitution: %s",s);
00144                                         }
00145 /*[06nov2003 mt]:*/
00146 #ifdef WITHMPI
00147                                         else { /*RHSide*/
00148                                             /* NOTE: We always set AC.RhsExprInModuleFlag regardless
      of
00149                                              *       AP.PreAssignFlag or AP.PreInsideLevel because we
      have to detect
00150                                              *       RHS expressions even in those cases. */
00151                                             AC.RhsExprInModuleFlag = 1;
00152                                         }
00153                                         if ( !AP.PreAssignFlag && !AP.PreInsideLevel )
00154                                             Expressions[number].vflags |= ISINRHS;
00155 #endif
00156 /*:[06nov2003 mt]*/
00157                                         if ( AC.exprfillwarning == 0 ) {
00158                                                 AC.exprfillwarning = 1;
00159                                         }
00160                                         break;
00161                     case CDELTA:        *out++ = TDELTA;      *in = c;
00162                                         object = 1; continue;
00163                     case CDUBIOUS:      *out++ = TDUBIOUS;    break;
00164                     default:            *out++ = TDUBIOUS;
00165                                         if ( !error ) error = 1;
00166                                         MesPrint("&Undeclared variable %s",s);
00167                                         number = AddDubious(s);
00168                                         break;
00169                 }
00170                 object = 1;
00171 donumber:       i = 0;
00172                 do { num[i++] = (SBYTE)(number & 0x7F); number »= 7; } while ( number );
```

```
00173                 while ( --i >= 0 ) *out++ = num[i];
00174                 *in = c;
00175                 break;
00176             case 1:     /* 0-9 */
00177               {
00178 #ifdef WITHFLOAT
00179                 int spec;
00180                 UBYTE *in2;
00181 #endif
00182                 CHECKPOLY
00183                 s = in;
00184                 while ( *s == '0' && FG.cTable[s[1]] == 1 ) s++;
00185                 in = s+1; i = 1;
00186                 while ( FG.cTable[*in] == 1 ) { in++; i++; }
00187                 if ( object > 0 ) {
00188                     c = *in; *in = 0;
00189                     MesPrint("&Illegal position for %s",s);
00190                     *in = c;
00191                     if ( !error ) error = 1;
00192                 }
00193                 if ( i == 1 && *in == '_' && ( *s == '5' || *s == '6'
00194                 || *s == '7' ) ) {
00195                     in++; *out++ = TSGAMMA; *out++ = (SBYTE)(*s - '4');
00196                     object = 1;
00197                     break;
00198                 }
00199 #ifdef WITHFLOAT
00200                 in2 = CheckFloat(in,&spec);
00201                 if ( in2 > in ) {
00202                     if ( spec == -1 ) {
00203                         MesPrint("&The floating point system has not been started: %s",in);
00204                         if ( !error ) error = 1;
00205                     }
00206                     else {
00207                       in = in2;
00208 dofloat:
00209                       while ( out + (in-s) >= AC.toptokens ) {
00210                         LONG oldsize = (LONG)(out - AC.tokens);
00211                         SBYTE **ppp = &(AC.tokens); /* to avoid a compiler warning */
00212                         SBYTE **pppp = &(AC.toptokens);
00213                         DoubleBuffer((void **)ppp,(void **)pppp,sizeof(SBYTE),"more tokens");
00214                         out = AC.tokens + oldsize;
00215                         outtop = AC.toptokens - MAXNUMSIZE;
00216                       }
00217                       *out++ = TFLOAT;
00218                       while ( s < in ) *out++ = *s++;
00219                     }
00220                 }
00221                 else
00222 #endif
00223                 {
00224                 *out++ = TNUMBER;
00225                 if ( ( i & 1 ) != 0 ) *out++ = (SBYTE)(*s++ - '0');
00226                 while ( out + (in-s)/2 >= AC.toptokens ) {
00227                     LONG oldsize = (LONG)(out - AC.tokens);
00228                     SBYTE **ppp = &(AC.tokens); /* to avoid a compiler warning */
00229                     SBYTE **pppp = &(AC.toptokens);
00230                     DoubleBuffer((void **)ppp,(void **)pppp,sizeof(SBYTE),"more tokens");
00231                     out = AC.tokens + oldsize;
00232                     outtop = AC.toptokens - MAXNUMSIZE;
00233                 }
00234                 while ( s < in ) {   /* We store in base 100 */
00235                     *out++ = (SBYTE)(( *s - '0' ) * 10 + ( s[1] - '0' ));
00236                     s += 2;
00237                 }
00238                 }
00239                 object = 2;
00240               }
00241               break;
00242             case 2:     /* . $ _ ? # ' */
00243                 CHECKPOLY
00244                 if ( *in == '?' ) {
00245                     if ( leftright == LHSIDE ) {
00246                         if ( object == 1 ) {    /* follows a name */
00247                             in++; *out++ = TWILDCARD;
00248                             if ( FG.cTable[in[0]] == 0 || in[0] == '[' || in[0] == '{' ) object = 0;
00249                         }
00250                         else if ( object == -1 ) {  /* follows comma or ( */
00251                             in++; s = in;
00252                             while ( FG.cTable[*in] == 0 || FG.cTable[*in] == 1 ) in++;
00253                             c = *in; *in = 0;
00254                             if ( FG.cTable[*s] != 0 ) {
00255                                 MesPrint("&Illegal name for argument list variable %s",s);
00256                                 error = 1;
00257                             }
00258                             else {
00259                                 i = AddWildcardName((UBYTE *)s);
```

```
00260                               *in = c;
00261                               *out++ = TWILDARG;
00262                               *out++ = (SBYTE)i;
00263                           }
00264                           object = 1;
00265                       }
00266                       else {
00267                           MesPrint("&Illegal position for ?");
00268                           error = 1;
00269                           in++;
00270                       }
00271                   }
00272                   else {
00273                       if ( object != -1 ) goto IllPos;
00274                       in++;
00275                       if ( FG.cTable[*in] == 0 || FG.cTable[*in] == 1 ) {
00276                           s = in;
00277                           while ( FG.cTable[*in] == 0 || FG.cTable[*in] == 1 ) in++;
00278                           c = *in; *in = 0;
00279                           i = GetWildcardName((UBYTE *)s);
00280                           if ( i <= 0 ) {
00281                               MesPrint("&Undefined argument list variable %s",s);
00282                               error = 1;
00283                           }
00284                           *in = c;
00285                           *out++ = TWILDARG;
00286                           *out++ = (SBYTE)i;
00287                       }
00288                       else {
00289                           if ( AC.vectorlikeLHS == 0 ) {
00290                               MesPrint("&Generated index ? only allowed in vector substitution",s);
00291                               error = 1;
00292                           }
00293                           *out++ = TGENINDEX;
00294                       }
00295                       object = 1;
00296                   }
00297               }
00298               else if ( *in == '.' ) {
00299                   if ( object == 1 ) {    /* follows a name */
00300                       *out++ = TDOT;
00301                       object = 0;
00302                       in++;
00303                   }
00304 #ifdef WITHFLOAT
00305                   else if ( object == 0 || object == -1 ) {
00306 /*
00307                       Test for floating point number
00308 */
00309                       int spec;
00310                       s = CheckFloat(in,&spec);
00311                       if ( s > in ) {
00312                           if ( spec == -1 ) {
00313                               MesPrint("&The floating point system has not been started: %s",in);
00314                               if ( !error ) error = 1;
00315                               in++;
00316                           }
00317                           else {
00318                               UBYTE *a = s; s = in; in = a;
00319                               goto dofloat;
00320                           }
00321                       }
00322                       else goto IllPos;
00323                   }
00324 #endif
00325                   else goto IllPos;
00326               }
00327               else if ( *in == '$' ) {    /* $ variable */
00328                   in++;
00329                   s = in;
00330                   if ( FG.cTable[*in] == 0 ) {
00331                       while ( FG.cTable[*in] == 0 || FG.cTable[*in] == 1 ) in++;
00332                       if ( *in == '_' && AP.PreAssignFlag == 2 ) in++;
00333                       c = *in; *in = 0;
00334                       if ( object > 0 ) {
00335                           if ( object != 1 || leftright == RHSIDE ) {
00336                               MesPrint("&Illegal position for $%s",s);
00337                               if ( !error ) error = 1;
00338                           }   /* else can be assignment in wildcard */
00339                           else {
00340                               if ( ( number = GetDollar(s) ) < 0 ) {
00341                                   number = AddDollar(s,0,0,0);
00342                               }
00343                           }
00344                       }
00345                       else if ( ( number = GetDollar(s) ) < 0 ) {
00346                           MesPrint("&Undefined variable $%s",s);
```

```
00347                           if ( !error ) error = 1;
00348                           number = AddDollar(s,0,0,0);
00349                       }
00350                       *out++ = TDOLLAR;
00351                       object = 1;
00352                       if ( ( AC.exprfillwarning == 0 ) &&
00353                            ( ( out > AC.tokens+1 ) && ( out[-2] != TWILDCARD ) ) ) {
00354                           AC.exprfillwarning = 1;
00355                       }
00356                       goto donumber;
00357                   }
00358                   else {
00359                       MesPrint("Illegal name for $ variable after %s",in);
00360                       if ( !error ) error = 1;
00361                   }
00362               }
00363               else if ( *in == '#' ) {
00364                   in++;
00365                   if ( object == 1 ) {    /* follows a name */
00366                       *out++ = TCONJUGATE;
00367                   }
00368                   else {
00369                       MesPrint("&Illegal position for %#");
00370                       error = 1;
00371                   }
00372               }
00373               else goto IllPos;
00374               break;
00375           case 3:     /* [ ] */
00376               CHECKPOLY
00377               if ( *in == '[' ) {
00378                   if ( object == 1 ) {    /* after name */
00379                       t = out-1;
00380                       if ( *t == RPARENTHESIS ) {
00381                           *out++ = LBRACE; *out++ = LPARENTHESIS;
00382                           bracelevel++; explevel = bracelevel;
00383                       }
00384                       else {
00385                           while ( *t >= 0 && t > AC.tokens ) t--;
00386                           if ( *t == TEXPRESSION ) {
00387                               *out++ = LBRACE; *out++ = LPARENTHESIS;
00388                               bracelevel++; explevel = bracelevel;
00389                           }
00390                           else {*out++ = LBRACE; bracelevel++; }
00391                       }
00392                       object = 0;
00393                   }
00394                   else {                  /* name. find matching ] */
00395                       s = in;
00396                       in = SkipAName(in);
00397                       goto dovariable;
00398                   }
00399               }
00400               else {
00401                   if ( explevel > 0 && explevel == bracelevel ) {
00402                       *out++ = RPARENTHESIS; explevel = 0;
00403                   }
00404                   *out++ = RBRACE; object = 1; bracelevel--;
00405               }
00406               in++;
00407               break;
00408           case 4:     /* ( ) = ; , */
00409               if ( *in == '(' ) {
00410                   if ( funlevel >= AM.MaxParLevel ) {
00411                       MesPrint("&More than %d levels of parentheses",AM.MaxParLevel);
00412                       return(-1);
00413                   }
00414                   if ( object == 1 ) {    /* After name -> function,vector */
00415                       AC.tokenarglevel[funlevel++] = TYPEISFUN;
00416                       *out++ = TFUNOPEN;
00417                       if ( polyflag ) {
00418                           if ( in[1] != ')' && in[1] != ',' ) {
00419                               *out++ = TNUMBER; *out++ = (SBYTE)(polyflag);
00420                               *out++ = TCOMMA;
00421                               *out++ = LPARENTHESIS;
00422                           }
00423                           else {
00424                               *out++ = LPARENTHESIS;
00425                               *out++ = TNUMBER; *out++ = (SBYTE)(polyflag);
00426                           }
00427                           polyflag = 0;
00428                       }
00429                       else if ( in[1] != ')' && in[1] != ',' ) {
00430                           *out++ = LPARENTHESIS;
00431                       }
00432                   }
00433                   else if ( object <= 0 ) {
```

```
00434                       CHECKPOLY
00435                       AC.tokenarglevel[funlevel++] = TYPEISSUB;
00436                       *out++ = LPARENTHESIS;
00437                   }
00438                   else {
00439                       polyflag = 0;
00440                       AC.tokenarglevel[funlevel++] = TYPEISMYSTERY;
00441                       MesPrint("&Illegal position for (: %s",in);
00442                       if ( error >= 0 ) error = -1;
00443                   }
00444                   object = -1;
00445               }
00446               else if ( *in == ')' ) {
00447                   funlevel--;
00448                   if ( funlevel < 0 ) {
00449 /*                    if ( funflag == 0 ) { */
00450                           MesPrint("&There is an unmatched parenthesis");
00451                           if ( error >= 0 ) error = -1;
00452 /*                    } */
00453                   }
00454                   else if ( object <= 0
00455                   && ( AC.tokenarglevel[funlevel] != TYPEISFUN
00456                   || out[-1] != TFUNOPEN ) ) {
00457                       MesPrint("&Illegal position for closing parenthesis.");
00458                       if ( error >= 0 ) error = -1;
00459                       if ( AC.tokenarglevel[funlevel] == TYPEISFUN ) object = 1;
00460                       else object = 3;
00461                   }
00462                   else {
00463                       if ( AC.tokenarglevel[funlevel] == TYPEISFUN ) {
00464                           if ( out[-1] == TFUNOPEN ) out--;
00465                           else {
00466                               if ( out[-1] != TCOMMA ) *out++ = RPARENTHESIS;
00467                               *out++ = TFUNCLOSE;
00468                           }
00469                           object = 1;
00470                       }
00471                       else if ( AC.tokenarglevel[funlevel] == TYPEISSUB ) {
00472                           *out++ = RPARENTHESIS;
00473                           object = 3;
00474                       }
00475                   }
00476               }
00477               else if ( *in == ',' ) {
00478                   if ( /* object > 0 && */ funlevel > 0 &&
00479                   AC.tokenarglevel[funlevel-1] == TYPEISFUN ) {
00480                       if ( out[-1] != TFUNOPEN && out[-1] != TCOMMA )
00481                           *out++ = RPARENTHESIS;
00482                       else { *out++ = TNUMBER; *out++ = 0; }
00483                       *out++ = TCOMMA;
00484                       if ( in[1] != ',' && in[1] != ')' )
00485                           *out++ = LPARENTHESIS;
00486                       else if ( in[1] == ')' ) {
00487                           *out++ = TNUMBER; *out++ = 0;
00488                       }
00489                   }
00490 /*
00491                   else if ( object > 0 ) {
00492                   }
00493 */
00494                   else {
00495                       MesPrint("&Illegal position for comma: %s",in);
00496                       MesPrint("&Forgotten ; ?");
00497                       if ( error >= 0 ) error = -1;
00498                   }
00499                   object = -1;
00500               }
00501               else goto IllPos;
00502               in++;
00503               break;
00504           case 5:     /* + - * % / ^ : */
00505               CHECKPOLY
00506               if ( *in == ':' || *in == '%' ) goto IllPos;
00507               if ( *in == '*' || *in == '/' || *in == '^' ) {
00508                   if ( object <= 0 ) {
00509                       MesPrint("&Illegal position for operator: %s",in);
00510                       if ( error >= 0 ) error = -1;
00511                   }
00512                   else if ( *in == '*' ) *out++ = TMULTIPLY;
00513                   else if ( *in == '/' ) *out++ = TDIVIDE;
00514                   else                   *out++ = TPOWER;
00515                   in++;
00516               }
00517               else {
00518                   i = 1;
00519                   while ( *in == '+' || *in == '-' ) {
00520                       if ( *in == '-' ) i = -i;
```

```
00521                       in++;
00522                   }
00523                   if ( i == 1 ) {
00524                       if ( out > AC.tokens && out[-1] != TFUNOPEN &&
00525                       out[-1] != LPARENTHESIS && out[-1] != TCOMMA
00526                       && out[-1] != LBRACE )
00527                           *out++ = TPLUS;
00528                   }
00529                   else *out++ = TMINUS;
00530               }
00531               object = 0;
00532               break;
00533           case 6:     /* Whitespace */
00534               in++; break;
00535           case 7:     /* { | } */
00536               CHECKPOLY
00537               if ( *in == '{' ) {
00538                   if ( object > 0 ) {
00539                       MesPrint("&Illegal position for %s",in);
00540                       if ( !error ) error = 1;
00541                   }
00542                   s = in+1;
00543                   SKIPBRA2(in)
00544                   number = DoTempSet(s,in);
00545                   in++;
00546                   if ( number >= 0 ) {
00547                       *out++ = TSET;
00548                       i = 0;
00549                       do { num[i++] = (SBYTE)(number & 0x7F); number »= 7; } while ( number );
00550                       while ( --i >= 0 ) *out++ = num[i];
00551                   }
00552                   else if ( error == 0 ) error = 1;
00553                   object = 1;
00554               }
00555               else goto IllPos;
00556               break;
00557           case 8:     /* ! & < > */
00558               CHECKPOLY
00559               if ( *in != '!' || leftright == RHSIDE
00560               || object != 1 || out[-1] != TWILDCARD ) goto IllPos;
00561               *out++ = TNOT;
00562               if ( FG.cTable[in[1]] == 0 || in[1] == '[' || in[1] == '{' ) object = 0;
00563               in++;
00564               break;
00565           default:
00566 IllPos:       MesPrint("&Illegal character at this position: %s",in);
00567               if ( error >= 0 ) error = -1;
00568               in++;
00569               polyflag = 0;
00570               break;
00571       }
00572   }
00573   *out++ = TENDOFIT;
00574   AC.endoftokens = out;
00575   if ( funlevel > 0 || bracelevel != 0 ) {
00576       if ( funlevel > 0 ) MesPrint("&Unmatched parentheses");
00577       if ( bracelevel != 0 ) MesPrint("&Unmatched braces");
00578       return(-1);
00579   }
00580   if ( AC.TokensWriteFlag ) WriteTokens(AC.tokens);
00581 /*
00582   Simplify fixed set elements
00583 */
00584   if ( error == 0 && simp1token(AC.tokens) ) error = 1;
00585 /*
00586   Collect wildcards for the prototype. Simplify the leftover wildcards
00587 */
00588   if ( error == 0 && leftright == LHSIDE && simpwtoken(AC.tokens) )
00589               error = 1;
00590 /*
00591   Now prepare the set[n] objects in the RHS.
00592 */
00593   if ( error == 0 && leftright == RHSIDE && simp4token(AC.tokens) )
00594               error = 1;
00595 /*
00596   Simplify simple function arguments (and 1/fac_ and 1/invfac_)
00597 */
00598   if ( error == 0 && simp2token(AC.tokens) ) error = 1;
00599 /*
00600   Next we try to remove composite denominators or exponents and
00601   replace them by their internal functions. This may involve expanding
00602   the buffer. The return code of 3a is negative if there is an error
00603   and positive if indeed we need to do some work.
00604   simp3btoken does the work
00605 */
00606   numexp = 0;
00607   if ( error == 0 && ( numexp = simp3atoken(AC.tokens,leftright) ) < 0 )
```

```
00608         error = 1;
00609     if ( numexp > 0 ) {
00610         // Make some space at the beginning of AC.tokens, filled with TEMPTY.
00611         SBYTE *tt;
00612         out = AC.tokens;
00613         while ( *out != TENDOFIT ) out++;
00614         while ( out+numexp*9 > outtop ) {
00615             LONG oldsize = (LONG)(out - AC.tokens);
00616             SBYTE **ppp = &(AC.tokens); /* to avoid a compiler warning */
00617             SBYTE **pppp = &(AC.toptokens);
00618             DoubleBuffer((void **)ppp,(void **)pppp,sizeof(SBYTE),"out tokens");
00619             out = AC.tokens + oldsize;
00620             outtop = AC.toptokens - MAXNUMSIZE;
00621         }
00622         tt = out + numexp*9;
00623         while ( out >= AC.tokens ) { *tt-- = *out--; }
00624         while ( tt >= AC.tokens ) { *tt-- = TEMPTY; }
00625         if ( error == 0 && simp3btoken(AC.tokens,leftright) ) error = 1;
00626         if ( error == 0 && simp2token(AC.tokens) ) error = 1;
00627     }
00628 /*
00629     In simp5token we test for special cases like sumvariables that are
00630     already wildcards, etc.
00631 */
00632     if ( error == 0 && simp5token(AC.tokens,leftright) ) error = 1;
00633 /*
00634     In simp6token we test for special cases like factorized expressions
00635     that occur in the RHS in an improper way.
00636 */
00637     if ( error == 0 && simp6token(AC.tokens,leftright) ) error = 1;
00638
00639     return(error);
00640 }
00641
00642 /*
00643         #] tokenize :
00644         #[ WriteTokens :
00645 */
00646
00647 char *ttypes[] = { "\n", "S", "I", "V", "F", "set", "E", "dotp", "#",
00648    "sub", "d_", "$", "dub", "(", ")", "?", "??", ".", "[", "]",
00649    ",", "((", "))", "*", "/", "^", "+", "-", "!", "end", "{{", "}}",
00650    "N_?", "conj", "()", "#d", "^d", "_", "snum"  };
00651
00652 void WriteTokens(SBYTE *in)
00653 {
00654     int numinline = 0, x, n = sizeof(ttypes)/sizeof(char *);
00655     char outbuf[81], *s, *out, c;
00656     out = outbuf;
00657     while ( *in != TENDOFIT ) {
00658         if ( *in < 0 ) {
00659             if ( *in >= -n ) {
00660                 s = ttypes[-*in];
00661                 while ( *s ) { *out++ = *s++; numinline++; }
00662             }
00663             else {
00664                 *out++ = '-'; x = -*in; numinline++;
00665                 goto writenumber;
00666             }
00667         }
00668         else {
00669             x = *in;
00670 writenumber:
00671             s = out;
00672             do {
00673                 *out++ = (char)(( x % 10 ) + '0');
00674                 numinline++;
00675                 x = x / 10;
00676             } while ( x );
00677             c = out[-1]; out[-1] = *s; *s = c;
00678         }
00679         if ( numinline > 70 ) {
00680             *out = 0;
00681             MesPrint("%s",outbuf);
00682             out = outbuf; numinline = 0;
00683         }
00684         else {
00685             *out++ = ' '; numinline++;
00686         }
00687         in++;
00688     }
00689     if ( numinline > 0 ) { *out = 0; MesPrint("%s",outbuf); }
00690 }
00691
00692 /*
00693         #] WriteTokens :
00694         #[ simp1token :
```

```
00695
00696         Routine substitutes set elements if possible.
00697         This means sets with a fixed argument like setname[3].
00698 */
00699
00700 int simp1token(SBYTE *s)
00701 {
00702     int error = 0, n, i, base;
00703     WORD numsub;
00704     SBYTE *fill = s, *start, *t, numtab[10];
00705     SETS set;
00706     while ( *s != TENDOFIT ) {
00707         if ( *s == RBRACE ) {
00708             start = fill-1;
00709             while ( *start != LBRACE ) start--;
00710             t = start - 1;
00711             while ( *t >= 0 ) t--;
00712             if ( *t == TSET && ( start[1] == TNUMBER || start[1] == TNUMBER1 ) ) {
00713                 base = start[1] == TNUMBER ? 100: 128;
00714                 start += 2;
00715                 numsub = *start++;
00716                 while ( *start >= 0 && start < fill )
00717                     { numsub = base*numsub + *start++; }
00718                 if ( start == fill ) {
00719                     start = t;
00720                     t++; n = *t++; while ( *t >= 0 ) { n = 128*n + *t++; }
00721                     set = Sets+n;
00722                     if ( ( set->type != CRANGE )
00723                     && ( numsub > 0 && numsub <= set->last-set->first ) ) {
00724                         fill = start;
00725                         n = SetElements[set->first+numsub-1];
00726                         switch (set->type) {
00727                             case CSYMBOL:
00728                                 if ( n > MAXPOWER ) {
00729                                     n -= 2*MAXPOWER;
00730                                     if ( n < 0 ) { n = -n; *fill++ = TMINUS; }
00731                                     *fill++ = TNUMBER1;
00732                                 }
00733                                 else *fill++ = TSYMBOL;
00734                                 break;
00735                             case CINDEX:
00736                                 if ( n < AM.OffsetIndex ) *fill++ = TNUMBER1;
00737                                 else {
00738                                     *fill++ = TINDEX;
00739                                     n -= AM.OffsetIndex;
00740                                 }
00741                                 break;
00742                             case CVECTOR:   *fill++ = TVECTOR;
00743                                 n -= AM.OffsetVector;   break;
00744                             case CFUNCTION: *fill++ = TFUNCTION;
00745                                 n -= FUNCTION; break;
00746                             case CNUMBER:   *fill++ = TNUMBER1;  break;
00747                             case CDUBIOUS:  *fill++ = TDUBIOUS; n = 1; break;
00748                         }
00749                         i = 0;
00750 if ( n < 0 ) {
00751     MesPrint("Value of n = %d",n);
00752 }
00753                         do { numtab[i++] = (SBYTE)(n & 0x7F); n »= 7; } while ( n );
00754                         while ( --i >= 0 ) *fill++ = numtab[i];
00755                     }
00756                     else {
00757                         MesPrint("&Illegal element %d in set",numsub);
00758                         error++;
00759                     }
00760                     s++; continue;
00761                 }
00762             }
00763             *fill++ = *s++;
00764         }
00765         else *fill++ = *s++;
00766     }
00767     *fill++ = TENDOFIT;
00768     return(error);
00769 }
00770
00771 /*
00772         #] simp1token :
00773         #[ simpwtoken :
00774
00775         Only to be called in the LHS.
00776         Hunts down the wildcards and writes them to the wildcardbuffer.
00777         Next it causes the ProtoType to be constructed.
00778         All wildcards are simplified into the trailing TWILDCARD,
00779         because the specifics are stored in the prototype.
00780         These specifics also include the transfer of wildcard values
00781         to $variables.
```

```
00782
00783         Types of wildcards:
00784         a?, a?set, a?!set, a?set[i], A?set1?set2, ?a
00785         After this we can strip the set information.
00786         We still need the ? because of the wildcarding offset in code generation
00787 */
00788
00789 int simpwtoken(SBYTE *s)
00790 {
00791     int error = 0, first = 1, notflag;
00792     WORD num, numto, numdollar, *w = AC.WildC, *wstart, *wtop;
00793     SBYTE *fill = s, *t, *v, *s0 = s;
00794     while ( *s != TENDOFIT ) {
00795         if ( *s == TWILDCARD ) {
00796             notflag = 0; t = fill;
00797             while ( t > s0 && t[-1] >= 0 ) t--;
00798             v = t; num = 0; *fill++ = *s++;
00799             while ( *v >= 0 ) num = 128*num + *v++;
00800             if ( t > s0 ) t--;
00801             AC.NwildC += 4;
00802             if ( AC.NwildC > 4*AM.MaxWildcards ) goto firsterr;
00803             switch ( *t ) {
00804                 case TSYMBOL:
00805                 case TDUBIOUS:
00806                     *w++ = SYMTOSYM; *w++ = 4; *w++ = num; *w++ = num; break;
00807                 case TINDEX:
00808                     num += AM.OffsetIndex;
00809                     *w++ = INDTOIND; *w++ = 4; *w++ = num; *w++ = num;  break;
00810                 case TVECTOR:
00811                     num += AM.OffsetVector;
00812                     *w++ = VECTOVEC; *w++ = 4; *w++ = num; *w++ = num;  break;
00813                 case TFUNCTION:
00814                     num += FUNCTION;
00815                     *w++ = FUNTOFUN; *w++ = 4; *w++ = num; *w++ = num;  break;
00816                 default:
00817                     MesPrint("&Illegal type of wildcard in LHS");
00818                     error = -1;
00819                     *w++ = SYMTOSYM; *w++ = 4; *w++ = num; *w++ = num;  break;
00820                     break;
00821             }
00822 /*
00823             Now the sets. The s pointer sits after the ?
00824 */
00825             wstart = w;
00826             if ( *s == TNOT && s[1] == TSET ) { notflag = 1; s++; }
00827             if ( *s == TSET ) {
00828                 s++; num = 0; while ( *s >= 0 ) num = 128*num + *s++;
00829                 if ( notflag == 0 && *s == TWILDCARD && s[1] == TSET ) {
00830                     s += 2; numto = 0; while ( *s >= 0 ) numto = 128*numto + *s++;
00831                     if ( num < AM.NumFixedSets || numto < AM.NumFixedSets
00832                     || Sets[num].type == CRANGE || Sets[numto].type == CRANGE ) {
00833                         MesPrint("&This type of set not allowed in this wildcard construction");
00834                         error = 1;
00835                     }
00836                     else {
00837                     AC.NwildC += 4;
00838                     if ( AC.NwildC > 4*AM.MaxWildcards ) goto firsterr;
00839                     *w++ = FROMSET; *w++ = 4; *w++ = num; *w++ = numto;
00840                     wstart = w;
00841                     }
00842                 }
00843                 else if ( notflag == 0 && *s == LBRACE && s[1] == TSYMBOL ) {
00844                     if ( num < AM.NumFixedSets || Sets[num].type == CRANGE ) {
00845                         MesPrint("&This type of set not allowed in this wildcard construction");
00846                         error = 1;
00847                     }
00848                     v = s; s += 2;
00849                     numto = 0; while ( *s >= 0 ) numto = 128*numto + *s++;
00850                     if ( *s == TWILDCARD ) s++; /* most common mistake */
00851                     if ( *s == RBRACE ) {
00852                         s++;
00853                         AC.NwildC += 8;
00854                         if ( AC.NwildC > 4*AM.MaxWildcards ) goto firsterr;
00855                         *w++ = SETTONUM; *w++ = 4; *w++ = num; *w++ = numto;
00856                         wstart = w;
00857                         *w++ = SYMTOSYM; *w++ = 4; *w++ = numto; *w++ = 0;
00858                     }
00859                     else if ( *s == TDOLLAR ) {
00860                         s++; numdollar = 0;
00861                         while ( *s >= 0 ) numdollar = 128*numdollar + *s++;
00862                         if ( *s == RBRACE ) {
00863                             s++;
00864                             AC.NwildC += 12;
00865                             if ( AC.NwildC > 4*AM.MaxWildcards ) goto firsterr;
00866                             *w++ = SETTONUM; *w++ = 4; *w++ = num; *w++ = numto;
00867                             wstart = w;
00868                             *w++ = SYMTOSYM; *w++ = 4; *w++ = numto; *w++ = 0;
```

```
00869                               *w++ = LOADDOLLAR; *w++ = 4; *w++ = numdollar;
00870                               *w++ = numdollar;
00871                           }
00872                           else { s = v; goto singlewild; }
00873                       }
00874                       else { s = v; goto singlewild; }
00875                   }
00876                   else {
00877 singlewild:          num += notflag * 2*WILDOFFSET;
00878                       AC.NwildC += 4;
00879                       if ( AC.NwildC > 4*AM.MaxWildcards ) goto firsterr;
00880                       *w++ = FROMSET; *w++ = 4; *w++ = num; *w++ = -WILDOFFSET;
00881                       wstart = w;
00882                   }
00883               }
00884               else if ( *s != TDOLLAR && *s != TENDOFIT && *s != RPARENTHESIS
00885               && *s != RBRACE && *s != TCOMMA && *s != TFUNCLOSE && *s != TMULTIPLY
00886               && *s != TPOWER && *s != TDIVIDE && *s != TPLUS && *s != TMINUS
00887               && *s != TPOWER1 && *s != TEMPTY && *s != TFUNOPEN && *s != TDOT ) {
00888                   MesPrint("&Illegal type of wildcard in LHS");
00889                   error = -1;
00890               }
00891               if ( *s == TDOLLAR ) {
00892                   s++; numdollar = 0;
00893                   while ( *s >= 0 ) numdollar = 128*numdollar + *s++;
00894                   AC.NwildC += 4;
00895                   if ( AC.NwildC > 4*AM.MaxWildcards ) goto firsterr;
00896                   wtop = w + 4;
00897                   if ( wstart < w ) {
00898                       while ( w > wstart ) { w[4] = w[0]; w--; }
00899                   }
00900                   *w++ = LOADDOLLAR; *w++ = 4; *w++ = numdollar; *w++ = numdollar;
00901                   w = wtop;
00902               }
00903           }
00904           else if ( *s == TWILDARG ) {
00905               *fill++ = *s++;
00906               num = 0;
00907               while ( *s >= 0 ) { num = 128*num + *s;  *fill++ = *s++; }
00908               AC.NwildC += 4;
00909               if ( AC.NwildC > 4*AM.MaxWildcards ) {
00910 firsterr:         if ( first ) {
00911                       MesPrint("&More than %d wildcards",AM.MaxWildcards);
00912                       error = -1;
00913                       first = 0;
00914                   }
00915               }
00916               else { *w++ = ARGTOARG; *w++ = 4; *w++ = num; *w++ = -1; }
00917               if ( *s == TDOLLAR ) {
00918                   s++; num = 0; while ( *s >= 0 ) num = 128*num + *s++;
00919                   AC.NwildC += 4;
00920                   if ( AC.NwildC > 4*AM.MaxWildcards ) goto firsterr;
00921                   *w++ = LOADDOLLAR; *w++ = 4; *w++ = num; *w++ = num;
00922               }
00923           }
00924           else *fill++ = *s++;
00925       }
00926       *fill++ = TENDOFIT;
00927       AC.WildC = w;
00928       return(error);
00929   }
00930
00931   /*
00932           #] simpwtoken :
00933           #[ simp2token :
00934
00935           Deals with function arguments.
00936           The tokenizer has given function arguments extra parentheses.
00937           We remove the double parentheses.
00938           Next we remove the parentheses around the simple arguments.
00939
00940           It also replaces /fac_() by *invfac_() and /invfac_() by *fac_()
00941   */
00942
00943   int simp2token(SBYTE *s)
00944   {
00945       SBYTE *to, *fill, *t, *v, *w, *s0 = s, *vv;
00946       int error = 0, n;
00947   /*
00948       Set substitutions
00949   */
00950       fill = to = s;
00951       while ( *s != TENDOFIT ) {
00952           if ( *s == LPARENTHESIS && s[1] == LPARENTHESIS ) {
00953               t = s+1; n = 0;
00954               while ( n >= 0 ) {
00955                   t++;
```

```
00956                 if ( *t == LPARENTHESIS ) n++;
00957                 else if ( *t == RPARENTHESIS ) n--;
00958             }
00959             if ( t[1] == RPARENTHESIS ) {
00960                 *t = TEMPTY; s++;
00961             }
00962             *fill++ = *s++;
00963         }
00964         else if ( *s == TEMPTY ) s++;
00965         else if ( *s == AM.facnum && ( fill > (s0+1) ) && fill[-2] == TDIVIDE
00966          && fill[-1] == TFUNCTION ) {
00967             fill[-2] = TMULTIPLY; *fill++ = (SBYTE)(AM.invfacnum); s++;
00968         }
00969         else if ( *s == AM.invfacnum && ( fill > (s0+1) ) && fill[-2] == TDIVIDE
00970          && fill[-1] == TFUNCTION ) {
00971             fill[-2] = TMULTIPLY; *fill++ = (SBYTE)(AM.facnum); s++;
00972         }
00973         else *fill++ = *s++;
00974     }
00975     *fill++ = TENDOFIT;
00976 /*
00977     Second round: try to locate 'simple' arguments and strip their brackets
00978
00979     We add (9-feb-2010) to the simple arguments integers of any size
00980 */
00981     fill = s = to;
00982     while ( *s != TENDOFIT ) {
00983         if ( *s == LPARENTHESIS ) {
00984             t = s; n = 0;
00985             while ( n >= 0 ) {
00986                 t++;
00987                 if ( *t == LPARENTHESIS ) n++;
00988                 else if ( *t == RPARENTHESIS ) n--;
00989             }
00990             if ( t[1] == TFUNCLOSE && s[1] != TWILDARG ) { /* Check for last argument in sum */
00991                 v = fill - 1; n = 0;
00992                 while ( n >= 0 && v >= to ) {
00993                     if ( *v == TFUNOPEN ) n--;
00994                     else if ( *v == TFUNCLOSE ) n++;
00995                     v--;
00996                 }
00997                 if ( v > to ) {
00998                     while ( *v >= 0 ) v--;
00999                     if ( *v == TFUNCTION ) { v++;
01000                         n = 0; while ( *v >= 0 && v < fill ) n = 128*n + *v++;
01001                         if ( n == AM.sumnum || n == AM.sumpnum ) {
01002                             *fill++ = *s++; continue;
01003                         }
01004                         else if ( ( n == (FIRSTBRACKET-FUNCTION)
01005                         || n == (TERMSINEXPR-FUNCTION)
01006                         || n == (SIZEOFFUNCTION-FUNCTION)
01007                         || n == (NUMFACTORS-FUNCTION)
01008                         || n == (GCDFUNCTION-FUNCTION)
01009                         || n == (DIVFUNCTION-FUNCTION)
01010                         || n == (REMFUNCTION-FUNCTION)
01011                         || n == (INVERSEFUNCTION-FUNCTION)
01012                         || n == (MULFUNCTION-FUNCTION)
01013                         || n == (FACTORIN-FUNCTION)
01014                         || n == (FIRSTTERM-FUNCTION)
01015                         || n == (CONTENTTERM-FUNCTION) )
01016                         && fill[-1] == TFUNOPEN ) {
01017                             v = s+1;
01018                             if ( *v == TEXPRESSION ) {
01019                                 v++;
01020                                 n = 0; while ( *v >= 0 ) n = 128*n + *v++;
01021                                 if ( v == t ) {
01022                                     *t = TEMPTY; s++;
01023                                 }
01024                             }
01025                         }
01026                     }
01027                 }
01028             }
01029             if ( ( fill > to )
01030              && ( ( fill[-1] == TFUNOPEN || fill[-1] == TCOMMA )
01031                && ( t[1] == TFUNCLOSE || t[1] == TCOMMA ) ) ) {
01032                 v = s + 1;
01033                 switch ( *v ) {
01034                     case TMINUS:
01035                         v++;
01036                         if ( *v == TVECTOR ) {
01037                             w = v+1; while ( *w >= 0 ) w++;
01038                             if ( w == t ) {
01039                                 *t = TEMPTY; s++;
01040                             }
01041                         }
01042                         else {
```

```
01043                           if ( *v == TNUMBER || *v == TNUMBER1 ) {
01044                             if ( BITSINWORD == 16 ) { ULONG x; WORD base;
01045                               base = ( *v == TNUMBER ) ? 100: 128;
01046                               vv = v+1; x = 0; while ( *vv >= 0 ) { x = x*base + *vv++; }
01047                               if ( ( vv != t ) || ( ( vv - v ) > 4 ) || ( x > (MAXPOSITIVE+1) ) )
01048                                   *fill++ = *s++;
01049                               else { *t = TEMPTY; s++; break; }
01050                             }
01051                             else if ( BITSINWORD == 32 ) { ULONG x; WORD base;
01052                               base = ( *v == TNUMBER ) ? 100: 128;
01053                               vv = v+1; x = 0; while ( *vv >= 0 ) { x = x*base + *vv++; }
01054                               if ( ( vv != t ) || ( ( vv - v ) > 6 ) || ( x > (MAXPOSITIVE+1) ) )
01055                                   *fill++ = *s++;
01056                               else { *t = TEMPTY; s++; break; }
01057                             }
01058                             else {
01059                               if ( ( v+2 == t ) || ( v+3 == t && v[2] >= 0 ) )
01060                                   { *t = TEMPTY; s++; break; }
01061                               else *fill++ = *s++;
01062                             }
01063                           }
01064                           else if ( *v == LPARENTHESIS && t[-1] == RPARENTHESIS ) {
01065                               w = v; n = 0;
01066                               while ( n >= 0 ) {
01067                                   w++;
01068                                   if ( *w == LPARENTHESIS ) n++;
01069                                   else if ( *w == RPARENTHESIS ) n--;
01070                               }
01071                               if ( w == ( t-1 ) ) { *t = TEMPTY; s++; }
01072                               else *fill++ = *s++;
01073                           }
01074                           else *fill++ = *s++;
01075                           break;
01076                       }
01077                       /* fall through */
01078                   case TSETNUM:
01079                       v++; while ( *v >= 0 ) v++;
01080                       goto tcommon;
01081                   case TSYMBOL:
01082                       if ( ( v[1] == COEFFSYMBOL || v[1] == NUMERATORSYMBOL
01083                       || v[1] == DENOMINATORSYMBOL ) && v[2] < 0 ) {
01084                           *fill++ = *s++; break;
01085                       }
01086                       /* fall through */
01087                   case TSET:
01088                   case TVECTOR:
01089                   case TINDEX:
01090                   case TFUNCTION:
01091                   case TDOLLAR:
01092                   case TDUBIOUS:
01093                   case TSGAMMA:
01094 tcommon:               v++; while ( *v >= 0 ) v++;
01095                       if ( v == t || ( v[0] == TWILDCARD && v+1 == t ) )
01096                       { *t = TEMPTY; s++; }
01097                       else *fill++ = *s++;
01098                       break;
01099                   case TGENINDEX:
01100                       v++;
01101                       if ( v == t ) { *t = TEMPTY; s++; }
01102                       else *fill++ = *s++;
01103                       break;
01104                   case TNUMBER:
01105                   case TNUMBER1:
01106                       if ( BITSINWORD == 16 ) { ULONG x; WORD base;
01107                           base = ( *v == TNUMBER ) ? 100: 128;
01108                           vv = v+1; x = 0; while ( *vv >= 0 ) { x = x*base + *vv++; }
01109                           if ( ( vv != t ) || ( ( vv - v ) > 4 ) || ( x > MAXPOSITIVE ) )
01110                               *fill++ = *s++;
01111                           else { *t = TEMPTY; s++; break; }
01112                       }
01113                       else if ( BITSINWORD == 32 ) { ULONG x; WORD base;
01114                           base = ( *v == TNUMBER ) ? 100: 128;
01115                           vv = v+1; x = 0; while ( *vv >= 0 ) { x = x*base + *vv++; }
01116                           if ( ( vv != t ) || ( ( vv - v ) > 6 ) || ( x > MAXPOSITIVE ) )
01117                               *fill++ = *s++;
01118                           else { *t = TEMPTY; s++; break; }
01119                       }
01120                       else {
01121                           if ( ( v+2 == t ) || ( v+3 == t && v[2] >= 0 ) )
01122                               { *t = TEMPTY; s++; break; }
01123                           else *fill++ = *s++;
01124                       }
01125                       break;
01126                   case TWILDARG:
01127                       v++; while ( *v >= 0 ) v++;
01128                       if ( v == t ) { *t = TEMPTY; s++; }
01129                       else *fill++ = *s++;
```

```
01130                       break;
01131                   case TEXPRESSION:
01132 /*
01133                       First establish that there is only the expression
01134                       in this argument.
01135 */
01136                       vv = s+1;
01137                       while ( vv < t ) {
01138                           if ( *vv != TEXPRESSION ) break;
01139                           vv++; while ( *vv >= 0 ) vv++;
01140                       }
01141                       if ( vv < t ) { *fill++ = *s++; break; }
01142 /*
01143                       Find the function
01144 */
01145                       w = fill-1; n = 0;
01146                       while ( n >= 0 && w >= to ) {
01147                           if ( *w == TFUNOPEN ) n--;
01148                           else if ( *w == TFUNCLOSE ) n++;
01149                           w--;
01150                       }
01151                       w--; while ( w > to && *w >= 0 ) w--;
01152                       if ( *w != TFUNCTION ) { *fill++ = *s++; break; }
01153                       w++; n = 0;
01154                       while ( *w >= 0 ) { n = 128*n + *w++; }
01155                       if ( n == GCDFUNCTION-FUNCTION
01156                       || n == DIVFUNCTION-FUNCTION
01157                       || n == REMFUNCTION-FUNCTION
01158                       || n == INVERSEFUNCTION-FUNCTION
01159                       || n == MULFUNCTION-FUNCTION ) {
01160                           *t = TEMPTY; s++;
01161                       }
01162                       else *fill++ = *s++;
01163                       break;
01164                   default: *fill++ = *s++; break;
01165               }
01166           }
01167           else *fill++ = *s++;
01168       }
01169       else if ( *s == TEMPTY ) s++;
01170       else *fill++ = *s++;
01171   }
01172   *fill++ = TENDOFIT;
01173   return(error);
01174 }
01175
01176 /*
01177       #] simp2token :
01178       #[ simp3atoken :
01179
01180       We hunt for denominators and exponents that seem hidden.
01181       For the denominators we have to recognize:
01182           /fun  /fun()  /fun^power  /fun()^power
01183           /set[n]  /set[n]()  /set[n]^power  /set[n]()^power
01184           /symbol^power (power no number or symbol wildcard)
01185           /dotpr^power (id)
01186           /#^power (id)
01187           /()  /()^power
01188           /vect /index /vect(anything) /vect(anything)^power
01189 */
01190
01191 int simp3atoken(SBYTE *s, int mode)
01192 {
01193   int error = 0, n, numexp = 0, denom, base, numprot, i;
01194   SBYTE *t, c;
01195   LONG num;
01196   WORD *prot;
01197   if ( mode == RHSIDE ) {
01198       prot = AC.ProtoType;
01199       numprot = prot[1] - SUBEXPSIZE;
01200       prot += SUBEXPSIZE;
01201   }
01202   else { prot = 0; numprot = 0; }
01203   while ( *s != TENDOFIT ) {
01204       denom = 1;
01205       if ( *s == TDIVIDE ) { denom = -1; s++; }
01206       c = *s;
01207       switch(c) {
01208           case TSYMBOL:
01209           case TNUMBER:
01210           case TNUMBER1:
01211               s++; while ( *s >= 0 ) s++; /* skip the object */
01212               if ( *s == TWILDCARD ) s++; /* and the possible wildcard */
01213 dosymbol:
01214               if ( *s != TPOWER ) continue;   /* No power -> done */
01215               s++;                            /* Skip the power */
01216               if ( *s == TMINUS ) s++;        /* negative: no difference here */
```

```
01217                 if ( *s == TNUMBER || *s == TNUMBER1 ) {
01218                     base = *s == TNUMBER ? 100: 128; /* NUMBER = base 100 */
01219                     s++;                /* Now we compose the power */
01220                     num = *s++;         /* If the number is way too large */
01221                     while ( *s >= 0 ) { /* it may look like not too big */
01222                         if ( num > MAXPOWER ) break;    /* Hence... */
01223                         num = base*num + *s++;
01224                     }
01225                     while ( *s >= 0 ) s++;  /* Finish the number if needed */
01226                     if ( *s == TPOWER ) goto doublepower;
01227                     if ( num <= MAXPOWER ) continue;    /* Simple case */
01228                 }
01229                 else if ( *s == TSYMBOL && c != TNUMBER && c != TNUMBER1 ) {
01230                     s++; n = 0; while ( *s >= 0 ) { n = 128*n + *s++; }
01231                     if ( *s == TWILDCARD ) { s++;
01232                         if ( *s == TPOWER ) goto doublepower;
01233                         continue; }
01234 /*
01235                     Now we have to test whether n happens to be a wildcard
01236 */
01237                     if ( mode == RHSIDE ) {
01238                         n += 2*MAXPOWER;
01239                         for ( i = 0; i < numprot; i += 4 ) {
01240                             if ( prot[i+2] == n && prot[i] == SYMTOSYM ) break;
01241                         }
01242                         if ( i < numprot ) break;
01243                     }
01244                     if ( *s == TPOWER ) goto doublepower;
01245                 }
01246                 numexp++;
01247                 break;
01248             case TINDEX:
01249                 s++; while ( *s >= 0 ) s++;
01250                 if ( *s == TWILDCARD ) s++;
01251 doindex:
01252                 if ( denom < 0 || *s == TPOWER ) {
01253                     MesPrint("&Index to a power or in denominator is illegal");
01254                     error = 1;
01255                 }
01256                 break;
01257             case TVECTOR:
01258                 s++; while ( *s >= 0 ) s++;
01259                 if ( *s == TWILDCARD ) s++;
01260 dovector:
01261                 if ( *s == TFUNOPEN ) {
01262                     s++; n = 1;
01263                     for(;;) {
01264                         if ( *s == TFUNOPEN ) {
01265                             n++;
01266                             MesPrint("&Illegal vector index");
01267                             error = 1;
01268                         }
01269                         else if ( *s == TFUNCLOSE ) {
01270                             n--;
01271                             if ( n <= 0 ) break;
01272                         }
01273                         s++;
01274                     }
01275                     s++;
01276                 }
01277                 else if ( *s == TDOT ) goto dodot;
01278                 if ( denom < 0 || *s == TPOWER || *s == TPOWER1 ) numexp++;
01279                 break;
01280             case TFUNCTION:
01281                 s++; while ( *s >= 0 ) s++;
01282                 if ( *s == TWILDCARD ) s++;
01283 dofunction:
01284                 t = s;
01285                 if ( *t == TFUNOPEN ) {
01286                     t++; n = 1;
01287                     for(;;) {
01288                         if ( *t == TFUNOPEN ) n++;
01289                         else if ( *t == TFUNCLOSE ) { if ( --n <= 0 ) break; }
01290                         t++;
01291                     }
01292                     t++; s++;
01293                 }
01294                 if ( denom < 0 || *t == TPOWER || *t == TPOWER1 ) numexp++;
01295                 break;
01296 #ifdef WITHFLOAT
01297             case TFLOAT:
01298                 s++; while ( *s >= 0 ) s++;
01299                 if ( denom < 0 || *s == TPOWER || *s == TPOWER1 ) numexp++;
01300                 break;
01301 #endif
01302             case TEXPRESSION:
01303                 s++; while ( *s >= 0 ) s++;
```

```
01304                 t = s;
01305                 if ( *t == TFUNOPEN ) {
01306                     t++; n = 1;
01307                     for(;;) {
01308                         if ( *t == TFUNOPEN ) n++;
01309                         else if ( *t == TFUNCLOSE ) { if ( --n <= 0 ) break; }
01310                         t++;
01311                     }
01312                     t++;
01313                 }
01314                 if ( *t == LBRACE ) {
01315                     t++; n = 1;
01316                     for(;;) {
01317                         if ( *t == LBRACE ) n++;
01318                         else if ( *t == RBRACE ) { if ( --n <= 0 ) break; }
01319                         t++;
01320                     }
01321                     t++;
01322                 }
01323                 if ( denom < 0 || ( ( *t == TPOWER || *t == TPOWER1 )
01324                 && t[1] == TMINUS ) ) numexp++;
01325                 break;
01326             case TDOLLAR:
01327                 s++; while ( *s >= 0 ) s++;
01328                 if ( denom < 0 || ( ( *s == TPOWER || *s == TPOWER1 )
01329                 && s[1] == TMINUS ) ) numexp++;
01330                 break;
01331             case LPARENTHESIS:
01332                 s++; n = 1; t = s;
01333                 for(;;) {
01334                     if ( *t == LPARENTHESIS ) n++;
01335                     else if ( *t == RPARENTHESIS ) { if ( --n <= 0 ) break; }
01336                     t++;
01337                 }
01338                 t++;
01339                 if ( denom > 0 && ( *t == TPOWER || *t == TPOWER1 ) ) {
01340                     if ( ( t[1] == TNUMBER || t[1] == TNUMBER1 ) && t[2] >= 0
01341                     && t[3] < 0 ) break;
01342                     numexp++;
01343                 }
01344                 else if ( denom < 0 && ( *t == TPOWER || *t == TPOWER1 ) ) {
01345                     if ( t[1] == TMINUS && ( t[2] == TNUMBER
01346                      || t[2] == TNUMBER1 ) && t[3] >= 0
01347                     && t[4] < 0 ) break;
01348                     numexp++;
01349                 }
01350                 else if ( denom < 0 || ( ( *t == TPOWER || *t == TPOWER1 )
01351                 && ( t[1] == TMINUS || t[1] == LPARENTHESIS ) ) ) numexp++;
01352                 break;
01353             case TSET:
01354                 s++; n = *s++; while ( *s >= 0 ) { n = 128*n + *s++; }
01355                 n = Sets[n].type;
01356                 switch ( n ) {
01357                     case CSYMBOL: goto dosymbol;
01358                     case CINDEX: goto doindex;
01359                     case CVECTOR: goto dovector;
01360                     case CFUNCTION: goto dofunction;
01361                     case CNUMBER: goto dosymbol;
01362                     case CMODEL:
01363                         if ( denom < 0 || *s == TPOWER ) {
01364                             MesPrint("&A model to a power or in denominator is illegal");
01365                             error = 1;
01366                         }
01367                         break;
01368                     case ANYTYPE:
01369                         if ( denom < 0 || *s == TPOWER ) {
01370                             MesPrint("&A set without type to a power or in denominator is illegal");
01371                             error = 1;
01372                         }
01373                         break;
01374                     default: error = 1; break;
01375                 }
01376                 break;
01377             case TDOT:
01378 dodot:          s++;
01379                 if ( *s == TVECTOR ) { s++; while ( *s >= 0 ) s++; }
01380                 else if ( *s == TSET ) {
01381                     s++; n = *s++; while ( *s >= 0 ) { n = 128*n + *s++; }
01382                     if ( Sets[n].type != CVECTOR ) {
01383                         MesPrint("&Set in dotproduct is not a set of vectors");
01384                         error = 1;
01385                     }
01386                     if ( *s == LBRACE ) {
01387                         s++; n = 1;
01388                         for(;;) {
01389                             if ( *s == LBRACE ) n++;
01390                             else if ( *s == RBRACE ) { if ( --n <= 0 ) break; }
```

```
01391                           s++;
01392                       }
01393                       s++;
01394                   }
01395                   else {
01396                       MesPrint("&Set without argument in dotproduct");
01397                       error = 1;
01398                   }
01399               }
01400               else if ( *s == TSETNUM ) {
01401                   s++; n = *s++; while ( *s >= 0 ) { n = 128*n + *s++; }
01402                   if ( *s != TVECTOR ) goto nodot;
01403                   s++; n = *s++; while ( *s >= 0 ) { n = 128*n + *s++; }
01404                   if ( Sets[n].type != CVECTOR ) {
01405                       MesPrint("&Set in dotproduct is not a set of vectors");
01406                       error = 1;
01407                   }
01408               }
01409               else {
01410 nodot:             MesPrint("&Illegal second element in dotproduct");
01411                   error = 1;
01412                   s++; while ( *s >= 0 ) s++;
01413               }
01414               goto dosymbol;
01415           default:
01416               s++; while ( *s >= 0 ) s++;
01417               break;
01418       }
01419   }
01420   if ( error ) return(-1);
01421   return(numexp);
01422 doublepower:
01423   MesPrint("&Dubious notation with object^power1^power2");
01424   return(-1);
01425 }
01426
01427 /*
01428       #] simp3atoken :
01429       #[ simp3btoken :
01430 */
01431
01432 int simp3btoken(SBYTE *s, int mode)
01433 {
01434   int error = 0, i, numprot, n, denom, base, inset = 0, dotp, sube = 0;
01435   SBYTE *t, c, *fill, *ff, *ss;
01436   LONG num;
01437   WORD *prot;
01438
01439   // Work in a temporary buffer, to avoid clashes between input and output
01440   SBYTE *tmptokens = (SBYTE*)Malloc1((AC.toptokens-AC.tokens)*sizeof(SBYTE), "simp3btoken scratch");
01441   SBYTE* tmptoptokens = tmptokens + (AC.toptokens-AC.tokens);
01442   fill = tmptokens;
01443
01444   if ( mode == RHSIDE ) {
01445       prot = AC.ProtoType;
01446       numprot = prot[1] - SUBEXPSIZE;
01447       prot += SUBEXPSIZE;
01448   }
01449   else { prot = 0; numprot = 0; }
01450   while ( *s == TEMPTY ) s++;
01451   while ( *s != TENDOFIT ) {
01452       // If we are near the end of the output buffer, we'd better reallocate
01453       // both tmptokens and AC.tokens (which must fit the final result).
01454       // Experimentally, fill can move by at most 5 per loop iteration, but
01455       // this buffer of 20 of might not be enough for all cases!
01456       while ( tmptoptokens - fill < 20 ) {
01457           LONG tmppos;
01458           tmppos = s - AC.tokens;
01459           DoubleBuffer((void**)&(AC.tokens), (void**)&(AC.toptokens), sizeof(SBYTE), "simp3btoken
    double");
01460           s = AC.tokens + tmppos;
01461           tmppos = fill - tmptokens;
01462           DoubleBuffer((void**)&tmptokens, (void**)&tmptoptokens, sizeof(SBYTE), "simp3btoken
    scratch double");
01463           fill = tmptokens + tmppos;
01464       }
01465       if ( *s == TEMPTY ) { s++; continue; }
01466       denom = 1;
01467       if ( *s == TDIVIDE ) { denom = -1; *fill++ = *s++; }
01468       ff = fill; ss = s; c = *s;
01469       if ( c == TSETNUM ) {
01470           *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01471           c = *s;
01472       }
01473       dotp = 0;
01474       switch(c) {
01475           case TSYMBOL:
```

```
01476             case TNUMBER:
01477             case TNUMBER1:
01478                 *fill++ = *s++;
01479                 while ( *s >= 0 ) *fill++ = *s++;
01480                 if ( *s == TWILDCARD ) *fill++ = *s++;
01481 dosymbol:
01482                 t = s;
01483                 if ( *s != TPOWER ) continue;
01484                 *fill++ = *s++;
01485                 if ( *s == TMINUS ) *fill++ = *s++;
01486                 if ( *s == TPLUS ) s++;
01487                 if ( *s == TSETNUM ) {
01488                     *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01489                     inset = 1;
01490                 }
01491                 else inset = 0;
01492                 if ( *s == TNUMBER || *s == TNUMBER1 ) {
01493                     base = *s == TNUMBER ? 100: 128;
01494                     *fill++ = *s++;
01495                     num = *s++; *fill++ = num;
01496                     while ( *s >= 0 ) {
01497                         if ( num > MAXPOWER ) break;
01498                         *fill++ = *s;
01499                         num = base*num + *s++;
01500                     }
01501                     while ( *s >= 0 ) *fill++ = *s++;
01502                     if ( num <= MAXPOWER ) continue;
01503                     goto putexp1;
01504                 }
01505                 else if ( *s == TSYMBOL && c != TNUMBER && c != TNUMBER1 ) {
01506                     *fill++ = *s++;
01507                     n = 0; while ( *s >= 0 ) { n = 128*n + *s; *fill++ = *s++; }
01508                     if ( *s == TWILDCARD ) { *fill++ = *s++;
01509                         if ( *s == TPOWER ) goto doublepower;
01510                     break; }
01511 /*
01512                     Now we have to test whether n happens to be a wildcard
01513 */
01514                     if ( mode == RHSIDE && inset == 0 ) {
01515 /*                      n += WILDOFFSET;*/
01516                         for ( i = 0; i < numprot; i += 4 ) {
01517                             if ( prot[i+2] == n && prot[i] == SYMTOSYM ) break;
01518                         }
01519                         if ( i < numprot ) break;
01520                     }
01521
01522 putexp1:            fill = ff;
01523                     if ( denom < 0 ) fill[-1] = TMULTIPLY;
01524                     *fill++ = TFUNCTION; *fill++ = (SBYTE)(AM.expnum); *fill++ = TFUNOPEN;
01525                     if ( dotp ) *fill++ = LPARENTHESIS;
01526                     while ( ss < t ) *fill++ = *ss++;
01527                     if ( dotp ) *fill++ = RPARENTHESIS;
01528                     *fill++ = TCOMMA;
01529                     ss++;                       /* Skip TPOWER */
01530                     if ( *ss == TMINUS ) { denom = -denom; ss++; }
01531                     if ( denom < 0 ) {
01532                         *fill++ = LPARENTHESIS;
01533                         *fill++ = TMINUS;
01534                         while ( ss < s ) *fill++ = *ss++;
01535                         *fill++ = RPARENTHESIS;
01536                     }
01537                     else {
01538                         while ( ss < s ) *fill++ = *ss++;
01539                     }
01540                     *fill++ = TFUNCLOSE;
01541                     if ( *ss == TPOWER ) goto doublepower;
01542                 }
01543                 else {  /* other objects can be composite */
01544                     goto dofunpower;
01545                 }
01546                 break;
01547             case TINDEX:
01548                 *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01549                 if ( *s == TWILDCARD ) *fill++ = *s++;
01550                 break;
01551             case TVECTOR:
01552                 *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01553                 if ( *s == TWILDCARD ) *fill++ = *s++;
01554 dovector:
01555                 if ( *s == TFUNOPEN ) {
01556                     while ( *s != TFUNCLOSE ) *fill++ = *s++;
01557                     *fill++ = *s++;
01558                 }
01559                 else if ( *s == TDOT ) goto dodot;
01560                 t = s;
01561                 goto dofunpower;
01562 #ifdef WITHFLOAT
```

```
01563             case TFLOAT:
01564                 *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01565                 t = s;
01566                 sube = 0;
01567                 goto dofunpower;
01568 #endif
01569             case TFUNCTION:
01570                 *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01571                 if ( *s == TWILDCARD ) *fill++ = *s++;
01572 dofunction:
01573                 t = s;
01574                 if ( *t == TFUNOPEN ) {
01575                     t++; n = 1;
01576                     for(;;) {
01577                         if ( *t == TFUNOPEN ) n++;
01578                         else if ( *t == TFUNCLOSE ) { if ( --n <= 0 ) break; }
01579                         t++;
01580                     }
01581                     t++; *fill++ = *s++;
01582                 }
01583                 sube = 0;
01584 dofunpower:
01585                 if ( *t == TPOWER || *t == TPOWER1 ) {
01586                     if ( sube ) {
01587                         if ( ( t[1] == TNUMBER || t[1] == TNUMBER1 )
01588                          && denom > 0 ) {
01589                             if ( t[2] >= 0 && t[3] < 0 ) { sube = 0; break; }
01590                         }
01591                         else if ( t[1] == TMINUS && denom < 0 &&
01592                         ( t[2] == TNUMBER || t[2] == TNUMBER1 ) ) {
01593                             if ( t[2] >= 0 && t[3] < 0 ) { sube = 0; break; }
01594                         }
01595                         sube = 0;
01596                     }
01597                     fill = ff;
01598                     *fill++ = TFUNCTION; *fill++ = (SBYTE)(AM.expnum); *fill++ = TFUNOPEN;
01599                     *fill++ = LPARENTHESIS;
01600                     while ( ss < t ) *fill++ = *ss++;
01601                     t++;
01602                     *fill++ = RPARENTHESIS; *fill++ = TCOMMA;
01603                     if ( *t == TMINUS ) { t++; denom = -denom; }
01604                     *fill++ = LPARENTHESIS;
01605                     if ( denom < 0 ) *fill++ = TMINUS;
01606                     if ( *t == LPARENTHESIS ) {
01607                         *fill++ = *t++; n = 0;
01608                         while ( n >= 0 ) {
01609                             if ( *t == LPARENTHESIS ) n++;
01610                             else if ( *t == RPARENTHESIS ) n--;
01611                             *fill++ = *t++;
01612                         }
01613                     }
01614                     else if ( *t == TFUNCTION || *t == TDUBIOUS ) {
01615                         *fill++ = *t++; while ( *t >= 0 ) *fill++ = *t++;
01616                         if ( *t == TWILDCARD ) *fill++ = *t++;
01617                         if ( *t == TFUNOPEN ) {
01618                             *fill++ = *t++; n = 0;
01619                             while ( n >= 0 ) {
01620                                 if ( *t == TFUNOPEN ) n++;
01621                                 else if ( *t == TFUNCLOSE ) n--;
01622                                 *fill++ = *t++;
01623                             }
01624                         }
01625                     }
01626                     else if ( *t == TSET ) {
01627                         *fill++ = *t++; n = 0;
01628                         while ( *t >= 0 ) { n = 128*n + *t; *fill++ = *t++; }
01629                         if ( *t == LBRACE ) {
01630                             if ( n < AM.NumFixedSets || Sets[n].type == CRANGE ) {
01631                                 MesPrint("&This type of usage of sets is not allowed");
01632                                 error = 1;
01633                             }
01634                             *fill++ = *t++; n = 0;
01635                             while ( n >= 0 ) {
01636                                 if ( *t == LBRACE ) n++;
01637                                 else if ( *t == RBRACE ) n--;
01638                                 *fill++ = *t++;
01639                             }
01640                         }
01641                     }
01642                     else if ( *t == TEXPRESSION ) {
01643                         *fill++ = *t++; while ( *t >= 0 ) *fill++ = *t++;
01644                         if ( *t == TFUNOPEN ) {
01645                             *fill++ = *t++; n = 0;
01646                             while ( n >= 0 ) {
01647                                 if ( *t == TFUNOPEN ) n++;
01648                                 else if ( *t == TFUNCLOSE ) n--;
01649                                 *fill++ = *t++;
```

```
01650                           }
01651                       }
01652                       if ( *t == LBRACE ) {
01653                           *fill++ = *t++; n = 0;
01654                           while ( n >= 0 ) {
01655                               if ( *t == LBRACE ) n++;
01656                               else if ( *t == RBRACE ) n--;
01657                               *fill++ = *t++;
01658                           }
01659                       }
01660                   }
01661                   else if ( *t == TVECTOR ) {
01662                       *fill++ = *t++; while ( *t >= 0 ) *fill++ = *t++;
01663                       if ( *t == TFUNOPEN ) {
01664                           *fill++ = *t++; n = 0;
01665                           while ( n >= 0 ) {
01666                               if ( *t == TFUNOPEN ) n++;
01667                               else if ( *t == TFUNCLOSE ) n--;
01668                               *fill++ = *t++;
01669                           }
01670                       }
01671                       else if ( *t == TDOT ) {
01672                           *fill++ = *t++;
01673                           if ( *t == TVECTOR || *t == TDUBIOUS ) {
01674                               *fill++ = *t++; while ( *t >= 0 ) *fill++ = *t++;
01675                           }
01676                           else if ( *t == TSET ) {
01677                               *fill++ = *t++; num = 0;
01678                               while ( *t >= 0 ) { num = 128*num + *t; *fill++ = *t++; }
01679                               if ( Sets[num].type != CVECTOR ) {
01680                                   MesPrint("&Illegal set type in dotproduct");
01681                                   error = 1;
01682                               }
01683                               if ( *t == LBRACE ) {
01684                                   *fill++ = *t++; n = 0;
01685                                   while ( n >= 0 ) {
01686                                       if ( *t == LBRACE ) n++;
01687                                       else if ( *t == RBRACE ) n--;
01688                                       *fill++ = *t++;
01689                                   }
01690                               }
01691                           }
01692                           else if ( *t == TSETNUM ) {
01693                               *fill++ = *t++;
01694                               while ( *t >= 0 ) { *fill++ = *t++; }
01695                               *fill++ = *t++;
01696                               while ( *t >= 0 ) { *fill++ = *t++; }
01697                           }
01698                       }
01699                       else {
01700                           MesPrint("&Illegal second element in dotproduct");
01701                           error = 1;
01702                       }
01703                   }
01704                   else {
01705                       *fill++ = *t++; while ( *t >= 0 ) *fill++ = *t++;
01706                       if ( *t == TWILDCARD ) *fill++ = *t++;
01707                   }
01708                   *fill++ = RPARENTHESIS; *fill++ = TFUNCLOSE;
01709                   if ( *t == TPOWER ) goto doublepower;
01710                   while ( fill > ff ) *--t = *--fill;
01711                   s = t;
01712               }
01713               else if ( denom < 0 ) {
01714                   fill = ff; ff[-1] = TMULTIPLY;
01715                   *fill++ = TFUNCTION; *fill++ = (SBYTE)(AM.denomnum);
01716                   *fill++ = TFUNOPEN; *fill++ = LPARENTHESIS;
01717                   while ( ss < t ) *fill++ = *ss++;
01718                   *fill++ = RPARENTHESIS; *fill++ = TFUNCLOSE;
01719                   while ( fill > ff ) *--t = *--fill;
01720                   s = t; denom = 1; sube = 0;
01721                   break;
01722               }
01723               sube = 0;
01724               break;
01725           case TEXPRESSION:
01726               *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01727               t = s;
01728               if ( *t == TFUNOPEN ) {
01729                   t++; n = 1;
01730                   for(;;) {
01731                       if ( *t == TFUNOPEN ) n++;
01732                       else if ( *t == TFUNCLOSE ) { if ( --n <= 0 ) break; }
01733                       t++;
01734                   }
01735                   t++;
01736               }
```

```
01737                 if ( *t == LBRACE ) {
01738                     t++; n = 1;
01739                     for(;;) {
01740                         if ( *t == LBRACE ) n++;
01741                         else if ( *t == RBRACE ) { if ( --n <= 0 ) break; }
01742                         t++;
01743                     }
01744                     t++;
01745                 }
01746                 if ( t > s || denom < 0 || ( ( *t == TPOWER || *t == TPOWER1 )
01747                 && t[1] == TMINUS ) ) goto dofunpower;
01748                 else goto dosymbol;
01749             case TDOLLAR:
01750                 *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01751                 goto dosymbol;
01752             case LPARENTHESIS:
01753                 *fill++ = *s++; n = 1; t = s;
01754                 for(;;) {
01755                     if ( *t == LPARENTHESIS ) n++;
01756                     else if ( *t == RPARENTHESIS ) { if ( --n <= 0 ) break; }
01757                     t++;
01758                 }
01759                 t++; sube = 1;
01760                 goto dofunpower;
01761             case TSET:
01762                 *fill++ = *s++; n = *s++; *fill++ = (SBYTE)n;
01763                 while ( *s >= 0 ) { *fill++ = *s; n = 128*n + *s++; }
01764                 n = Sets[n].type;
01765                 switch ( n ) {
01766                     case CSYMBOL: goto dosymbol;
01767                     case CINDEX: break;
01768                     case CVECTOR: goto dovector;
01769                     case CFUNCTION: goto dofunction;
01770                     case CNUMBER: goto dosymbol;
01771                     case CMODEL: break;
01772                     case ANYTYPE: break;
01773                     default: error = 1; break;
01774                 }
01775                 break;
01776             case TDOT:
01777 dodot:          *fill++ = *s++;
01778                 if ( *s == TVECTOR ) {
01779                     *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01780                 }
01781                 else if ( *s == TSET ) {
01782                     *fill++ = *s++; n = *s++; *fill++ = (SBYTE)n;
01783                     while ( *s >= 0 ) { *fill++ = *s; n = 128*n + *s++; }
01784                     if ( *s == LBRACE ) {
01785                         if ( n < AM.NumFixedSets || Sets[n].type == CRANGE ) {
01786                             MesPrint("&This type of usage of sets is not allowed");
01787                             error = 1;
01788                         }
01789                         *fill++ = *s++; n = 1;
01790                         for(;;) {
01791                             if ( *s == LBRACE ) n++;
01792                             else if ( *s == RBRACE ) { if ( --n <= 0 ) break; }
01793                             *fill++ = *s++;
01794                         }
01795                         *fill++ = *s++;
01796                     }
01797                     else {
01798                         MesPrint("&Set without argument in dotproduct");
01799                         error = 1;
01800                     }
01801                 }
01802                 else if ( *s == TSETNUM ) {
01803                     *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01804                     if ( *s != TVECTOR ) goto nodot;
01805                     *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01806                 }
01807                 else {
01808 nodot:              MesPrint("&Illegal second element in dotproduct");
01809                     error = 1;
01810                     *fill++ = *s++;
01811                     while ( *s >= 0 ) *fill++ = *s++;
01812                 }
01813                 dotp = 1;
01814                 goto dosymbol;
01815             default:
01816                 *fill++ = *s++;
01817                 while ( *s >= 0 ) *fill++ = *s++;
01818                 break;
01819         }
01820     }
01821     *fill = TENDOFIT;
01822     // Now copy the modified tokens back to the original buffer
01823     fill = tmptokens;
```

```
01824     s = AC.tokens;
01825     do {
01826         *s++ = *fill;
01827     } while ( *fill++ != TENDOFIT );
01828     M_free(tmptokens, "simp3btoken scratch");
01829     return(error);
01830 doublepower:;
01831     MesPrint("&Dubious notation with power of power");
01832     M_free(tmptokens, "simp3btoken scratch");
01833     return(-1);
01834 }
01835
01836 /*
01837         #] simp3btoken :
01838         #[ simp4token :
01839
01840         Deal with the set[n] objects in the RHS.
01841 */
01842
01843 int simp4token(SBYTE *s)
01844 {
01845     int error = 0, n, nsym, settype;
01846     WORD i, *w, *wstop, level;
01847     SBYTE *const s0 = s;
01848     SBYTE *fill = s, *s1, *s2, *s3, type, s1buf[10];
01849     SBYTE *tbuf = s, *t, *t1;
01850
01851     while ( *s != TENDOFIT ) {
01852         if ( *s != TSET ) {
01853             if ( *s == TEMPTY ) s++;
01854             else *fill++ = *s++;
01855             continue;
01856         }
01857         if ( fill >= (s0+1) && fill[-1] == TWILDCARD ) { *fill++ = *s++; continue; }
01858         if ( fill >= (s0+2) && fill[-1] == TNOT && fill[-2] == TWILDCARD ) { *fill++ = *s++; continue;
    }
01859         s1 = s++; n = 0; while ( *s >= 0 ) { n = 128*n + *s++; }
01860         i = Sets[n].type;
01861         if ( *s != LBRACE ) { while ( s1 < s ) *fill++ = *s1++; continue; }
01862         if ( n < AM.NumFixedSets || i == CRANGE ) {
01863             MesPrint("&It is not allowed to refer to individual elements of built in or ranged sets");
01864             error = 1;
01865         }
01866         s++;
01867         if ( *s != TSYMBOL && *s != TDOLLAR ) {
01868             MesPrint("&Set index in RHS is not a wildcard symbol or $-variable");
01869             error = 1;
01870             while ( s1 < s ) *fill++ = *s1++;
01871             continue;
01872         }
01873         settype = ( *s == TDOLLAR );
01874         s++; nsym = 0; s2 = s;
01875         while ( *s >= 0 ) nsym = 128*nsym + *s++;
01876         if ( *s != RBRACE ) {
01877             MesPrint("&Improper set argument in RHS");
01878             error = 1;
01879             while ( s1 < s ) *fill++ = *s1++;
01880             continue;
01881         }
01882         s++;
01883 /*
01884         Verify that nsym is a wildcard
01885 */
01886         if ( !settype ) {
01887             w = AC.ProtoType; wstop = w + w[1]; w += SUBEXPSIZE;
01888             while ( w < wstop ) {
01889                 if ( *w == SYMTOSYM && w[2] == nsym ) break;
01890                 w += w[1];
01891             }
01892             if ( w >= wstop ) {
01893 /*
01894                 It could still be a summation parameter!
01895 */
01896                 t = fill - 1;
01897                 while ( t >= tbuf ) {
01898                     if ( *t == TFUNCLOSE ) {
01899                         level = 1; t--;
01900                         while ( t >= tbuf ) {
01901                             if ( *t == TFUNCLOSE ) level++;
01902                             else if ( *t == TFUNOPEN ) {
01903                                 level--;
01904                                 if ( level == 0 ) break;
01905                             }
01906                             t--;
01907                         }
01908                     }
01909                     else if ( *t == RBRACE ) {
```

```
01910                       level = 1; t--;
01911                       while ( t >= tbuf ) {
01912                           if ( *t == RBRACE ) level++;
01913                           else if ( *t == LBRACE ) {
01914                               level--;
01915                               if ( level == 0 ) break;
01916                           }
01917                           t--;
01918                       }
01919                   }
01920                   else if ( *t == RPARENTHESIS ) {
01921                       level = 1; t--;
01922                       while ( t >= tbuf ) {
01923                           if ( *t == RPARENTHESIS ) level++;
01924                           else if ( *t == LPARENTHESIS ) {
01925                               level--;
01926                               if ( level == 0 ) break;
01927                           }
01928                           t--;
01929                       }
01930                   }
01931                   else if ( *t == TFUNOPEN ) {
01932                       t1 = t-1;
01933                       while ( *t1 > 0 && t1 > tbuf ) t1--;
01934                       if ( *t1 == TFUNCTION ) {
01935                           t1++; level = 0;
01936                           while ( *t1 > 0 ) level = level*128+*t1++;
01937                           if ( level == (SUMF1-FUNCTION)
01938                            || level == (SUMF2-FUNCTION) ) {
01939                              t1 = t + 1;
01940                              if ( *t1 == LPARENTHESIS ) t1++;
01941                              if ( *t1 == TSYMBOL ) {
01942                                if ( ( t1[1] == COEFFSYMBOL
01943                                || t1[1] == NUMERATORSYMBOL
01944                                || t1[1] == DENOMINATORSYMBOL )
01945                                && t1[2] < 0 ) {}
01946                                else {
01947                                  t1++; level = 0;
01948                                  while ( *t1 >= 0 && t1 < fill ) level = 128*level + *t1++;
01949                                  if ( level == nsym && t1 < fill ) {
01950                                      if ( t[1] == LPARENTHESIS
01951                                       && *t1 == RPARENTHESIS && t1[1] == TCOMMA ) break;
01952                                      if ( t[1] != LPARENTHESIS && *t1 == TCOMMA ) break;
01953                                  }
01954                                    }
01955                              }
01956                           }
01957                       }
01958                   }
01959                   t--;
01960               }
01961               if ( t < tbuf ) {
01962                   fill--;
01963                   MesPrint("&Set index in RHS is not a wildcard symbol");
01964                   error = 1;
01965                   while ( s1 < s ) *fill++ = *s1++;
01966                   continue;
01967               }
01968           }
01969       }
01970 /*
01971       Now replace by a set marker: TSETNUM,nsym,TYPE,setnumber
01972 */
01973       switch ( i ) {
01974           case CSYMBOL: type = TSYMBOL; break;
01975           case CINDEX: type = TINDEX; break;
01976           case CVECTOR: type = TVECTOR; break;
01977           case CFUNCTION: type = TFUNCTION; break;
01978           case CNUMBER: type = TNUMBER1; break;
01979           case CDUBIOUS: type = TDUBIOUS; break;
01980           default:
01981 /* INTERNAL_ERROR_EXCL_START */
01982               MesPrint("!>Unknown set type in simp4token");
01983               error = 1; type = CDUBIOUS; break;
01984 /* INTERNAL_ERROR_EXCL_STOP */
01985       }
01986       s3 = s1buf; s1++;
01987       while ( *s1 >= 0 ) *s3++ = *s1++;
01988       *s3 = -1; s1 = s1buf;
01989       if ( settype ) *fill++ = TSETDOL;
01990       else           *fill++ = TSETNUM;
01991       while ( *s2 >= 0 ) *fill++ = *s2++;
01992       *fill++ = type; while ( *s1 >= 0 ) *fill++ = *s1++;
01993   }
01994   *fill++ = TENDOFIT;
01995   return(error);
01996 }
```

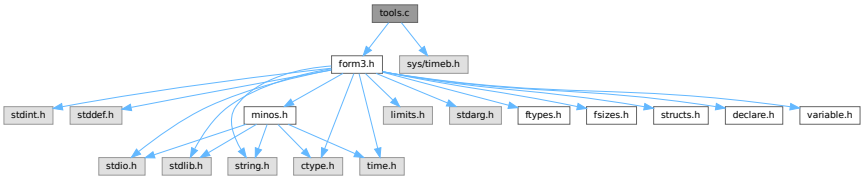
```
01997
01998 /*
01999         #] simp4token :
02000         #[ simp5token :
02001
02002     Making sure that first argument of sumfunction is not a wildcard already
02003 */
02004
02005 int simp5token(SBYTE *s, int mode)
02006 {
02007     int error = 0, n, type;
02008     WORD *w, *wstop;
02009     if ( mode == RHSIDE ) {
02010         while ( *s != TENDOFIT ) {
02011             if ( *s == TFUNCTION ) {
02012                 s++; n = 0; while ( *s >= 0 ) n = 128*n + *s++;
02013                 if ( n == AM.sumnum || n == AM.sumpnum ) {
02014                     if ( *s != TFUNOPEN ) continue;
02015                     s++;
02016                     if ( *s != TSYMBOL && *s != TINDEX ) continue;
02017                     type = *s++;
02018                     n = 0; while ( *s >= 0 ) n = 128*n + *s++;
02019                     if ( type == TINDEX ) n += AM.OffsetIndex;
02020                     if ( *s != TCOMMA ) continue;
02021                     w = AC.ProtoType;
02022                     wstop = w + w[1];
02023                     w += SUBEXPSIZE;
02024                     while ( w < wstop ) {
02025                         if ( w[2] == n ) {
02026                             if ( ( type == TSYMBOL && ( w[0] == SYMTOSYM
02027                             || w[0] == SYMTONUM || w[0] == SYMTOSUB ) ) || (
02028                             type == TINDEX && ( w[0] == INDTOIND
02029                             || w[0] == INDTOSUB ) ) ) {
02030                                 error = 1;
02031                                 MesPrint("&Parameter of sum function is already a wildcard");
02032                             }
02033                         }
02034                         w += w[1];
02035                     }
02036                 }
02037             }
02038             else s++;
02039         }
02040     }
02041     return(error);
02042 }
02043
02044 /*
02045         #] simp5token :
02046         #[ simp6token :
02047
02048     Making sure that factorized expressions are used properly
02049 */
02050
02051 int simp6token(SBYTE *tokens, int mode)
02052 {
02053 /*  EXPRESSIONS e = Expressions; */
02054     int error = 0, n;
02055     int level = 0, haveone = 0;
02056     SBYTE *s = tokens, *ss;
02057     LONG numterms;
02058     WORD funnum = 0;
02059     GETIDENTITY
02060     if ( mode == RHSIDE ) {
02061         while ( *s == TPLUS || *s == TMINUS ) s++;
02062         numterms = 1;
02063         while ( *s != TENDOFIT ) {
02064             if ( *s == LPARENTHESIS ) level++;
02065             else if ( *s == RPARENTHESIS ) level--;
02066             else if ( *s == TFUNOPEN ) level++;
02067             else if ( *s == TFUNCLOSE ) level--;
02068             else if ( ( *s == TPLUS || *s == TMINUS ) && level == 0 ) {
02069 /*
02070                 Special exception: x^-1 etc.
02071 */
02072                 if ( s[-1] != TPOWER && s[-1] != TPLUS && s[-1] != TMINUS ) {
02073                     numterms++;
02074                 }
02075             }
02076             else if ( *s == TEXPRESSION ) {
02077                 ss = s;
02078                 s++; n = 0; while ( *s >= 0 ) n = 128*n + *s++;
02079
02080                 if ( Expressions[n].status == STOREDEXPRESSION ) {
02081                     POSITION position;
02082 /*
02083 #ifdef WITHPTHREADS
```

```
02084                     RENUMBER renumber;
02085 #endif
02086 */
02087                     RENUMBER renumber;
02088
02089                     WORD TMproto[SUBEXPSIZE];
02090                     TMproto[0] = EXPRESSION;
02091                     TMproto[1] = SUBEXPSIZE;
02092                     TMproto[2] = n;
02093                     TMproto[3] = 1;
02094                     { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
02095                     AT.TMaddr = TMproto;
02096                     PUTZERO(position);
02097 /*
02098                     if ( (
02099 #ifdef WITHPTHREADS
02100                         renumber =
02101 #endif
02102                             GetTable(n,&position,0) ) == 0 )
02103 */
02104                     if ( ( renumber = GetTable(n,&position,0) ) == 0 )
02105                     {
02106 /* INTERNAL_ERROR_EXCL_START */
02107                         error = 1;
02108                         MesPrint("!>Problems getting information about stored expression %s(4)"
02109                         ,EXPRNAME(n));
02110 /* INTERNAL_ERROR_EXCL_STOP */
02111                     }
02112 /*
02113 #ifdef WITHPTHREADS
02114 */
02115                     if ( renumber->symb.lo != AN.dummyrenumlist )
02116                         M_free(renumber->symb.lo,"VarSpace");
02117                     M_free(renumber,"Renumber");
02118 /*
02119 #endif
02120 */
02121                 }
02122
02123                 if ( ( ( AS.Oldvflags[n] & ISFACTORIZED ) != 0 ) && *s != LBRACE ) {
02124                     if ( level == 0 ) {
02125                         haveone = 1;
02126                     }
02127                     else if ( error == 0 ) {
02128                         if ( ss[-1] != TFUNOPEN || funnum != NUMFACTORS-FUNCTION ) {
02129                             MesPrint("&Illegal use of factorized expression(s) in RHS");
02130                             error = 1;
02131                         }
02132                     }
02133                 }
02134                 continue;
02135             }
02136             else if ( *s == TFUNCTION ) {
02137                 s++; funnum = 0; while ( *s >= 0 ) funnum = 128*funnum + *s++;
02138                 continue;
02139             }
02140             s++;
02141         }
02142         if ( haveone ) {
02143             if ( numterms > 1 ) {
02144                 MesPrint("&Factorized expression in RHS in an expression of more than one term.");
02145                 error = 1;
02146             }
02147             else if ( AC.ToBeInFactors == 0 ) {
02148                 MesPrint("&Attempt to put a factorized expression inside an unfactorized
      expression.");
02149                 error = 1;
02150             }
02151         }
02152     }
02153     return(error);
02154 }
02155
02156 /*
02157         #] simp6token :
02158     #] Compiler :
02159 */
```

4.149 tools.c File Reference

```
#include "form3.h"
#include <sys/timeb.h>
```

Include dependency graph for tools.c:



Macros

- #define FILLVALUE 0
- #define **COPYFILEBUFSIZE** 40960L
- #define TERMMEMSTARTNUM 16
- #define TERMEXTRAWORDS 10
- #define NUMBERMEMSTARTNUM 16
- #define NUMBEREXTRAWORDS 10L
- #define DODOUBLE(x)
- #define DOEXPAND(x)

Functions

- UBYTE ∗ LoadInputFile (UBYTE ∗filename, int type)
- UBYTE ReadFromStream (STREAM ∗stream)
- UBYTE GetFromStream (STREAM ∗stream)
- UBYTE LookInStream (STREAM ∗stream)
- STREAM ∗ OpenStream (UBYTE ∗name, int type, int prevarmode, int raiselow)
- int LocateFile (UBYTE ∗∗name, int type)
- STREAM ∗ CloseStream (STREAM ∗stream)
- STREAM ∗ CreateStream (UBYTE ∗where)
- LONG GetStreamPosition (STREAM ∗stream)
- void PositionStream (STREAM ∗stream, LONG position)
- int ReverseStatements (STREAM ∗stream)
- void StartFiles (void)
- int OpenFile (char ∗name)
- int OpenAddFile (char ∗name)
- int ReOpenFile (char ∗name)
- int CreateFile (char ∗name)
- int CreateLogFile (char ∗name)
- void CloseFile (int handle)
- int CopyFile (char ∗source, char ∗dest)
- int CreateHandle (void)
- LONG ReadFile (int handle, UBYTE ∗buffer, LONG size)
- LONG ReadPosFile (PHEAD FILEHANDLE ∗fi, UBYTE ∗buffer, LONG size, POSITION ∗pos)
- LONG WriteFileToFile (int handle, UBYTE ∗buffer, LONG size)
- void SeekFile (int handle, POSITION ∗offset, int origin)
- LONG TellFile (int handle)
- void TELLFILE (int handle, POSITION ∗position)
- void FlushFile (int handle)
- int GetPosFile (int handle, fpos_t ∗pospointer)
- int SetPosFile (int handle, fpos_t ∗pospointer)

- void SynchFile (int handle)
- void TruncateFile (int handle)
- int GetChannel (char *name, int mode)
- int GetAppendChannel (char *name)
- int CloseChannel (char *name)
- void UpdateMaxSize (void)
- int StrCmp (UBYTE *s1, UBYTE *s2)
- int StrICmp (UBYTE *s1, UBYTE *s2)
- int StrHICmp (UBYTE *s1, UBYTE *s2)
- int StrICont (UBYTE *s1, UBYTE *s2)
- int CmpArray (WORD *t1, WORD *t2, WORD n)
- int ConWord (UBYTE *s1, UBYTE *s2)
- int StrLen (UBYTE *s)
- void NumToStr (UBYTE *s, LONG x)
- void WriteString (int type, UBYTE *str, int num)
- void WriteUnfinString (int type, UBYTE *str, int num)
- UBYTE * AddToString (UBYTE *outstring, UBYTE *extrastring, int par)
- UBYTE * strDup1 (UBYTE *instring, char *ifwrong)
- UBYTE * EndOfToken (UBYTE *s)
- UBYTE * ToToken (UBYTE *s)
- UBYTE * SkipField (UBYTE *s, int level)
- WORD ReadSnum (UBYTE **p)
- UBYTE * NumCopy (WORD y, UBYTE *to)
- char * LongCopy (LONG y, char *to)
- char * LongLongCopy (off_t *y, char *to)
- UBYTE * MakeDate (void)
- int set_in (UBYTE ch, set_of_char set)
- one_byte set_set (UBYTE ch, set_of_char set)
- one_byte set_del (UBYTE ch, set_of_char set)
- one_byte set_sub (set_of_char set, set_of_char set1, set_of_char set2)
- void iniTools (void)
- void * Malloc1 (LONG size, const char *messageifwrong)
- void M_free (void *x, const char *where)
- void M_check1 (void)
- void M_print (void)
- void TermMallocAddMemory (PHEAD0)
- void NumberMallocAddMemory (PHEAD0)
- void CacheNumberMallocAddMemory (PHEAD0)
- void * FromList (LIST *L)
- void * From0List (LIST *L)
- void * FromVarList (LIST *L)
- int DoubleList (void ***lijst, int *oldsize, int objectsize, char *nameoftype)
- int DoubleLList (void ***lijst, LONG *oldsize, int objectsize, char *nameoftype)
- void DoubleBuffer (void **start, void **stop, int size, char *text)
- void ExpandBuffer (void **buffer, LONG *oldsize, int type)
- LONG iexp (LONG x, int p)
- void ToGeneral (WORD *r, WORD *m, WORD par)
- int ToFast (WORD *r, WORD *m)
- WORD ToPolyFunGeneral (PHEAD WORD *term)
- int IsLikeVector (WORD *arg)
- int AreArgsEqual (WORD *arg1, WORD *arg2)
- int CompareArgs (WORD *arg1, WORD *arg2)
- int CompArg (WORD *s1, WORD *s2)
- LONG TimeWallClock (WORD par)
- LONG TimeChildren (WORD par)

- LONG TimeCPU (WORD par)
- int Crash (void)
- int TestTerm (WORD *term)
- int DistrN (int n, int *cpl, int ncpl, int *scratch)

Variables

- FILES ** filelist
- int numinfilelist = 0
- int filelistsize = 0
- WRITEFILE WriteFile = &WriteFileToFile

4.149.1 Detailed Description

Low level routines for many types of task. There are routines for manipulating the input system (streams and files) routines for string manipulation, the memory allocation interface, and the clock. The last is the most sensitive to ports. In the past nearly every port to another OS or computer gave trouble. Nowadays it is slightly better but the poor POSIX compliance of LINUX again gave problems for the multithreaded version.

Definition in file tools.c.

4.149.2 Macro Definition Documentation

4.149.2.1 FILLVALUE

```
#define FILLVALUE 0
```

Definition at line 49 of file tools.c.

4.149.2.2 TERMMEMSTARTNUM

```
#define TERMMEMSTARTNUM 16
```

Provides memory for one term (or one small polynomial) This means that the memory is limited to a buffer of size AM.MaxTer plus a few extra words. In parallel versions, each worker has its own memory pool.

The way we use the memory is by: term = TermMalloc(BHEAD0); and later we free it by TermFree(BHEAD term);

Layout: We have a list of available pointers to buffers: AT.TermMemHeap Its size is AT.TermMemMax We take from the top (indicated by AT.TermMemTop). When we run out of buffers we assign new ones (doubling the amount) and we have to extend the AT.TermMemHeap array. Important: There is no checking that the returned memory is legal, ie is memory that was handed out earlier.

Definition at line 2487 of file tools.c.

4.149.2.3 TERMEXTRAWORDS

```
#define TERMEXTRAWORDS 10
```

Definition at line 2488 of file tools.c.

4.149.2.4 NUMBERMEMSTARTNUM

```
#define NUMBERMEMSTARTNUM 16
```

Provides memory for one Long number This means that the memory is limited to a buffer of size AM.MaxTal In parallel versions, each worker has its own memory pool.

The way we use the memory is by: num = NumberMalloc(BHEAD0); Number = AT.NumberMemHeap[num]; and later we free it by NumberFree(BHEAD num);

Layout: We have a list of available pointers to buffers: AT.NumberMemHeap Its size is AT.NumberMemMax We take from the top (indicated by AT.NumberMemTop). When we run out of buffers we assign new ones (doubling the amount) and we have to extend the AT.NumberMemHeap array. Important: There is no checking on the returned memory!!!!

Definition at line 2587 of file tools.c.

4.149.2.5 NUMBEREXTRAWORDS

```
#define NUMBEREXTRAWORDS 10L
```

Definition at line 2588 of file tools.c.

4.149.2.6 DODOUBLE

```
#define DODOUBLE(
               x )
```

Value:

```
{ x *s, *t, *u; if ( *start ) { \
oldsize = *(x **)stop - *(x **)start; newsize = 2*oldsize; \
t = u = (x *)Malloc1(newsize*sizeof(x),text); s = *(x **)start; \
for ( i = 0; i < oldsize; i++ ) {*t++ = *s++;} M_free(*start,"double"); } \
else { newsize = 100; u = (x *)Malloc1(newsize*sizeof(x),text); } \
*start = (void *)u; *stop = (void *)(u+newsize); }
```

Definition at line 2914 of file tools.c.

4.149.2.7 DOEXPAND

```
#define DOEXPAND(
               x )
```

Value:

```
{ x *newbuffer, *t, *m;                                \
t = newbuffer = (x *)Malloc1((newsize+2)*type,"ExpandBuffer");      \
if ( *buffer ) { m = (x *)*buffer; i = *oldsize;                    \
    while ( --i >= 0 ) {*t++ = *m++;} M_free(*buffer,"ExpandBuffer"); \
} *buffer = newbuffer; *oldsize = newsize; }
```

Definition at line 2940 of file tools.c.

4.149.3 Function Documentation

4.149.3.1 LoadInputFile()

```
UBYTE * LoadInputFile (
            UBYTE * filename,
            int type )
```

Definition at line 112 of file tools.c.

4.149.3.2 ReadFromStream()

```
UBYTE ReadFromStream (
            STREAM * stream )
```

Definition at line 151 of file tools.c.

4.149.3.3 GetFromStream()

```
UBYTE GetFromStream (
            STREAM * stream )
```

Definition at line 286 of file tools.c.

4.149.3.4 LookInStream()

```
UBYTE LookInStream (
            STREAM * stream )
```

Definition at line 309 of file tools.c.

4.149.3.5 OpenStream()

```
STREAM * OpenStream (
            UBYTE * name,
            int type,
            int prevarmode,
            int raiselow )
```

Definition at line 321 of file tools.c.

4.149.3.6 LocateFile()

```
int LocateFile (
            UBYTE ** name,
            int type )
```

Definition at line 576 of file tools.c.

4.149.3.7 CloseStream()

```
STREAM * CloseStream (
            STREAM * stream )
```

Definition at line 663 of file tools.c.

4.149.3.8 CreateStream()

```
STREAM * CreateStream (
            UBYTE * where )
```

Definition at line 784 of file tools.c.

4.149.3.9 GetStreamPosition()

```
LONG GetStreamPosition (
            STREAM * stream )
```

Definition at line 815 of file tools.c.

4.149.3.10 PositionStream()

```
void PositionStream (
            STREAM * stream,
            LONG position )
```

Definition at line 825 of file tools.c.

4.149.3.11 ReverseStatements()

```
int ReverseStatements (
            STREAM * stream )
```

Definition at line 857 of file tools.c.

4.149.3.12 StartFiles()

```
void StartFiles (
            void  )
```

Definition at line 958 of file tools.c.

4.149.3.13 OpenFile()

```
int OpenFile (
            char * name )
```

Definition at line 985 of file tools.c.

4.149.3.14 OpenAddFile()

```
int OpenAddFile (
            char * name )
```

Definition at line 1004 of file tools.c.

4.149.3.15 ReOpenFile()

```
int ReOpenFile (
            char * name )
```

Definition at line 1025 of file tools.c.

4.149.3.16 CreateFile()

```
int CreateFile (
            char * name )
```

Definition at line 1045 of file tools.c.

4.149.3.17 CreateLogFile()

```
int CreateLogFile (
            char * name )
```

Definition at line 1062 of file tools.c.

4.149.3.18 CloseFile()

```
void CloseFile (
            int handle )
```

Definition at line 1080 of file tools.c.

4.149.3.19 CopyFile()

```
int CopyFile (
            char * source,
            char * dest )
```

Copies a file with name ∗source to a file named ∗dest. The involved files must not be open. Returns non-zero if an error occurred. Uses if possible the combined large and small sorting buffers as cache.

Definition at line 1104 of file tools.c.

Referenced by DoRecovery().

4.149.3.20 CreateHandle()

```
int CreateHandle (
            void  )
```

Definition at line 1166 of file tools.c.

4.149.3.21 ReadFile()

```
LONG ReadFile (
            int handle,
            UBYTE * buffer,
            LONG size )
```

Definition at line 1221 of file tools.c.

4.149.3.22 ReadPosFile()

```
LONG ReadPosFile (
            PHEAD FILEHANDLE * fi,
            UBYTE * buffer,
            LONG size,
            POSITION * pos )
```

Definition at line 1271 of file tools.c.

4.149.3.23 WriteFileToFile()

```
LONG WriteFileToFile (
            int handle,
            UBYTE * buffer,
            LONG size )
```

Definition at line 1337 of file tools.c.

4.149.3.24 SeekFile()

```
void SeekFile (
            int handle,
            POSITION * offset,
            int origin )
```

Definition at line 1375 of file tools.c.

4.149.3.25 TellFile()

```
LONG TellFile (
            int handle )
```

Definition at line 1400 of file tools.c.

4.149.3.26 TELLFILE()

```
void TELLFILE (
            int handle,
            POSITION * position )
```

Definition at line 1410 of file tools.c.

4.149.3.27 FlushFile()

```
void FlushFile (
            int handle )
```

Definition at line 1424 of file tools.c.

4.149.3.28 GetPosFile()

```
int GetPosFile (
            int handle,
            fpos_t * pospointer )
```

Definition at line 1438 of file tools.c.

4.149.3.29 SetPosFile()

```
int SetPosFile (
            int handle,
            fpos_t * pospointer )
```

Definition at line 1452 of file tools.c.

4.149.3.30 SynchFile()

```
void SynchFile (
            int handle )
```

Definition at line 1472 of file tools.c.

4.149.3.31 TruncateFile()

```
void TruncateFile (
            int handle )
```

Definition at line 1494 of file tools.c.

4.149.3.32 GetChannel()

```
int GetChannel (
            char * name,
            int mode )
```

Definition at line 1514 of file tools.c.

4.149.3.33 GetAppendChannel()

```
int GetAppendChannel (
            char * name )
```

Definition at line 1550 of file tools.c.

4.149.3.34 CloseChannel()

```
int CloseChannel (
            char * name )
```

Definition at line 1580 of file tools.c.

4.149.3.35 UpdateMaxSize()

```
void UpdateMaxSize (
            void  )
```

Definition at line 1614 of file tools.c.

4.149.3.36 StrCmp()

```
int StrCmp (
            UBYTE * s1,
            UBYTE * s2 )
```

Definition at line 1704 of file tools.c.

4.149.3.37 StrICmp()

```
int StrICmp (
            UBYTE * s1,
            UBYTE * s2 )
```

Definition at line 1715 of file tools.c.

4.149.3.38 StrHICmp()

```
int StrHICmp (
             UBYTE * s1,
             UBYTE * s2 )
```

Definition at line 1726 of file tools.c.

4.149.3.39 StrICont()

```
int StrICont (
             UBYTE * s1,
             UBYTE * s2 )
```

Definition at line 1737 of file tools.c.

4.149.3.40 CmpArray()

```
int CmpArray (
             WORD * t1,
             WORD * t2,
             WORD n )
```

Definition at line 1749 of file tools.c.

4.149.3.41 ConWord()

```
int ConWord (
             UBYTE * s1,
             UBYTE * s2 )
```

Definition at line 1763 of file tools.c.

4.149.3.42 StrLen()

```
int StrLen (
             UBYTE * s )
```

Definition at line 1775 of file tools.c.

4.149.3.43 NumToStr()

```
void NumToStr (
             UBYTE * s,
             LONG x )
```

Definition at line 1787 of file tools.c.

4.149.3.44 WriteString()

```
void WriteString (
            int type,
            UBYTE * str,
            int num )
```

Definition at line 1811 of file tools.c.

4.149.3.45 WriteUnfinString()

```
void WriteUnfinString (
            int type,
            UBYTE * str,
            int num )
```

Definition at line 1845 of file tools.c.

4.149.3.46 AddToString()

```
UBYTE * AddToString (
            UBYTE * outstring,
            UBYTE * extrastring,
            int par )
```

Definition at line 1870 of file tools.c.

4.149.3.47 strDup1()

```
UBYTE * strDup1 (
            UBYTE * instring,
            char * ifwrong )
```

Definition at line 1908 of file tools.c.

4.149.3.48 EndOfToken()

```
UBYTE * EndOfToken (
            UBYTE * s )
```

Skips over alphanumeric characters to find the end of the token.

Note

This function does not handle formal names (e.g., `[x+a]`). To handle them, use `SkipAName` instead.

Parameters

in	*s*	Pointer to a null-terminated input buffer.

Returns

Pointer to the first non-alphanumeric character (i.e., the character immediately after the token), or the null terminator if the token reaches the end of the string.

Definition at line 1934 of file tools.c.

4.149.3.49 ToToken()

```
UBYTE * ToToken (
            UBYTE * s )
```

Skips over non-alphanumeric characters to find the start of a token.

Note

This function does not handle formal names (e.g., `[x+a]`). To handle them, consider simply skipping whitespace, e.g., by using `SkipSpaces`.

Parameters

in	*s*	Pointer to a null-terminated input buffer.

Returns

Pointer to the first alphanumeric character (i.e., the start of the token), or the null terminator if none is found.

Definition at line 1957 of file tools.c.

4.149.3.50 SkipField()

```
UBYTE * SkipField (
            UBYTE * s,
            int level )
```

Skips from `s` to the end of a declaration field (e.g., `x`, `1+2*[x+a]`, or `f({x,[x+a]},1)`).

Parameters

in	*s*	Pointer to a null-terminated input buffer.
in	*level*	Number of parentheses that still have to be closed.

Returns

Pointer to the character after the field, which is either a comma at level 0 or the null terminator.

Definition at line 1978 of file tools.c.

4.149.3.51 ReadSnum()

```
WORD ReadSnum (
            UBYTE ** p )
```

Definition at line 2005 of file tools.c.

4.149.3.52 NumCopy()

```
UBYTE * NumCopy (
            WORD y,
            UBYTE * to )
```

Definition at line 2029 of file tools.c.

4.149.3.53 LongCopy()

```
char * LongCopy (
            LONG y,
            char * to )
```

Definition at line 2056 of file tools.c.

4.149.3.54 LongLongCopy()

```
char * LongLongCopy (
            off_t * y,
            char * to )
```

Definition at line 2083 of file tools.c.

4.149.3.55 MakeDate()

```
UBYTE * MakeDate (
            void  )
```

Definition at line 2119 of file tools.c.

4.149.3.56 set_in()

```
int set_in (
            UBYTE ch,
            set_of_char set )
```

Definition at line 2135 of file tools.c.

4.149.3.57 set_set()

```
one_byte set_set (
            UBYTE ch,
            set_of_char set )
```

Definition at line 2155 of file tools.c.

4.149.3.58 set_del()

```
one_byte set_del (
            UBYTE ch,
            set_of_char set )
```

Definition at line 2176 of file tools.c.

4.149.3.59 set_sub()

```
one_byte set_sub (
            set_of_char set,
            set_of_char set1,
            set_of_char set2 )
```

Definition at line 2198 of file tools.c.

4.149.3.60 iniTools()

```
void iniTools (
            void  )
```

Definition at line 2224 of file tools.c.

4.149.3.61 Malloc1()

```
void * Malloc1 (
            LONG size,
            const char * messageifwrong )
```

Definition at line 2245 of file tools.c.

4.149.3.62 M_free()

```
void M_free (
            void * x,
            const char * where )
```

Definition at line 2323 of file tools.c.

4.149.3.63 M_check1()

```
void M_check1 (
            void  )
```

Definition at line 2456 of file tools.c.

4.149.3.64 M_print()

```
void M_print (
            void  )
```

Definition at line 2457 of file tools.c.

4.149.3.65 TermMallocAddMemory()

```
void TermMallocAddMemory (
            PHEAD0  )
```

Definition at line 2490 of file tools.c.

4.149.3.66 NumberMallocAddMemory()

```
void NumberMallocAddMemory (
            PHEAD0  )
```

Definition at line 2594 of file tools.c.

4.149.3.67 CacheNumberMallocAddMemory()

```
void CacheNumberMallocAddMemory (
            PHEAD0  )
```

Definition at line 2668 of file tools.c.

4.149.3.68 FromList()

```
void * FromList (
            LIST * L )
```

Definition at line 2720 of file tools.c.

4.149.3.69 From0List()

```
void * From0List (
            LIST * L )
```

Definition at line 2746 of file tools.c.

4.149.3.70 FromVarList()

```
void * FromVarList (
            LIST * L )
```

Definition at line 2775 of file tools.c.

4.149.3.71 DoubleList()

```
int DoubleList (
            void *** lijst,
            int * oldsize,
            int objectsize,
            char * nameoftype )
```

Definition at line 2821 of file tools.c.

4.149.3.72 DoubleLList()

```
int DoubleLList (
            void *** lijst,
            LONG * oldsize,
            int objectsize,
            char * nameoftype )
```

Definition at line 2873 of file tools.c.

4.149.3.73 DoubleBuffer()

```
void DoubleBuffer (
            void ** start,
            void ** stop,
            int size,
            char * text )
```

Definition at line 2921 of file tools.c.

4.149.3.74 ExpandBuffer()

```
void ExpandBuffer (
            void ** buffer,
            LONG * oldsize,
            int type )
```

Definition at line 2946 of file tools.c.

4.149.3.75 iexp()

```
LONG iexp (
              LONG x,
              int p )
```

Definition at line 2970 of file tools.c.

4.149.3.76 ToGeneral()

```
void ToGeneral (
              WORD * r,
              WORD * m,
              WORD par )
```

Definition at line 3001 of file tools.c.

4.149.3.77 ToFast()

```
int ToFast (
              WORD * r,
              WORD * m )
```

Definition at line 3045 of file tools.c.

4.149.3.78 ToPolyFunGeneral()

```
WORD ToPolyFunGeneral (
              PHEAD WORD * term )
```

Definition at line 3098 of file tools.c.

4.149.3.79 IsLikeVector()

```
int IsLikeVector (
              WORD * arg )
```

Definition at line 3175 of file tools.c.

4.149.3.80 AreArgsEqual()

```
int AreArgsEqual (
              WORD * arg1,
              WORD * arg2 )
```

Definition at line 3203 of file tools.c.

4.149.3.81 CompareArgs()

```
int CompareArgs (
            WORD * arg1,
            WORD * arg2 )
```

Definition at line 3222 of file tools.c.

4.149.3.82 CompArg()

```
int CompArg (
            WORD * s1,
            WORD * s2 )
```

Definition at line 3253 of file tools.c.

4.149.3.83 TimeWallClock()

```
LONG TimeWallClock (
            WORD par )
```

Returns the wall-clock time.

Parameters

par	If zero, the wall-clock time will be reset to 0.

Returns

The wall-clock time in centiseconds.

Definition at line 3425 of file tools.c.

Referenced by DoCheckpoint(), DoRecovery(), find_Horner_MCTS(), optimize_expression_given_Horner(), optimize_greedy(), and WriteStats().

4.149.3.84 TimeChildren()

```
LONG TimeChildren (
            WORD par )
```

Definition at line 3481 of file tools.c.

4.149.3.85 TimeCPU()

```
LONG TimeCPU (
            WORD par )
```

Returns the CPU time.

Parameters

par	If zero, the CPU time will be reset to 0.

Returns

The CPU time in milliseconds.

Definition at line 3499 of file tools.c.

Referenced by PF_GetSlaveTimes(), PF_Processor(), and WriteStats().

4.149.3.86 Crash()

```
int Crash (
            void  )
```

Definition at line 3784 of file tools.c.

4.149.3.87 TestTerm()

```
int TestTerm (
            WORD * term )
```

Tests the consistency of the term. Returns 0 when the term is OK. Any nonzero value is trouble. In the current version the testing isn't 100% complete. For instance, we don't check the validity of the symbols nor do we check the range of their powers. Etc. This should be extended when the need is there.

Parameters

term	the term to be tested

Definition at line 3812 of file tools.c.

References TestTerm().

Referenced by TestTerm().

Here is the call graph for this function:

4.149.3.88 DistrN()

```
int DistrN (
            int n,
            int * cpl,
            int ncpl,
            int * scratch )
```

Definition at line 4028 of file tools.c.

4.149.4 Variable Documentation

4.149.4.1 filelist

```
FILES** filelist
```

Definition at line 65 of file tools.c.

4.149.4.2 numinfilelist

```
int numinfilelist = 0
```

Definition at line 66 of file tools.c.

4.149.4.3 filelistsize

```
int filelistsize = 0
```

Definition at line 67 of file tools.c.

4.149.4.4 WriteFile

```
WRITEFILE WriteFile = &WriteFileToFile
```

Definition at line 1361 of file tools.c.

4.150 tools.c

Go to the documentation of this file.

```
00001
00011 /* #[ License : */
00012 /*
00013  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00014  *   When using this file you are requested to refer to the publication
00015  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00016  *   This is considered a matter of courtesy as the development was paid
00017  *   for by FOM the Dutch physics granting agency and we would like to
00018  *   be able to track its scientific use to convince FOM of its value
00019  *   for the community.
00020  *
00021  *   This file is part of FORM.
00022  *
00023  *   FORM is free software: you can redistribute it and/or modify it under the
00024  *   terms of the GNU General Public License as published by the Free Software
00025  *   Foundation, either version 3 of the License, or (at your option) any later
00026  *   version.
00027  *
00028  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00029  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00030  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00031  *   details.
00032  *
00033  *   You should have received a copy of the GNU General Public License along
00034  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00035  */
00036 /* #] License : */
00037 /*
00038     #[ Includes :
00039     Note: TERMMALLOCDEBUG tests part of the TermMalloc and NumberMalloc
00040           system. To work properly it needs MEMORYMACROS in declare.h
00041           not to be defined to make sure that all calls will be diverted
00042           to the routines here.
00043 #define TERMMALLOCDEBUG
00044 #define FILLVALUE 126
00045 #define MALLOCDEBUGOUTPUT
00046 #define MALLOCDEBUG 1
00047 */
00048 #ifndef FILLVALUE
00049     #define FILLVALUE 0
00050 #endif
00051
00052 /*
00053     The enhanced malloc debugger, see comments in the beginning of the
00054     file mallocprotect.h
00055     MALLOCPROTECT == -1  -- protect left side, used block is left-aligned.
00056     MALLOCPROTECT == 0  -- protect both sides, used block is left-aligned;
00057     MALLOCPROTECT == 1  -- protect both sides, used block is right-aligned;
00058     ATTENTION! The macro MALLOCPROTECT must be defined
00059     BEFORE #include mallocprotect.h
00060 #define MALLOCPROTECT 1
00061 */
00062
00063 #include "form3.h"
00064
00065 FILES **filelist;
00066 int numinfilelist = 0;
00067 int filelistsize = 0;
00068 #ifdef MALLOCDEBUG
00069 #define BANNER (4*sizeof(LONG))
00070 void *malloclist[60000];
00071 LONG mallocsizes[60000];
00072 char *mallocstrings[60000];
00073 int nummalloclist = 0;
00074 #endif
00075
00076 #ifdef GPP
00077 extern "C" getdtablesize();
00078 #endif
00079
00080 #ifdef WITHSTATS
00081 LONG numwrites = 0;
00082 LONG numreads = 0;
00083 LONG numseeks = 0;
00084 LONG nummallocs = 0;
00085 LONG numfrees = 0;
00086 #endif
00087
00088 #ifdef MALLOCPROTECT
00089 #ifdef TRAPSIGNALS
00090 #error "MALLOCPROTECT":  undefine "TRAPSIGNALS" in unix.h first!
00091 #endif
```

```
00092 #include "mallocprotect.h"
00093
00094 #ifdef M_alloc
00095 #undef M_alloc
00096 #endif
00097
00098 #define M_alloc mprotectMalloc
00099
00100 #endif
00101
00102 #ifdef TERMMALLOCDEBUG
00103 WORD **DebugHeap1, **DebugHeap2;
00104 #endif
00105
00106 /*
00107     #] Includes :
00108     #[ Streams :
00109         #[ LoadInputFile :
00110 */
00111
00112 UBYTE *LoadInputFile(UBYTE *filename, int type)
00113 {
00114     int handle;
00115     LONG filesize;
00116     UBYTE *buffer, *name = filename;
00117     POSITION scrpos;
00118     handle = LocateFile(&name,type);
00119     if ( handle < 0 ) return(0);
00120     PUTZERO(scrpos);
00121     SeekFile(handle,&scrpos,SEEK_END);
00122     TELLFILE(handle,&scrpos);
00123     filesize = BASEPOSITION(scrpos);
00124     PUTZERO(scrpos);
00125     SeekFile(handle,&scrpos,SEEK_SET);
00126     buffer = (UBYTE *)Malloc1(filesize+2,"LoadInputFile");
00127     if ( ReadFile(handle,buffer,filesize) != filesize ) {
00128         Error1("Read error for file ",name);
00129         M_free(buffer,"LoadInputFile");
00130         if ( name != filename ) M_free(name,"FromLoadInputFile");
00131         CloseFile(handle);
00132         return(0);
00133     }
00134     CloseFile(handle);
00135     if ( type == PROCEDUREFILE || type == SETUPFILE ) {
00136         buffer[filesize] = '\n';
00137         buffer[filesize+1] = 0;
00138     }
00139     else {
00140         buffer[filesize] = 0;
00141     }
00142     if ( name != filename ) M_free(name,"FromLoadInputFile");
00143     return(buffer);
00144 }
00145
00146 /*
00147         #] LoadInputFile :
00148         #[ ReadFromStream :
00149 */
00150
00151 UBYTE ReadFromStream(STREAM *stream)
00152 {
00153     UBYTE c;
00154     POSITION scrpos;
00155 #ifdef WITHPIPE
00156     if ( stream->type == PIPESTREAM ) {
00157 #ifndef WITHMPI
00158         FILE *f;
00159         int cc;
00160         RWLOCKR(AM.handlelock);
00161         f = (FILE *)(filelist[stream->handle]);
00162         UNRWLOCK(AM.handlelock);
00163         cc = getc(f);
00164         if ( cc == EOF ) return(ENDOFSTREAM);
00165         c = (UBYTE)cc;
00166 #else
00167         if ( stream->pointer >= stream->top ) {
00168             /* The master reads the pipe and broadcasts it to the slaves. */
00169             LONG len;
00170             if ( PF.me == MASTER ) {
00171                 FILE *f;
00172                 UBYTE *p, *end;
00173                 RWLOCKR(AM.handlelock);
00174                 f = (FILE *)filelist[stream->handle];
00175                 UNRWLOCK(AM.handlelock);
00176                 p = stream->buffer;
00177                 end = stream->buffer + stream->buffersize;
00178                 while ( p < end ) {
```

```
00179                     int cc = getc(f);
00180                     if ( cc == EOF ) {
00181                         break;
00182                     }
00183                     *p++ = (UBYTE)cc;
00184                 }
00185                 len = p - stream->buffer;
00186                 PF_BroadcastNumber(len);
00187             }
00188             else {
00189                 len = PF_BroadcastNumber(0);
00190             }
00191             if ( len > 0 ) {
00192                 PF_Bcast(stream->buffer, len);
00193             }
00194             stream->pointer = stream->buffer;
00195             stream->inbuffer = len;
00196             stream->top = stream->buffer + stream->inbuffer;
00197             if ( stream->pointer == stream->top ) return ENDOFSTREAM;
00198         }
00199         c = (UBYTE)*stream->pointer++;
00200 #endif
00201         if ( stream->eqnum == 1 ) { stream->eqnum = 0; stream->linenumber++; }
00202         if ( c == LINEFEED ) stream->eqnum = 1;
00203         return(c);
00204     }
00205 #endif
00206 /*[14apr2004 mt]:*/
00207 #ifdef WITHEXTERNALCHANNEL
00208     if ( stream->type == EXTERNALCHANNELSTREAM ) {
00209         int cc;
00210         cc = getcFromExtChannel();
00211         /*[18may20006 mt]:*/
00212         /*if ( cc == EOF ) return(ENDOFSTREAM);*/
00213         if ( cc < 0 ){
00214             if( cc == EOF )
00215               return(ENDOFSTREAM);
00216             else{
00217                 Error0("No current external channel");
00218                 Terminate(-1);
00219             }
00220         }/*if ( cc < 0 )*/
00221         /*:[18may20006 mt]*/
00222         c = (UBYTE)cc;
00223         if ( stream->eqnum == 1 ) { stream->eqnum = 0; stream->linenumber++; }
00224         if ( c == LINEFEED ) stream->eqnum = 1;
00225         return(c);
00226     }
00227 #endif /*ifdef WITHEXTERNALCHANNEL*/
00228 /*:[14apr2004 mt]*/
00229     if ( stream->type == INPUTSTREAM ) {
00230         if ( stream->pointer < stream->top ) {
00231             c = *stream->pointer++;
00232         }
00233         else {
00234             if ( ReadFile(stream->handle,&c,1) != 1 ) {
00235                 return(ENDOFSTREAM);
00236             }
00237             if ( stream->fileposition == 0 ) {
00238                 if ( !stream->buffer ) {
00239                     stream->buffersize = 32;
00240                     stream->buffer = (UBYTE *)Mallocl(stream->buffersize,"input stream buffer");
00241                     stream->pointer = stream->top = stream->buffer;
00242                 }
00243                 else {
00244                     if ( stream->top - stream->buffer >= stream->buffersize ) {
00245                         LONG oldsize = stream->buffersize;
00246
      DoubleBuffer((void**)&stream->buffer,(void**)&stream->top,sizeof(UBYTE),"double input stream buffer");
00247                         stream->buffersize = stream->top - stream->buffer;
00248                         stream->pointer = stream->top = stream->buffer + oldsize;
00249                     }
00250                 }
00251                 *stream->pointer = c;
00252                 stream->pointer = ++stream->top;
00253                 stream->inbuffer++;
00254             }
00255         }
00256         if ( stream->eqnum == 1 ) { stream->eqnum = 0; stream->linenumber++; }
00257         if ( c == LINEFEED ) stream->eqnum = 1;
00258         return(c);
00259     }
00260     if ( stream->pointer >= stream->top ) {
00261         if ( stream->type != FILESTREAM ) return(ENDOFSTREAM);
00262         if ( stream->fileposition != stream->bufferposition+stream->inbuffer ) {
00263             stream->fileposition = stream->bufferposition+stream->inbuffer;
00264             SETBASEPOSITION(scrpos,stream->fileposition);
```

```
00265             SeekFile(stream->handle,&scrpos,SEEK_SET);
00266         }
00267         stream->bufferposition = stream->fileposition;
00268         stream->inbuffer = ReadFile(stream->handle,
00269                 stream->buffer,stream->buffersize);
00270         if ( stream->inbuffer <= 0 ) return(ENDOFSTREAM);
00271         stream->top = stream->buffer + stream->inbuffer;
00272         stream->pointer = stream->buffer;
00273         stream->fileposition = stream->bufferposition + stream->inbuffer;
00274     }
00275     if ( stream->eqnum == 1 ) { stream->eqnum = 0; stream->linenumber++; }
00276     c = *(stream->pointer)++;
00277     if ( c == LINEFEED ) stream->eqnum = 1;
00278     return(c);
00279 }
00280
00281 /*
00282         #] ReadFromStream :
00283         #[ GetFromStream :
00284 */
00285
00286 UBYTE GetFromStream(STREAM *stream)
00287 {
00288     UBYTE c1, c2;
00289     if ( stream->isnextchar > 0 ) {
00290         return(stream->nextchar[--stream->isnextchar]);
00291     }
00292     c1 = ReadFromStream(stream);
00293     if ( c1 == LINEFEED || c1 == CARRIAGERETURN ) {
00294         c2 = ReadFromStream(stream);
00295         if ( c2 == c1 || ( c2 != LINEFEED && c2 != CARRIAGERETURN ) ) {
00296             stream->isnextchar = 1;
00297             stream->nextchar[0] = c2;
00298         }
00299         return(LINEFEED);
00300     }
00301     else return(c1);
00302 }
00303
00304 /*
00305         #] GetFromStream :
00306         #[ LookInStream :
00307 */
00308
00309 UBYTE LookInStream(STREAM *stream)
00310 {
00311     UBYTE c = GetFromStream(stream);
00312     UngetFromStream(stream,c);
00313     return(c);
00314 }
00315
00316 /*
00317         #] LookInStream :
00318         #[ OpenStream :
00319 */
00320
00321 STREAM *OpenStream(UBYTE *name, int type, int prevarmode, int raiselow)
00322 {
00323     STREAM *stream;
00324     UBYTE *rhsofvariable, *s, *newname, c;
00325     POSITION scrpos;
00326     int handle, num;
00327     LONG filesize;
00328     switch ( type ) {
00329         case REVERSEFILESTREAM:
00330         case FILESTREAM:
00331 /*
00332             Notice that FILESTREAM is only used for text files:
00333             The #include files and the main input file (.frm)
00334             Hence we do not worry about files longer than 2 Gbytes.
00335 */
00336             newname = name;
00337             handle = LocateFile(&newname,-1);
00338             if ( handle < 0 ) return(0);
00339             PUTZERO(scrpos);
00340             SeekFile(handle,&scrpos,SEEK_END);
00341             TELLFILE(handle,&scrpos);
00342             filesize = BASEPOSITION(scrpos);
00343             PUTZERO(scrpos);
00344             SeekFile(handle,&scrpos,SEEK_SET);
00345             if ( filesize > AM.MaxStreamSize && type == FILESTREAM )
00346                     filesize = AM.MaxStreamSize;
00347             stream = CreateStream((UBYTE *)"filestream");
00348 /*
00349             The extra +1 in the Malloc1 is potentially needed in ReverseStatements!
00350 */
00351             stream->buffer = (UBYTE *)Malloc1(filesize+1,"name of input stream");
```

```
00352             stream->inbuffer = ReadFile(handle,stream->buffer,filesize);
00353             if ( type == REVERSEFILESTREAM ) {
00354                 if ( ReverseStatements(stream) ) {
00355                     M_free(stream->buffer,"name of input stream");
00356                     return(0);
00357                 }
00358             }
00359             stream->top = stream->buffer + stream->inbuffer;
00360             stream->pointer = stream->buffer;
00361             stream->handle = handle;
00362             stream->buffersize = filesize;
00363             stream->fileposition = stream->inbuffer;
00364             if ( newname != name ) stream->name = newname;
00365             else if ( name ) stream->name = strDup1(name,"name of input stream");
00366             else
00367                 stream->name = 0;
00368             stream->prevline = stream->linenumber = 1;
00369             stream->eqnum = 0;
00370             break;
00371         case PREVARSTREAM:
00372             if ( ( rhsofvariable = GetPreVar(name,WITHERROR) ) == 0 ) return(0);
00373             stream = CreateStream((UBYTE *)"var-stream");
00374             stream->buffer = stream->pointer = s = rhsofvariable;
00375             while ( *s ) s++;
00376             stream->top = s;
00377             stream->inbuffer = s - stream->buffer;
00378             stream->name = AC.CurrentStream->name;
00379             stream->linenumber = AC.CurrentStream->linenumber;
00380             stream->prevline = AC.CurrentStream->prevline;
00381             stream->eqnum = AC.CurrentStream->eqnum;
00382             stream->pname = strDup1(name,"stream->pname");
00383             stream->olddelay = AP.AllowDelay;
00384             s = stream->pname; while ( *s ) s++;
00385             while ( s[-1] == '+' || s[-1] == '-' ) s--;
00386             *s = 0;
00387             UnsetAllowDelay();
00388             break;
00389         case DOLLARSTREAM:
00390             if ( ( num = GetDollar(name) ) < 0 ) {
00391                 WORD numfac = 0;
00392 /*
00393                     Here we have to test first whether we have $x[1], $x[0]
00394                     or just an undefined $x.
00395 */
00396                 s = name; while ( *s && *s != '[' ) s++;
00397                 if ( *s == 0 ) return(0);
00398                 c = *s; *s = 0;
00399                 if ( ( num = GetDollar(name) ) < 0 ) return(0);
00400                 *s = c;
00401                 s++;
00402                 if ( *s == 0 || FG.cTable[*s] != 1 || *s == ']' ) {
00403                     MesPrint("@Illegal factor number for dollar variable");
00404                     return(0);
00405                 }
00406                 while ( *s && FG.cTable[*s] == 1 ) {
00407                     numfac = 10*numfac+*s++-'0';
00408                 }
00409                 if ( *s != ']' || s[1] != 0 ) {
00410                     MesPrint("@Illegal factor number for $ variable");
00411                     return(0);
00412                 }
00413                 stream = CreateStream((UBYTE *)"dollar-stream");
00414                 stream->buffer = stream->pointer = s = WriteDollarFactorToBuffer(num,numfac,1);
00415             }
00416             else {
00417                 stream = CreateStream((UBYTE *)"dollar-stream");
00418                 stream->buffer = stream->pointer = s = WriteDollarToBuffer(num,1);
00419             }
00420             while ( *s ) s++;
00421             stream->top = s;
00422             stream->inbuffer = s - stream->buffer;
00423             stream->name = AC.CurrentStream->name;
00424             stream->linenumber = AC.CurrentStream->linenumber;
00425             stream->prevline= AC.CurrentStream->prevline;
00426             stream->eqnum = AC.CurrentStream->eqnum;
00427             stream->pname = strDup1(name,"stream->pname");
00428             s = stream->pname; while ( *s ) s++;
00429             while ( s[-1] == '+' || s[-1] == '-' ) s--;
00430             *s = 0;
00431             /* We 'stole' the buffer. Later we can free it. */
00432             AO.DollarOutSizeBuffer = 0;
00433             AO.DollarOutBuffer = 0;
00434             AO.DollarInOutBuffer = 0;
00435             break;
00436         case PREREADSTREAM:
00437         case PREREADSTREAM2:
00438         case PREREADSTREAM3:
```

```
00439         case PRECALCSTREAM:
00440             stream = CreateStream((UBYTE *)"calculator");
00441             stream->buffer = stream->pointer = s = name;
00442             while ( *s ) s++;
00443             stream->top = s;
00444             stream->inbuffer = s - stream->buffer;
00445             stream->name = AC.CurrentStream->name;
00446             stream->linenumber = AC.CurrentStream->linenumber;
00447             stream->prevline = AC.CurrentStream->prevline;
00448             stream->eqnum = 0;
00449             break;
00450 #ifdef WITHPIPE
00451         case PIPESTREAM:
00452             stream = CreateStream((UBYTE *)"pipe");
00453 #ifndef WITHMPI
00454             {
00455                 FILE *f;
00456                 if ( ( f = popen((char *)name,"r") ) == 0 ) {
00457                     Error0("@Cannot create pipe");
00458                 }
00459                 stream->handle = CreateHandle();
00460                 RWLOCKW(AM.handlelock);
00461                 filelist[stream->handle] = (FILES *)f;
00462                 UNRWLOCK(AM.handlelock);
00463             }
00464             stream->buffer = stream->top = 0;
00465             stream->inbuffer = 0;
00466 #else
00467             {
00468                 /* Only the master opens the pipe. */
00469                 FILE *f;
00470                 if ( PF.me == MASTER ) {
00471                     f = popen((char *)name, "r");
00472                     PF_BroadcastNumber(f == 0);
00473                     if ( f == 0 ) Error0("@Cannot create pipe");
00474                 }
00475                 else {
00476                     if ( PF_BroadcastNumber(0) ) Error0("@Cannot create pipe");
00477                     f = (FILE *)123;  /* dummy */
00478                 }
00479                 stream->handle = CreateHandle();
00480                 RWLOCKW(AM.handlelock);
00481                 filelist[stream->handle] = (FILES *)f;
00482                 UNRWLOCK(AM.handlelock);
00483             }
00484             /* stream->buffer as a send/receive buffer. */
00485             stream->buffersize = AM.MaxStreamSize;
00486             stream->buffer = (UBYTE *)Malloc1(stream->buffersize, "pipe buffer");
00487             stream->inbuffer = 0;
00488             stream->top = stream->buffer;
00489             stream->pointer = stream->buffer;
00490 #endif
00491             stream->name = strDup1((UBYTE *)"pipe","pipe");
00492             stream->prevline = stream->linenumber = 1;
00493             stream->eqnum = 0;
00494             break;
00495 #endif
00496 /*[14apr2004 mt]:*/
00497 #ifdef WITHEXTERNALCHANNEL
00498         case EXTERNALCHANNELSTREAM:
00499             {/*Block*/
00500                 int n, *tmpn;
00501                 if( (n=getCurrentExternalChannel()) == 0 )
00502                     Error0("@No current external channel");
00503                 stream = CreateStream((UBYTE *)"externalchannel");
00504                 stream->handle = CreateHandle();
00505                 tmpn = (int *)Malloc1(sizeof(int),"external channel handle");
00506                 *tmpn = n;
00507                 RWLOCKW(AM.handlelock);
00508                 filelist[stream->handle] = (FILES *)tmpn;
00509                 UNRWLOCK(AM.handlelock);
00510             }/*Block*/
00511             stream->buffer = stream->top = 0;
00512             stream->inbuffer = 0;
00513             stream->name = strDup1((UBYTE *)"externalchannel","externalchannel");
00514             stream->prevline = stream->linenumber = 1;
00515             stream->eqnum = 0;
00516             break;
00517 #endif /*ifdef WITHEXTERNALCHANNEL*/
00518 /*:[14apr2004 mt]*/
00519         case INPUTSTREAM:
00520             /*
00521              * Assume that "name" stores a file descriptor (UNIX) or a FILE
00522              * pointer (Windows). We don't close it automatically on closing
00523              * the INPUTSTREAM stream (e.g., for stdin).
00524              */
00525             stream = CreateStream((UBYTE *)"input stream");
```

```
00526             stream->handle = CreateHandle();
00527             {
00528                 FILES *f = (FILES *)Malloc1(sizeof(int),"input stream handle");
00529                 /* NOTE: in both cases name=0 indicates stdin. */
00530 #ifdef UNIX
00531                 f->descriptor = (int)(ssize_t)name;
00532 #else
00533                 f = name ? (FILES *)name : stdin;
00534 #endif
00535                 RWLOCKW(AM.handlelock);
00536                 filelist[stream->handle] = f;
00537                 UNRWLOCK(AM.handlelock);
00538             }
00539             stream->buffer = stream->pointer = stream->top = 0;
00540             stream->inbuffer = 0;
00541             stream->name = strDup1((UBYTE *)(name ? "INPUT" : "STDIN"),"input stream name");
00542             stream->prevline = stream->linenumber = 1;
00543             stream->eqnum = 0;
00544             /*
00545              * fileposition == -1: default
00546              * fileposition ==  0: cache the input
00547              * See also: ReadFromStream, TryFileSetups
00548              */
00549             stream->fileposition = -1;
00550             break;
00551         default:
00552             return(0);
00553     }
00554     stream->bufferposition = 0;
00555     stream->isnextchar = 0;
00556     stream->type = type;
00557     stream->previousNoShowInput = AC.NoShowInput;
00558     stream->afterwards = raiselow;
00559     if ( AC.CurrentStream ) stream->previous = AC.CurrentStream - AC.Streams;
00560     else stream->previous = -1;
00561     stream->FoldName = 0;
00562     if ( prevarmode == 0 ) stream->prevars = -1;
00563     else if ( prevarmode > 0 ) stream->prevars = NumPre;
00564     else if ( prevarmode < 0 ) stream->prevars = -prevarmode-1;
00565     AC.CurrentStream = stream;
00566     if ( type == PREREADSTREAM || type == PREREADSTREAM3 || type == PRECALCSTREAM
00567         || type == DOLLARSTREAM ) AC.NoShowInput = 1;
00568     return(stream);
00569 }
00570
00571 /*
00572         #] OpenStream :
00573         #[ LocateFile :
00574 */
00575
00576 int LocateFile(UBYTE **name, int type)
00577 {
00578     int handle, namesize, i;
00579     UBYTE *s, *to, *u1, *u2, *newname, *indir;
00580     handle = OpenFile((char *)(*name));
00581     if ( handle >= 0 ) return(handle);
00582     if ( type == SETUPFILE && AM.SetupFile ) {
00583         handle = OpenFile((char *)(AM.SetupFile));
00584         if ( handle >= 0 ) return(handle);
00585         MesPrint("Could not open setup file %s",(char *)(AM.SetupFile));
00586     }
00587     namesize = 4; s = *name;
00588     while ( *s ) { s++; namesize++; }
00589     if ( type == SETUPFILE ) indir = AM.SetupDir;
00590     else indir = AM.IncDir;
00591     if ( indir ) {
00592
00593         s = indir; i = 0;
00594         while ( *s ) { s++; i++; }
00595         newname = (UBYTE *)Malloc1(namesize+i,"LocateFile");
00596         s = indir; to = newname;
00597         while ( *s ) *to++ = *s++;
00598         if ( to > newname && to[-1] != SEPARATOR ) *to++ = SEPARATOR;
00599         s = *name;
00600         while ( *s ) *to++ = *s++;
00601         *to = 0;
00602         handle = OpenFile((char *)newname);
00603         if ( handle >= 0 ) {
00604             *name = newname;
00605             return(handle);
00606         }
00607         M_free(newname,"LocateFile, incdir/file");
00608     }
00609     if ( type == SETUPFILE ) {
00610         handle = OpenFile(setupfilename);
00611         if ( handle >= 0 ) return(handle);
00612         s = (UBYTE *)getenv("FORMSETUP");
```

```
00613         if ( s ) {
00614             handle = OpenFile((char *)s);
00615             if ( handle >= 0 ) return(handle);
00616             MesPrint("Could not open setup file %s",s);
00617         }
00618     }
00619     if ( type != SETUPFILE && AM.Path ) {
00620         u1 = AM.Path;
00621         while ( *u1 ) {
00622             u2 = u1; i = 0;
00623 #ifdef WINDOWS
00624             while ( *u1 && *u1 != ';' ) {
00625                 u1++; i++;
00626             }
00627 #else
00628             while ( *u1 && *u1 != ':' ) {
00629                 if ( *u1 == '\\' ) u1++;
00630                 u1++; i++;
00631             }
00632 #endif
00633             newname = (UBYTE *)Malloc1(namesize+i,"LocateFile");
00634             s = u2; to = newname;
00635             while ( s < u1 ) {
00636 #ifndef WINDOWS
00637                 if ( *s == '\\' ) s++;
00638 #endif
00639                 *to++ = *s++;
00640             }
00641             if ( to > newname && to[-1] != SEPARATOR ) *to++ = SEPARATOR;
00642             s = *name;
00643             while ( *s ) *to++ = *s++;
00644             *to = 0;
00645             handle = OpenFile((char *)newname);
00646             if ( handle >= 0 ) {
00647                 *name = newname;
00648                 return(handle);
00649             }
00650             M_free(newname,"LocateFile Path/file");
00651             if ( *u1 ) u1++;
00652         }
00653     }
00654     if ( type != SETUPFILE && type >= -1 ) Error1("LocateFile: Cannot find file",*name);
00655     return(-1);
00656 }
00657
00658 /*
00659         #] LocateFile :
00660         #[ CloseStream :
00661 */
00662
00663 STREAM *CloseStream(STREAM *stream)
00664 {
00665     int newstr = stream->previous, sgn;
00666     UBYTE *t, numbuf[24];
00667     LONG x;
00668     if ( stream->FoldName ) {
00669         M_free(stream->FoldName,"stream->FoldName");
00670         stream->FoldName = 0;
00671     }
00672     if ( stream->type == FILESTREAM || stream->type == REVERSEFILESTREAM ) {
00673         CloseFile(stream->handle);
00674         if ( stream->buffer != 0 ) M_free(stream->buffer,"name of input stream");
00675         stream->buffer = 0;
00676     }
00677 #ifdef WITHPIPE
00678     else if ( stream->type == PIPESTREAM ) {
00679         RWLOCKW(AM.handlelock);
00680 #ifdef WITHMPI
00681         if ( PF.me == MASTER )
00682 #endif
00683         pclose((FILE *)(filelist[stream->handle]));
00684         filelist[stream->handle] = 0;
00685         numinfilelist--;
00686         UNRWLOCK(AM.handlelock);
00687 #ifdef WITHMPI
00688         if ( stream->buffer != 0 ) {
00689             M_free(stream->buffer, "pipe buffer");
00690             stream->buffer = 0;
00691         }
00692 #endif
00693     }
00694 #endif
00695 /*[14apr2004 mt]:*/
00696 #ifdef WITHEXTERNALCHANNEL
00697     else if ( stream->type == EXTERNALCHANNELSTREAM ) {
00698         int *tmpn;
00699         RWLOCKW(AM.handlelock);
```

```
00700         tmpn = (int *)(filelist[stream->handle]);
00701         filelist[stream->handle] = 0;
00702         numinfilelist--;
00703         UNRWLOCK(AM.handlelock);
00704         M_free(tmpn,"external channel handle");
00705     }
00706 #endif /*ifdef WITHEXTERNALCHANNEL*/
00707 /*:[14apr2004 mt]*/
00708     else if ( stream->type == INPUTSTREAM ) {
00709         FILES *f;
00710         RWLOCKW(AM.handlelock);
00711         f = filelist[stream->handle];
00712         filelist[stream->handle] = 0;
00713         numinfilelist--;
00714         UNRWLOCK(AM.handlelock);
00715         M_free(f,"input stream handle");
00716     }
00717     else if ( stream->type == PREVARSTREAM && (
00718     stream->afterwards == PRERAISEAFTER || stream->afterwards == PRELOWERAFTER ) ) {
00719         t = stream->buffer; x = 0; sgn = 1;
00720         while ( *t == '-' || *t == '+' ) {
00721             if ( *t == '-' ) sgn = -sgn;
00722             t++;
00723         }
00724         if ( FG.cTable[*t] == 1 ) {
00725             while ( *t && FG.cTable[*t] == 1 ) x = 10*x + *t++ - '0';
00726             if ( *t == 0 ) {
00727                 if ( stream->afterwards == PRERAISEAFTER ) x = sgn*x + 1;
00728                 else x = sgn*x - 1;
00729                 NumToStr(numbuf,x);
00730                 PutPreVar(stream->pname,numbuf,0,1);
00731             }
00732         }
00733     }
00734     else if ( stream->type == DOLLARSTREAM && (
00735     stream->afterwards == PRERAISEAFTER || stream->afterwards == PRELOWERAFTER ) ) {
00736         if ( stream->afterwards == PRERAISEAFTER ) x = 1;
00737         else x = -1;
00738         DollarRaiseLow(stream->pname,x);
00739         if ( stream->buffer ) M_free(stream->buffer,"stream->buffer");
00740         stream->buffer = 0;
00741     }
00742     else if ( stream->type == PRECALCSTREAM || stream->type == DOLLARSTREAM ) {
00743         if ( stream->buffer ) M_free(stream->buffer,"stream->buffer");
00744         stream->buffer = 0;
00745     }
00746     if ( stream->name && stream->type != PREVARSTREAM
00747     && stream->type != PREREADSTREAM && stream->type != PREREADSTREAM2 && stream->type !=
    PREREADSTREAM3
00748     && stream->type != PRECALCSTREAM && stream->type != DOLLARSTREAM ) {
00749         M_free(stream->name,"stream->name");
00750     }
00751     stream->name = 0;
00752 /*  if ( stream->type != FILESTREAM )  */
00753     AC.NoShowInput = stream->previousNoShowInput;
00754     stream->buffer = 0;       /* To make sure we will not reuse it */
00755     stream->pointer = 0;
00756 /*
00757     Look whether we have to pop preprocessor variables.
00758 */
00759     if ( stream->prevars >= 0 ) {
00760         while ( NumPre > stream->prevars ) {
00761             NumPre--;
00762             M_free(PreVar[NumPre].name,"PreVar[NumPre].name");
00763             PreVar[NumPre].name = PreVar[NumPre].value = 0;
00764         }
00765     }
00766     if ( stream->type == PREVARSTREAM ) {
00767         AP.AllowDelay = stream->olddelay;
00768         ClearMacro(stream->pname);
00769         M_free(stream->pname,"stream->pname");
00770     }
00771     else if ( stream->type == DOLLARSTREAM ) {
00772         M_free(stream->pname,"stream->pname");
00773     }
00774     AC.NumStreams--;
00775     if ( newstr >= 0 ) return(AC.Streams + newstr);
00776     else return(0);
00777 }
00778
00779 /*
00780         #] CloseStream :
00781         #[ CreateStream :
00782 */
00783
00784 STREAM *CreateStream(UBYTE *where)
00785 {
```

```
00786     STREAM *newstreams;
00787     int numnewstreams,i;
00788     int offset;
00789     if ( AC.NumStreams >= AC.MaxNumStreams ) {
00790         if ( AC.MaxNumStreams == 0 ) numnewstreams = 10;
00791         else                         numnewstreams = 2*AC.MaxNumStreams;
00792         newstreams = (STREAM *)Malloc1(sizeof(STREAM)*(numnewstreams+1),"CreateStream");
00793         if ( AC.MaxNumStreams > 0 ) {
00794             offset = AC.CurrentStream - AC.Streams;
00795             for ( i = 0; i < AC.MaxNumStreams; i++ ) {
00796                 newstreams[i] = AC.Streams[i];
00797             }
00798             AC.CurrentStream = newstreams + offset;
00799         }
00800         else newstreams[0].previous = -1;
00801         AC.MaxNumStreams = numnewstreams;
00802         if ( AC.Streams ) M_free(AC.Streams,(char *)where);
00803         AC.Streams = newstreams;
00804     }
00805     newstreams = AC.Streams+AC.NumStreams++;
00806     newstreams->name = 0;
00807     return(newstreams);
00808 }
00809
00810 /*
00811         #] CreateStream :
00812         #[ GetStreamPosition :
00813 */
00814
00815 LONG GetStreamPosition(STREAM *stream)
00816 {
00817     return(stream->bufferposition + ((LONG)stream->pointer-(LONG)stream->buffer));
00818 }
00819
00820 /*
00821         #] GetStreamPosition :
00822         #[ PositionStream :
00823 */
00824
00825 void PositionStream(STREAM *stream, LONG position)
00826 {
00827     POSITION scrpos;
00828     if ( position >= stream->bufferposition
00829     && position < stream->bufferposition + stream->inbuffer ) {
00830         stream->pointer = stream->buffer + (position-stream->bufferposition);
00831     }
00832     else if ( stream->type == FILESTREAM ) {
00833         SETBASEPOSITION(scrpos,position);
00834         SeekFile(stream->handle,&scrpos,SEEK_SET);
00835         stream->inbuffer = ReadFile(stream->handle,stream->buffer,stream->buffersize);
00836         stream->pointer = stream->buffer;
00837         stream->top = stream->buffer + stream->inbuffer;
00838         stream->bufferposition = position;
00839         stream->fileposition = position + stream->inbuffer;
00840         stream->isnextchar = 0;
00841     }
00842     else {
00843         Error0("Illegal position for stream");
00844         Terminate(-1);
00845     }
00846 }
00847
00848 /*
00849         #] PositionStream :
00850         #[ ReverseStatements :
00851
00852         Reverses the order of the statements in the buffer.
00853         We allocate an extra buffer and copy a bit to and from.
00854         Note that there are some nasties that cannot be resolved.
00855 */
00856
00857 int ReverseStatements(STREAM *stream)
00858 {
00859     UBYTE *spare = (UBYTE *)Malloc1((stream->inbuffer+1)*sizeof(UBYTE),"Reverse copy");
00860     UBYTE *top = stream->buffer + stream->inbuffer, *in, *s, *ss, *out;
00861     out = spare+stream->inbuffer+1;
00862     in = stream->buffer;
00863     while ( in < top ) {
00864         s = in;
00865         if ( *s == AP.ComChar ) {
00866 toeol:;
00867             for(;;) {
00868                 if ( s == top ) { *--out = '\n'; break; }
00869                 if ( *s == '\\' ) {
00870                     s++;
00871                     if ( s >= top ) { /* This is an error! */
00872 irrend:                 MesPrint("@Irregular end of reverse include file.");
```

```
00873                       return(1);
00874                   }
00875               }
00876               else if ( *s == '\n' ) {
00877                   s++; ss = s;
00878                   while ( ss > in ) *--out = *--ss;
00879                   in = s;
00880                   if ( out[0] == AP.ComChar && ss+6 < s && out[3] == '#' ) {
00881 /*
00882                       For folds we have to exchange begin and end
00883 */
00884                       if ( out[4] == '[' ) out[4] = ']';
00885                       else if ( out[4] == ']' ) out[4] = '[';
00886                   }
00887                   break;
00888               }
00889               s++;
00890           }
00891           continue;
00892       }
00893       while ( s < top && ( *s == ' ' || *s == '\t' ) ) s++;
00894       if ( *s == '#' ) {  /* preprocessor instruction */
00895           goto toeol;     /* read to end of line */
00896       }
00897       if ( *s == '.' ) {  /* end-of-module instruction */
00898           goto toeol;     /* read to end of line */
00899       }
00900 /*
00901       Here we have a regular statement. In principle we scan to ; and its \n
00902       but there are special cases.
00903       1: ; inside a string (in print ".....;";)
00904       2: multiple statements on one line.
00905       3: ; + commentary after some blanks.
00906       4: `var' can cause problems.....
00907 */
00908       while ( s < top ) {
00909           if ( *s == ';' ) {
00910               s++;
00911               while ( s < top && ( *s == ' ' || *s == '\t' ) ) s++;
00912               while ( s < top && *s == '\n' ) s++;
00913               if ( s >= top && s[-1] != '\n' ) *s++ = '\n';
00914               ss = s;
00915               while ( ss > in ) *--out = *--ss;
00916               in = s;
00917               break;
00918           }
00919           else if ( *s == '"' ) {
00920               s++;
00921               while ( s < top ) {
00922                   if ( *s == '"' ) break;
00923                   if ( *s == '\\' ) { s++; }
00924                   s++;
00925               }
00926               if ( s >= top ) goto irrend;
00927           }
00928           else if ( *s == '\\' ) {
00929               s++;
00930               if ( s >= top ) goto irrend;
00931           }
00932           s++;
00933       }
00934       if ( in < top ) { /* Like blank lines at the end */
00935           if ( s >= top && s[-1] != '\n' ) *s++ = '\n';
00936           ss = s;
00937           while ( ss > in ) *--out = *--ss;
00938           in = s;
00939       }
00940   }
00941   if ( out == spare ) stream->inbuffer++;
00942   if ( out > spare+1 ) {
00943       MesPrint("@Internal error in #reverseinclude instruction.");
00944       return(1);
00945   }
00946   memcpy((void *)(stream->buffer),(void *)out,(size_t)(stream->inbuffer*sizeof(UBYTE)));
00947   M_free(spare,"Reverse copy");
00948   return(0);
00949 }
00950
00951 /*
00952       #] ReverseStatements :
00953   #] Streams :
00954   #[ Files :
00955       #[ StartFiles :
00956 */
00957
00958 void StartFiles(void)
00959 {
```

```
00960     int i = CreateHandle();
00961     filelist[i] = Ustdout;
00962     AM.StdOut = i;
00963     AC.StoreHandle = -1;
00964     AC.LogHandle = -1;
00965 #ifndef WITHPTHREADS
00966     AR.Fscr[0].handle = -1;
00967     AR.Fscr[1].handle = -1;
00968     AR.Fscr[2].handle = -1;
00969     AR.FoStage4[0].handle = -1;
00970     AR.FoStage4[1].handle = -1;
00971     AR.infile = &(AR.Fscr[0]);
00972     AR.outfile = &(AR.Fscr[1]);
00973     AR.hidefile = &(AR.Fscr[2]);
00974     AR.StoreData.Handle = -1;
00975 #endif
00976     AC.Streams = 0;
00977     AC.MaxNumStreams = 0;
00978 }
00979
00980 /*
00981         #] StartFiles :
00982         #[ OpenFile :
00983 */
00984
00985 int OpenFile(char *name)
00986 {
00987     FILES *f;
00988     int i;
00989
00990     if ( ( f = Uopen(name,"rb") ) == 0 ) return(-1);
00991 /*  Usetbuf(f,0); */
00992     i = CreateHandle();
00993     RWLOCKW(AM.handlelock);
00994     filelist[i] = f;
00995     UNRWLOCK(AM.handlelock);
00996     return(i);
00997 }
00998
00999 /*
01000         #] OpenFile :
01001         #[ OpenAddFile :
01002 */
01003
01004 int OpenAddFile(char *name)
01005 {
01006     FILES *f;
01007     int i;
01008     POSITION scrpos;
01009     if ( ( f = Uopen(name,"a+b") ) == 0 ) return(-1);
01010 /*  Usetbuf(f,0); */
01011     i = CreateHandle();
01012     RWLOCKW(AM.handlelock);
01013     filelist[i] = f;
01014     UNRWLOCK(AM.handlelock);
01015     TELLFILE(i,&scrpos);
01016     SeekFile(i,&scrpos,SEEK_SET);
01017     return(i);
01018 }
01019
01020 /*
01021         #] OpenAddFile :
01022         #[ ReOpenFile :
01023 */
01024 /* UNFINISHED_FEATURE_EXCL_START */
01025 int ReOpenFile(char *name)
01026 {
01027     FILES *f;
01028     int i;
01029     POSITION scrpos;
01030     if ( ( f = Uopen(name,"r+b") ) == 0 ) return(-1);
01031     i = CreateHandle();
01032     RWLOCKW(AM.handlelock);
01033     filelist[i] = f;
01034     UNRWLOCK(AM.handlelock);
01035     TELLFILE(i,&scrpos);
01036     SeekFile(i,&scrpos,SEEK_SET);
01037     return(i);
01038 }
01039 /* UNFINISHED_FEATURE_EXCL_STOP */
01040 /*
01041         #] ReOpenFile :
01042         #[ CreateFile :
01043 */
01044
01045 int CreateFile(char *name)
01046 {
```

```
01047     FILES *f;
01048     int i;
01049     if ( ( f = Uopen(name,"w+b") ) == 0 ) return(-1);
01050     i = CreateHandle();
01051     RWLOCKW(AM.handlelock);
01052     filelist[i] = f;
01053     UNRWLOCK(AM.handlelock);
01054     return(i);
01055 }
01056
01057 /*
01058         #] CreateFile :
01059         #[ CreateLogFile :
01060 */
01061
01062 int CreateLogFile(char *name)
01063 {
01064     FILES *f;
01065     int i;
01066     if ( ( f = Uopen(name,"w+b") ) == 0 ) return(-1);
01067     Usetbuf(f,0);
01068     i = CreateHandle();
01069     RWLOCKW(AM.handlelock);
01070     filelist[i] = f;
01071     UNRWLOCK(AM.handlelock);
01072     return(i);
01073 }
01074
01075 /*
01076         #] CreateLogFile :
01077         #[ CloseFile :
01078 */
01079
01080 void CloseFile(int handle)
01081 {
01082     if ( handle >= 0 ) {
01083         FILES *f;   /* we need this variable to be thread-safe */
01084         RWLOCKW(AM.handlelock);
01085         f = filelist[handle];
01086         filelist[handle] = 0;
01087         numinfilelist--;
01088         UNRWLOCK(AM.handlelock);
01089         Uclose(f);
01090     }
01091 }
01092
01093 /*
01094         #] CloseFile :
01095         #[ CopyFile :
01096 */
01097
01103 /* UNFINISHED_FEATURE_EXCL_START */
01104 int CopyFile(char *source, char *dest)
01105 {
01106     #define COPYFILEBUFSIZE 40960L
01107     FILE *in, *out;
01108     size_t countin, countout, sumcount;
01109     char *buffer = NULL;
01110
01111     sumcount = (AM.S0->LargeSize+AM.S0->SmallEsize)*sizeof(WORD);
01112     if ( sumcount <= COPYFILEBUFSIZE ) {
01113         sumcount = COPYFILEBUFSIZE;
01114         buffer = (char*)Malloc1(sumcount, "file copy buffer");
01115     }
01116     else {
01117         buffer = (char *)(AM.S0->lBuffer);
01118     }
01119
01120     in = fopen(source, "rb");
01121     if ( in == NULL ) {
01122         perror("CopyFile: ");
01123         return(1);
01124     }
01125     out = fopen(dest, "wb");
01126     if ( out == NULL ) {
01127         fclose(in);
01128         perror("CopyFile: ");
01129         return(2);
01130     }
01131
01132     while ( !feof(in) ) {
01133         countin = fread(buffer, 1, sumcount, in);
01134         if ( countin != sumcount ) {
01135             if ( ferror(in) ) {
01136                 perror("CopyFile: ");
01137                 return(3);
01138             }
```

```
01139         }
01140         countout = fwrite(buffer, 1, countin, out);
01141         if ( countin != countout ) {
01142             perror("CopyFile: ");
01143             return(4);
01144         }
01145     }
01146 
01147     fclose(in);
01148     fclose(out);
01149     if ( sumcount <= COPYFILEBUFSIZE ) {
01150         M_free(buffer, "file copy buffer");
01151     }
01152     return(0);
01153 }
01154 /* UNFINISHED_FEATURE_EXCL_STOP */
01155 /*
01156         #] CopyFile :
01157         #[ CreateHandle :
01158 
01159         We need a lock here.
01160         Problem: the same lock is needed inside Malloc1 and M_free which
01161         is used in DoubleList when we use MALLOCDEBUG
01162 
01163         Conclusion: MALLOCDEBUG will have to be a bit unsafe
01164 */
01165 
01166 int CreateHandle(void)
01167 {
01168     int i, j;
01169 #ifndef MALLOCDEBUG
01170     RWLOCKW(AM.handlelock);
01171 #endif
01172     if ( filelistsize == 0 ) {
01173         filelistsize = 10;
01174         filelist = (FILES **)Malloc1(sizeof(FILES *)*filelistsize,"file handle");
01175         for ( j = 0; j < filelistsize; j++ ) filelist[j] = 0;
01176         numinfilelist = 1;
01177         i = 0;
01178     }
01179     else if ( numinfilelist >= filelistsize ) {
01180         void **fl = (void **)filelist;
01181         i = filelistsize;
01182         if ( DoubleList((void ***)(&fl),&filelistsize,(int)sizeof(FILES *),
01183             "list of open files") != 0 ) Terminate(-1);
01184         filelist = (FILES **)fl;
01185         for ( j = i; j < filelistsize; j++ ) filelist[j] = 0;
01186         numinfilelist = i + 1;
01187     }
01188     else {
01189         i = filelistsize;
01190         for ( j = 0; j < filelistsize; j++ ) {
01191             if ( filelist[j] == 0 ) { i = j; break; }
01192         }
01193         numinfilelist++;
01194     }
01195     filelist[i] = (FILES *)(filelist); /* Just for now to not get into problems */
01196 /*
01197     The next code is not needed when we use open.
01198     It may be needed when we use fopen.
01199     fopen is used in minos.c without this central administration.
01200 */
01201     if ( numinfilelist > MAX_OPEN_FILES ) {
01202 #ifndef MALLOCDEBUG
01203         UNRWLOCK(AM.handlelock);
01204 #endif
01205         MesPrint("More than %d open files",MAX_OPEN_FILES);
01206         Error0("System limit. This limit is not due to FORM!");
01207     }
01208     else {
01209 #ifndef MALLOCDEBUG
01210         UNRWLOCK(AM.handlelock);
01211 #endif
01212     }
01213     return(i);
01214 }
01215 
01216 /*
01217         #] CreateHandle :
01218         #[ ReadFile :
01219 */
01220 
01221 LONG ReadFile(int handle, UBYTE *buffer, LONG size)
01222 {
01223     LONG inbuf = 0, r;
01224     FILES *f;
01225     char *b;
```

```
01226     b = (char *)buffer;
01227     for(;;) {   /* Gotta do difficult because of VMS! */
01228         RWLOCKR(AM.handlelock);
01229         f = filelist[handle];
01230         UNRWLOCK(AM.handlelock);
01231 #ifdef WITHSTATS
01232         numreads++;
01233 #endif
01234         r = Uread(b,1,size,f);
01235         if ( r < 0 ) return(r);
01236         if ( r == 0 ) return(inbuf);
01237         inbuf += r;
01238         if ( r == size ) return(inbuf);
01239         if ( r > size ) return(-1);
01240         size -= r;
01241         b += r;
01242     }
01243 }
01244
01245 /*
01246         #] ReadFile :
01247         #[ ReadPosFile :
01248
01249         Gets words from a file(handle).
01250         First tries to get the information from the buffers.
01251         Reads a file at a position. Updates the position.
01252         Places a lock in the case of multithreading.
01253         Exists for multiple reading from the same file.
01254         size is the number of WORDs to read!!!!
01255
01256         We may need some strategy in the caching. This routine is used from
01257         GetOneTerm only. The problem is when it reads brackets and the
01258         brackets are read backwards. This is very uneconomical because
01259         each time it may read a large buffer.
01260         On the other hand, reading piece by piece in GetOneTerm takes
01261         much overhead as well.
01262         Two strategies come to mind:
01263         1: keep things as they are but limit the size of the buffers.
01264         2: have the position of 'pos' at about 1/3 of the buffer.
01265            this is of course guess work.
01266         Currently we have implemented the first method by creating the
01267         setup parameter threadscratchsize with the default value 100K.
01268         In the test program much bigger values gave a slower program.
01269 */
01270
01271 LONG ReadPosFile(PHEAD FILEHANDLE *fi, UBYTE *buffer, LONG size, POSITION *pos)
01272 {
01273     GETBIDENTITY
01274     LONG i, retval = 0;
01275     WORD *b = (WORD *)buffer, *t;
01276
01277     if ( fi->handle < 0 ) {
01278         fi->POfill = (WORD *)((UBYTE *)(fi->PObuffer) + BASEPOSITION(*pos));
01279         t = fi->POfill;
01280         while ( size > 0 && fi->POfill < fi->POfull ) { *b++ = *t++; size--; }
01281     }
01282     else {
01283         if ( ISLESSPOS(*pos,fi->POposition) || ISGEPOSINC(*pos,fi->POposition,
01284             ((UBYTE *)(fi->POfull)-(UBYTE *)(fi->PObuffer))) ) {
01285 /*
01286             The start is not inside the buffer. Fill the buffer.
01287 */
01288
01289             fi->POposition = *pos;
01290             LOCK(AS.inputslock);
01291             SeekFile(fi->handle,pos,SEEK_SET);
01292             retval = ReadFile(fi->handle,(UBYTE *)(fi->PObuffer),fi->POsize);
01293             UNLOCK(AS.inputslock);
01294             fi->POfull = fi->PObuffer+retval/sizeof(WORD);
01295             fi->POfill = fi->PObuffer;
01296             if ( fi != AR.hidefile ) AR.InInBuf = retval/sizeof(WORD);
01297             else                     AR.InHiBuf = retval/sizeof(WORD);
01298         }
01299         else {
01300             fi->POfill = (WORD *)((UBYTE *)(fi->PObuffer) + DIFBASE(*pos,fi->POposition));
01301         }
01302         if ( fi->POfill + size <= fi->POfull ) {
01303             t = fi->POfill;
01304             while ( size > 0 ) { *b++ = *t++; size--; }
01305         }
01306         else {
01307           for (;;) {
01308             i = fi->POfull - fi->POfill; t = fi->POfill;
01309             if ( i > size ) i = size;
01310             size -= i;
01311             while ( --i >= 0 ) *b++ = *t++;
01312             if ( size == 0 ) break;
```

```
01313             ADDPOS(fi->POposition,(UBYTE *)(fi->POfull)-(UBYTE *)(fi->PObuffer));
01314             LOCK(AS.inputslock);
01315             SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
01316             retval = ReadFile(fi->handle,(UBYTE *)(fi->PObuffer),fi->POsize);
01317             UNLOCK(AS.inputslock);
01318             fi->POfull = fi->PObuffer+retval/sizeof(WORD);
01319             fi->POfill = fi->PObuffer;
01320             if ( fi != AR.hidefile ) AR.InInBuf = retval/sizeof(WORD);
01321             else                     AR.InHiBuf = retval/sizeof(WORD);
01322             if ( retval == 0 ) { t = fi->POfill; break; }
01323           }
01324         }
01325     }
01326     retval = (UBYTE *)b - buffer;
01327     fi->POfill = t;
01328     ADDPOS(*pos,retval);
01329     return(retval);
01330 }
01331
01332 /*
01333         #] ReadPosFile :
01334         #[ WriteFile :
01335 */
01336
01337 LONG WriteFileToFile(int handle, UBYTE *buffer, LONG size)
01338 {
01339     FILES *f;
01340     LONG retval, totalwritten = 0, stilltowrite;
01341     RWLOCKR(AM.handlelock);
01342     f = filelist[handle];
01343     UNRWLOCK(AM.handlelock);
01344     while ( totalwritten < size ) {
01345         stilltowrite = size - totalwritten;
01346 #ifdef WITHSTATS
01347         numwrites++;
01348 #endif
01349         retval = Uwrite((char *)buffer+totalwritten,1,stilltowrite,f);
01350         if ( retval < 0 ) return(retval);
01351         if ( retval == 0 ) return(totalwritten);
01352         totalwritten += retval;
01353     }
01354 /*
01355 if ( handle == AC.LogHandle || handle == ERROROUT ) FlushFile(handle);
01356 */
01357     return(totalwritten);
01358 }
01359 #ifndef WITHMPI
01360 /*[17nov2005]:*/
01361 WRITEFILE WriteFile = &WriteFileToFile;
01362 /*
01363 LONG (*WriteFile)(int handle, UBYTE *buffer, LONG size) = &WriteFileToFile;
01364 */
01365 /*:[17nov2005]*/
01366 #else
01367 WRITEFILE WriteFile = &PF_WriteFileToFile;
01368 #endif
01369
01370 /*
01371         #] WriteFile :
01372         #[ SeekFile :
01373 */
01374
01375 void SeekFile(int handle, POSITION *offset, int origin)
01376 {
01377     FILES *f;
01378     RWLOCKR(AM.handlelock);
01379     f = filelist[handle];
01380     UNRWLOCK(AM.handlelock);
01381 #ifdef WITHSTATS
01382     numseeks++;
01383 #endif
01384     if ( origin == SEEK_SET ) {
01385         Useek(f,BASEPOSITION(*offset),origin);
01386         SETBASEPOSITION(*offset,(Utell(f)));
01387         return;
01388     }
01389     else if ( origin == SEEK_END ) {
01390         Useek(f,0,origin);
01391     }
01392     SETBASEPOSITION(*offset,(Utell(f)));
01393 }
01394
01395 /*
01396         #] SeekFile :
01397         #[ TellFile :
01398 */
01399
```

```
01400 LONG TellFile(int handle)
01401 {
01402     POSITION pos;
01403     TELLFILE(handle,&pos);
01404 #ifdef WITHSTATS
01405     numseeks++;
01406 #endif
01407     return(BASEPOSITION(pos));
01408 }
01409
01410 void TELLFILE(int handle, POSITION *position)
01411 {
01412     FILES *f;
01413     RWLOCKR(AM.handlelock);
01414     f = filelist[handle];
01415     UNRWLOCK(AM.handlelock);
01416     SETBASEPOSITION(*position,(Utell(f)));
01417 }
01418
01419 /*
01420         #] TellFile :
01421         #[ FlushFile :
01422 */
01423
01424 void FlushFile(int handle)
01425 {
01426     FILES *f;
01427     RWLOCKR(AM.handlelock);
01428     f = filelist[handle];
01429     UNRWLOCK(AM.handlelock);
01430     Uflush(f);
01431 }
01432
01433 /*
01434         #] FlushFile :
01435         #[ GetPosFile :
01436 */
01437
01438 int GetPosFile(int handle, fpos_t *pospointer)
01439 {
01440     FILES *f;
01441     RWLOCKR(AM.handlelock);
01442     f = filelist[handle];
01443     UNRWLOCK(AM.handlelock);
01444     return(Ugetpos(f,pospointer));
01445 }
01446
01447 /*
01448         #] GetPosFile :
01449         #[ SetPosFile :
01450 */
01451
01452 int SetPosFile(int handle, fpos_t *pospointer)
01453 {
01454     FILES *f;
01455     RWLOCKR(AM.handlelock);
01456     f = filelist[handle];
01457     UNRWLOCK(AM.handlelock);
01458     return(Usetpos(f,(fpos_t *)pospointer));
01459 }
01460
01461 /*
01462         #] SetPosFile :
01463         #[ SynchFile :
01464
01465         It may be that when we use many sort files at the same time there
01466         is a big traffic jam in the cache. This routine is experimental,
01467         just to see whether this improves the situation.
01468         It could also be that the internal disk of the Quad opteron norma
01469         is very slow.
01470 */
01471
01472 void SynchFile(int handle)
01473 {
01474     FILES *f;
01475     if ( handle >= 0 ) {
01476         RWLOCKR(AM.handlelock);
01477         f = filelist[handle];
01478         UNRWLOCK(AM.handlelock);
01479         Usync(f);
01480     }
01481 }
01482
01483 /*
01484         #] SynchFile :
01485         #[ TruncateFile :
01486
```

```
01487         It may be that when we use many sort files at the same time there
01488         is a big traffic jam in the cache. This routine is experimental,
01489         just to see whether this improves the situation.
01490         It could also be that the internal disk of the Quad opteron norma
01491         is very slow.
01492 */
01493
01494 void TruncateFile(int handle)
01495 {
01496     FILES *f;
01497     if ( handle >= 0 ) {
01498         RWLOCKR(AM.handlelock);
01499         f = filelist[handle];
01500         UNRWLOCK(AM.handlelock);
01501         Utruncate(f);
01502     }
01503 }
01504
01505 /*
01506         #] TruncateFile :
01507         #[ GetChannel :
01508
01509         Checks whether we have this file already. If so, we return its
01510         handle. If not and mode == 0, we open the file first and add it
01511         to the buffers.
01512 */
01513
01514 int GetChannel(char *name,int mode)
01515 {
01516     CHANNEL *ch;
01517     int i;
01518     FILES *f;
01519     for ( i = 0; i < NumOutputChannels; i++ ) {
01520         if ( channels[i].name == 0 ) continue;
01521         if ( StrCmp((UBYTE *)name,(UBYTE *)(channels[i].name)) == 0 ) return(channels[i].handle);
01522     }
01523     if ( mode == 1 ) {
01524         MesPrint("&File %s in print statement is not open",name);
01525         MesPrint("   You should open it first with a #write or #append instruction");
01526         return(-1);
01527     }
01528     for ( i = 0; i < NumOutputChannels; i++ ) {
01529         if ( channels[i].name == 0 ) break;
01530     }
01531     if ( i < NumOutputChannels ) { ch = &(channels[i]); }
01532     else { ch = (CHANNEL *)FromList(&AC.ChannelList); }
01533     ch->name = (char *)strDup1((UBYTE *)name,"name of channel");
01534     ch->handle = CreateFile(name);
01535     RWLOCKR(AM.handlelock);
01536     f = filelist[ch->handle];
01537     UNRWLOCK(AM.handlelock);
01538     Usetbuf(f,0);    /* We turn the buffer off!!!!!!*/
01539     return(ch->handle);
01540 }
01541
01542 /*
01543         #] GetChannel :
01544         #[ GetAppendChannel :
01545
01546         Checks whether we have this file already. If so, we return its
01547         handle. If not, we open the file first and add it to the buffers.
01548 */
01549
01550 int GetAppendChannel(char *name)
01551 {
01552     CHANNEL *ch;
01553     int i;
01554     FILES *f;
01555     for ( i = 0; i < NumOutputChannels; i++ ) {
01556         if ( channels[i].name == 0 ) continue;
01557         if ( StrCmp((UBYTE *)name,(UBYTE *)(channels[i].name)) == 0 ) return(channels[i].handle);
01558     }
01559     for ( i = 0; i < NumOutputChannels; i++ ) {
01560         if ( channels[i].name == 0 ) break;
01561     }
01562     if ( i < NumOutputChannels ) { ch = &(channels[i]); }
01563     else { ch = (CHANNEL *)FromList(&AC.ChannelList); }
01564     ch->name = (char *)strDup1((UBYTE *)name,"name of channel");
01565     ch->handle = OpenAddFile(name);
01566     RWLOCKR(AM.handlelock);
01567     f = filelist[ch->handle];
01568     UNRWLOCK(AM.handlelock);
01569     Usetbuf(f,0);    /* We turn the buffer off!!!!!!*/
01570     return(ch->handle);
01571 }
01572
01573 /*
```

```
01574         #] GetAppendChannel :
01575         #[ CloseChannel :
01576
01577         Checks whether we have this file already. If so, we close it.
01578 */
01579
01580 int CloseChannel(char *name)
01581 {
01582     int i;
01583     for ( i = 0; i < NumOutputChannels; i++ ) {
01584         if ( channels[i].name == 0 ) continue;
01585         if ( channels[i].name[0] == 0 ) continue;
01586         if ( StrCmp((UBYTE *)name,(UBYTE *)(channels[i].name)) == 0 ) {
01587             CloseFile(channels[i].handle);
01588             M_free(channels[i].name,"CloseChannel");
01589             channels[i].name = 0;
01590             return(0);
01591         }
01592     }
01593     return(0);
01594 }
01595
01596 /*
01597         #] CloseChannel :
01598         #[ UpdateMaxSize :
01599
01600         Updates the maximum size of the combined input/output/hide scratch
01601         files, the sort files and the .str file.
01602         The result becomes only visible with either
01603             ON totalsize;
01604             #: totalsize ON;
01605         or the -T in the command tail.
01606
01607         To be called, whenever a file is closed/removed or truncated to zero.
01608
01609         We have no provisions yet for expressions that remain inside the
01610         small or large buffer during the sort. The space they use there is
01611         currently ignored.
01612 */
01613
01614 void UpdateMaxSize(void)
01615 {
01616     POSITION position, sumsize;
01617     int i;
01618     FILEHANDLE *scr;
01619 #ifdef WITHMPI
01620     /* Currently, it works only on the master. The sort files on the slaves
01621      * are ignored. (TU 11 Oct 2011) */
01622     if ( PF.me != MASTER ) return;
01623 #endif
01624     PUTZERO(sumsize);
01625     if ( AM.PrintTotalSize ) {
01626 /*
01627         First the three scratch files
01628 */
01629 #ifdef WITHPTHREADS
01630     scr = AB[0]->R.Fscr;
01631 #else
01632     scr = AR.Fscr;
01633 #endif
01634     for ( i = 0; i <=2; i++ ) {
01635         if ( scr[i].handle < 0 ) {
01636             SETBASEPOSITION(position,(scr[i].POfull-scr[i].PObuffer)*sizeof(WORD));
01637         }
01638         else {
01639             position = scr[i].filesize;
01640         }
01641         ADD2POS(sumsize,position);
01642     }
01643 /*
01644         Now the sort file(s)
01645 */
01646 #ifdef WITHPTHREADS
01647     {
01648         int j;
01649         ALLPRIVATES *B;
01650         for ( j = 0; j < AM.totalnumberofthreads; j++ ) {
01651             B = AB[j];
01652             if ( AT.SS && AT.SS->file.handle >= 0 ) {
01653                 position = AT.SS->file.filesize;
01654 /*
01655 MLOCK(ErrorMessageLock);
01656 MesPrint("%d: %10p",j,&(AT.SS->file.filesize));
01657 MUNLOCK(ErrorMessageLock);
01658 */
01659                 ADD2POS(sumsize,position);
01660             }
```

```
01661             if ( AR.FoStage4[0].handle >= 0 ) {
01662                 position = AR.FoStage4[0].filesize;
01663                 ADD2POS(sumsize,position);
01664             }
01665         }
01666     }
01667 #else
01668     if ( AT.SS && AT.SS->file.handle >= 0 ) {
01669         position = AT.SS->file.filesize;
01670         ADD2POS(sumsize,position);
01671     }
01672     if ( AR.FoStage4[0].handle >= 0 ) {
01673         position = AR.FoStage4[0].filesize;
01674         ADD2POS(sumsize,position);
01675     }
01676 #endif
01677 /*
01678         And of course the str file.
01679 */
01680     ADD2POS(sumsize,AC.StoreFileSize);
01681 /*
01682         Finally the test whether it is bigger
01683 */
01684     if ( ISLESSPOS(AS.MaxExprSize,sumsize) ) {
01685 #ifdef WITHPTHREADS
01686         LOCK(AS.MaxExprSizeLock);
01687         if ( ISLESSPOS(AS.MaxExprSize,sumsize) ) AS.MaxExprSize = sumsize;
01688         UNLOCK(AS.MaxExprSizeLock);
01689 #else
01690         AS.MaxExprSize = sumsize;
01691 #endif
01692     }
01693     }
01694     return;
01695 }
01696
01697 /*
01698         #] UpdateMaxSize :
01699     #] Files :
01700     #[ Strings :
01701         #[ StrCmp :
01702 */
01703
01704 int StrCmp(UBYTE *s1, UBYTE *s2)
01705 {
01706     while ( *s1 && *s1 == *s2 ) { s1++; s2++; }
01707     return((int)*s1-(int)*s2);
01708 }
01709
01710 /*
01711         #] StrCmp :
01712         #[ StrICmp :
01713 */
01714
01715 int StrICmp(UBYTE *s1, UBYTE *s2)
01716 {
01717     while ( *s1 && tolower(*s1) == tolower(*s2) ) { s1++; s2++; }
01718     return((int)tolower(*s1)-(int)tolower(*s2));
01719 }
01720
01721 /*
01722         #] StrICmp :
01723         #[ StrHICmp :
01724 */
01725
01726 int StrHICmp(UBYTE *s1, UBYTE *s2)
01727 {
01728     while ( *s1 && tolower(*s1) == *s2 ) { s1++; s2++; }
01729     return((int)tolower(*s1)-(int)(*s2));
01730 }
01731
01732 /*
01733         #] StrHICmp :
01734         #[ StrICont :
01735 */
01736
01737 int StrICont(UBYTE *s1, UBYTE *s2)
01738 {
01739     while ( *s1 && tolower(*s1) == tolower(*s2) ) { s1++; s2++; }
01740     if ( *s1 == 0 ) return(0);
01741     return((int)tolower(*s1)-(int)tolower(*s2));
01742 }
01743
01744 /*
01745         #] StrICont :
01746         #[ CmpArray :
01747 */
```

```
01748 /* UNFINISHED_FEATURE_EXCL_START */
01749 int CmpArray(WORD *t1, WORD *t2, WORD n)
01750 {
01751     int i,x;
01752     for ( i = 0; i < n; i++ ) {
01753         if ( ( x = (int)(t1[i]-t2[i]) ) != 0 ) return(x);
01754     }
01755     return(0);
01756 }
01757 /* UNFINISHED_FEATURE_EXCL_STOP */
01758 /*
01759         #] CmpArray :
01760         #[ ConWord :
01761 */
01762
01763 int ConWord(UBYTE *s1, UBYTE *s2)
01764 {
01765     while ( *s1 && ( tolower(*s1) == tolower(*s2) ) ) { s1++; s2++; }
01766     if ( *s1 == 0 ) return(1);
01767     return(0);
01768 }
01769
01770 /*
01771         #] ConWord :
01772         #[ StrLen :
01773 */
01774
01775 int StrLen(UBYTE *s)
01776 {
01777     int i = 0;
01778     while ( *s ) { s++; i++; }
01779     return(i);
01780 }
01781
01782 /*
01783         #] StrLen :
01784         #[ NumToStr :
01785 */
01786
01787 void NumToStr(UBYTE *s, LONG x)
01788 {
01789     UBYTE *t, str[24];
01790     ULONG xx;
01791     t = str;
01792     if ( x < 0 ) { *s++ = '-'; xx = -x; }
01793     else xx = x;
01794     do {
01795         *t++ = xx % 10 + '0';
01796         xx /= 10;
01797     } while ( xx );
01798     while ( t > str ) *s++ = *--t;
01799     *s = 0;
01800 }
01801
01802 /*
01803         #] NumToStr :
01804         #[ WriteString :
01805
01806         Writes a characterstring to the various outputs.
01807         The action may depend on the flags involved.
01808         The type of output is given by type, the string by str and the
01809         number of characters in it by num
01810 */
01811 void WriteString(int type, UBYTE *str, int num)
01812 {
01813     int error = 0;
01814
01815     if ( num > 0 && str[num-1] == 0 ) { num--; }
01816     else if ( num <= 0 || str[num-1] != LINEFEED ) {
01817         AddLineFeed(str,num);
01818     }
01819     /*[15apr2004 mt]:*/
01820     if(type == EXTERNALCHANNELOUT){
01821         if(WriteFile(0,str,num) != num) error = 1;
01822     }else
01823     /*:[15apr2004 mt]*/
01824     if ( AM.silent == 0 || type == ERROROUT ) {
01825         if ( type == INPUTOUT ) {
01826             if ( !AM.FileOnlyFlag && WriteFile(AM.StdOut,(UBYTE *)"    ",4) != 4 ) error = 1;
01827             if ( AC.LogHandle >= 0 && WriteFile(AC.LogHandle,(UBYTE *)"    ",4) != 4 ) error = 1;
01828         }
01829         if ( !AM.FileOnlyFlag && WriteFile(AM.StdOut,str,num) != num ) error = 1;
01830         if ( AC.LogHandle >= 0 && WriteFile(AC.LogHandle,str,num) != num ) error = 1;
01831     }
01832     if ( error ) Terminate(-1);
01833 }
01834
```

```
01835 /*
01836         #] WriteString :
01837         #[ WriteUnfinString :
01838
01839         Writes a characterstring to the various outputs.
01840         The action may depend on the flags involved.
01841         The type of output is given by type, the string by str and the
01842         number of characters in it by num
01843 */
01844
01845 void WriteUnfinString(int type, UBYTE *str, int num)
01846 {
01847     int error = 0;
01848
01849     /*[15apr2004 mt]:*/
01850     if(type == EXTERNALCHANNELOUT){
01851         if(WriteFile(0,str,num) != num) error = 1;
01852     }else
01853     /*:[15apr2004 mt]*/
01854     if ( AM.silent == 0 || type == ERROROUT ) {
01855         if ( type == INPUTOUT ) {
01856             if ( !AM.FileOnlyFlag && WriteFile(AM.StdOut,(UBYTE *)"    ",4) != 4 ) error = 1;
01857             if ( AC.LogHandle >= 0 && WriteFile(AC.LogHandle,(UBYTE *)"    ",4) != 4 ) error = 1;
01858         }
01859         if ( !AM.FileOnlyFlag && WriteFile(AM.StdOut,str,num) != num ) error = 1;
01860         if ( AC.LogHandle >= 0 && WriteFile(AC.LogHandle,str,num) != num ) error = 1;
01861     }
01862     if ( error ) Terminate(-1);
01863 }
01864
01865 /*
01866         #] WriteUnfinString :
01867         #[ AddToString :
01868 */
01869
01870 UBYTE *AddToString(UBYTE *outstring, UBYTE *extrastring, int par)
01871 {
01872     UBYTE *s = extrastring, *t, *newstring;
01873     int n, nn;
01874     while ( *s ) { s++; }
01875     n = s-extrastring;
01876     if ( outstring == 0 ) {
01877         s = extrastring;
01878         t = outstring = (UBYTE *)Malloc1(n+1,"AddToString");
01879         NCOPY(t,s,n)
01880         *t++ = 0;
01881         return(outstring);
01882     }
01883     else {
01884         t = outstring;
01885         while ( *t ) t++;
01886         nn = t - outstring;
01887         t = newstring = (UBYTE *)Malloc1(n+nn+2,"AddToString");
01888         s = outstring;
01889         NCOPY(t,s,nn)
01890         if ( par == 1 ) *t++ = ',';
01891         s = extrastring;
01892         NCOPY(t,s,n)
01893         *t = 0;
01894         M_free(outstring,"AddToString");
01895         return(newstring);
01896     }
01897 }
01898
01899 /*
01900         #] AddToString :
01901         #[ strDup1 :
01902
01903         string duplication with message passing for Malloc1, allowing
01904         this routine to give a more detailed error message if there
01905         is not enough memory.
01906 */
01907
01908 UBYTE *strDup1(UBYTE *instring, char *ifwrong)
01909 {
01910     UBYTE *s = instring, *to;
01911     while ( *s ) s++;
01912     to = s = (UBYTE *)Malloc1((s-instring)+1,ifwrong);
01913     while ( *instring ) *to++ = *instring++;
01914     *to = 0;
01915     return(s);
01916 }
01917
01918 /*
01919         #] strDup1 :
01920         #[ EndOfToken :
01921 */
```

```
01922
01934 UBYTE *EndOfToken(UBYTE *s)
01935 {
01936     UBYTE c;
01937     while ( ( c = (UBYTE)(FG.cTable[*s]) ) == 0 || c == 1 ) s++;
01938     return(s);
01939 }
01940
01941 /*
01942         #] EndOfToken :
01943         #[ ToToken :
01944 */
01945
01957 UBYTE *ToToken(UBYTE *s)
01958 {
01959     UBYTE c;
01960     while ( *s && ( c = (UBYTE)(FG.cTable[*s]) ) != 0 && c != 1 ) s++;
01961     return(s);
01962 }
01963
01964 /*
01965         #] ToToken :
01966         #[ SkipField :
01967 */
01968
01978 UBYTE *SkipField(UBYTE *s, int level)
01979 {
01980     while ( *s ) {
01981         if ( *s == ',' && level == 0 ) return(s);
01982         if ( *s == '(' ) level++;
01983         else if ( *s == ')' ) { level--; if ( level < 0 ) level = 0; }
01984         else if ( *s == '[' ) {
01985             SKIPBRA1(s)
01986         }
01987         else if ( *s == '{' ) {
01988             SKIPBRA2(s)
01989         }
01990         s++;
01991     }
01992     return(s);
01993 }
01994
01995 /*
01996         #] SkipField :
01997         #[ ReadSnum :           WORD ReadSnum(p)
01998
01999         Reads a number that should fit in a word.
02000         The number should be unsigned and a negative return value
02001         indicates an irregularity.
02002
02003 */
02004
02005 WORD ReadSnum(UBYTE **p)
02006 {
02007     LONG x = 0;
02008     UBYTE *s;
02009     s = *p;
02010     if ( FG.cTable[*s] == 1 ) {
02011         do {
02012             x = ( x << 3 ) + ( x << 1 ) + ( *s++ - '0' );
02013             if ( x > MAXPOSITIVE ) return(-1);
02014         } while ( FG.cTable[*s] == 1 );
02015         *p = s;
02016         return((WORD)x);
02017     }
02018     else return(-1);
02019 }
02020
02021 /*
02022         #] ReadSnum :
02023         #[ NumCopy :
02024
02025     Adds the decimal representation of a number to a string.
02026
02027 */
02028
02029 UBYTE *NumCopy(WORD y, UBYTE *to)
02030 {
02031     UBYTE *s;
02032     WORD i = 0, j;
02033     UWORD x;
02034     if ( y < 0 ) {
02035         *to++ = '-';
02036     }
02037     x = WordAbs(y);
02038     s = to;
02039     do { *s++ = (UBYTE)((x % 10)+'0'); i++; } while ( ( x /= 10 ) != 0 );
```

```
02040     *s-- = '\0';
02041     j = ( i - 1 ) » 1;
02042     while ( j >= 0 ) {
02043         i = to[j]; to[j] = s[-j]; s[-j] = (UBYTE)i; j--;
02044     }
02045     return(s+1);
02046 }
02047
02048 /*
02049         #] NumCopy :
02050         #[ LongCopy :
02051
02052     Adds the decimal representation of a number to a string.
02053
02054 */
02055
02056 char *LongCopy(LONG y, char *to)
02057 {
02058     char *s;
02059     WORD i = 0, j;
02060     ULONG x;
02061     if ( y < 0 ) {
02062         *to++ = '-';
02063     }
02064     x = LongAbs(y);
02065     s = to;
02066     do { *s++ = (x % 10)+'0'; i++; } while ( ( x /= 10 ) != 0 );
02067     *s-- = '\0';
02068     j = ( i - 1 ) » 1;
02069     while ( j >= 0 ) {
02070         i = to[j]; to[j] = s[-j]; s[-j] = (char)i; j--;
02071     }
02072     return(s+1);
02073 }
02074
02075 /*
02076         #] LongCopy :
02077         #[ LongLongCopy :
02078
02079     Adds the decimal representation of a number to a string.
02080     Bugfix feb 2003. y was not pointer!
02081 */
02082
02083 char *LongLongCopy(off_t *y, char *to)
02084 {
02085     /*
02086      * This code fails to print the maximum negative value on systems with two's
02087      * complement. To fix this, we need the unsigned version of off_t with the
02088      * same size, but unfortunately it is undefined. On the other hand, if a
02089      * system is configured with a 64-bit off_t, in practice one never reaches
02090      * 2^63 ~ 10^18 as of 2016. If one really reach such a big number, then it
02091      * would be the time to move on a 128-bit off_t.
02092      */
02093     off_t x = *y;
02094     char *s;
02095     WORD i = 0, j;
02096     if ( x < 0 ) { x = -x; *to++ = '-'; }
02097     s = to;
02098     do { *s++ = (x % 10)+'0'; i++; } while ( ( x /= 10 ) != 0 );
02099     *s-- = '\0';
02100     j = ( i - 1 ) » 1;
02101     while ( j >= 0 ) {
02102         i = to[j]; to[j] = s[-j]; s[-j] = (char)i; j--;
02103     }
02104     return(s+1);
02105 }
02106
02107 /*
02108         #] LongLongCopy :
02109         #[ MakeDate :
02110
02111         Routine produces a string with the date and time of the run
02112 */
02113
02114 #if defined(ANSI) || defined(mBSD)
02115 #else
02116 static char notime[] = "";
02117 #endif
02118
02119 UBYTE *MakeDate(void)
02120 {
02121 #if defined(ANSI) || defined(mBSD)
02122     time_t tp;
02123     time(&tp);
02124     return((UBYTE *)ctime(&tp));
02125 #else
02126     return((UBYTE *)notime);
```

```
02127 #endif
02128 }
02129
02130 /*
02131         #] MakeDate :
02132         #[ set_in :
02133          Returns 1 if ch is in set ; 0 if ch is not in set:
02134 */
02135 int set_in(UBYTE ch, set_of_char set)
02136 {
02137     set += ch/8;
02138     switch (ch % 8){
02139         case 0: return(set->bit_0);
02140         case 1: return(set->bit_1);
02141         case 2: return(set->bit_2);
02142         case 3: return(set->bit_3);
02143         case 4: return(set->bit_4);
02144         case 5: return(set->bit_5);
02145         case 6: return(set->bit_6);
02146         case 7: return(set->bit_7);
02147     }/*switch (ch % 8)*/
02148     return(-1);
02149 }/*set_in*/
02150 /*
02151         #] set_in :
02152         #[ set_set :
02153             sets ch into set; returns *set:
02154 */
02155 one_byte set_set(UBYTE ch, set_of_char set)
02156 {
02157     one_byte tmp=(one_byte)set;
02158     set += ch/8;
02159     switch (ch % 8){
02160         case 0: set->bit_0=1;break;
02161         case 1: set->bit_1=1;break;
02162         case 2: set->bit_2=1;break;
02163         case 3: set->bit_3=1;break;
02164         case 4: set->bit_4=1;break;
02165         case 5: set->bit_5=1;break;
02166         case 6: set->bit_6=1;break;
02167         case 7: set->bit_7=1;break;
02168     }
02169     return(tmp);
02170 }/*set_set*/
02171 /*
02172         #] set_set :
02173         #[ set_del :
02174             deletes ch from set; returns *set:
02175 */
02176 one_byte set_del(UBYTE ch, set_of_char set)
02177 {
02178     one_byte tmp=(one_byte)set;
02179     set += ch/8;
02180     switch (ch % 8){
02181         case 0: set->bit_0=0;break;
02182         case 1: set->bit_1=0;break;
02183         case 2: set->bit_2=0;break;
02184         case 3: set->bit_3=0;break;
02185         case 4: set->bit_4=0;break;
02186         case 5: set->bit_5=0;break;
02187         case 6: set->bit_6=0;break;
02188         case 7: set->bit_7=0;break;
02189     }
02190     return(tmp);
02191 }/*set_del*/
02192 /*
02193         #] set_del :
02194         #[ set_sub :
02195             returns *set = set1\set2. This function may be usd for initialising,
02196                 set_sub(a,a,a) => now a is empty set :
02197 */
02198 one_byte set_sub(set_of_char set, set_of_char set1, set_of_char set2)
02199 {
02200     one_byte tmp=(one_byte)set;
02201     int i=0,j=0;
02202     while(j=0,i++<32)
02203     while(j<9)
02204         switch (j++){
02205             case 0: set->bit_0=(set1->bit_0&&(!set2->bit_0));break;
02206             case 1: set->bit_1=(set1->bit_1&&(!set2->bit_1));break;
02207             case 2: set->bit_2=(set1->bit_2&&(!set2->bit_2));break;
02208             case 3: set->bit_3=(set1->bit_3&&(!set2->bit_3));break;
02209             case 4: set->bit_4=(set1->bit_4&&(!set2->bit_4));break;
02210             case 5: set->bit_5=(set1->bit_5&&(!set2->bit_5));break;
02211             case 6: set->bit_6=(set1->bit_6&&(!set2->bit_6));break;
02212             case 7: set->bit_7=(set1->bit_7&&(!set2->bit_7));break;
02213             case 8: set++;set1++;set2++;
```

```
02214       };
02215     return(tmp);
02216 }/*set_sub*/
02217 /*
02218         #] set_sub :
02219     #] Strings :
02220     #[ Mixed :
02221         #[ iniTools :
02222 */
02223
02224 void iniTools(void)
02225 {
02226 #ifdef MALLOCPROTECT
02227     if ( mprotectInit() ) exit(0);
02228 #endif
02229     return;
02230 }
02231
02232 /*
02233         #] iniTools :
02234         #[ Malloc1 :
02235
02236         Malloc routine with built in error checking,
02237         and a more detailed error message.
02238         Gives the user some idea of what is happening.
02239 */
02240
02241 #ifdef MALLOCDEBUG
02242 INILOCK(MallocLock)
02243 #endif
02244
02245 void *Malloc1(LONG size, const char *messageifwrong)
02246 {
02247     void *mem;
02248 #ifdef MALLOCDEBUG
02249     char *t, *u;
02250     int i;
02251     LOCK(MallocLock);
02252     if ( size == 0 ) {
02253 /* INTERNAL_ERROR_EXCL_START */
02254         MesPrint("!>%wAsking for 0 bytes in Malloc1");
02255 /* INTERNAL_ERROR_EXCL_STOP */
02256     }
02257 #endif
02258 #ifdef WITHSTATS
02259     nummallocs++;
02260 #endif
02261     if ( ( size & 7 ) != 0 ) { size = size - ( size&7 ) + 8; }
02262 #ifdef MALLOCDEBUG
02263     size += 2*BANNER;
02264 #endif
02265     mem = (void *)M_alloc(size);
02266     if ( mem == 0 ) {
02267 #ifndef MALLOCDEBUG
02268         MLOCK(ErrorMessageLock);
02269 #endif
02270         MesPrint("Attempted to allocate %l bytes.", size);
02271         MesPrint("@No memory while allocating %s", (UBYTE *)messageifwrong);
02272 #ifndef MALLOCDEBUG
02273         MUNLOCK(ErrorMessageLock);
02274 #else
02275         UNLOCK(MallocLock);
02276 #endif
02277         Terminate(-1);
02278     }
02279 #ifdef MALLOCDEBUG
02280     mallocsizes[nummalloclist] = size;
02281     mallocstrings[nummalloclist] = (char *)messageifwrong;
02282     malloclist[nummalloclist++] = mem;
02283     if ( AC.MemDebugFlag && filelist ) MesPrint("%wMem1 at 0x%x: %l bytes.
    %s",mem,size,messageifwrong);
02284     {
02285         int i = nummalloclist-1;
02286         while ( --i >= 0 ) {
02287             if ( (char *)mem < (((char *)malloclist[i]) + mallocsizes[i])
02288             && (char *)(malloclist[i]) < ((char *)mem + size) ) {
02289                 if ( filelist ) MesPrint("This memory overlaps with the block at 0x%x"
02290                     ,malloclist[i]);
02291             }
02292         }
02293     }
02294
02295 #ifdef MALLOCDEBUGOUTPUT
02296     printf ("Malloc1: %s, allocated %li bytes at %.8lx\n",messageifwrong,size,(unsigned long)mem);
02297     fflush (stdout);
02298 #endif
02299
```

```
02300     t = (char *)mem;
02301     u = t + size;
02302     for ( i = 0; i < (int)BANNER; i++ ) { *t++ = FILLVALUE; *--u = FILLVALUE; }
02303     mem = (void *)t;
02304     M_check();
02305 /*  MUNLOCK(ErrorMessageLock); */
02306     UNLOCK(MallocLock);
02307 #endif
02308 /*
02309     if ( size > 500000000L ) {
02310         MLOCK(ErrorMessageLock);
02311         MesPrint("Malloc1: %s, allocated %l bytes\n",messageifwrong,size);
02312         MUNLOCK(ErrorMessageLock);
02313     }
02314 */
02315     return(mem);
02316 }
02317
02318 /*
02319         #] Malloc1 :
02320         #[ M_free :
02321 */
02322
02323 void M_free(void *x, const char *where)
02324 {
02325 #ifdef MALLOCDEBUG
02326     char *t = (char *)x;
02327     int i, j, k;
02328     LONG size = 0;
02329     x = (void *)(((char *)x)-BANNER);
02330 /*  MLOCK(ErrorMessageLock); */
02331     if ( AC.MemDebugFlag ) MesPrint("%wFreeing 0x%x: %s",x,where);
02332     LOCK(MallocLock);
02333     for ( i = nummalloclist-1; i >= 0; i-- ) {
02334         if ( x == malloclist[i] ) {
02335             size = mallocsizes[i];
02336             for ( j = i+1; j < nummalloclist; j++ ) {
02337                 malloclist[j-1] = malloclist[j];
02338                 mallocsizes[j-1] = mallocsizes[j];
02339                 mallocstrings[j-1] = mallocstrings[j];
02340             }
02341             nummalloclist--;
02342             break;
02343         }
02344     }
02345     if ( i < 0 ) {
02346         unsigned int xx = ((ULONG)x);
02347         printf("Error returning non-allocated address: 0x%x from %s\n"
02348             ,xx,where);
02349 /*      MUNLOCK(ErrorMessageLock); */
02350         UNLOCK(MallocLock);
02351         exit(-1);
02352     }
02353     else {
02354         for ( k = 0, j = 0; k < (int)BANNER; k++ ) {
02355             if ( *--t != FILLVALUE ) j++;
02356         }
02357         if ( j ) {
02358             LONG *tt = (LONG *)x;
02359             MesPrint("%w!!!!! Banner has been written in !!!!!: %x %x %x %x",
02360             tt[0],tt[1],tt[2],tt[3]);
02361         }
02362         t += size;
02363         for ( k = 0, j = 0; k < (int)BANNER; k++ ) {
02364             if ( *--t != FILLVALUE ) j++;
02365         }
02366         if ( j ) {
02367             LONG *tt = (LONG *)x;
02368             MesPrint("%w!!!!! Tail has been written in !!!!!: %x %x %x %x",
02369             tt[0],tt[1],tt[2],tt[3]);
02370         }
02371         M_check();
02372 /*      MUNLOCK(ErrorMessageLock); */
02373         UNLOCK(MallocLock);
02374     }
02375 #else
02376     DUMMYUSE(where);
02377 #endif
02378 #ifdef WITHSTATS
02379     numfrees++;
02380 #endif
02381     if ( x ) {
02382 #ifdef MALLOCDEBUGOUTPUT
02383         printf ("M_free: %s, memory freed at %.8lx\n",where,(unsigned long)x);
02384         fflush(stdout);
02385 #endif
02386
```

```
02387 #ifdef MALLOCPROTECT
02388         mprotectFree((void *)x);
02389 #else
02390         free(x);
02391 #endif
02392     }
02393 }
02394
02395 /*
02396         #] M_free :
02397         #[ M_check :
02398 */
02399
02400 #ifdef MALLOCDEBUG
02401
02402 void M_check1() { MesPrint("Checking Malloc"); M_check(); }
02403
02404 void M_check()
02405 {
02406     int i,j,k,error = 0;
02407     char *t;
02408     LONG *tt;
02409     for ( i = 0; i < nummalloclist; i++ ) {
02410         t = (char *)(malloclist[i]);
02411         for ( k = 0, j = 0; k < (int)BANNER; k++ ) {
02412             if ( *t++ != FILLVALUE ) j++;
02413         }
02414         if ( j ) {
02415             tt = (LONG *)(malloclist[i]);
02416             MesPrint("%w!!!!! Banner %d (%s) has been written in !!!!!: %x %x %x %x",
02417             i,mallocstrings[i],tt[0],tt[1],tt[2],tt[3]);
02418             tt[0] = tt[1] = tt[2] = tt[3] = 0;
02419             error = 1;
02420         }
02421         t = (char *)(malloclist[i]) + mallocsizes[i];
02422         for ( k = 0, j = 0; k < (int)BANNER; k++ ) {
02423             if ( *--t != FILLVALUE ) j++;
02424         }
02425         if ( j ) {
02426             tt = (LONG *)t;
02427             MesPrint("%w!!!!! Tail %d (%s) has been written in !!!!!: %x %x %x %x",
02428             i,mallocstrings[i],tt[0],tt[1],tt[2],tt[3]);
02429             tt[0] = tt[1] = tt[2] = tt[3] = 0;
02430             error = 1;
02431         }
02432         if ( ( mallocstrings[i][0] == ' ' ) || ( mallocstrings[i][0] == '#' ) ) {
02433             MesPrint("%w!!!!! Funny mallocstring");
02434             error = 1;
02435         }
02436     }
02437     if ( error ) {
02438         M_print();
02439 /*      MUNLOCK(ErrorMessageLock); */
02440         UNLOCK(MallocLock);
02441         Terminate(-1);
02442     }
02443 }
02444
02445 void M_print()
02446 {
02447     int i;
02448     MesPrint("We have the following memory allocations left:");
02449     for ( i = 0; i < nummalloclist; i++ ) {
02450         MesPrint("0x%x: %l bytes. number %d: '%s'",malloclist[i],mallocsizes[i],i,mallocstrings[i]);
02451     }
02452 }
02453
02454 #else
02455
02456 void M_check1(void) {}
02457 void M_print(void) {}
02458
02459 #endif
02460
02461 /*
02462         #] M_check :
02463         #[ TermMalloc :
02464 */
02487 #define TERMMEMSTARTNUM 16
02488 #define TERMEXTRAWORDS 10
02489
02490 void TermMallocAddMemory(PHEAD0)
02491 {
02492     WORD *newbufs;
02493     int i, extra;
02494     if ( AT.TermMemMax == 0 ) extra = TERMMEMSTARTNUM;
02495     else                      extra = AT.TermMemMax;
```

```
02496     if ( AT.TermMemHeap ) M_free(AT.TermMemHeap,"TermMalloc");
02497     newbufs = (WORD *)Malloc1(extra*(AM.MaxTer+TERMEXTRAWORDS*sizeof(WORD)),"TermMalloc");
02498     AT.TermMemHeap = (WORD **)Malloc1((extra+AT.TermMemMax)*sizeof(WORD *),"TermMalloc");
02499     for ( i = 0; i < extra; i++ ) {
02500         AT.TermMemHeap[i] = newbufs + i*(AM.MaxTer/sizeof(WORD)+TERMEXTRAWORDS);
02501     }
02502 #ifdef TERMMALLOCDEBUG
02503     DebugHeap2 = (WORD **)Malloc1((extra+AT.TermMemMax)*sizeof(WORD *),"TermMalloc");
02504     for ( i = 0; i < AT.TermMemMax; i++ ) { DebugHeap2[i] = DebugHeap1[i]; }
02505     for ( i = 0; i < extra; i++ ) {
02506         DebugHeap2[i+AT.TermMemMax] = newbufs + i*(AM.MaxTer/sizeof(WORD)+TERMEXTRAWORDS);
02507     }
02508     if ( DebugHeap1 ) M_free(DebugHeap1,"TermMalloc");
02509     DebugHeap1 = DebugHeap2;
02510 #endif
02511     AT.TermMemTop = extra;
02512     AT.TermMemMax += extra;
02513 #ifdef TERMMALLOCDEBUG
02514     MesPrint("AT.TermMemMax is now %l",AT.TermMemMax);
02515 #endif
02516 }
02517
02518 #ifndef MEMORYMACROS
02519
02520 WORD *TermMalloc2(PHEAD char *text)
02521 {
02522     if ( AT.TermMemTop <= 0 ) TermMallocAddMemory(BHEAD0);
02523
02524 #ifdef TERMMALLOCDEBUG
02525     MesPrint("TermMalloc: %s, %d",text,(AT.TermMemMax-AT.TermMemTop));
02526 #endif
02527
02528 #ifdef MALLOCDEBUGOUTPUT
02529     MesPrint("TermMalloc: %s, %l/%l
      (%x)",text,AT.TermMemTop,AT.TermMemMax,AT.TermMemHeap[AT.TermMemTop-1]);
02530 #endif
02531
02532     DUMMYUSE(text);
02533     return(AT.TermMemHeap[--AT.TermMemTop]);
02534 }
02535
02536 void TermFree2(PHEAD WORD *TermMem, char *text)
02537 {
02538 #ifdef TERMMALLOCDEBUG
02539
02540     int i;
02541
02542     for ( i = 0; i < AT.TermMemMax; i++ ) {
02543         if ( TermMem == DebugHeap1[i] ) break;
02544     }
02545     if ( i >= AT.TermMemMax ) {
02546         MesPrint(" ERROR: TermFree called with an address not given by TermMalloc.");
02547         Terminate(-1);
02548     }
02549 #endif
02550     DUMMYUSE(text);
02551     AT.TermMemHeap[AT.TermMemTop++] = TermMem;
02552
02553 #ifdef TERMMALLOCDEBUG
02554     MesPrint("TermFree: %s, %d",text,(AT.TermMemMax-AT.TermMemTop));
02555 #endif
02556 #ifdef MALLOCDEBUGOUTPUT
02557     MesPrint("TermFree: %s, %l/%l (%x)",text,AT.TermMemTop,AT.TermMemMax,TermMem);
02558 #endif
02559 }
02560
02561 #endif
02562
02563 /*
02564         #] TermMalloc :
02565         #[ NumberMalloc :
02566 */
02587 #define NUMBERMEMSTARTNUM 16
02588 #define NUMBEREXTRAWORDS 10L
02589
02590 #ifdef TERMMALLOCDEBUG
02591 UWORD **DebugHeap3, **DebugHeap4;
02592 #endif
02593
02594 void NumberMallocAddMemory(PHEAD0)
02595 {
02596     UWORD *newbufs;
02597     WORD extra;
02598     int i;
02599     if ( AT.NumberMemMax == 0 ) extra = NUMBERMEMSTARTNUM;
02600     else                        extra = AT.NumberMemMax;
02601     if ( AT.NumberMemHeap ) M_free(AT.NumberMemHeap,"NumberMalloc");
```

```
02602     newbufs = (UWORD *)Malloc1(extra*(AM.MaxTal+NUMBEREXTRAWORDS)*sizeof(UWORD),"NumberMalloc");
02603     AT.NumberMemHeap = (UWORD **)Malloc1((extra+AT.NumberMemMax)*sizeof(UWORD *),"NumberMalloc");
02604     for ( i = 0; i < extra; i++ ) {
02605         AT.NumberMemHeap[i] = newbufs + i*(LONG)(AM.MaxTal+NUMBEREXTRAWORDS);
02606     }
02607 #ifdef TERMMALLOCDEBUG
02608     DebugHeap4 = (UWORD **)Malloc1((extra+AT.NumberMemMax)*sizeof(WORD *),"NumberMalloc");
02609     for ( i = 0; i < AT.NumberMemMax; i++ ) { DebugHeap4[i] = DebugHeap3[i]; }
02610     for ( i = 0; i < extra; i++ ) {
02611         DebugHeap4[i+AT.NumberMemMax] = newbufs + i*(LONG)(AM.MaxTal+NUMBEREXTRAWORDS);
02612     }
02613     if ( DebugHeap3 ) M_free(DebugHeap3,"NumberMalloc");
02614     DebugHeap3 = DebugHeap4;
02615 #endif
02616     AT.NumberMemTop = extra;
02617     AT.NumberMemMax += extra;
02618 /*
02619 MesPrint("AT.NumberMemMax is now %l",AT.NumberMemMax);
02620 */
02621 }
02622
02623 #ifndef MEMORYMACROS
02624
02625 UWORD *NumberMalloc2(PHEAD char *text)
02626 {
02627     if ( AT.NumberMemTop <= 0 ) NumberMallocAddMemory(BHEAD text);
02628
02629 #ifdef MALLOCDEBUGOUTPUT
02630     if ( (AT.NumberMemMax-AT.NumberMemTop) > 10 )
02631     MesPrint("NumberMalloc: %s, %l/%l
      (%x)",text,AT.NumberMemTop,AT.NumberMemMax,AT.NumberMemHeap[AT.NumberMemTop-1]);
02632 #endif
02633
02634     DUMMYUSE(text);
02635     return(AT.NumberMemHeap[--AT.NumberMemTop]);
02636 }
02637
02638 void NumberFree2(PHEAD UWORD *NumberMem, char *text)
02639 {
02640 #ifdef TERMMALLOCDEBUG
02641     int i;
02642     for ( i = 0; i < AT.NumberMemMax; i++ ) {
02643         if ( NumberMem == DebugHeap3[i] ) break;
02644     }
02645     if ( i >= AT.NumberMemMax ) {
02646         MesPrint(" ERROR: NumberFree called with an address not given by NumberMalloc.");
02647         Terminate(-1);
02648     }
02649 #endif
02650     DUMMYUSE(text);
02651     AT.NumberMemHeap[AT.NumberMemTop++] = NumberMem;
02652
02653 #ifdef MALLOCDEBUGOUTPUT
02654     if ( (AT.NumberMemMax-AT.NumberMemTop) > 10 )
02655     MesPrint("NumberFree: %s, %l/%l (%x)",text,AT.NumberMemTop,AT.NumberMemMax,NumberMem);
02656 #endif
02657 }
02658
02659 #endif
02660
02661 /*
02662         #] NumberMalloc :
02663         #[ CacheNumberMalloc :
02664
02665     Similar to NumberMalloc
02666  */
02667
02668 void CacheNumberMallocAddMemory(PHEAD0)
02669 {
02670     UWORD *newbufs;
02671     WORD extra;
02672     int i;
02673     if ( AT.CacheNumberMemMax == 0 ) extra = NUMBERMEMSTARTNUM;
02674     else                             extra = AT.CacheNumberMemMax;
02675     if ( AT.CacheNumberMemHeap ) M_free(AT.CacheNumberMemHeap,"NumberMalloc");
02676     newbufs = (UWORD *)Malloc1(extra*(AM.MaxTal+NUMBEREXTRAWORDS)*sizeof(UWORD),"CacheNumberMalloc");
02677     AT.CacheNumberMemHeap = (UWORD **)Malloc1((extra+AT.NumberMemMax)*sizeof(UWORD
      *),"CacheNumberMalloc");
02678     for ( i = 0; i < extra; i++ ) {
02679         AT.CacheNumberMemHeap[i] = newbufs + i*(LONG)(AM.MaxTal+NUMBEREXTRAWORDS);
02680     }
02681     AT.CacheNumberMemTop = extra;
02682     AT.CacheNumberMemMax += extra;
02683 }
02684
02685 #ifndef MEMORYMACROS
02686
```

```
02687 UWORD *CacheNumberMalloc2(PHEAD char *text)
02688 {
02689     if ( AT.CacheNumberMemTop <= 0 ) CacheNumberMallocAddMemory(BHEAD0);
02690
02691 #ifdef MALLOCDEBUGOUTPUT
02692     MesPrint("NumberMalloc: %s, %l/%l
      (%x)",text,AT.NumberMemTop,AT.NumberMemMax,AT.NumberMemHeap[AT.NumberMemTop-1]);
02693 #endif
02694
02695     DUMMYUSE(text);
02696     return(AT.CacheNumberMemHeap[--AT.CacheNumberMemTop]);
02697 }
02698
02699 void CacheNumberFree2(PHEAD UWORD *NumberMem, char *text)
02700 {
02701     DUMMYUSE(text);
02702     AT.CacheNumberMemHeap[AT.CacheNumberMemTop++] = NumberMem;
02703
02704 #ifdef MALLOCDEBUGOUTPUT
02705     MesPrint("NumberFree: %s, %l/%l (%x)",text,AT.NumberMemTop,AT.NumberMemMax,NumberMem);
02706 #endif
02707 }
02708
02709 #endif
02710
02711 /*
02712         #] CacheNumberMalloc :
02713         #[ FromList :
02714
02715     Returns the next object in a list.
02716     If the list has been exhausted we double it (like a realloc)
02717     If the list has not been initialized yet we start with 12 elements.
02718 */
02719
02720 void *FromList(LIST *L)
02721 {
02722     void *newlist;
02723     int i, *old, *newL;
02724     if ( L->num >= L->maxnum || L->lijst == 0 ) {
02725         if ( L->maxnum == 0 ) L->maxnum = 12;
02726         else if ( L->lijst ) L->maxnum *= 2;
02727         newlist = Malloc1(L->maxnum * L->size,L->message);
02728         if ( L->lijst ) {
02729             i = ( L->num * L->size ) / sizeof(int);
02730             old = (int *)L->lijst; newL = (int *)newlist;
02731             while ( --i >= 0 ) *newL++ = *old++;
02732             if ( L->lijst ) M_free(L->lijst,"L->lijst FromList");
02733         }
02734         L->lijst = newlist;
02735     }
02736     return( ((char *)(L->lijst)) + L->size * (L->num)++ );
02737 }
02738
02739 /*
02740         #] FromList :
02741         #[ From0List :
02742
02743         Same as FromList, but we zero excess variables.
02744 */
02745
02746 void *From0List(LIST *L)
02747 {
02748     void *newlist;
02749     int i, *old, *newL;
02750     if ( L->num >= L->maxnum || L->lijst == 0 ) {
02751         if ( L->maxnum == 0 ) L->maxnum = 12;
02752         else if ( L->lijst ) L->maxnum *= 2;
02753         newlist = Malloc1(L->maxnum * L->size,L->message);
02754         i = ( L->num * L->size ) / sizeof(int);
02755         old = (int *)(L->lijst); newL = (int *)newlist;
02756         while ( --i >= 0 ) *newL++ = *old++;
02757         i = ( L->maxnum - L->num ) / sizeof(int);
02758         while ( --i >= 0 ) *newL++ = 0;
02759         if ( L->lijst ) M_free(L->lijst,"L->lijst From0List");
02760         L->lijst = newlist;
02761     }
02762     return( ((char *)(L->lijst)) + L->size * (L->num)++ );
02763 }
02764
02765 /*
02766         #] From0List :
02767         #[ FromVarList :
02768
02769     Returns the next object in a list of variables.
02770     If the list has been exhausted we double it (like a realloc)
02771     If the list has not been initialized yet we start with 12 elements.
02772     We allow at most MAXVARIABLES elements!
```

```
02773 */
02774
02775 void *FromVarList(LIST *L)
02776 {
02777     void *newlist;
02778     int i, *old, *newL;
02779     if ( L->num >= L->maxnum || L->lijst == 0 ) {
02780         if ( L->maxnum == 0 ) L->maxnum = 12;
02781         else if ( L->lijst ) {
02782             L->maxnum *= 2;
02783             if ( L == &(AP.DollarList) ) {
02784                 if ( L->maxnum > MAXDOLLARVARIABLES ) L->maxnum = MAXDOLLARVARIABLES;
02785                 if ( L->num >= MAXDOLLARVARIABLES ) {
02786 /* INTERNAL_ERROR_EXCL_START */
02787                     MesPrint("!>More than %l objects in list of $-variables",
02788                         MAXDOLLARVARIABLES);
02789                     Terminate(-1);
02790 /* INTERNAL_ERROR_EXCL_STOP */
02791                 }
02792             }
02793             else {
02794                 if ( L->maxnum > MAXVARIABLES ) L->maxnum = MAXVARIABLES;
02795                 if ( L->num >= MAXVARIABLES ) {
02796 /* INTERNAL_ERROR_EXCL_START */
02797                     MesPrint("!>More than %l objects in list of variables",
02798                         MAXVARIABLES);
02799                     Terminate(-1);
02800 /* INTERNAL_ERROR_EXCL_STOP */
02801                 }
02802             }
02803         }
02804         newlist = Malloc1(L->maxnum * L->size,L->message);
02805         if ( L->lijst ) {
02806             i = ( L->num * L->size ) / sizeof(int);
02807             old = (int *)(L->lijst); newL = (int *)newlist;
02808             while ( --i >= 0 ) *newL++ = *old++;
02809             if ( L->lijst ) M_free(L->lijst,"L->lijst from VarList");
02810         }
02811         L->lijst = newlist;
02812     }
02813     return( ((char *)(L->lijst)) + L->size * ((L->num)++) );
02814 }
02815
02816 /*
02817         #] FromVarList :
02818         #[ DoubleList :
02819 */
02820
02821 int DoubleList(void ***lijst, int *oldsize, int objectsize, char *nameoftype)
02822 {
02823     void **newlist;
02824     LONG i, newsize, fullsize;
02825     void **to, **from;
02826     static LONG maxlistsize = (LONG)(MAXPOSITIVE);
02827     if ( *lijst == 0 ) {
02828         if ( *oldsize > 0 ) newsize = *oldsize;
02829         else newsize = 100;
02830     }
02831     else newsize = *oldsize * 2;
02832     if ( newsize > maxlistsize ) {
02833         if ( *oldsize == maxlistsize ) {
02834             MesPrint("No memory for extra space in %s",nameoftype);
02835             return(-1);
02836         }
02837         newsize = maxlistsize;
02838     }
02839     fullsize = ( newsize * objectsize + sizeof(void *)-1 ) & (-sizeof(void *));
02840     newlist = (void **)Malloc1(fullsize,nameoftype);
02841     if ( *lijst ) { /* Now some punning. DANGEROUS CODE in principle */
02842         to = newlist; from = *lijst; i = (*oldsize * objectsize)/sizeof(void *);
02843 /*
02844 #ifdef MALLOCDEBUG
02845 if ( filelist ) MesPrint("    oldsize: %l, objectsize: %d, fullsize: %l"
02846         ,*oldsize,objectsize,fullsize);
02847 #endif
02848 */
02849         while ( --i >= 0 ) *to++ = *from++;
02850     }
02851     if ( *lijst ) M_free(*lijst,"DoubleLList");
02852     *lijst = newlist;
02853     *oldsize = newsize;
02854     return(0);
02855 /*
02856     int error;
02857     LONG lsize = *oldsize;
02858
02859     maxlistsize = (LONG)(MAXPOSITIVE);
```

```
02860     error = DoubleLList(lijst,&lsize,objectsize,nameoftype);
02861     *oldsize = lsize;
02862     maxlistsize = (LONG)(MAXLONG);
02863
02864     return(error);
02865 */
02866 }
02867
02868 /*
02869         #] DoubleList :
02870         #[ DoubleLList :
02871 */
02872
02873 int DoubleLList(void ***lijst, LONG *oldsize, int objectsize, char *nameoftype)
02874 {
02875     void **newlist;
02876     LONG i, newsize, fullsize;
02877     void **to, **from;
02878     static LONG maxlistsize = (LONG)(MAXLONG);
02879     if ( *lijst == 0 ) {
02880         if ( *oldsize > 0 ) newsize = *oldsize;
02881         else newsize = 100;
02882     }
02883     else newsize = *oldsize * 2;
02884     if ( newsize > maxlistsize ) {
02885         if ( *oldsize == maxlistsize ) {
02886             MesPrint("No memory for extra space in %s",nameoftype);
02887             return(-1);
02888         }
02889         newsize = maxlistsize;
02890     }
02891     fullsize = ( newsize * objectsize + sizeof(void *)-1 ) & (-sizeof(void *));
02892     newlist = (void **)Malloc1(fullsize,nameoftype);
02893     if ( *lijst ) { /* Now some punning. DANGEROUS CODE in principle */
02894         to = newlist; from = *lijst; i = (*oldsize * objectsize)/sizeof(void *);
02895 /*
02896 #ifdef MALLOCDEBUG
02897 if ( filelist ) MesPrint("    oldsize: %l, objectsize: %d, fullsize: %l"
02898         ,*oldsize,objectsize,fullsize);
02899 #endif
02900 */
02901         while ( --i >= 0 ) *to++ = *from++;
02902     }
02903     if ( *lijst ) M_free(*lijst,"DoubleLList");
02904     *lijst = newlist;
02905     *oldsize = newsize;
02906     return(0);
02907 }
02908
02909 /*
02910         #] DoubleLList :
02911         #[ DoubleBuffer :
02912 */
02913
02914 #define DODOUBLE(x) { x *s, *t, *u; if ( *start ) { \
02915     oldsize = *(x **)stop - *(x **)start; newsize = 2*oldsize; \
02916     t = u = (x *)Malloc1(newsize*sizeof(x),text); s = *(x **)start; \
02917     for ( i = 0; i < oldsize; i++ ) {*t++ = *s++;} M_free(*start,"double"); } \
02918     else { newsize = 100; u = (x *)Malloc1(newsize*sizeof(x),text); } \
02919     *start = (void *)u; *stop = (void *)(u+newsize); }
02920
02921 void DoubleBuffer(void **start, void **stop, int size, char *text)
02922 {
02923     LONG oldsize, newsize, i;
02924     if ( size == sizeof(char) ) DODOUBLE(char)
02925     else if ( size == sizeof(short) ) DODOUBLE(short)
02926     else if ( size == sizeof(int) ) DODOUBLE(int)
02927     else if ( size == sizeof(LONG) ) DODOUBLE(LONG)
02928     else if ( size % sizeof(int) == 0 ) DODOUBLE(int)
02929     else {
02930         MesPrint("---Cannot handle doubling buffers of size %d",size);
02931         Terminate(-1);
02932     }
02933 }
02934
02935 /*
02936         #] DoubleBuffer :
02937         #[ ExpandBuffer :
02938 */
02939
02940 #define DOEXPAND(x) { x *newbuffer, *t, *m;                              \
02941     t = newbuffer = (x *)Malloc1((newsize+2)*type,"ExpandBuffer");       \
02942     if ( *buffer ) { m = (x *)*buffer; i = *oldsize;                     \
02943         while ( --i >= 0 ) {*t++ = *m++;} M_free(*buffer,"ExpandBuffer"); \
02944     } *buffer = newbuffer; *oldsize = newsize; }
02945
02946 void ExpandBuffer(void **buffer, LONG *oldsize, int type)
```

```
02947 {
02948     LONG newsize, i;
02949     if ( *oldsize <= 0 ) { newsize = 100; }
02950     else newsize = 2*(*oldsize);
02951     if ( type == sizeof(char) ) DOEXPAND(char)
02952     else if ( type == sizeof(short) ) DOEXPAND(short)
02953     else if ( type == sizeof(int) ) DOEXPAND(int)
02954     else if ( type == sizeof(LONG) ) DOEXPAND(LONG)
02955     else if ( type == sizeof(POSITION) ) DOEXPAND(POSITION)
02956     else {
02957         MesPrint("---Cannot handle expanding buffers with objects of size %d",type);
02958         Terminate(-1);
02959     }
02960 }
02961
02962 /*
02963         #] ExpandBuffer :
02964         #[ iexp :
02965
02966         Raises the long integer y to the power p.
02967         Returnvalue is long, regardless of overflow.
02968 */
02969
02970 LONG iexp(LONG x, int p)
02971 {
02972     int sign;
02973     ULONG y;
02974     ULONG ux;
02975     if ( x == 0 ) return(0);
02976     if ( p == 0 ) return(1);
02977     sign = x < 0 ? -1 : 1;
02978     if ( sign < 0 && ( p & 1 ) == 0 ) sign = 1;
02979     ux = LongAbs(x);
02980     if ( ux == 1 ) return(sign);
02981     if ( p < 0 ) return(0);
02982     y = 1;
02983     while ( p ) {
02984         if ( ( p & 1 ) != 0 ) y *= ux;
02985         p >>= 1;
02986         ux = ux*ux;
02987     }
02988     if ( sign < 0 ) y = -y;
02989     return ULongToLong(y);
02990 }
02991
02992 /*
02993         #] iexp :
02994         #[ ToGeneral :
02995
02996         Convert a fast argument to a general argument
02997         Input in r, output in m.
02998         If par == 0 we need the argument header also.
02999 */
03000
03001 void ToGeneral(WORD *r, WORD *m, WORD par)
03002 {
03003     WORD *mm = m, j, k;
03004     if ( par ) m++;
03005     else { m[1] = 0; m += ARGHEAD + 1; }
03006     j = -*r++;
03007     k = 3;
03008 /*      JV: Bugfix 1-feb-2016. Old code assumed FUNHEAD to be 2 */
03009     if ( j >= FUNCTION ) { *m++ = j; *m++ = FUNHEAD; FILLFUN(m) }
03010     else {
03011         switch ( j ) {
03012             case SYMBOL: *m++ = j; *m++ = 4; *m++ = *r++; *m++ = 1; break;
03013             case SNUMBER:
03014                 if ( *r > 0 ) { *m++ =  *r; *m++ = 1; *m++ =  3; }
03015                 else if ( *r == 0 ) { m--; }
03016                 else          { *m++ = -*r; *m++ = 1; *m++ = -3; }
03017                 goto MakeSize;
03018             case MINVECTOR:
03019                 k = -k;
03020                 /* fall through */
03021             case INDEX:
03022             case VECTOR:
03023                 *m++ = INDEX; *m++ = 3; *m++ = *r++;
03024                 break;
03025         }
03026     }
03027     *m++ = 1; *m++ = 1; *m++ = k;
03028 MakeSize:
03029     *mm = m-mm;
03030     if ( !par ) mm[ARGHEAD] = *mm-ARGHEAD;
03031 }
03032
03033 /*
```

```
03034         #] ToGeneral :
03035         #[ ToFast :
03036
03037         Checks whether an argument can be converted to fast notation
03038         If this can be done it does it.
03039         Important: m should be allowed to be equal to r!
03040         Return value is 1 if conversion took place.
03041         If there was conversion the answer is in m.
03042         If there was no conversion m hasn't been touched.
03043 */
03044
03045 int ToFast(WORD *r, WORD *m)
03046 {
03047     WORD i;
03048     if ( *r == ARGHEAD ) { *m++ = -SNUMBER; *m++ = 0; return(1); }
03049     if ( *r != r[ARGHEAD]+ARGHEAD ) return(0);  /* > 1 term */
03050     r += ARGHEAD;
03051     if ( *r == 4 ) {
03052         if ( r[2] != 1 || r[1] <= 0 ) return(0);
03053         *m++ = -SNUMBER; *m = ( r[3] < 0 ) ? -r[1] : r[1]; return(1);
03054     }
03055     i = *r - 1;
03056     if ( r[i-1] != 1 || r[i-2] != 1 ) return(0);
03057     if ( r[i] != 3 ) {
03058         if ( r[i] == -3 && r[2] == *r-4 && r[2] == 3 && r[1] == INDEX
03059         && r[3] < MINSPEC ) {}
03060         else return(0);
03061     }
03062     else if ( r[2] != *r - 4 ) return(0);
03063     r++;
03064     if ( *r >= FUNCTION ) {
03065         if ( r[1] <= FUNHEAD ) { *m++ = -*r; return(1); }
03066     }
03067     else if ( *r == SYMBOL ) {
03068         if ( r[1] == 4 && r[3] == 1 )
03069             { *m++ = -SYMBOL; *m++ = r[2]; return(1); }
03070     }
03071     else if ( *r == INDEX ) {
03072         if ( r[1] == 3 ) {
03073             if ( r[2] >= MINSPEC ) {
03074                 if ( r[2] >= 0 && r[2] < AM.OffsetIndex ) *m++ = -SNUMBER;
03075                 else *m++ = -INDEX;
03076             }
03077             else {
03078                 if ( r[5] == -3 ) *m++ = -MINVECTOR;
03079                 else *m++ = -VECTOR;
03080             }
03081             *m++ = r[2];
03082             return(1);
03083         }
03084     }
03085     return(0);
03086 }
03087
03088 /*
03089         #] ToFast :
03090         #[ ToPolyFunGeneral :
03091
03092     Routine forces a polyratfun into general notation if needed.
03093     If no action was needed, the return value is zero.
03094     A positive return value indicates how many arguments were converted.
03095     The new term overwrite the old.
03096 */
03097
03098 WORD ToPolyFunGeneral(PHEAD WORD *term)
03099 {
03100     WORD *t = term+1, *tt, *to, *to1, *termout, *tstop, *tnext;
03101     WORD numarg, i, change = 0;
03102     tstop = term + *term; tstop -= ABS(tstop[-1]);
03103     termout = to = AT.WorkPointer;
03104     to++;
03105     while ( t < tstop ) { /* go through the subterms */
03106         if ( *t == AR.PolyFun ) {
03107             tt = t+FUNHEAD; tnext = t + t[1];
03108             numarg = 0;
03109             while ( tt < tnext ) { numarg++; NEXTARG(tt); }
03110             if ( numarg == 2 ) { /* this needs attention */
03111                 tt = t + FUNHEAD;
03112                 to1 = to;
03113                 i = FUNHEAD; NCOPY(to,t,i);
03114                 while ( tt < tnext ) { /* Do the arguments */
03115                     if ( *tt > 0 ) {
03116                         i = *tt; NCOPY(to,tt,i);
03117                     }
03118                     else if ( *tt == -SYMBOL ) {
03119                         to1[1] += 6+ARGHEAD; to1[2] |= MUSTCLEANPRF; change++;
03120                         *to++ = 8+ARGHEAD; *to++ = 0; FILLARG(to);
```

```
03121                       *to++ = 8; *to++ = SYMBOL; *to++ = 4; *to++ = tt[1];
03122                       *to++ = 1; *to++ = 1; *to++ = 1; *to++ = 3;
03123                       tt += 2;
03124                   }
03125                   else if ( *tt == -SNUMBER ) {
03126                       if ( tt[1] > 0 ) {
03127                           to1[1] += 2+ARGHEAD; to1[2] |= MUSTCLEANPRF; change++;
03128                           *to++ = 4+ARGHEAD; *to++ = 0; FILLARG(to);
03129                           *to++ = 4; *to++ = tt[1]; *to++ = 1; *to++ = 3;
03130                           tt += 2;
03131                       }
03132                       else if ( tt[1] < 0 ) {
03133                           to1[1] += 2+ARGHEAD; to1[2] |= MUSTCLEANPRF; change++;
03134                           *to++ = 4+ARGHEAD; *to++ = 0; FILLARG(to);
03135                           *to++ = 4; *to++ = -tt[1]; *to++ = 1; *to++ = -3;
03136                           tt += 2;
03137                       }
03138                       else {
03139 /* INTERNAL_ERROR_EXCL_START */
03140                           MLOCK(ErrorMessageLock);
03141                           MesPrint("!>Internal error: Zero in PolyRatFun");
03142                           MUNLOCK(ErrorMessageLock);
03143                           Terminate(-1);
03144 /* INTERNAL_ERROR_EXCL_STOP */
03145                       }
03146                   }
03147               }
03148               t = tnext;
03149               continue;
03150           }
03151       }
03152       i = t[1]; NCOPY(to,t,i)
03153   }
03154   if ( change ) {
03155       tt = term + *term;
03156       while ( t < tt ) *to++ = *t++;
03157       *termout = to - termout;
03158       t = term; i = *termout; tt = termout;
03159       NCOPY(t,tt,i)
03160       AT.WorkPointer = term + *term;
03161   }
03162   return(change);
03163 }
03164
03165 /*
03166       #] ToPolyFunGeneral :
03167       #[ IsLikeVector :
03168
03169       Routine determines whether a function argument is like a vector.
03170       Returnvalue: 1: is vector or index
03171                    0: is not vector or index
03172                   -1: may be an index
03173 */
03174
03175 int IsLikeVector(WORD *arg)
03176 {
03177   WORD *sstop, *t, *tstop;
03178   if ( *arg < 0 ) {
03179       if ( *arg == -VECTOR || *arg == -INDEX ) return(1);
03180       if ( *arg == -SNUMBER && arg[1] >= 0 && arg[1] < AM.OffsetIndex )
03181           return(-1);
03182       return(0);
03183   }
03184   sstop = arg + *arg; arg += ARGHEAD;
03185   while ( arg < sstop ) {
03186       t = arg + *arg;
03187       tstop = t - ABS(t[-1]);
03188       arg++;
03189       while ( arg < tstop ) {
03190           if ( *arg == INDEX ) return(1);
03191           arg += arg[1];
03192       }
03193       arg = t;
03194   }
03195   return(0);
03196 }
03197
03198 /*
03199       #] IsLikeVector :
03200       #[ AreArgsEqual :
03201 */
03202
03203 int AreArgsEqual(WORD *arg1, WORD *arg2)
03204 {
03205   int i;
03206   if ( *arg2 != *arg1 ) return(0);
03207   if ( *arg1 > 0 ) {
```

```
03208         i = *arg1;
03209         while ( --i > 0 ) { if ( arg1[i] != arg2[i] ) return(0); }
03210         return(1);
03211     }
03212     else if ( *arg1 <= -FUNCTION ) return(1);
03213     else if ( arg1[1] == arg2[1] ) return(1);
03214     return(0);
03215 }
03216
03217 /*
03218         #] AreArgsEqual :
03219         #[ CompareArgs :
03220 */
03221
03222 int CompareArgs(WORD *arg1, WORD *arg2)
03223 {
03224     int i1,i2;
03225     if ( *arg1 > 0 ) {
03226         if ( *arg2 < 0 ) return(-1);
03227         i1 = *arg1-ARGHEAD; arg1 += ARGHEAD;
03228         i2 = *arg2-ARGHEAD; arg2 += ARGHEAD;
03229         while ( i1 > 0 && i2 > 0 ) {
03230             if ( *arg1 != *arg2 ) return((int)(*arg1)-(int)(*arg2));
03231             i1--; i2--; arg1++; arg2++;
03232         }
03233         return(i1-i2);
03234     }
03235     else if ( *arg2 > 0 ) return(1);
03236     else {
03237         if ( *arg1 != *arg2 ) {
03238             if ( *arg1 < *arg2 ) return(-1);
03239             else return(1);
03240         }
03241         if ( *arg1 <= -FUNCTION ) return(0);
03242         return((int)(arg1[1])-(int)(arg2[1]));
03243     }
03244 }
03245
03246 /*
03247         #] CompareArgs :
03248         #[ CompArg :
03249
03250     returns 1 if arg1 comes first, -1 if arg2 comes first, 0 if equal
03251 */
03252
03253 int CompArg(WORD *s1, WORD *s2)
03254 {
03255     GETIDENTITY
03256     WORD *st1, *st2, x[7];
03257     int k;
03258     if ( *s1 < 0 ) {
03259         if ( *s2 < 0 ) {
03260             if ( *s1 <= -FUNCTION && *s2 <= -FUNCTION ) {
03261                 if ( *s1 > *s2 ) return(-1);
03262                 if ( *s1 < *s2 ) return(1);
03263                 return(0);
03264             }
03265             if ( *s1 > *s2 ) return(1);
03266             if ( *s1 < *s2 ) return(-1);
03267             if ( *s1 <= -FUNCTION ) return(0);
03268             s1++; s2++;
03269             if ( *s1 > *s2 ) return(1);
03270             if ( *s1 < *s2 ) return(-1);
03271             return(0);
03272         }
03273         x[1] = AT.comsym[3];
03274         x[2] = AT.comnum[1];
03275         x[3] = AT.comnum[3];
03276         x[4] = AT.comind[3];
03277         x[5] = AT.comind[6];
03278         x[6] = AT.comfun[1];
03279         if ( *s1 == -SYMBOL ) {
03280             AT.comsym[3] = s1[1];
03281             st1 = AT.comsym+8; s1 = AT.comsym;
03282         }
03283         else if ( *s1 == -SNUMBER ) {
03284             if ( s1[1] < 0 ) {
03285                 AT.comnum[1] = -s1[1]; AT.comnum[3] = -3;
03286             }
03287             else {
03288                 AT.comnum[1] = s1[1]; AT.comnum[3] = 3;
03289             }
03290             st1 = AT.comnum+4;
03291             s1 = AT.comnum;
03292         }
03293         else if ( *s1 == -INDEX || *s1 == -VECTOR ) {
03294             AT.comind[3] = s1[1]; AT.comind[6] = 3;
```

```
03295             st1 = AT.comind+7; s1 = AT.comind;
03296         }
03297         else if ( *s1 == -MINVECTOR ) {
03298             AT.comind[3] = s1[1]; AT.comind[6] = -3;
03299             st1 = AT.comind+7; s1 = AT.comind;
03300         }
03301         else if ( *s1 <= -FUNCTION ) {
03302             AT.comfun[1] = -*s1;
03303             st1 = AT.comfun+FUNHEAD+4; s1 = AT.comfun;
03304         }
03305 /*
03306             Symmetrize during compilation of id statement when properorder
03307             needs this one. Code added 10-nov-2001
03308 */
03309         else if ( *s1 == -ARGWILD ) {
03310              return(-1);
03311         }
03312         else { goto argerror; }
03313         st2 = s2 + *s2; s2 += ARGHEAD;
03314         goto docompare;
03315     }
03316     else if ( *s2 < 0 ) {
03317         x[1] = AT.comsym[3];
03318         x[2] = AT.comnum[1];
03319         x[3] = AT.comnum[3];
03320         x[4] = AT.comind[3];
03321         x[5] = AT.comind[6];
03322         x[6] = AT.comfun[1];
03323         if ( *s2 == -SYMBOL ) {
03324             AT.comsym[3] = s2[1];
03325             st2 = AT.comsym+8; s2 = AT.comsym;
03326         }
03327         else if ( *s2 == -SNUMBER ) {
03328             if ( s2[1] < 0 ) {
03329                 AT.comnum[1] = -s2[1]; AT.comnum[3] = -3;
03330                 st2 = AT.comnum+4;
03331             }
03332             else if ( s2[1] == 0 ) {
03333                 st2 = AT.comnum+4; s2 = st2;
03334             }
03335             else {
03336                 AT.comnum[1] = s2[1]; AT.comnum[3] = 3;
03337                 st2 = AT.comnum+4;
03338             }
03339             s2 = AT.comnum;
03340         }
03341         else if ( *s2 == -INDEX || *s2 == -VECTOR ) {
03342             AT.comind[3] = s2[1]; AT.comind[6] = 3;
03343             st2 = AT.comind+7; s2 = AT.comind;
03344         }
03345         else if ( *s2 == -MINVECTOR ) {
03346             AT.comind[3] = s2[1]; AT.comind[6] = -3;
03347             st2 = AT.comind+7; s2 = AT.comind;
03348         }
03349         else if ( *s2 <= -FUNCTION ) {
03350             AT.comfun[1] = -*s2;
03351             st2 = AT.comfun+FUNHEAD+4; s2 = AT.comfun;
03352         }
03353 /*
03354             Symmetrize during compilation of id statement when properorder
03355             needs this one. Code added 10-nov-2001
03356 */
03357         else if ( *s2 == -ARGWILD ) {
03358              return(1);
03359         }
03360         else { goto argerror; }
03361         st1 = s1 + *s1; s1 += ARGHEAD;
03362         goto docompare;
03363     }
03364     else {
03365         x[1] = AT.comsym[3];
03366         x[2] = AT.comnum[1];
03367         x[3] = AT.comnum[3];
03368         x[4] = AT.comind[3];
03369         x[5] = AT.comind[6];
03370         x[6] = AT.comfun[1];
03371         st1 = s1 + *s1; st2 = s2 + *s2;
03372         s1 += ARGHEAD; s2 += ARGHEAD;
03373 docompare:
03374         while ( s1 < st1 && s2 < st2 ) {
03375             if ( ( k = CompareTerms(BHEAD s1,s2,(WORD)2) ) != 0 ) {
03376                 AT.comsym[3] = x[1];
03377                 AT.comnum[1] = x[2];
03378                 AT.comnum[3] = x[3];
03379                 AT.comind[3] = x[4];
03380                 AT.comind[6] = x[5];
03381                 AT.comfun[1] = x[6];
```

```
03382                 return(-k);
03383             }
03384             s1 += *s1; s2 += *s2;
03385         }
03386         AT.comsym[3] = x[1];
03387         AT.comnum[1] = x[2];
03388         AT.comnum[3] = x[3];
03389         AT.comind[3] = x[4];
03390         AT.comind[6] = x[5];
03391         AT.comfun[1] = x[6];
03392         if ( s1 < st1 ) return(1);
03393         if ( s2 < st2 ) return(-1);
03394     }
03395     return(0);
03396
03397 argerror:
03398 /* INTERNAL_ERROR_EXCL_START */
03399     MesPrint("!>Illegal type of short function argument in Normalize");
03400     Terminate(-1); return(0);
03401 /* INTERNAL_ERROR_EXCL_STOP */
03402 }
03403
03404 /*
03405         #] CompArg :
03406         #[ TimeWallClock :
03407 */
03408
03409 #ifdef HAVE_CLOCK_GETTIME
03410 #include <time.h> /* for clock_gettime() */
03411 #else
03412 #ifdef HAVE_GETTIMEOFDAY
03413 #include <sys/time.h> /* for gettimeofday() */
03414 #else
03415 #include <sys/timeb.h> /* for ftime() */
03416 #endif
03417 #endif
03418
03425 LONG TimeWallClock(WORD par)
03426 {
03427     /*
03428      * NOTE: this function is not thread-safe. Operations on tp are not atomic.
03429      */
03430
03431 #ifdef HAVE_CLOCK_GETTIME
03432     struct timespec ts;
03433     clock_gettime(CLOCK_MONOTONIC, &ts);
03434
03435     if ( par ) {
03436         return(((LONG)(ts.tv_sec)-AM.OldSecTime)*100 +
03437             ((LONG)(ts.tv_nsec / 1000000)-AM.OldMilliTime)/10);
03438     }
03439     else {
03440         AM.OldSecTime   = (LONG)(ts.tv_sec);
03441         AM.OldMilliTime = (LONG)(ts.tv_nsec / 1000000);
03442         return(0L);
03443     }
03444 #else
03445 #ifdef HAVE_GETTIMEOFDAY
03446     struct timeval t;
03447     LONG sec, msec;
03448     gettimeofday(&t, NULL);
03449     sec = (LONG)t.tv_sec;
03450     msec = (LONG)(t.tv_usec/1000);
03451     if ( par ) {
03452         return (sec-AM.OldSecTime)*100 + (msec-AM.OldMilliTime)/10;
03453     }
03454     else {
03455         AM.OldSecTime   = sec;
03456         AM.OldMilliTime = msec;
03457         return(0L);
03458     }
03459 #else
03460     struct timeb tp;
03461     ftime(&tp);
03462
03463     if ( par ) {
03464         return(((LONG)(tp.time)-AM.OldSecTime)*100 +
03465             ((LONG)(tp.millitm)-AM.OldMilliTime)/10);
03466     }
03467     else {
03468         AM.OldSecTime   = (LONG)(tp.time);
03469         AM.OldMilliTime = (LONG)(tp.millitm);
03470         return(0L);
03471     }
03472 #endif
03473 #endif
03474 }
```

```
03475
03476 /*
03477         #] TimeWallClock :
03478         #[ TimeChildren :
03479 */
03480
03481 LONG TimeChildren(WORD par)
03482 {
03483     if ( par ) return(Timer(1)-AM.OldChildTime);
03484     AM.OldChildTime = Timer(1);
03485     return(0L);
03486 }
03487
03488 /*
03489         #] TimeChildren :
03490         #[ TimeCPU :
03491 */
03492
03499 LONG TimeCPU(WORD par)
03500 {
03501     GETIDENTITY
03502     if ( par ) return(Timer(0)-AR.OldTime);
03503     AR.OldTime = Timer(0);
03504     return(0L);
03505 }
03506
03507 /*
03508         #] TimeCPU :
03509         #[ Timer :
03510 */
03511 #if defined(WINDOWS)
03512
03513 LONG Timer(int par)
03514 {
03515 #ifndef WITHPTHREADS
03516     static int initialized = 0;
03517     static HANDLE hProcess;
03518     FILETIME ftCreate, ftExit, ftKernel, ftUser;
03519     DUMMYUSE(par);
03520
03521     if ( !initialized ) {
03522         hProcess = OpenProcess(PROCESS_QUERY_INFORMATION, FALSE, GetCurrentProcessId());
03523     }
03524     if ( GetProcessTimes(hProcess, &ftCreate, &ftExit, &ftKernel, &ftUser) ) {
03525         PFILETIME pftKernel = &ftKernel;  /* to avoid strict-aliasing rule warnings */
03526         PFILETIME pftUser   = &ftUser;
03527         __int64 t = *(__int64 *)pftKernel + *(__int64 *)pftUser;  /* in 100 nsec. */
03528         return (LONG)(t / 10000);  /* in msec. */
03529     }
03530     return 0;
03531 #else
03532     LONG lResult = 0;
03533     HANDLE hThread;
03534     FILETIME ftCreate, ftExit, ftKernel, ftUser;
03535     DUMMYUSE(par);
03536
03537     hThread = OpenThread(THREAD_QUERY_INFORMATION, FALSE, GetCurrentThreadId());
03538     if ( hThread ) {
03539         if ( GetThreadTimes(hThread, &ftCreate, &ftExit, &ftKernel, &ftUser) ) {
03540             PFILETIME pftKernel = &ftKernel;  /* to avoid strict-aliasing rule warnings */
03541             PFILETIME pftUser   = &ftUser;
03542             __int64 t = *(__int64 *)pftKernel + *(__int64 *)pftUser;  /* in 100 nsec. */
03543             lResult = (LONG)(t / 10000);  /* in msec. */
03544         }
03545         CloseHandle(hThread);
03546     }
03547     return lResult;
03548 #endif
03549 }
03550
03551 #elif defined(UNIX)
03552 #include <sys/time.h>
03553 #include <sys/resource.h>
03554 #ifdef WITHPOSIXCLOCK
03555 #include <time.h>
03556 /*
03557     And include -lrt in the link statement (on blade02)
03558 */
03559 #endif
03560
03561 LONG Timer(int par)
03562 {
03563 #ifdef WITHPOSIXCLOCK
03564 /*
03565     Only to be used in combination with WITHPTHREADS
03566     This clock seems to be supported by the standard.
03567     The getrusage clock returns according to the standard only the combined
```

```
03568     time of the whole process. But in older versions of Linux LinuxThreads
03569     is used which gives a separate id to each thread and individual timings.
03570     In NPTL we get, according to the standard, one combined timing.
03571     To get individual timings we need to use
03572         clock_gettime(CLOCK_THREAD_CPUTIME_ID, &timing)
03573     with timing of the time
03574     struct timespec {
03575         time_t tv_sec;            Seconds.
03576         long   tv_nsec;           Nanoseconds.
03577     };
03578
03579 */
03580     struct timespec t;
03581     if ( par == 0 ) {
03582         if ( clock_gettime(CLOCK_THREAD_CPUTIME_ID, &t) ) {
03583 /* INTERNAL_ERROR_EXCL_START */
03584             MesPrint("!>Error in getting timing information");
03585 /* INTERNAL_ERROR_EXCL_STOP */
03586         }
03587         return (LONG)t.tv_sec * 1000 + (LONG)t.tv_nsec / 1000000;
03588     }
03589     return(0);
03590 #else
03591     struct rusage rusage;
03592     if ( par == 1 ) {
03593         getrusage(RUSAGE_CHILDREN,&rusage);
03594         return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
03595               +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
03596     }
03597     else {
03598         getrusage(RUSAGE_SELF,&rusage);
03599         return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
03600               +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
03601     }
03602 #endif
03603 }
03604
03605 #elif defined(SUN)
03606 #define _TIME_T_
03607 #include <sys/time.h>
03608 #include <sys/resource.h>
03609
03610 LONG Timer(int par)
03611 {
03612     struct rusage rusage;
03613     if ( par == 1 ) {
03614         getrusage(RUSAGE_CHILDREN,&rusage);
03615         return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
03616               +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
03617     }
03618     else {
03619         getrusage(RUSAGE_SELF,&rusage);
03620         return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
03621               +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
03622     }
03623 }
03624
03625 #elif defined(RS6K)
03626 #include <sys/time.h>
03627 #include <sys/resource.h>
03628
03629 LONG Timer(int par)
03630 {
03631     struct rusage rusage;
03632     if ( par == 1 ) {
03633         getrusage(RUSAGE_CHILDREN,&rusage);
03634         return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
03635               +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
03636     }
03637     else {
03638         getrusage(RUSAGE_SELF,&rusage);
03639         return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
03640               +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
03641     }
03642 }
03643
03644 #elif defined(ANSI)
03645 LONG Timer(int par)
03646 {
03647 #ifdef ALPHA
03648 /*  clock_t t,tikken = clock();                                    */
03649 /*  MesPrint("ALPHA-clock = %l",(LONG)tikken);                     */
03650 /*  t = tikken % CLOCKS_PER_SEC;                                   */
03651 /*  tikken /= CLOCKS_PER_SEC;                                      */
03652 /*  tikken *= 1000;                                                */
03653 /*  tikken += (t*1000)/CLOCKS_PER_SEC;                             */
03654 /*  return((LONG)tikken);                                          */
```

```
03655 /* #define _TIME_T_                                                */
03656 #include <sys/time.h>
03657 #include <sys/resource.h>
03658     struct rusage rusage;
03659     if ( par == 1 ) {
03660         getrusage(RUSAGE_CHILDREN,&rusage);
03661         return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
03662               +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
03663     }
03664     else {
03665         getrusage(RUSAGE_SELF,&rusage);
03666         return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
03667               +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
03668     }
03669 #else
03670 #ifdef DEC_STATION
03671     clock_t tikken = clock();
03672     return((LONG)tikken/1000);
03673 #else
03674     clock_t t, tikken = clock();
03675     t = tikken % CLK_TCK;
03676     tikken /= CLK_TCK;
03677     tikken *= 1000;
03678     tikken += (t*1000)/CLK_TCK;
03679     return(tikken);
03680 #endif
03681 #endif
03682 }
03683 #elif defined(VMS)
03684
03685 #include <time.h>
03686 void times(tbuffer_t *buffer);
03687
03688 LONG
03689 Timer(int par)
03690 {
03691     tbuffer_t buffer;
03692     if ( par == 1 ) { return(0); }
03693     else {
03694         times(&buffer);
03695         return(buffer.proc_user_time * 10);
03696     }
03697 }
03698
03699 #elif defined(mBSD)
03700
03701 #ifdef MICROTIME
03702 /*
03703     There is only a CP time clock in microseconds here
03704     This can cause problems with AO.wrap around
03705 */
03706 #else
03707 #ifdef mBSD2
03708 #include <sys/types.h>
03709 #include <sys/times.h>
03710 #include <time.h>
03711 LONG pretime = 0;
03712 #else
03713 #define _TIME_T_
03714 #include <sys/time.h>
03715 #include <sys/resource.h>
03716 #endif
03717 #endif
03718
03719 LONG Timer(int par)
03720 {
03721 #ifdef MICROTIME
03722     LONG t;
03723     if ( par == 1 ) { return(0); }
03724     t = clock();
03725     if ( ( AO.wrapnum & 1 ) != 0 ) t ^= 0x80000000;
03726     if ( t < 0 ) {
03727         t ^= 0x80000000;
03728         warpnum++;
03729         AO.wrap += 2147584;
03730     }
03731     return(AO.wrap+(t/1000));
03732 #else
03733 #ifdef mBSD2
03734     struct tms buffer;
03735     LONG ret;
03736     ULONG a1, a2, a3, a4;
03737     if ( par == 1 ) { return(0); }
03738     times(&buffer);
03739     a1 = (ULONG)buffer.tms_utime;
03740     a2 = a1 » 16;
03741     a3 = a1 & 0xFFFFL;
```

```
03742     a3 *= 1000;
03743     a2 = 1000*a2 + (a3 » 16);
03744     a3 &= 0xFFFFL;
03745     a4 = a2/CLK_TCK;
03746     a2 %= CLK_TCK;
03747     a3 += a2 « 16;
03748     ret = (LONG)((a4 « 16) + a3 / CLK_TCK);
03749 /*  ret = ((LONG)buffer.tms_utime * 1000)/CLK_TCK; */
03750     return(ret);
03751 #else
03752 #ifdef REALTIME
03753     struct timeval tp;
03754     struct timezone tzp;
03755     if ( par == 1 ) { return(0); }
03756     gettimeofday(&tp,&tzp); */
03757     return(tp.tv_sec*1000+tp.tv_usec/1000);
03758 #else
03759     struct rusage rusage;
03760     if ( par == 1 ) {
03761         getrusage(RUSAGE_CHILDREN,&rusage);
03762         return((rusage.ru_utime.tv_sec+rusage.ru_stime.tv_sec)*1000
03763               +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
03764     }
03765     else {
03766         getrusage(RUSAGE_SELF,&rusage);
03767         return((rusage.ru_utime.tv_sec+rusage.ru_stime.tv_sec)*1000
03768               +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
03769     }
03770 #endif
03771 #endif
03772 #endif
03773 }
03774
03775 #endif
03776
03777 /*
03778         #] Timer :
03779         #[ Crash :
03780
03781         Routine for debugging purposes
03782 */
03783
03784 int Crash(void)
03785 {
03786     int retval;
03787 #ifdef DEBUGGING
03788     int *zero = 0;
03789     retval = *zero;
03790 #else
03791     retval = 0;
03792 #endif
03793     return(retval);
03794 }
03795
03796 /*
03797         #] Crash :
03798         #[ TestTerm :
03799 */
03800
03812 int TestTerm(WORD *term)
03813 {
03814     int errorcode = 0, coeffsize;
03815     WORD *t, *tt, *tstop, *endterm, *targ, *targstop, *funstop, *argterm;
03816     endterm = term + *term;
03817     coeffsize = ABS(endterm[-1]);
03818     if ( coeffsize >= *term ) {
03819 /* INTERNAL_ERROR_EXCL_START */
03820         MLOCK(ErrorMessageLock);
03821         MesPrint("!>TestTerm: Internal inconsistency in term. Coefficient too big.");
03822         MUNLOCK(ErrorMessageLock);
03823         errorcode = 1;
03824         goto finish;
03825 /* INTERNAL_ERROR_EXCL_STOP */
03826     }
03827     if ( ( coeffsize < 3 ) || ( ( coeffsize & 1 ) != 1 ) ) {
03828 /* INTERNAL_ERROR_EXCL_START */
03829         MLOCK(ErrorMessageLock);
03830         MesPrint("!>TestTerm: Internal inconsistency in term. Wrong size coefficient.");
03831         MUNLOCK(ErrorMessageLock);
03832         errorcode = 2;
03833         goto finish;
03834 /* INTERNAL_ERROR_EXCL_STOP */
03835     }
03836     t = term+1;
03837     tstop = endterm - coeffsize;
03838     while ( t < tstop ) {
03839         switch ( *t ) {
```

```
03840             case SYMBOL:
03841             case DOTPRODUCT:
03842             case INDEX:
03843             case VECTOR:
03844             case DELTA:
03845             case HAAKJE:
03846                 break;
03847             case SNUMBER:
03848             case LNUMBER:
03849 /* INTERNAL_ERROR_EXCL_START */
03850                 MLOCK(ErrorMessageLock);
03851                 MesPrint("!>TestTerm: Internal inconsistency in term. L or S number");
03852                 MUNLOCK(ErrorMessageLock);
03853                 errorcode = 3;
03854                 goto finish;
03855                 break;
03856 /* INTERNAL_ERROR_EXCL_STOP */
03857             case EXPRESSION:
03858             case SUBEXPRESSION:
03859             case DOLLAREXPRESSION:
03860 /*
03861                 MLOCK(ErrorMessageLock);
03862                 MesPrint("TestTerm: Internal inconsistency in term. Expression survives.");
03863                 MUNLOCK(ErrorMessageLock);
03864                 errorcode = 4;
03865                 goto finish;
03866 */
03867                 break;
03868             case SETSET:
03869             case MINVECTOR:
03870             case SETEXP:
03871             case ARGFIELD:
03872 /* INTERNAL_ERROR_EXCL_START */
03873                 MLOCK(ErrorMessageLock);
03874                 MesPrint("!>TestTerm: Internal inconsistency in term. Illegal subterm.");
03875                 MUNLOCK(ErrorMessageLock);
03876                 errorcode = 5;
03877                 goto finish;
03878                 break;
03879 /* INTERNAL_ERROR_EXCL_STOP */
03880             case ARGWILD:
03881                 break;
03882             default:
03883                 if ( *t <= 0 ) {
03884 /* INTERNAL_ERROR_EXCL_START */
03885                     MLOCK(ErrorMessageLock);
03886                     MesPrint("!>TestTerm: Internal inconsistency in term. Illegal subterm number.");
03887                     MUNLOCK(ErrorMessageLock);
03888                     errorcode = 6;
03889                     goto finish;
03890 /* INTERNAL_ERROR_EXCL_STOP */
03891                 }
03892 /*
03893                 This is a regular function.
03894 */
03895                 if ( *t-FUNCTION >= NumFunctions ) {
03896 /* INTERNAL_ERROR_EXCL_START */
03897                     MLOCK(ErrorMessageLock);
03898                     MesPrint("!>TestTerm: Internal inconsistency in term. Illegal function number");
03899                     MUNLOCK(ErrorMessageLock);
03900                     errorcode = 7;
03901                     goto finish;
03902 /* INTERNAL_ERROR_EXCL_STOP */
03903                 }
03904                 funstop = t + t[1];
03905                 if ( funstop > tstop ) goto subtermsize;
03906                 if ( t[2] != 0 ) {
03907 /* INTERNAL_ERROR_EXCL_START */
03908                     MLOCK(ErrorMessageLock);
03909                     MesPrint("!>TestTerm: Internal inconsistency in term. Dirty flag nonzero.");
03910                     MUNLOCK(ErrorMessageLock);
03911                     errorcode = 8;
03912                     goto finish;
03913 /* INTERNAL_ERROR_EXCL_STOP */
03914                 }
03915                 targ = t + FUNHEAD;
03916                 if ( targ > funstop ) {
03917 /* INTERNAL_ERROR_EXCL_START */
03918                     MLOCK(ErrorMessageLock);
03919                     MesPrint("!>TestTerm: Internal inconsistency in term. Illegal function size.");
03920                     MUNLOCK(ErrorMessageLock);
03921                     errorcode = 9;
03922                     goto finish;
03923 /* INTERNAL_ERROR_EXCL_STOP */
03924                 }
03925                 if ( functions[*t-FUNCTION].spec >= TENSORFUNCTION ) {
03926                 }
```

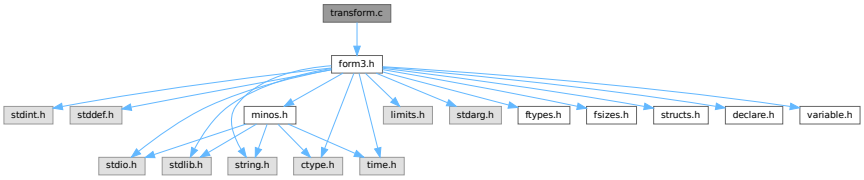
```
03927                 else {
03928                   while ( targ < funstop ) {
03929                     if ( *targ < 0 ) {
03930                         if ( *targ <= -(FUNCTION+NumFunctions) ) {
03931 /* INTERNAL_ERROR_EXCL_START */
03932                             MLOCK(ErrorMessageLock);
03933                             MesPrint("!>TestTerm: Internal inconsistency in term. Illegal function
      number in argument.");
03934                             MUNLOCK(ErrorMessageLock);
03935                             errorcode = 10;
03936                             goto finish;
03937 /* INTERNAL_ERROR_EXCL_STOP */
03938                         }
03939                         if ( *targ <= -FUNCTION ) { targ++; }
03940                         else {
03941                             if ( ( *targ != -SYMBOL ) && ( *targ != -VECTOR )
03942                             && ( *targ != -MINVECTOR )
03943                             && ( *targ != -SNUMBER )
03944                             && ( *targ != -ARGWILD )
03945                             && ( *targ != -INDEX ) ) {
03946 /* INTERNAL_ERROR_EXCL_START */
03947                                 MLOCK(ErrorMessageLock);
03948                                 MesPrint("!>TestTerm: Internal inconsistency in term. Illegal object
      in argument.");
03949                                 MUNLOCK(ErrorMessageLock);
03950                                 errorcode = 11;
03951                                 goto finish;
03952 /* INTERNAL_ERROR_EXCL_STOP */
03953                             }
03954                             targ += 2;
03955                         }
03956                     }
03957                     else if ( ( *targ < ARGHEAD ) || ( targ+*targ > funstop ) ) {
03958 /* INTERNAL_ERROR_EXCL_START */
03959                         MLOCK(ErrorMessageLock);
03960                         MesPrint("!>TestTerm: Internal inconsistency in term. Illegal size of
      argument.");
03961                         MUNLOCK(ErrorMessageLock);
03962                         errorcode = 12;
03963                         goto finish;
03964 /* INTERNAL_ERROR_EXCL_STOP */
03965                     }
03966                     else if ( targ[1] != 0 ) {
03967 /* INTERNAL_ERROR_EXCL_START */
03968                         MLOCK(ErrorMessageLock);
03969                         MesPrint("!>TestTerm: Internal inconsistency in term. Dirty flag in
      argument.");
03970                         MUNLOCK(ErrorMessageLock);
03971                         errorcode = 13;
03972                         goto finish;
03973 /* INTERNAL_ERROR_EXCL_STOP */
03974                     }
03975                     else {
03976                         targstop = targ + *targ;
03977                         argterm = targ + ARGHEAD;
03978                         while ( argterm < targstop ) {
03979                             if ( ( *argterm < 4 ) || ( argterm + *argterm > targstop ) ) {
03980 /* INTERNAL_ERROR_EXCL_START */
03981                                 MLOCK(ErrorMessageLock);
03982                                 MesPrint("!>TestTerm: Internal inconsistency in term. Illegal termsize
      in argument.");
03983                                 MUNLOCK(ErrorMessageLock);
03984                                 errorcode = 14;
03985                                 goto finish;
03986 /* INTERNAL_ERROR_EXCL_STOP */
03987                             }
03988                             if ( TestTerm(argterm) != 0 ) {
03989 /* INTERNAL_ERROR_EXCL_START */
03990                                 MLOCK(ErrorMessageLock);
03991                                 MesPrint("!>TestTerm: Internal inconsistency in term. Called from
      TestTerm.");
03992                                 MUNLOCK(ErrorMessageLock);
03993                                 errorcode = 15;
03994                                 goto finish;
03995 /* INTERNAL_ERROR_EXCL_STOP */
03996                             }
03997                             argterm += *argterm;
03998                         }
03999                         targ = targstop;
04000                     }
04001                   }
04002                 }
04003                 break;
04004         }
04005         tt = t + t[1];
04006         if ( tt > tstop ) {
04007 subtermsize:
```

```
04008 /* INTERNAL_ERROR_EXCL_START */
04009             MLOCK(ErrorMessageLock);
04010             MesPrint("!>TestTerm: Internal inconsistency in term. Illegal subterm size.");
04011             MUNLOCK(ErrorMessageLock);
04012             errorcode = 100;
04013             goto finish;
04014 /* INTERNAL_ERROR_EXCL_STOP */
04015         }
04016         t = tt;
04017     }
04018     return(errorcode);
04019 finish:
04020     return(errorcode);
04021 }
04022
04023 /*
04024         #] TestTerm :
04025         #[ DistrN :
04026 */
04027
04028 int DistrN(int n, int *cpl, int ncpl, int *scratch)
04029 {
04030 /*
04031     Divides n objects over ncpl bins (cpl), each time returning one
04032     of those distributions until there are no more after which the
04033     routine returns the value zero (otherwise one).
04034     The array scratch (size n) is kept for the intermediate information.
04035     The whole starts with scratch[0] == -2;
04036 */
04037     int i, j;
04038     if ( ncpl == 0 ) {
04039         if ( scratch[0] == -2 ) { scratch[0] = 0; return(1); }
04040         else return(0);
04041     }
04042     if ( scratch[0] == ncpl-1 ) {
04043         return(0);
04044     }
04045     else if ( scratch[0] == -2 ) {
04046         for ( i = 0; i < n; i++ ) scratch[i] = 0;
04047     }
04048     else {
04049         j = n-1;
04050         while ( j >= 0 ) {
04051             scratch[j]++;
04052             if ( scratch[j] < ncpl ) break;
04053             j--;
04054         }
04055         j++;
04056         while ( j < n ) { scratch[j] = scratch[j-1]; j++; }
04057     }
04058     for ( i = 0; i < ncpl; i++ ) cpl[i] = 0;
04059     for ( i = 0; i < n; i++ ) { cpl[scratch[i]]++; }
04060     return(1);
04061 }
04062
04063 /*
04064         #] DistrN :
04065     #] Mixed :
04066 */
```

4.151 transform.c File Reference

`#include "form3.h"`

Include dependency graph for transform.c:



Functions

- int CoTransform (UBYTE *in)
- int RunTransform (PHEAD WORD *term, WORD *params)
- int RunEncode (PHEAD WORD *fun, WORD *args, WORD *info)
- int RunDecode (PHEAD WORD *fun, WORD *args, WORD *info)
- int RunReplace (PHEAD WORD *fun, WORD *args, WORD *info)
- int RunImplode (WORD *fun, WORD *args)
- int RunExplode (PHEAD WORD *fun, WORD *args)
- int RunPermute (PHEAD WORD *fun, WORD *args, WORD *info)
- int RunReverse (PHEAD WORD *fun, WORD *args)
- int RunDedup (PHEAD WORD *fun, WORD *args)
- int RunCycle (PHEAD WORD *fun, WORD *args, WORD *info)
- int RunAddArg (PHEAD WORD *fun, WORD *args)
- int RunMulArg (PHEAD WORD *fun, WORD *args)
- int RunIsLyndon (PHEAD WORD *fun, WORD *args, int par)
- WORD RunToLyndon (PHEAD WORD *fun, WORD *args, int par)
- int RunDropArg (PHEAD WORD *fun, WORD *args)
- int RunSelectArg (PHEAD WORD *fun, WORD *args)
- int RunZtoHArg (PHEAD WORD *fun, WORD *args)
- int RunHtoZArg (PHEAD WORD *fun, WORD *args)
- int TestArgNum (int n, int totarg, WORD *args)
- WORD PutArgInScratch (WORD *arg, UWORD *scrat)
- UBYTE * ReadRange (UBYTE *s, WORD *out, int par)
- int FindRange (PHEAD WORD *args, WORD *arg1, WORD *arg2, WORD totarg)

4.151.1 Detailed Description

Routines that deal with the transform statement and its varieties.

Definition in file transform.c.

4.151.2 Function Documentation

4.151.2.1 CoTransform()

```
int CoTransform (
            UBYTE * in )
```

Definition at line 87 of file transform.c.

4.151.2.2 RunTransform()

```
int RunTransform (
            PHEAD WORD * term,
            WORD * params )
```

Definition at line 750 of file transform.c.

4.151.2.3 RunEncode()

```
int RunEncode (
            PHEAD WORD * fun,
            WORD * args,
            WORD * info )
```

Definition at line 1008 of file transform.c.

4.151.2.4 RunDecode()

```
int RunDecode (
            PHEAD WORD * fun,
            WORD * args,
            WORD * info )
```

Definition at line 1197 of file transform.c.

4.151.2.5 RunReplace()

```
int RunReplace (
            PHEAD WORD * fun,
            WORD * args,
            WORD * info )
```

Definition at line 1369 of file transform.c.

4.151.2.6 RunImplode()

```
int RunImplode (
            WORD * fun,
            WORD * args )
```

Definition at line 1849 of file transform.c.

4.151.2.7 RunExplode()

```
int RunExplode (
            PHEAD WORD * fun,
            WORD * args )
```

Definition at line 2050 of file transform.c.

4.151.2.8 RunPermute()

```
int RunPermute (
            PHEAD WORD * fun,
            WORD * args,
            WORD * info )
```

Definition at line 2164 of file transform.c.

4.151.2.9 RunReverse()

```
int RunReverse (
            PHEAD WORD * fun,
            WORD * args )
```

Definition at line 2391 of file transform.c.

4.151.2.10 RunDedup()

```
int RunDedup (
            PHEAD WORD * fun,
            WORD * args )
```

Definition at line 2478 of file transform.c.

4.151.2.11 RunCycle()

```
int RunCycle (
            PHEAD WORD * fun,
            WORD * args,
            WORD * info )
```

Definition at line 2567 of file transform.c.

4.151.2.12 RunAddArg()

```
int RunAddArg (
            PHEAD WORD * fun,
            WORD * args )
```

Definition at line 2719 of file transform.c.

4.151.2.13 RunMulArg()

```
int RunMulArg (
            PHEAD WORD * fun,
            WORD * args )
```

Definition at line 2808 of file transform.c.

4.151.2.14 RunIsLyndon()

```
int RunIsLyndon (
            PHEAD WORD * fun,
            WORD * args,
            int par )
```

Definition at line 2949 of file transform.c.

4.151.2.15 RunToLyndon()

```
WORD RunToLyndon (
            PHEAD WORD * fun,
            WORD * args,
            int par )
```

Definition at line 3027 of file transform.c.

4.151.2.16 RunDropArg()

```
int RunDropArg (
            PHEAD WORD * fun,
            WORD * args )
```

Definition at line 3139 of file transform.c.

4.151.2.17 RunSelectArg()

```
int RunSelectArg (
            PHEAD WORD * fun,
            WORD * args )
```

Definition at line 3165 of file transform.c.

4.151.2.18 RunZtoHArg()

```
int RunZtoHArg (
            PHEAD WORD * fun,
            WORD * args )
```

Definition at line 3193 of file transform.c.

4.151.2.19 RunHtoZArg()

```
int RunHtoZArg (
            PHEAD WORD * fun,
            WORD * args )
```

Definition at line 3249 of file transform.c.

4.151.2.20 TestArgNum()

```
int TestArgNum (
            int n,
            int totarg,
            WORD * args )
```

Definition at line 3315 of file transform.c.

4.151.2.21 PutArgInScratch()

```
WORD PutArgInScratch (
              WORD * arg,
              UWORD * scrat )
```

Definition at line 3384 of file transform.c.

4.151.2.22 ReadRange()

```
UBYTE * ReadRange (
              UBYTE * s,
              WORD * out,
              int par )
```

Definition at line 3420 of file transform.c.

4.151.2.23 FindRange()

```
int FindRange (
              PHEAD WORD * args,
              WORD * arg1,
              WORD * arg2,
              WORD totarg )
```

Definition at line 3580 of file transform.c.

4.152 transform.c

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
00026  *
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
00031 /*
00032     #[ Includes : transform.c
00033 */
00034
00035 #include "form3.h"
00036
```

```
00037 /*
00038     #] Includes :
00039     #[ Transform :
00040         #[ Intro :
00041
00042         Here are the routines for the transform statement. This is a
00043         group of transformations on function arguments or groups of
00044         function arguments. The purpose of this command is that it
00045         avoids repetitive pattern matching.
00046         Syntax:
00047             Transform,SetOfFunctions,OneOrMoreTransformations;
00048         Each transformation is given by
00049             Replace(argfirst,arglast)=(„,)
00050             Encode(argfirst,arglast):base=#
00051             Decode(argfirst,arglast):base=#
00052             Implode(argfirst,arglast)
00053             Explode(argfirst,arglast)
00054             Permute(cycle)(cycle)(cycle)...(cycle)
00055             Reverse(argfirst,arglast)
00056             Dedup(argfirst,arglast)
00057             Cycle(argfirst,arglast)=+/-num
00058             IsLyndon(argfirst,arglast)=(yes,no)
00059             ToLyndon(argfirst,arglast)=(yes,no)
00060         In replace the extra information is
00061             a replace_() without the name of the replace_ function.
00062             This can be as in (0,1,1,0) or (xarg_,1-xarg_) to indicate
00063             a symbolic argument or (x,y,y,x) to exchange x and y, etc.
00064         In Encode and Decode argfirst is the most significant 'word' and
00065         arglast is the least significant 'word'.
00066         Note that we need to introduce the generic symbolic arguments xarg_,
00067         parg_, iarg_ and farg_.
00068         Examples:
00069             Transform,{H,E}
00070                     ,Replace(1:'WEIGHT')=(0,1,1,0)
00071                     ,Encode(1:'WEIGHT')=base(2);
00072             Transform,{H,E}
00073                     ,Decode(1:'WEIGHT')=base(3)
00074                     ,Replace(1:'WEIGHT')=(2,-1,1,0,0,1);
00075         Others that can be added:
00076             symmetrize?
00077
00078         6-may-2016: Changed MAXPOSITIVE2 into MAXPOSITIVE4. This makes room
00079                     for the use of dollar variables as arguments.
00080
00081         #] Intro :
00082         #[ CoTransform :
00083 */
00084
00085 static WORD tranarray[10] = { SUBEXPRESSION, SUBEXPSIZE, 0, 1, 0, 0, 0, 0, 0, 0 };
00086
00087 int CoTransform(UBYTE *in)
00088 {
00089     GETIDENTITY
00090     UBYTE *s = in, c, *ss, *Tempbuf;
00091     WORD number, type, i, *work = AT.WorkPointer+2, *wp, range[2], one = 1;
00092     WORD numdol, *wstart;
00093     int error = 0, irhs;
00094     LONG x;
00095     while ( *in == ',' ) in++;
00096     wp = work + 1;
00097 /*
00098     #[ Sets :
00099
00100     First the set specification(s). No sets means all functions (dangerous!)
00101 */
00102     for(;;) {
00103         if ( *in == '{' ) {
00104             s = in+1;
00105             SKIPBRA2(in)
00106             number = DoTempSet(s,in);
00107             in++;
00108             if ( *in != ',' ) {
00109                 c = in[1]; in[1] = 0;
00110                 MesPrint("& %s: A set in a transform statement should be followed by a comma",s);
00111                 in[1] = c; in++;
00112                 if ( error == 0 ) error = 1;
00113             }
00114         }
00115         else if ( *in == '[' || FG.cTable[*in] == 0 ) {
00116             s = in;
00117             in = SkipAName(in);
00118             if ( *in != ',' ) break;
00119             c = *in; *in = 0;
00120             type = GetName(AC.varnames,s,&number,NOAUTO);
00121             if ( type == CFUNCTION ) {
00122 #ifdef WITHFLOAT
00123                 if ( (number+FUNCTION) == FLOATFUN ) {
```

```
00124                       MesPrint("&Illegal use of a transform statement and float_");
00125                       if ( error == 0 ) error = 1;
00126                   }
00127 #endif
00128                   number += MAXVARIABLES + FUNCTION; }
00129               else if ( type != CSET ) {
00130                   MesPrint("& %s: A transform statement starts with sets of functions",s);
00131                   if ( error == 0 ) error = 1;
00132               }
00133               *in++ = c;
00134           }
00135           else {
00136               MesPrint("&Illegal syntax in Transform statement",s);
00137               if ( error == 0 ) error = 1;
00138               return(error);
00139           }
00140           if ( number >= 0 ) {
00141             if ( number < MAXVARIABLES ) {
00142 /*
00143               Check that this is a set of functions
00144 */
00145               if ( Sets[number].type != CFUNCTION ) {
00146                   MesPrint("&A set in a transform statement should be a set of functions");
00147                   if ( error == 0 ) error = 1;
00148               }
00149 #ifdef WITHFLOAT
00150               WORD *r1, *r2;
00151               r1 = SetElements + Sets[number].first;
00152               r2 = SetElements + Sets[number].last;
00153               while ( r1 < r2 ) {
00154                   if ( *r1++ == FLOATFUN ) {
00155                       MesPrint("&Illegal use of a transform statement and float_");
00156                       if ( error == 0 ) error = 1;
00157                   }
00158               }
00159 #endif
00160             }
00161           }
00162           else if ( error == 0 ) error = 1;
00163 /*
00164           Now write the number to the right place
00165 */
00166           *wp++ = number;
00167           while ( *in == ',' ) in++;
00168       }
00169       *work = wp - work;
00170       work = wp; wp++;
00171 /*
00172       #] Sets :
00173
00174       Now we should loop over the various transformations
00175 */
00176       while ( *s ) {
00177           in = s;
00178           if ( FG.cTable[*in] != 0 ) {
00179               MesPrint("&Illegal character in Transform statement");
00180               if ( error == 0 ) error = 1;
00181               return(error);
00182           }
00183           in = SkipAName(in);
00184           if ( *in == '>' || *in == '<' || *in == '+' || *in == '-' ) in++;
00185           ss = in;
00186           c = *ss; *ss = 0;
00187           if ( c != '(' ) {
00188               MesPrint("&Illegal syntax in specifying a transformation inside a Transform statement");
00189               if ( error == 0 ) error = 1;
00190               return(error);
00191           }
00192 /*
00193           #[ replace :
00194 */
00195           if ( StrICmp(s,(UBYTE *)"replace") == 0 ) {
00196 /*
00197                   Subkeys: (,,) as in replace_(,,)
00198               The idea here is to read the subkeys as the argument
00199               of a replace_ function.
00200               We put the whole together as in the multiply statement (which
00201               could just be a replace_(....)) and compile it.
00202               Then we expand the tree with Generator and check the complete
00203               expression for legality.
00204 */
00205               type = REPLACEARG;
00206 doreplace:
00207               *ss = c;
00208               if ( ( in = ReadRange(in,range,0) ) == 0 ) {
00209                   if ( error == 0 ) error = 1;
00210                   return(error);
```

```
00211             }
00212             in++;
00213 /*
00214             We have replace(#,#)=(...), and we want dum_(...)  (DUMFUN)
00215             to send to the compiler. The pointer is after the '=';
00216 */
00217             s = in;
00218             if ( *s != '(' ) {
00219                 MesPrint("&");
00220                 if ( error == 0 ) error = 1;
00221                 return(error);
00222             }
00223             SKIPBRA3(in);
00224             if ( *in != ')' ) {
00225                 MesPrint("&");
00226                 if ( error == 0 ) error = 1;
00227                 return(error);
00228             }
00229             in++;
00230             if ( *in != ',' && *in != '\0' ) {
00231                 MesPrint("&");
00232                 if ( error == 0 ) error = 1;
00233                 return(error);
00234             }
00235             i = in - s;
00236             ss = Tempbuf = (UBYTE *)Malloc1(i+5,"CoTransform/replace");
00237             *ss++ = 'd'; *ss++ = 'u'; *ss++ = 'm'; *ss++ = '_';
00238             NCOPY(ss,s,i)
00239             *ss++ = 0;
00240             AC.ProtoType = tranarray;
00241             tranarray[4] = AC.cbufnum;
00242             irhs = CompileAlgebra(Tempbuf,RHSIDE,AC.ProtoType);
00243             M_free(Tempbuf,"CoTransform/replace");
00244             if ( irhs < 0 ) {
00245                 if ( error == 0 ) error = 1;
00246                 return(error);
00247             }
00248             tranarray[2] = irhs;
00249 /*
00250             The result of the compilation goes through Generator during
00251             execution, because that takes care of $-variables.
00252             This is why we could not use replace_ and had to use dum_.
00253 */
00254             *wp++ = ARGRANGE;
00255             *wp++ = range[0];
00256             *wp++ = range[1];
00257             *wp++ = type;
00258             *wp++ = SUBEXPSIZE+4;
00259             for ( i = 0; i < SUBEXPSIZE; i++ ) *wp++ = tranarray[i];
00260             *wp++ = 1;
00261             *wp++ = 1;
00262             *wp++ = 3;
00263             *work = wp-work;
00264             work = wp; *wp++ = 0;
00265             s = in;
00266         }
00267 /*
00268         #] replace :
00269         #[ encode/decode :
00270 */
00271         else if ( StrICmp(s,(UBYTE *)"decode" ) == 0 ) {
00272             type = DECODEARG;
00273             goto doencode;
00274         }
00275         else if ( StrICmp(s,(UBYTE *)"encode" ) == 0 ) {
00276             type = ENCODEARG;
00277 doencode:   *ss = c;
00278             if ( ( in = ReadRange(in,range,2) ) == 0 ) {
00279                 if ( error == 0 ) error = 1;
00280                 return(error);
00281             }
00282             in++;
00283             s = in; while ( FG.cTable[*in] == 0 ) in++;
00284             c = *in; *in = 0;
00285 /*
00286                 Subkeys: base=# or base=$var
00287 */
00288             if ( StrICmp(s,(UBYTE *)"base") == 0 ) {
00289                 *in = c;
00290                 if ( *in != '=' ) {
00291                     MesPrint("&Illegal base specification in encode/decode transformation");
00292                     if ( error == 0 ) error = 1;
00293                     return(error);
00294                 }
00295                 in++;
00296                 if ( *in == '$' ) {
00297                     in++; ss = in;
```

```
00298                     in = SkipAName(in);
00299                     c = *in; *in = 0;
00300                     if ( GetName(AC.dollarnames,ss,&numdol,NOAUTO) != CDOLLAR ) {
00301                         MesPrint("&%s is undefined",ss-1);
00302                         numdol = AddDollar(ss,DOLINDEX,&one,1);
00303                         return(1);
00304                     }
00305                     *in = c;
00306                     x = -numdol;
00307                 }
00308                 else {
00309                     x = 0;
00310                     while ( FG.cTable[*in] == 1 ) {
00311                         x = 10*x + *in++ - '0';
00312                         if ( x > MAXPOSITIVE4 ) {
00313 illsize:                    MesPrint("&Illegal value for base in encode/decode transformation");
00314                             if ( error == 0 ) error = 1;
00315                             return(error);
00316                         }
00317                     }
00318                     if ( x <= 1 ) goto illsize;
00319                 }
00320                 if ( *in != ',' && *in != '\0' ) {
00321                     MesPrint("&Illegal termination of transformation");
00322                     if ( error == 0 ) error = 1;
00323                     return(error);
00324                 }
00325             }
00326             else {
00327                 MesPrint("&Illegal option in encode/decode transformation");
00328                 if ( error == 0 ) error = 1;
00329                 return(error);
00330             }
00331 /*
00332             Now we can put the whole statement together
00333             We have the set(s) in work up to wp and the range in range.
00334             The base is in x and the type tells whether it is encode or decode.
00335 */
00336             *wp++ = ARGRANGE;
00337             *wp++ = range[0];
00338             *wp++ = range[1];
00339             *wp++ = type;
00340             *wp++ = 4;
00341             *wp++ = BASECODE;
00342             *wp++ = (WORD)x;
00343             *work = wp-work;
00344             work = wp; *wp++ = 0;
00345             s = in;
00346         }
00347 /*
00348         #] encode/decode :
00349         #[ implode :
00350 */
00351         else if ( StrICmp(s,(UBYTE *)"implode") == 0
00352                 || StrICmp(s,(UBYTE *)"tosumnotation") == 0 ) {
00353 /*
00354                 Subkeys: ?
00355 */
00356             type = IMPLODEARG;
00357             *ss = c;
00358             if ( ( in = ReadRange(in,range,1) ) == 0 ) {
00359                 if ( error == 0 ) error = 1;
00360                 return(error);
00361             }
00362             *wp++ = ARGRANGE;
00363             *wp++ = range[0];
00364             *wp++ = range[1];
00365             *wp++ = type;
00366             *work = wp-work;
00367             work = wp; *wp++ = 0;
00368             s = in;
00369         }
00370 /*
00371         #] implode :
00372         #[ explode :
00373 */
00374         else if ( StrICmp(s,(UBYTE *)"explode") == 0
00375                 || StrICmp(s,(UBYTE *)"tointegralnotation") == 0 ) {
00376 /*
00377                 Subkeys: ?
00378 */
00379             type = EXPLODEARG;
00380             *ss = c;
00381             if ( ( in = ReadRange(in,range,1) ) == 0 ) {
00382                 if ( error == 0 ) error = 1;
00383                 return(error);
00384             }
```

```
00385             *wp++ = ARGRANGE;
00386             *wp++ = range[0];
00387             *wp++ = range[1];
00388             *wp++ = type;
00389             *work = wp-work;
00390             work = wp; *wp++ = 0;
00391             s = in;
00392         }
00393 /*
00394         #] explode :
00395         #[ permute :
00396 */
00397         else if ( StrICmp(s,(UBYTE *)"permute") == 0 ) {
00398             type = PERMUTEARG;
00399             *ss = c;
00400             *wp++ = ARGRANGE;
00401             *wp++ = 1;
00402             *wp++ = MAXPOSITIVE4;
00403             *wp++ = type;
00404 /*
00405             Now a sequence of cycles
00406 */
00407             do {
00408               wstart = wp; wp++;
00409               do {
00410                 in++;
00411                 if ( *in == '$' ) {
00412                     WORD number; UBYTE *t;
00413                     in++; t = in;
00414                     while ( FG.cTable[*in] < 2 ) in++;
00415                     c = *in; *in = 0;
00416                     if ( ( number = GetDollar(t) ) < 0 ) {
00417                         MesPrint("&Undefined variable $%s",t);
00418                         if ( !error ) error = 1;
00419                         number = AddDollar(t,0,0,0);
00420                     }
00421                     *in = c;
00422                     *wp++ = -number-1;
00423                 }
00424                 else {
00425                   x = 0;
00426                   while ( FG.cTable[*in] == 1 ) {
00427                     x = 10*x + *in++ - '0';
00428                     if ( x > MAXPOSITIVE4 ) {
00429                         MesPrint("&value in permute transformation too large");
00430                         if ( error == 0 ) error = 1;
00431                         return(error);
00432                     }
00433                   }
00434                   if ( x == 0 ) {
00435                     MesPrint("&value 0 in permute transformation not allowed");
00436                     if ( error == 0 ) error = 1;
00437                     return(error);
00438                   }
00439                   *wp++ = (WORD)x-1;
00440                 }
00441               } while ( *in == ',' );
00442               if ( *in != ')' ) {
00443                 MesPrint("&Illegal syntax in permute transformation");
00444                 if ( error == 0 ) error = 1;
00445                 return(error);
00446               }
00447               in++;
00448               if ( *in != ',' && *in != '(' && *in != '\0' ) {
00449                 MesPrint("&Illegal ending in permute transformation");
00450                 if ( error == 0 ) error = 1;
00451                 return(error);
00452               }
00453               *wstart = wp-wstart;
00454               if ( *wstart == 1 ) wstart--;
00455             } while ( *in == '(' );
00456             *work = wp-work;
00457             work = wp; *wp++ = 0;
00458             s = in;
00459         }
00460 /*
00461         #] permute :
00462         #[ reverse :
00463 */
00464         else if ( StrICmp(s,(UBYTE *)"reverse") == 0 ) {
00465             type = REVERSEARG;
00466             *ss = c;
00467             if ( ( in = ReadRange(in,range,1) ) == 0 ) {
00468                 if ( error == 0 ) error = 1;
00469                 return(error);
00470             }
00471             *wp++ = ARGRANGE;
```

```
00472             *wp++ = range[0];
00473             *wp++ = range[1];
00474             *wp++ = type;
00475             *work = wp-work;
00476             work = wp; *wp++ = 0;
00477             s = in;
00478         }
00479 /*
00480         #] reverse :
00481         #[ dedup :
00482 */
00483         else if ( StrICmp(s,(UBYTE *)"dedup") == 0 ) {
00484             type = DEDUPARG;
00485             *ss = c;
00486             if ( ( in = ReadRange(in,range,1) ) == 0 ) {
00487                 if ( error == 0 ) error = 1;
00488                 return(error);
00489             }
00490             *wp++ = ARGRANGE;
00491             *wp++ = range[0];
00492             *wp++ = range[1];
00493             *wp++ = type;
00494             *work = wp-work;
00495             work = wp; *wp++ = 0;
00496             s = in;
00497         }
00498 /*
00499         #] dedup :
00500         #[ cycle :
00501 */
00502         else if ( StrICmp(s,(UBYTE *)"cycle") == 0 ) {
00503             type = CYCLEARG;
00504             *ss = c;
00505             if ( ( in = ReadRange(in,range,0) ) == 0 ) {
00506                 if ( error == 0 ) error = 1;
00507                 return(error);
00508             }
00509             *wp++ = ARGRANGE;
00510             *wp++ = range[0];
00511             *wp++ = range[1];
00512             *wp++ = type;
00513 /*
00514             Now a sequence of cycles
00515 */
00516             in++;
00517             if ( *in == '+' ) {
00518             }
00519             else if ( *in == '-' ) {
00520                 one = -1;
00521             }
00522             else {
00523                 MesPrint("&Cycle in a Transform statement should be followed by =+/-number/$");
00524                 if ( error == 0 ) error = 1;
00525                 return(error);
00526             }
00527             in++; x = 0;
00528             if ( *in == '$' ) {
00529                 UBYTE *si = in;
00530                 in++; si = in;
00531                 while ( FG.cTable[*in] == 0 || FG.cTable[*in] == 1 ) in++;
00532                 c = *in; *in = 0;
00533                 if ( ( x = GetDollar(si) ) < 0 ) {
00534                     MesPrint("&Undefined $-variable in transform,cycle statement.");
00535                     error = 1;
00536                 }
00537                 *in = c;
00538                 if ( one < 0 ) x += MAXPOSITIVE4;
00539                 x += MAXPOSITIVE2;
00540                 *wp++ = x;
00541             }
00542             else {
00543               while ( FG.cTable[*in] == 1 ) {
00544                 x = 10*x + *in++ - '0';
00545                 if ( x > MAXPOSITIVE4 ) {
00546                     MesPrint("&Number in cycle in a Transform statement too big");
00547                     if ( error == 0 ) error = 1;
00548                     return(error);
00549                 }
00550               }
00551               *wp++ = x*one;
00552             }
00553             *work = wp-work;
00554             work = wp; *wp++ = 0;
00555             s = in;
00556         }
00557 /*
00558         #] cycle :
```

```
00559         #[ islyndon/tolyndon :
00560 */
00561         else if ( StrICmp(s,(UBYTE *)"islyndon" ) == 0 ) {
00562             type = ISLYNDON;
00563             goto doreplace;
00564         }
00565         else if ( StrICmp(s,(UBYTE *)"islyndon<" ) == 0 ) {
00566             type = ISLYNDON;
00567             goto doreplace;
00568         }
00569         else if ( StrICmp(s,(UBYTE *)"islyndon-" ) == 0 ) {
00570             type = ISLYNDON;
00571             goto doreplace;
00572         }
00573         else if ( StrICmp(s,(UBYTE *)"islyndon>" ) == 0 ) {
00574             type = ISLYNDONR;
00575             goto doreplace;
00576         }
00577         else if ( StrICmp(s,(UBYTE *)"islyndon+" ) == 0 ) {
00578             type = ISLYNDONR;
00579             goto doreplace;
00580         }
00581         else if ( StrICmp(s,(UBYTE *)"tolyndon" ) == 0 ) {
00582             type = TOLYNDON;
00583             goto doreplace;
00584         }
00585         else if ( StrICmp(s,(UBYTE *)"tolyndon<" ) == 0 ) {
00586             type = TOLYNDON;
00587             goto doreplace;
00588         }
00589         else if ( StrICmp(s,(UBYTE *)"tolyndon-" ) == 0 ) {
00590             type = TOLYNDON;
00591             goto doreplace;
00592         }
00593         else if ( StrICmp(s,(UBYTE *)"tolyndon>" ) == 0 ) {
00594             type = TOLYNDONR;
00595             goto doreplace;
00596         }
00597         else if ( StrICmp(s,(UBYTE *)"tolyndon+" ) == 0 ) {
00598             type = TOLYNDONR;
00599             goto doreplace;
00600         }
00601 /*
00602         #] islyndon/tolyndon :
00603         #[ addarg :
00604 */
00605         else if ( StrICmp(s,(UBYTE *)"addargs" ) == 0 ) {
00606             type = ADDARG;
00607             *ss = c;
00608             if ( ( in = ReadRange(in,range,1) ) == 0 ) {
00609                 if ( error == 0 ) error = 1;
00610                 return(error);
00611             }
00612             *wp++ = ARGRANGE;
00613             *wp++ = range[0];
00614             *wp++ = range[1];
00615             *wp++ = type;
00616             *work = wp-work;
00617             work = wp; *wp++ = 0;
00618             s = in;
00619         }
00620 /*
00621         #] addarg :
00622         #[ mularg :
00623 */
00624         else if ( ( StrICmp(s,(UBYTE *)"mulargs" ) == 0 )
00625                 || ( StrICmp(s,(UBYTE *)"multiplyargs" ) == 0 ) ) {
00626             type = MULTIPLYARG;
00627             *ss = c;
00628             if ( ( in = ReadRange(in,range,1) ) == 0 ) {
00629                 if ( error == 0 ) error = 1;
00630                 return(error);
00631             }
00632             *wp++ = ARGRANGE;
00633             *wp++ = range[0];
00634             *wp++ = range[1];
00635             *wp++ = type;
00636             *work = wp-work;
00637             work = wp; *wp++ = 0;
00638             s = in;
00639         }
00640 /*
00641         #] mularg :
00642         #[ droparg :
00643 */
00644         else if ( StrICmp(s,(UBYTE *)"dropargs" ) == 0 ) {
00645             type = DROPARG;
```

```
00646             *ss = c;
00647             if ( ( in = ReadRange(in,range,1) ) == 0 ) {
00648                 if ( error == 0 ) error = 1;
00649                 return(error);
00650             }
00651             *wp++ = ARGRANGE;
00652             *wp++ = range[0];
00653             *wp++ = range[1];
00654             *wp++ = type;
00655             *work = wp-work;
00656             work = wp; *wp++ = 0;
00657             s = in;
00658         }
00659 /*
00660         #] droparg :
00661         #[ selectarg :
00662 */
00663         else if ( StrICmp(s,(UBYTE *)"selectargs" ) == 0 ) {
00664             type = SELECTARG;
00665             *ss = c;
00666             if ( ( in = ReadRange(in,range,1) ) == 0 ) {
00667                 if ( error == 0 ) error = 1;
00668                 return(error);
00669             }
00670             *wp++ = ARGRANGE;
00671             *wp++ = range[0];
00672             *wp++ = range[1];
00673             *wp++ = type;
00674             *work = wp-work;
00675             work = wp; *wp++ = 0;
00676             s = in;
00677         }
00678 /*
00679         #] selectarg :
00680         #[ ZtoH :
00681 */
00682         else if ( StrICmp(s,(UBYTE *)"ztoh") == 0 ) {
00683 /*
00684                 Subkeys: ?
00685 */
00686             type = ZTOHARG;
00687             *ss = c;
00688             if ( ( in = ReadRange(in,range,1) ) == 0 ) {
00689                 if ( error == 0 ) error = 1;
00690                 return(error);
00691             }
00692             *wp++ = ARGRANGE;
00693             *wp++ = range[0];
00694             *wp++ = range[1];
00695             *wp++ = type;
00696             *work = wp-work;
00697             work = wp; *wp++ = 0;
00698             s = in;
00699         }
00700 /*
00701         #] ZtoH :
00702         #[ HtoZ :
00703 */
00704         else if ( StrICmp(s,(UBYTE *)"htoz") == 0 ) {
00705 /*
00706                 Subkeys: ?
00707 */
00708             type = HTOZARG;
00709             *ss = c;
00710             if ( ( in = ReadRange(in,range,1) ) == 0 ) {
00711                 if ( error == 0 ) error = 1;
00712                 return(error);
00713             }
00714             *wp++ = ARGRANGE;
00715             *wp++ = range[0];
00716             *wp++ = range[1];
00717             *wp++ = type;
00718             *work = wp-work;
00719             work = wp; *wp++ = 0;
00720             s = in;
00721         }
00722 /*
00723         #] HtoZ :
00724 */
00725         else {
00726             MesPrint("&Unknown transformation inside a Transform statement: %s",s);
00727             *ss = c;
00728             if ( error == 0 ) error = 1;
00729             return(error);
00730         }
00731         while ( *s == ',') s++;
00732     }
```

```
00733     AT.WorkPointer[0] = TYPETRANSFORM;
00734     AT.WorkPointer[1] = i = wp - AT.WorkPointer;
00735     AddNtoL(i,AT.WorkPointer);
00736     return(error);
00737 }
00738
00739 /*
00740         #] CoTransform :
00741         #[ RunTransform :
00742
00743         Executes the transform statement.
00744         This routine hunts down the functions and sends them to the various
00745         action routines.
00746         params: size,#set1,...,#setn, transformations
00747
00748 */
00749
00750 int RunTransform(PHEAD WORD *term, WORD *params)
00751 {
00752     WORD *t, *tstop, *w, *m, *out, *in, *tt, retval;
00753     WORD *fun, *args, *info, *infoend, *onetransform, *funs, *endfun;
00754     WORD *thearg = 0, *iterm, *newterm, *nt, *oldwork = AT.WorkPointer, sign = 1;
00755     int i;
00756     out = tstop = term + *term;
00757     tstop -= ABS(tstop[-1]);
00758     in = term;
00759     t = term + 1;
00760     while ( t < tstop ) {
00761         endfun = onetransform = params + *params;
00762         funs = params + 1;
00763         if ( *t < FUNCTION ) {}
00764         else if ( funs == endfun ) {  /* we do all functions */
00765 hit:;
00766 #ifdef WITHFLOAT
00767             if ( *t == FLOATFUN ) goto next;
00768 #endif
00769             while ( in < t ) *out++ = *in++;
00770             tt = t + t[1]; fun = out;
00771             while ( in < tt ) *out++ = *in++;
00772             do {
00773                 args = onetransform + 1;
00774                 info = args; while ( *info <= MAXRANGEINDICATOR ) {
00775                     if ( *info == ALLARGS ) info++;
00776                     else if ( *info == NUMARG ) info += 2;
00777                     else if ( *info == ARGRANGE ) info += 3;
00778                     else if ( *info == MAKEARGS ) info += 3;
00779                 }
00780                 switch ( *info ) {
00781                     case REPLACEARG:
00782                         if ( RunReplace(BHEAD fun,args,info) ) goto abo;
00783                         out = fun + fun[1];
00784                         break;
00785                     case ENCODEARG:
00786                         if ( RunEncode(BHEAD fun,args,info) ) goto abo;
00787                         out = fun + fun[1];
00788                         break;
00789                     case DECODEARG:
00790                         if ( RunDecode(BHEAD fun,args,info) ) goto abo;
00791                         out = fun + fun[1];
00792                         break;
00793                     case IMPLODEARG:
00794                         if ( RunImplode(fun,args) ) goto abo;
00795                         out = fun + fun[1];
00796                         break;
00797                     case EXPLODEARG:
00798                         if ( RunExplode(BHEAD fun,args) ) goto abo;
00799                         out = fun + fun[1];
00800                         break;
00801                     case PERMUTEARG:
00802                         if ( RunPermute(BHEAD fun,args,info) ) goto abo;
00803                         out = fun + fun[1];
00804                         break;
00805                     case REVERSEARG:
00806                         if ( RunReverse(BHEAD fun,args) ) goto abo;
00807                         out = fun + fun[1];
00808                         break;
00809                     case DEDUPARG:
00810                         if ( RunDedup(BHEAD fun,args) ) goto abo;
00811                         out = fun + fun[1];
00812                         break;
00813                     case CYCLEARG:
00814                         if ( RunCycle(BHEAD fun,args,info) ) goto abo;
00815                         out = fun + fun[1];
00816                         break;
00817                     case ADDARG:
00818                         if ( RunAddArg(BHEAD fun,args) ) goto abo;
00819                         out = fun + fun[1];
```

```
00820                       break;
00821                   case MULTIPLYARG:
00822                       if ( RunMulArg(BHEAD fun,args) ) goto abo;
00823                       out = fun + fun[1];
00824                       break;
00825                   case ISLYNDON:
00826                       if ( ( retval = RunIsLyndon(BHEAD fun,args,1) ) < -1 ) goto abo;
00827                       goto returnvalues;
00828                       break;
00829                   case ISLYNDONR:
00830                       if ( ( retval = RunIsLyndon(BHEAD fun,args,-1) ) < -1 ) goto abo;
00831                       goto returnvalues;
00832                       break;
00833                   case TOLYNDON:
00834                       if ( ( retval = RunToLyndon(BHEAD fun,args,1) ) < -1 ) goto abo;
00835                       goto returnvalues;
00836                       break;
00837                   case TOLYNDONR:
00838                       if ( ( retval = RunToLyndon(BHEAD fun,args,-1) ) < -1 ) goto abo;
00839 returnvalues:;
00840                       out = fun + fun[1];
00841                       if ( retval == -1 ) break;
00842 /*
00843                       Work out the yes/no stuff
00844 */
00845                       AT.WorkPointer += 2*AM.MaxTer;
00846                       if ( AT.WorkPointer > AT.WorkTop ) {
00847                           MLOCK(ErrorMessageLock);
00848                           MesWork();
00849                           MUNLOCK(ErrorMessageLock);
00850                           return(-1);
00851                       }
00852                       iterm = AT.WorkPointer;
00853                       info++;
00854                       for ( i = 0; i < *info; i++ ) iterm[i] = info[i];
00855                       AT.WorkPointer = iterm + *iterm;
00856                       AR.Eside = LHSIDEX;
00857                       NewSort(BHEAD0);
00858                       if ( Generator(BHEAD iterm,AR.Cnumlhs) ) {
00859                           LowerSortLevel();
00860                           AT.WorkPointer = oldwork;
00861                           return(-1);
00862                       }
00863                       newterm = AT.WorkPointer;
00864                       if ( EndSort(BHEAD newterm,1) < 0 ) {}
00865                       if ( ( *newterm && *(newterm+*newterm) != 0 ) || *newterm == 0 ) {
00866                           MLOCK(ErrorMessageLock);
00867                           MesPrint("&yes/no information in islyndon/tolyndon does not evaluate into
      a single term");
00868                           MUNLOCK(ErrorMessageLock);
00869                           return(-1);
00870                       }
00871                       AR.Eside = RHSIDE;
00872                       i = *newterm; tt = iterm; nt = newterm;
00873                       NCOPY(tt,nt,i);
00874                       AT.WorkPointer = iterm + *iterm;
00875                       info = iterm + 1;
00876                       infoend = info+info[1];
00877                       info += FUNHEAD;
00878
00879                       if ( retval == 0 ) {
00880 /*
00881                           Need second argument (=no)
00882 */
00883                           if ( info >= infoend ) {
00884 abortlyndon:;
00885                               MLOCK(ErrorMessageLock);
00886                               MesPrint("There should be a yes and a no argument in
      islyndon/tolyndon");
00887                               MUNLOCK(ErrorMessageLock);
00888                               Terminate(-1);
00889                           }
00890                           NEXTARG(info)
00891                           if ( info >= infoend ) goto abortlyndon;
00892                           thearg = info;
00893                       }
00894                       else if ( retval == 1 ) {
00895 /*
00896                           Need first argument (=yes)
00897 */
00898                           if ( info >= infoend ) goto abortlyndon;
00899                           thearg = info;
00900                           NEXTARG(info)
00901                           if ( info >= infoend ) goto abortlyndon;
00902                       }
00903                       NEXTARG(info)
00904                       if ( info < infoend ) goto abortlyndon;
```

```
00905 /*
00906                         The argument in thearg needs to be copied
00907                         We did not pull it through generator to guarantee
00908                         that it is a single argument.
00909                         The easiest way is to let the routine Normalize
00910                         do the job and put everything in an exponent function
00911                         with the power one.
00912 */
00913                         if ( *thearg == -SNUMBER && thearg[1] == 0 ) {
00914                             *term = 0; return(0);
00915                         }
00916                         if ( *thearg == -SNUMBER && thearg[1] == 1 ) { }
00917                         else {
00918                             fun = out;
00919                             *out++ = EXPONENT; out++; *out++ = 1; FILLFUN3(out);
00920                             COPY1ARG(out,thearg);
00921                             *out++ = -SNUMBER; *out++ = 1;
00922                             fun[1] = out-fun;
00923                         }
00924                         break;
00925                     case DROPARG:
00926                         if ( RunDropArg(BHEAD fun,args) ) goto abo;
00927                         out = fun + fun[1];
00928                         break;
00929                     case SELECTARG:
00930                         if ( RunSelectArg(BHEAD fun,args) ) goto abo;
00931                         out = fun + fun[1];
00932                         break;
00933                     case ZTOHARG:
00934                         {
00935                           WORD s = RunZtoHArg(BHEAD fun,args);
00936                           if ( s < 0 ) goto abo;
00937                           if ( s == 1 ) sign = -sign;
00938                           out = fun + fun[1];
00939                         }
00940                         break;
00941                     case HTOZARG:
00942                         {
00943                           WORD s = RunHtoZArg(BHEAD fun,args);
00944                           if ( s < 0 ) goto abo;
00945                           if ( s == 1 ) sign = -sign;
00946                           out = fun + fun[1];
00947                         }
00948                         break;
00949                     default:
00950 /* INTERNAL_ERROR_EXCL_START */
00951                         MLOCK(ErrorMessageLock);
00952                         MesPrint("!>Irregular code in execution of transform statement");
00953                         MUNLOCK(ErrorMessageLock);
00954                         Terminate(-1);
00955 /* INTERNAL_ERROR_EXCL_STOP */
00956                 }
00957                 onetransform += *onetransform;
00958             } while ( *onetransform );
00959         }
00960         else {
00961           while ( funs < endfun ) {  /* sum over sets */
00962             if ( *funs > MAXVARIABLES ) {
00963                 if ( *t == *funs-MAXVARIABLES ) goto hit;
00964             }
00965             else {
00966               w = SetElements + Sets[*funs].first;
00967               m = SetElements + Sets[*funs].last;
00968               while ( w < m ) {  /* sum over set elements */
00969                 if ( *w == *t ) goto hit;
00970                 w++;
00971               }
00972             }
00973             funs++;
00974           }
00975         }
00976 #ifdef WITHFLOAT
00977 next:
00978 #endif
00979         t += t[1];
00980     }
00981     tt = term + *term; while ( in < tt ) *out++ = *in++;
00982     if ( sign == -1 ) out[-1] = -out[-1];
00983     *tt = i = out - tt;
00984 /*
00985     Now copy the whole thing back
00986 */
00987     NCOPY(term,tt,i)
00988     return(0);
00989 abo:
00990     MLOCK(ErrorMessageLock);
00991     MesCall("RunTransform");
```

```
00992     MUNLOCK(ErrorMessageLock);
00993     return(-1);
00994 }
00995
00996 /*
00997         #] RunTransform :
00998         #[ RunEncode :
00999
01000         The info is given by
01001             ENCODEARG,size,BASECODE,num
01002         and possibly more codes to follow.
01003         Only one range is allowed and for now, it should be fully numerical
01004         If the range is in reverse order, we need to either revert it
01005         first or work with an array of pointers.
01006 */
01007
01008 int RunEncode(PHEAD WORD *fun, WORD *args, WORD *info)
01009 {
01010     WORD base, *f, *funstop, *fun1, *t, size1, size2, size3, *arg;
01011     int num, num1, num2, n, i, i1, i2;
01012     UWORD *scrat1, *scrat2, *scrat3;
01013     WORD *tt, *tstop, totarg, arg1, arg2;
01014     if ( functions[fun[0]-FUNCTION].spec != 0 ) return(0);
01015     if ( *args != ARGRANGE ) {
01016         MLOCK(ErrorMessageLock);
01017         MesPrint("Illegal range encountered in RunEncode");
01018         MUNLOCK(ErrorMessageLock);
01019         Terminate(-1);
01020     }
01021     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
01022     while ( tt < tstop ) { totarg++; NEXTARG(tt); }
01023     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
01024     if ( arg1 > totarg || arg2 > totarg ) return(0);
01025
01026     if ( info[2] == BASECODE ) {
01027         base = info[3];
01028         if ( base <= 0 ) { /* is a dollar variable */
01029             i1 = -base;
01030             base = DolToNumber(BHEAD i1);
01031             if ( AN.ErrorInDollar || base < 2 ) {
01032                 MLOCK(ErrorMessageLock);
01033                 MesPrint("$%s does not have a number value > 1 in base/encode/transform statement in
      module %l",
01034                     DOLLARNAME(Dollars,i1),AC.CModule);
01035                 MUNLOCK(ErrorMessageLock);
01036                 Terminate(-1);
01037             }
01038         }
01039 /*
01040         Compute number of pointers needed and make sure there is space
01041 */
01042         if ( arg1 > arg2 ) { num1 = arg2; num2 = arg1; }
01043         else { num1 = arg1; num2 = arg2; }
01044         num = num2-num1+1;
01045         WantAddPointers(num);
01046 /*
01047         Collect the pointers in pWorkSpace
01048 */
01049         n = 1; funstop = fun+fun[1]; f = fun+FUNHEAD;
01050         while ( n < num1 ) {
01051             if ( f >= funstop ) return(0);
01052             NEXTARG(f);
01053             n++;
01054         }
01055         fun1 = f; i = 0;
01056         while ( n <= num2 ) {
01057             if ( f >= funstop ) return(0);
01058             if ( *f != -SNUMBER ) {
01059                 if ( *f < 0 ) return(0);
01060                 t = f + *f - 1;
01061                 i1 = ABS(*t);
01062                 if ( (*f-i1) != (ARGHEAD+1) ) return(0); /* Not numerical */
01063                 i1 = (i1-1)/2 - 1;
01064                 t--;
01065                 while ( i1 > 0 ) {
01066                     if ( *t != 0 ) return(0); /* Not an integer */
01067                     t--; i1--;
01068                 }
01069             }
01070             AT.pWorkSpace[AT.pWorkPointer+i] = f;
01071             i++;
01072             NEXTARG(f);
01073             n++;
01074         }
01075 /*
01076         f points now to after the arguments; fun1 at the first.
01077         Now check whether we need to revert the order
```

```
01078 */
01079         if ( arg1 > arg2 ) {
01080             i1 = 0; i2 = i-1;
01081             while ( i1 < i2 ) {
01082                 t = AT.pWorkSpace[AT.pWorkPointer+i1];
01083                 AT.pWorkSpace[AT.pWorkPointer+i1] = AT.pWorkSpace[AT.pWorkPointer+i2];
01084                 AT.pWorkSpace[AT.pWorkPointer+i2] = t;
01085                 i1++; i2--;
01086             }
01087         }
01088 /*
01089         Now we can put the thing together.
01090         x = arg1;
01091         x = base*x+arg2
01092         x = base*x+arg3  etc.
01093         We need three scratch arrays for long integers
01094                 (see NumberMalloc in tools.c).
01095 */
01096         scrat1 = NumberMalloc("RunEncode");
01097         scrat2 = NumberMalloc("RunEncode");
01098         scrat3 = NumberMalloc("RunEncode");
01099         arg = AT.pWorkSpace[AT.pWorkPointer];
01100         size1 = PutArgInScratch(arg,scrat1);
01101         i--;
01102         while ( i > 0 ) {
01103             if ( MulLong(scrat1,size1,(UWORD *)(&base),1,scrat2,&size2) ) {
01104                 NumberFree(scrat3,"RunEncode");
01105                 NumberFree(scrat2,"RunEncode");
01106                 NumberFree(scrat1,"RunEncode");
01107                 goto CalledFrom;
01108             }
01109             NEXTARG(arg);
01110             size3 = PutArgInScratch(arg,scrat3);
01111             if ( AddLong(scrat2,size2,scrat3,size3,scrat1,&size1) ) {
01112                 NumberFree(scrat3,"RunEncode");
01113                 NumberFree(scrat2,"RunEncode");
01114                 NumberFree(scrat1,"RunEncode");
01115                 goto CalledFrom;
01116             }
01117             i--;
01118         }
01119 /*
01120         Now put the output in place. There are two cases, one being much
01121         faster than the other. Hence we program both.
01122         Fast: it fits inside the old location.
01123         Slow: it does not.
01124         The total space is f-fun1
01125 */
01126         if ( size1 == 0 ) { /* Fits! */
01127             *fun1++ = -SNUMBER; *fun1++ = 0;
01128             while ( f < funstop ) *fun1++ = *f++;
01129             fun[1] = funstop-fun;
01130         }
01131         else if ( size1 == 1 && scrat1[0] <= MAXPOSITIVE ) { /* Fits! */
01132             *fun1++ = -SNUMBER; *fun1++ = scrat1[0];
01133             while ( f < funstop ) *fun1++ = *f++;
01134             fun[1] = fun1-fun;
01135         }
01136         else if ( size1 == -1 && scrat1[0] <= MAXPOSITIVE+1 ) { /* Fits! */
01137             *fun1++ = -SNUMBER;
01138             if ( scrat1[0] < MAXPOSITIVE ) *fun1++ = scrat1[0];
01139             else *fun1++ = (WORD)(MAXPOSITIVE+1);
01140             while ( f < funstop ) *fun1++ = *f++;
01141             fun[1] = fun1-fun;
01142         }
01143         else if ( ABS(size1)*2+2+ARGHEAD <= f-fun1 ) { /* Fits! */
01144             if ( size1 < 0 ) { size2 = size1*2-1; size1 = -size1; size3 = -size2; }
01145             else { size2 = 2*size1+1; size3 = size2; }
01146             *fun1++ = size3+ARGHEAD+1;
01147             *fun1++ = 0; FILLARG(fun1);
01148             *fun1++ = size3+1;
01149             for ( i = 0; i < size1; i++ ) *fun1++ = scrat1[i];
01150             *fun1++ = 1;
01151             for ( i = 1; i < size1; i++ ) *fun1++ = 0;
01152             *fun1++ = size2;
01153             while ( f < funstop ) *fun1++ = *f++;
01154             fun[1] = fun1-fun;
01155         }
01156         else {  /* Does not fit */
01157             t = funstop;
01158             if ( size1 < 0 ) { size2 = size1*2-1; size1 = -size1; size3 = -size2; }
01159             else { size2 = 2*size1+1; size3 = size2; }
01160             *t++ = size3+ARGHEAD+1;
01161             *t++ = 0; FILLARG(t);
01162             *t++ = size3+1;
01163             for ( i = 0; i < size1; i++ ) *t++ = scrat1[i];
01164             *t++ = 1;
```

```
01165             for ( i = 1; i < size1; i++ ) *t++ = 0;
01166             *t++ = size2;
01167             while ( f < funstop ) *t++ = *f++;
01168             f = funstop;
01169             while ( f < t ) *fun1++ = *f++;
01170             fun[1] = fun1-fun;
01171         }
01172         NumberFree(scrat3,"RunEncode");
01173         NumberFree(scrat2,"RunEncode");
01174         NumberFree(scrat1,"RunEncode");
01175     }
01176     else {
01177 /* INTERNAL_ERROR_EXCL_START */
01178         MLOCK(ErrorMessageLock);
01179         MesPrint("!>Unimplemented type of encoding encountered in RunEncode");
01180         MUNLOCK(ErrorMessageLock);
01181         Terminate(-1);
01182 /* INTERNAL_ERROR_EXCL_STOP */
01183     }
01184     return(0);
01185 CalledFrom:
01186     MLOCK(ErrorMessageLock);
01187     MesCall("RunEncode");
01188     MUNLOCK(ErrorMessageLock);
01189     return(-1);
01190 }
01191
01192 /*
01193         #] RunEncode :
01194         #[ RunDecode :
01195 */
01196
01197 int RunDecode(PHEAD WORD *fun, WORD *args, WORD *info)
01198 {
01199     WORD base, num, num1, num2, n, *f, *funstop, *fun1, size1, size2, size3, *t;
01200     WORD i1, i2, i, sig;
01201     UWORD *scrat1, *scrat2, *scrat3;
01202     WORD *tt, *tstop, totarg, arg1, arg2;
01203     if ( functions[fun[0]-FUNCTION].spec != 0 ) return(0);
01204     if ( *args != ARGRANGE ) {
01205         MLOCK(ErrorMessageLock);
01206         MesPrint("Illegal range encountered in RunDecode");
01207         MUNLOCK(ErrorMessageLock);
01208         Terminate(-1);
01209     }
01210     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
01211     while ( tt < tstop ) { totarg++; NEXTARG(tt); }
01212     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
01213     if ( arg1 > totarg && arg2 > totarg ) return(0);
01214     if ( info[2] == BASECODE ) {
01215         base = info[3];
01216         if ( base <= 0 ) { /* is a dollar variable */
01217             i1 = -base;
01218             base = DolToNumber(BHEAD i1);
01219             if ( AN.ErrorInDollar || base < 2 ) {
01220                 MLOCK(ErrorMessageLock);
01221                 MesPrint("$%s does not have a number value > 1 in base/decode/transform statement in
      module %l",
01222                     DOLLARNAME(Dollars,i1),AC.CModule);
01223                 MUNLOCK(ErrorMessageLock);
01224                 Terminate(-1);
01225             }
01226         }
01227 /*
01228         Compute number of output arguments needed
01229 */
01230         if ( arg1 > arg2 ) { num1 = arg2; num2 = arg1; }
01231         else { num1 = arg1; num2 = arg2; }
01232         num = num2-num1+1;
01233         if ( num <= 1 ) return(0);
01234 /*
01235         Find argument num1
01236 */
01237         funstop = fun + fun[1];
01238         f = fun + FUNHEAD; n = 1;
01239         while ( f < funstop ) {
01240             if ( n == num1 ) break;
01241             NEXTARG(f); n++;
01242         }
01243         if ( f >= funstop ) return(0);  /* not enough arguments */
01244 /*
01245         Check that f is integer
01246 */
01247         if ( *f == -SNUMBER ) {}
01248         else if ( *f < 0 ) return(0);
01249         else {
01250             t = f + *f - 1;
```

```
01251             i1 = ABS(*t);
01252             if ( (*f-i1) != (ARGHEAD+1) ) return(0); /* Not numerical */
01253             i1 = (i1-1)/2 - 1;
01254             t--;
01255             while ( i1 > 0 ) {
01256                 if ( *t != 0 ) return(0); /* Not an integer */
01257                 t--; i1--;
01258             }
01259         }
01260         fun1 = f;
01261 /*
01262         The argument that should be decoded is in fun1
01263         We have to copy it to scratch
01264 */
01265         scrat1 = NumberMalloc("RunEncode");
01266         scrat2 = NumberMalloc("RunEncode");
01267         scrat3 = NumberMalloc("RunEncode");
01268         size1 = PutArgInScratch(fun1,scrat1);
01269         if ( size1 < 0 ) { sig = -1; size1 = -size1; }
01270         else sig = 1;
01271 /*
01272         We can check first whether this number can be decoded
01273 */
01274         scrat2[0] = base; size2 = 1;
01275         if ( RaisPow(BHEAD scrat2,&size2,num) ) {
01276             NumberFree(scrat3,"RunEncode");
01277             NumberFree(scrat2,"RunEncode");
01278             NumberFree(scrat1,"RunEncode");
01279             goto CalledFrom;
01280         }
01281         if ( BigLong(scrat1,size1,scrat2,size2) >= 0 ) { /* Number too big */
01282             NumberFree(scrat3,"RunEncode");
01283             NumberFree(scrat2,"RunEncode");
01284             NumberFree(scrat1,"RunEncode");
01285             return(0);
01286         }
01287 /*
01288         We need num*2 spaces
01289 */
01290         if ( *fun1 > num*2 ) {  /* shrink space */
01291             t = fun1 + 2*num; f = fun1 + *fun1;
01292             while ( f < funstop ) *t++ = *f++;
01293             fun[1] = t - fun;
01294         }
01295         else if ( *fun1 < num*2 ) { /* case includes -SNUMBER */
01296             if ( *fun1 < 0 ) { /* expand space from -SNUMBER */
01297                 fun[1] += (num-1)*2;
01298                 t = funstop + (num-1)*2;
01299             }
01300             else { /* expand space from general argument */
01301                 fun[1] += 2*num - *fun1;
01302                 t = funstop +2*num - *fun1;
01303             }
01304             f = funstop;
01305             while ( f > fun1 ) *--t = *--f;
01306         }
01307 /*
01308         Now there is space for num -SNUMBER arguments filled from the top.
01309 */
01310         for ( i = num-1; i >= 0; i-- ) {
01311             DivLong(scrat1,size1,(UWORD *)(&base),1,scrat2,&size2,scrat3,&size3);
01312             fun1[2*i]   = -SNUMBER;
01313             if ( size3 == 0 ) fun1[2*i+1] = 0;
01314             else fun1[2*i+1] = (WORD)(scrat3[0])*sig;
01315             for ( i1 = 0; i1 < size2; i1++ ) scrat1[i1] = scrat2[i1];
01316             size1 = size2;
01317         }
01318         if ( size2 != 0 ) {
01319             MLOCK(ErrorMessageLock);
01320             MesPrint("RunDecode: number to be decoded is too big");
01321             MUNLOCK(ErrorMessageLock);
01322             NumberFree(scrat3,"RunEncode");
01323             NumberFree(scrat2,"RunEncode");
01324             NumberFree(scrat1,"RunEncode");
01325             goto CalledFrom;
01326         }
01327 /*
01328         Now check whether we should change the order of the arguments
01329 */
01330         if ( arg1 > arg2 ) {
01331             i1 = 1; i2 = 2*num-1;
01332             while ( i2 > i1 ) {
01333                 i = fun1[i1]; fun1[i1] = fun1[i2]; fun1[i2] = i;
01334                 i1 += 2; i2 -= 2;
01335             }
01336         }
01337         NumberFree(scrat3,"RunEncode");
```

```
01338         NumberFree(scrat2,"RunEncode");
01339         NumberFree(scrat1,"RunEncode");
01340     }
01341     else {
01342 /* INTERNAL_ERROR_EXCL_START */
01343         MLOCK(ErrorMessageLock);
01344         MesPrint("!>Unimplemented type of encoding encountered in RunDecode");
01345         MUNLOCK(ErrorMessageLock);
01346         Terminate(-1);
01347 /* INTERNAL_ERROR_EXCL_STOP */
01348     }
01349     return(0);
01350 CalledFrom:
01351     MLOCK(ErrorMessageLock);
01352     MesCall("RunDecode");
01353     MUNLOCK(ErrorMessageLock);
01354     return(-1);
01355 }
01356
01357 /*
01358         #] RunDecode :
01359         #[ RunReplace :
01360
01361         Gets the function, passes the arguments and looks whether they
01362         need to be treated. If so, the exact treatment is found in info.
01363         The info is given as if it is a function of type REPLACEMENT but
01364         its name is REPLACEARG (which is NOT a function).
01365         It is performed on the arguments.
01366         The output is at first written after fun and in the end overwrites fun.
01367 */
01368
01369 int RunReplace(PHEAD WORD *fun, WORD *args, WORD *info)
01370 {
01371     int n = 0, i, dirty = 0, totarg, nfix, nwild, ngeneral;
01372     WORD *t, *tt, *u, *tstop, *info1, *infoend, *oldwork = AT.WorkPointer;
01373     WORD *term, *newterm, *nt, *term1, *term2;
01374     WORD wild[4], mask, *term3, *term4, *oldmask = AT.WildMask;
01375     WORD n1, n2, doanyway;
01376     info++;
01377     t = fun; tstop = fun + fun[1]; u = tstop;
01378     for ( i = 0; i < FUNHEAD; i++ ) *u++ = *t++;
01379     tt = t;
01380     if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
01381         totarg = 0;
01382         while ( tt < tstop ) { totarg++; NEXTARG(tt); }
01383     }
01384     else {
01385         totarg = tstop - tt;
01386     }
01387 /*
01388     Now get the info through Generator to bring it to standard form.
01389     info points at a single term that should be sent to Generator.
01390
01391     We want to put the information in the WorkSpace but fun etc lies there
01392     already. This means that we have to move the WorkPointer quite high up.
01393 */
01394     AT.WorkPointer += 2*AM.MaxTer;
01395     if ( AT.WorkPointer > AT.WorkTop ) {
01396         MLOCK(ErrorMessageLock);
01397         MesWork();
01398         MUNLOCK(ErrorMessageLock);
01399         return(-1);
01400     }
01401     term = AT.WorkPointer;
01402     for ( i = 0; i < *info; i++ ) term[i] = info[i];
01403     AT.WorkPointer = term + *term;
01404     AR.Eside = LHSIDEX;
01405     NewSort(BHEAD0);
01406     if ( Generator(BHEAD term,AR.Cnumlhs) ) {
01407         LowerSortLevel();
01408         AT.WorkPointer = oldwork;
01409         return(-1);
01410     }
01411     newterm = AT.WorkPointer;
01412     if ( EndSort(BHEAD newterm,1) < 0 ) {}
01413     if ( ( *newterm && *(newterm+*newterm) != 0 ) || *newterm == 0 ) {
01414         MLOCK(ErrorMessageLock);
01415         MesPrint("&information in replace transformation does not evaluate into a single term");
01416         MUNLOCK(ErrorMessageLock);
01417         return(-1);
01418     }
01419     AR.Eside = RHSIDE;
01420     i = *newterm; tt = term; nt = newterm;
01421     NCOPY(tt,nt,i);
01422     AT.WorkPointer = term + *term;
01423     info = term + 1;
01424
```

```
01425     term1 = term + *term;
01426     term2 = term1+1;
01427     *term2++ = REPLACEMENT;
01428     term2++; FILLFUN(term2)
01429 /*
01430     First we count the different types of objects
01431 */
01432     infoend = info + info[1];
01433     info1 = info + FUNHEAD;
01434     nfix = nwild = ngeneral = 0;
01435     while ( info1 < infoend ) {
01436         if ( *info1 == -SNUMBER ) {
01437             nfix++;
01438             info1 += 2; NEXTARG(info1)
01439         }
01440         else if ( *info1 <= -FUNCTION ) {
01441             if ( *info1 == -WILDARGFUN ) {
01442                 nwild++;
01443                 info1++; NEXTARG(info1)
01444             }
01445             else {
01446                 *term2++ = *info1++; COPY1ARG(term2,info1)
01447                 ngeneral++;
01448             }
01449         }
01450         else if ( *info1 == -INDEX ) {
01451             if ( info1[1] == WILDARGINDEX + AM.OffsetIndex ) {
01452                 nwild++;
01453                 info1 += 2; NEXTARG(info1)
01454             }
01455             else {
01456                 *term2++ = *info1++; *term2++ = *info1++; COPY1ARG(term2,info1)
01457                 ngeneral++;
01458             }
01459         }
01460         else if ( *info1 == -SYMBOL ) {
01461             if ( info1[1] == WILDARGSYMBOL ) {
01462                 nwild++;
01463                 info1 += 2; NEXTARG(info1)
01464             }
01465             else {
01466                 *term2++ = *info1++; *term2++ = *info1++; COPY1ARG(term2,info1)
01467                 ngeneral++;
01468             }
01469         }
01470         else if ( *info1 == -MINVECTOR || *info1 == -VECTOR ) {
01471             if ( info1[1] == WILDARGVECTOR + AM.OffsetVector ) {
01472                 nwild++;
01473                 info1 += 2; NEXTARG(info1)
01474             }
01475             else {
01476                 *term2++ = *info1++; *term2++ = *info1++; COPY1ARG(term2,info1)
01477                 ngeneral++;
01478             }
01479         }
01480         else {
01481 /* INTERNAL_ERROR_EXCL_START */
01482             MLOCK(ErrorMessageLock);
01483             MesPrint("!>irregular code found in replace transformation (RunReplace)");
01484             MUNLOCK(ErrorMessageLock);
01485             Terminate(-1);
01486 /* INTERNAL_ERROR_EXCL_STOP */
01487         }
01488     }
01489     AT.WorkPointer = term2;
01490     *term1 = term2 - term1;
01491     term1[2] = *term1 - 1;
01492 /*
01493     And now stepping through the arguments
01494 */
01495     while ( t < tstop ) {
01496         n++;    /* The number of the argument. Now check whether we need it */
01497         if ( TestArgNum(n,totarg,args) == 0 ) {
01498             if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
01499                 if ( *t <= -FUNCTION ) { *u++ = *t++; }
01500                 else if ( *t < 0 ) { *u++ = *t++; *u++ = *t++; }
01501                 else { i = *t; NCOPY(u,t,i) }
01502             }
01503             else *u++ = *t++;
01504             continue;
01505         }
01506 /*
01507         Here we have in info effectively a replace_ function, but with
01508         additionally the possibility of integer arguments. We treat those first
01509         and for the rest we have to do some pattern matching.
01510         Note that the compilation routine should check that there is an
01511         even number of arguments in the replace function.
```

```
01512
01513         First we go for number -> something
01514 */
01515         doanyway = 0;
01516         if ( nfix > 0 ) {
01517           if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
01518             if ( *t == -SNUMBER ) {
01519               info1 = info + FUNHEAD;
01520               while ( info1 < infoend ) {
01521                 if ( *info1 == -SNUMBER ) {
01522                     if ( info1[1] == t[1] ) {
01523                       if ( info1[2] == -SNUMBER ) {
01524                         *u++ = -SNUMBER; *u++ = info1[3];
01525                         info1 += 4;
01526                       }
01527                       else {
01528                         info1 += 2;
01529                         if ( info1[0] <= -FUNCTION ) i = 1;
01530                         else if ( info1[0] < 0 ) i = 2;
01531                         else i = *info1;
01532                         NCOPY(u,info1,i)
01533                       }
01534                       t += 2; goto nextt;
01535                     }
01536                     info1 += 2;
01537                     NEXTARG(info1);
01538                 }
01539                 else {
01540                     NEXTARG(info1);
01541                     NEXTARG(info1);
01542                 }
01543               }
01544 /*
01545               Here we had no match in the style of 1->2. It could however
01546               be that xarg_ does something
01547 */
01548               doanyway = 1; n2 = t[1];
01549             }
01550           }
01551           else {  /* Tensor */
01552             if ( *t < AM.OffsetIndex && *t >= 0 ) {
01553               info1 = info + FUNHEAD;
01554               while ( info1 < infoend ) {
01555                 if ( ( *info1 == -SNUMBER ) && ( info1[1] == *t )
01556                  && ( ( ( info1[2] == -SNUMBER ) && ( info1[3] >= 0 )
01557                  && ( info1[3] < AM.OffsetIndex ) )
01558                  || ( info1[2] == -INDEX || info1[2] == -VECTOR
01559                  || info1[2] == -MINVECTOR ) ) ) {
01560                     *u++ = info1[3];
01561                     info1 += 4;
01562                     t++; goto nextt;
01563                 }
01564                 else {
01565                     NEXTARG(info1);
01566                     NEXTARG(info1);
01567                 }
01568               }
01569             }
01570           }
01571         }
01572         else if ( *t == -SNUMBER ) {
01573           doanyway = 1; n2 = t[1];
01574         }
01575 /*
01576         First we try to catch those elements that have an exact match
01577         in the traditional replace_ part.
01578         This means that *t should be less than zero and match an entry
01579         in the replace_ function that we prepared.
01580 */
01581         if ( ngeneral > 0 ) {
01582           if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
01583             if ( *t < 0 ) {
01584                 term3 = term1 + *term1;
01585                 term4 = term1 + FUNHEAD;
01586                 while ( term4 < term3 ) {
01587                     if ( *term4 == *t && ( *t <= -FUNCTION ||
01588                     ( t[1] == term4[1] ) ) ) break;
01589                     NEXTARG(term4)
01590                 }
01591                 if ( term4 < term3 ) goto dothisnow;
01592             }
01593           }
01594           else {
01595             term3 = term1 + *term1;
01596             term4 = term1 + FUNHEAD;
01597             while ( term4 < term3 ) {
01598                 if ( ( term4[1] == *t ) &&
```

```
01599                     ( ( *term4 == -INDEX || *term4 == -VECTOR ||
01600                      ( *term4 == -SYMBOL && term4[1] < AM.OffsetIndex
01601                         && term4[1] >= 0 ) ) ) ) break;
01602                 NEXTARG(term4)
01603             }
01604             if ( term4 < term3 ) goto dothisnow;
01605           }
01606         }
01607 /*
01608         First we eliminate the fixed arguments and make a 'new info'
01609         If there is anything left we can continue.
01610         Now we look for whole argument wildcards (arg_, parg_, iarg_ or farg_)
01611 */
01612         if ( nwild > 0 ) {
01613 /*
01614             If we have f(a)*replace_(xarg_,b(xarg_)) this gives f(b(a))
01615             In testing the wildcard we have CheckWild do the work.
01616             This means that we have to set op the special variables
01617             (AT.WildMask,AN.WildValue,AN.NumWild)
01618
01619 */
01620             wild[1] = 4;
01621             info1 = info + FUNHEAD;
01622             while ( info1 < infoend ) {
01623                 if ( *info1 == -SYMBOL && info1[1] == WILDARGSYMBOL
01624                 && ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) ) {
01625                     wild[0] = SYMTOSUB;
01626                     wild[2] = WILDARGSYMBOL;
01627                     wild[3] = 0;
01628                     AN.WildValue = wild;
01629                     AT.WildMask = &mask;
01630                     mask = 0;
01631                     AN.NumWild = 1;
01632                     if ( *t == -SYMBOL || ( *t > 0 && CheckWild(BHEAD WILDARGSYMBOL,SYMTOSUB,1,t) == 0
     )
01633                     || doanyway ) {
01634 /*
01635                         We put the part in replace in a function and make
01636                         a replace_(xarg_,(t argument)).
01637 */
01638                         n1 = SYMBOL; n2 = WILDARGSYMBOL;
01639                         info1 += 2;
01640 getthisone:;
01641                         term3 = term2+1;
01642                         if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
01643                             *term3++ = DUMFUN; term3++; FILLFUN(term3)
01644                             COPY1ARG(term3,info1)
01645                         }
01646                         else {
01647                             *term3++ = fun[0]; term3++; FILLFUN(term3)
01648                             *term3++ = *info1;
01649                         }
01650                         term2[2] = term3 - term2 - 1;
01651                         tt = term3;
01652                         *term3++ = REPLACEMENT;
01653                         term3++; FILLFUN(term3)
01654                         *term3++ = -n1;
01655                         if ( n1 < FUNCTION ) *term3++ = n2;
01656                         if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
01657                             term4 = t;
01658                             COPY1ARG(term3,term4)
01659                         }
01660                         else {
01661                             *term3++ = *t;
01662                         }
01663                         tt[1] = term3 - tt;
01664                         *term3++ = 1; *term3++ = 1; *term3++ = 3;
01665                         *term2 = term3 - term2;
01666
01667                         AT.WorkPointer = term3;
01668                         NewSort(BHEAD0);
01669                         if ( Generator(BHEAD term2,AR.Cnumlhs) ) {
01670                             LowerSortLevel();
01671                             AT.WorkPointer = oldwork;
01672                             AT.WildMask = oldmask;
01673                             return(-1);
01674                         }
01675                         term4 = AT.WorkPointer;
01676                         if ( EndSort(BHEAD term4,1) < 0 ) {}
01677                         if ( ( *term4 && *(term4+*term4) != 0 ) || *term4 == 0 ) {
01678                             MLOCK(ErrorMessageLock);
01679                             MesPrint("&information in replace transformation does not evaluate into a
     single term");
01680                             MUNLOCK(ErrorMessageLock);
01681                             return(-1);
01682                         }
01683 /*
```

```
01684                           Now we can copy the new function argument to the output u
01685 */
01686                           i = term4[2]-FUNHEAD;
01687                           term3 = term4+FUNHEAD+1;
01688                           NCOPY(u,term3,i)
01689                           if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
01690                               NEXTARG(t)
01691                           }
01692                           else t++;
01693                           AT.WorkPointer = term2;
01694 
01695                           goto nextt;
01696                       }
01697                       info1 += 2; NEXTARG(info1)
01698                   }
01699                   else if ( ( *info1 == -INDEX )
01700                       && ( info[1] == WILDARGINDEX + AM.OffsetIndex ) ) {
01701                       wild[0] = INDTOSUB;
01702                       wild[2] = WILDARGINDEX+AM.OffsetIndex;
01703                       wild[3] = 0;
01704                       AN.WildValue = wild;
01705                       AT.WildMask = &mask;
01706                       mask = 0;
01707                       AN.NumWild = 1;
01708                       if ( ( functions[fun[0]-FUNCTION].spec == TENSORFUNCTION )
01709                       || ( *t == -INDEX || ( *t > 0 && CheckWild(BHEAD WILDARGINDEX,INDTOSUB,1,t) == 0 )
      ) ) {
01710 /*
01711                           We put the part in replace in a function and make
01712                           a replace_(xarg_,(t argument)).
01713 */
01714                           n1 = INDEX; n2 = WILDARGINDEX+AM.OffsetIndex;
01715                           info1 += 2;
01716                           goto getthisone;
01717                       }
01718                       info1 += 2; NEXTARG(info1)
01719                   }
01720                   else if ( ( *info1 == -VECTOR )
01721                       && ( info1[1] == WILDARGVECTOR + AM.OffsetVector ) ) {
01722                       wild[0] = VECTOSUB;
01723                       wild[2] = WILDARGVECTOR+AM.OffsetVector;
01724                       wild[3] = 0;
01725                       AN.WildValue = wild;
01726                       AT.WildMask = &mask;
01727                       mask = 0;
01728                       AN.NumWild = 1;
01729                       if ( functions[fun[0]-FUNCTION].spec == TENSORFUNCTION ) {
01730                           if ( *t < MINSPEC ) {
01731                               n1 = VECTOR; n2 = WILDARGVECTOR+AM.OffsetVector;
01732                               info1 += 2;
01733                               goto getthisone;
01734                           }
01735                       }
01736                       else if ( *t == -VECTOR || *t == -MINVECTOR ||
01737                       ( *t > 0 && CheckWild(BHEAD WILDARGVECTOR,VECTOSUB,1,t) == 0 ) ) {
01738 /*
01739                           We put the part in replace in a function and make
01740                           a replace_(xarg_,(t argument)).
01741 */
01742                           n1 = VECTOR; n2 = WILDARGVECTOR+AM.OffsetVector;
01743                           info1 += 2;
01744                           goto getthisone;
01745                       }
01746                       info1 += 2; NEXTARG(info1)
01747                   }
01748                   else if ( *info1 == -WILDARGFUN ) {
01749                       wild[0] = FUNTOFUN;
01750                       wild[2] = WILDARGFUN;
01751                       wild[3] = 0;
01752                       AN.WildValue = wild;
01753                       AT.WildMask = &mask;
01754                       mask = 0;
01755                       AN.NumWild = 1;
01756                       if ( *t <= -FUNCTION || ( *t > 0 && CheckWild(BHEAD WILDARGFUN,FUNTOFUN,1,t) == 0
      ) ) {
01757 /*
01758                           We put the part in replace in a function and make
01759                           a replace_(xarg_,(t argument)).
01760 */
01761                           n2 = n1 = -WILDARGFUN; /* n2 is to keep the compiler quiet */
01762                           info1++;
01763                           goto getthisone;
01764                       }
01765                       info1++; NEXTARG(info1)
01766                   }
01767                   else {
01768                       NEXTARG(info1) NEXTARG(info1)
```

```
01769                 }
01770             }
01771         }
01772         if ( ngeneral > 0 ) {
01773 /*
01774             They are all in a replace_ function.
01775             Compose the whole thing into a term with replace_()*dum_(arg)
01776             which will be given to Generator.
01777             If we have f(a(x))*replace_(x,b) this gives f(a(b))
01778 */
01779 dothisnow:;
01780             term3 = term2; term4 = term1; i = *term1;
01781             NCOPY(term3,term4,i)
01782             term4 = term3;
01783             if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
01784                 *term3++ = DUMFUN; term3++; FILLFUN(term3);
01785                 tt = t;
01786                 COPY1ARG(term3,tt)
01787             }
01788             else {
01789                 *term3++ = fun[0]; term3++; FILLFUN(term3); *term3++ = *t;
01790             }
01791             term4[1] = term3-term4;
01792             *term3++ = 1; *term3++ = 1; *term3++ = 3;
01793             *term2 = term3-term2;
01794             AT.WorkPointer = term3;
01795             NewSort(BHEAD0);
01796             if ( Generator(BHEAD term2,AR.Cnumlhs) ) {
01797                 LowerSortLevel();
01798                 AT.WorkPointer = oldwork;
01799                 AT.WildMask = oldmask;
01800                 return(-1);
01801             }
01802             term4 = AT.WorkPointer;
01803             if ( EndSort(BHEAD term4,1) < 0 ) {}
01804             if ( ( *term4 && *(term4+*term4) != 0 ) || *term4 == 0 ) {
01805                 MLOCK(ErrorMessageLock);
01806                 MesPrint("&information in replace transformation does not evaluate into a single
      term");
01807                 MUNLOCK(ErrorMessageLock);
01808                 return(-1);
01809             }
01810 /*
01811             Now we can copy the new function argument to the output u
01812 */
01813             i = term4[2]-FUNHEAD;
01814             term3 = term4+FUNHEAD+1;
01815             NCOPY(u,term3,i)
01816             NEXTARG(t)
01817             AT.WorkPointer = term2;
01818
01819             goto nextt;
01820         }
01821
01822 /*
01823         No catch. Copy the argument and continue.
01824 */
01825         if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
01826             if ( *t <= -FUNCTION ) { *u++ = *t++; }
01827             else if ( *t < 0 ) { *u++ = *t++; *u++ = *t++; }
01828             else { i = *t; NCOPY(u,t,i) }
01829         }
01830         else {
01831             *u++ = *t++;
01832         }
01833 nextt:;
01834     }
01835     i = u - tstop; tstop[1] = i; tstop[2] = dirty;
01836     t = fun; u = tstop; NCOPY(t,u,i)
01837     AT.WorkPointer = oldwork;
01838     AT.WildMask = oldmask;
01839     return(0);
01840 }
01841
01842 /*
01843         #] RunReplace :
01844         #[ RunImplode :
01845
01846         Note that we restrict ourselves to short integers and/or single symbols
01847 */
01848
01849 int RunImplode(WORD *fun, WORD *args)
01850 {
01851     GETIDENTITY
01852     WORD *tt, *tstop, totarg, arg1, arg2, num1, num2, i1, n;
01853     WORD *f, *t, *ttt, *t4, *ff, *fff;
01854     WORD moveup, numzero, outspace;
```

```
01855     if ( functions[fun[0]-FUNCTION].spec != 0 ) return(0);
01856     if ( *args != ARGRANGE ) {
01857         MLOCK(ErrorMessageLock);
01858         MesPrint("Illegal range encountered in RunImplode");
01859         MUNLOCK(ErrorMessageLock);
01860         Terminate(-1);
01861     }
01862     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
01863     while ( tt < tstop ) { totarg++; NEXTARG(tt); }
01864     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
01865 /*
01866     Get the proper range in forward direction and the number of arguments
01867 */
01868     if ( arg1 > arg2 ) { num1 = arg2; num2 = arg1; }
01869     else { num1 = arg1; num2 = arg2; }
01870     if ( num1 > totarg || num2 > totarg ) return(0);
01871 /*
01872     We need, for the most general case 4 spots for each:
01873             x,pow,coef,sign
01874     Hence we put these in the workspace above the term after tstop
01875 */
01876     n = 1; f = fun+FUNHEAD;
01877     while ( n < num1 ) {
01878         if ( f >= tstop ) return(0);
01879         NEXTARG(f);
01880         n++;
01881     }
01882     ff = f;
01883 /*
01884     We are now at the first argument to be done
01885     Go through the terms and test their validity.
01886     If one of them doesn't conform to the rules we don't do anything.
01887     The terms to be done are put in special notation after the function.
01888     Notation: numsymbol, power, |coef|, sign
01889     If numsymbol is negative there is no symbol.
01890     We do it this way because otherwise stepping backwards (as in range=(4,1))
01891     would be very difficult.
01892 */
01893     tt = tstop;
01894     while ( n <= num2 ) {
01895         if ( f >= tstop ) return(0);
01896         if ( *f == -SNUMBER ) { *tt++ = -1; *tt++ = 0;
01897             if ( f[1] < 0 ) { *tt++ = -f[1]; *tt++ = -1; }
01898             else { *tt++ = f[1]; *tt++ = 1; }
01899             f += 2;
01900         }
01901         else if ( *f == -SYMBOL ) { *tt++ = f[1]; *tt++ = 1; *tt++ = 1; *tt++ = 1; f += 2; }
01902         else if ( *f < 0 ) return(0);
01903         else {
01904             if ( *f != ( f[ARGHEAD]+ARGHEAD ) ) return(0); /* Not a single term */
01905             t = f + *f - 1;
01906             i1 = ABS(*t);
01907             if ( ( i1 > 3 ) || ( t[-1] != 1 ) ) return(0); /* Not an integer or too big */
01908             if ( (UWORD)(t[-2]) > MAXPOSITIVE4 ) return(0); /* number too big */
01909             if ( f[ARGHEAD] == i1+1 ) { /* numerical which is fine */
01910                 *tt++ = -1; *tt++ = 0; *tt++ = t[-2];
01911                 if ( *t < 0 ) { *tt++ = -1; }
01912                 else { *tt++ = 1; }
01913             }
01914             else if ( ( f[ARGHEAD+1] != SYMBOL )
01915                  || ( f[ARGHEAD+2] != 4 )
01916                  || ( ( f+ARGHEAD+1+f[ARGHEAD+2] ) < ( t-i1 ) ) ) return(0);
01917                     /* not a single symbol with a coefficient */
01918             else {
01919                 *tt++ = f[ARGHEAD+3];
01920                 *tt++ = f[ARGHEAD+4];
01921                 *tt++ = t[-2];
01922                 if ( *t < 0 ) { *tt++ = -1; }
01923                 else { *tt++ = 1; }
01924             }
01925             f += *f;
01926         }
01927         n++;
01928     }
01929     fff = f;
01930 /*
01931     At this point we can do the implosion.
01932     Requirement: no coefficient shall take more than one word.
01933     (a stricter requirement may be needed to keep the explosion contained)
01934 */
01935     if ( arg1 > arg2 ) {
01936 /*
01937         Work backward.
01938 */
01939         t = tt - 4; numzero = 0;
01940         while ( t >= tstop ) {
01941             if ( t[2] == 0 ) numzero++;
```

```
01942             else {
01943                 if ( numzero > 0 ) {
01944                     t[2] += numzero;
01945                     t4 = t+4;
01946                     ttt = t4 + 4*numzero;
01947                     while ( ttt < tt ) *t4++ = *ttt++;
01948                     tt -= 4*numzero;
01949                     numzero = 0;
01950                 }
01951             }
01952             t -= 4;
01953         }
01954     }
01955     else {
01956         t = tstop;
01957         numzero = 0; ttt = t;
01958         while ( t < tt ) {
01959             if ( t[2] == 0 ) numzero++;
01960             else {
01961                 if ( numzero > 0 ) {
01962                     t[2] += numzero;
01963                     t4 = t;
01964                     while ( t4 < tt ) *ttt++ = *t4++;
01965                     tt -= 4*numzero;
01966                     t -= 4*numzero;
01967                     ttt = t + 4;
01968                     numzero = 0;
01969                 }
01970                 else {
01971                     ttt = t + 4;
01972                 }
01973             }
01974             t += 4;
01975         }
01976 /*
01977         We may have numzero > 0 at the end. We leave them.
01978         Output space is currently from tstop to tt
01979 */
01980     }
01981 /*
01982     Now we compute the real output space needed
01983 */
01984     t = tstop; outspace = 0;
01985     while ( t < tt ) {
01986         if ( t[0] == -1 ) {
01987             if ( t[2] > MAXPOSITIVE4 ) { return(0); /* Number too big */ }
01988             outspace += 2;
01989         }
01990         else if ( t[1] == 1 && t[2] == 1 && t[3] == 1 ) { outspace += 2; }
01991         else { outspace += 8 + ARGHEAD; }
01992         t += 4;
01993     }
01994     if ( outspace < (fff-ff) ) {
01995         t = tstop;
01996         while ( t < tt ) {
01997             if ( t[0] == -1 ) { *ff++ = -SNUMBER; *ff++ = t[2]*t[3]; }
01998             else if ( t[1] == 1 && t[2] == 1 && t[3] == 1 ) {
01999                 *ff++ = -SYMBOL; *ff++ = t[0];
02000             }
02001             else {
02002                 *ff++ = 8+ARGHEAD; *ff++ = 0; FILLARG(ff);
02003                 *ff++ = 8; *ff++ = SYMBOL; *ff++ = 4; *ff++ = t[0]; *ff++ = t[1];
02004                 *ff++ = t[2]; *ff++ = 1; *ff++ = t[3] > 0 ? 3: -3;
02005             }
02006             t += 4;
02007         }
02008         while ( fff < tstop ) *ff++ = *fff++;
02009         fun[1] = ff - fun;
02010     }
02011     else if ( outspace > (fff-ff) ) {
02012 /*
02013         Move the answer up by the required amount.
02014         Move the tail to its new location
02015         Move in things as for outspace == (fff-ff)
02016 */
02017         moveup = outspace-(fff-ff);
02018         ttt = tt + moveup;
02019         t = tt;
02020         while ( t > fff ) *--ttt = *--t;
02021         tt += moveup; tstop += moveup;
02022         fff += moveup;
02023         fun[1] += moveup;
02024         goto moveinto;
02025     }
02026     else {
02027 moveinto:
02028         t = tstop;
```

```
02029         while ( t < tt ) {
02030             if ( t[0] == -1 ) { *ff++ = -SNUMBER; *ff++ = t[2]*t[3]; }
02031             else if ( t[1] == 1 && t[2] == 1 && t[3] == 1 ) {
02032                 *ff++ = -SYMBOL; *ff++ = t[0];
02033             }
02034             else {
02035                 *ff++ = 8+ARGHEAD; *ff++ = 0; FILLARG(ff);
02036                 *ff++ = 8; *ff++ = SYMBOL; *ff++ = 4; *ff++ = t[0]; *ff++ = t[1];
02037                 *ff++ = t[2]; *ff++ = 1; *ff++ = t[3] > 0 ? 3: -3;
02038             }
02039             t += 4;
02040         }
02041     }
02042     return(0);
02043 }
02044
02045 /*
02046         #] RunImplode :
02047         #[ RunExplode :
02048 */
02049
02050 int RunExplode(PHEAD WORD *fun, WORD *args)
02051 {
02052     WORD arg1, arg2, num1, num2, *tt, *tstop, totarg, *tonew, *newfun;
02053     WORD *ff, *f;
02054     int reverse = 0, iarg, i, numzero;
02055     if ( functions[fun[0]-FUNCTION].spec != 0 ) return(0);
02056     if ( *args != ARGRANGE ) {
02057         MLOCK(ErrorMessageLock);
02058         MesPrint("Illegal range encountered in RunExplode");
02059         MUNLOCK(ErrorMessageLock);
02060         Terminate(-1);
02061     }
02062     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
02063     while ( tt < tstop ) { totarg++; NEXTARG(tt); }
02064     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
02065 /*
02066     Get the proper range in forward direction and the number of arguments
02067 */
02068     if ( arg1 > arg2 ) { num1 = arg2; num2 = arg1; reverse = 1; }
02069     else { num1 = arg1; num2 = arg2; }
02070     if ( num1 > totarg || num2 > totarg ) return(0);
02071     if ( tstop + AM.MaxTer > AT.WorkTop ) goto OverWork;
02072 /*
02073     We will make the new function after the old one in the workspace
02074     Find the first argument
02075 */
02076     tonew = newfun = tstop;
02077     ff = fun + FUNHEAD; iarg = 0;
02078     while ( ff < tstop ) {
02079         iarg++;
02080         if ( iarg == num1 ) {
02081             i = ff - fun; f = fun;
02082             NCOPY(tonew,f,i)
02083             break;
02084         }
02085         NEXTARG(ff)
02086     }
02087 /*
02088     We have reached the first argument to be done
02089 */
02090     while ( iarg <= num2 ) {
02091         if ( *ff == -SYMBOL || ( *ff == -SNUMBER && ff[1] == 0 ) )
02092             { *tonew++ = *ff++; *tonew++ = *ff++; }
02093         else if ( *ff == -SNUMBER ) {
02094             numzero = ABS(ff[1])-1;
02095             if ( reverse ) {
02096                 *tonew++ = -SNUMBER; *tonew++ = ff[1] < 0 ? -1: 1;
02097                 while ( numzero > 0 ) {
02098                     *tonew++ = -SNUMBER; *tonew++ = 0; numzero--;
02099                 }
02100             }
02101             else {
02102                 while ( numzero > 0 ) {
02103                     *tonew++ = -SNUMBER; *tonew++ = 0; numzero--;
02104                 }
02105                 *tonew++ = -SNUMBER; *tonew++ = ff[1] < 0 ? -1: 1;
02106             }
02107             ff += 2;
02108         }
02109         else if ( *ff < 0 ) { return(0); }
02110         else {
02111             if ( *ff != ARGHEAD+8 || ff[ARGHEAD] != 8
02112                  || ff[ARGHEAD+1] != SYMBOL || ABS(ff[ARGHEAD+7]) != 3
02113                  || ff[ARGHEAD+6] != 1 ) return(0);
02114             numzero = ff[ARGHEAD+5];
02115             if ( numzero >= MAXPOSITIVE4 ) return(0);
```

```
02116             numzero--;
02117             if ( reverse ) {
02118                 if ( ff[ARGHEAD+7] > 0 ) { *tonew++ = -SNUMBER; *tonew++ = 1; }
02119                 else {
02120                     *tonew++ = ARGHEAD+8; *tonew++ = 0; FILLARG(tonew)
02121                     *tonew++ = 8; *tonew++ = SYMBOL; *tonew++ = ff[ARGHEAD+3];
02122                     *tonew++ = ff[ARGHEAD+4]; *tonew++ = 1; *tonew++ = 1;
02123                     *tonew++ = -3;
02124                 }
02125                 while ( numzero > 0 ) {
02126                     *tonew++ = -SNUMBER; *tonew++ = 0; numzero--;
02127                 }
02128             }
02129             else {
02130                 while ( numzero > 0 ) {
02131                     *tonew++ = -SNUMBER; *tonew++ = 0; numzero--;
02132                 }
02133                 *tonew++ = ARGHEAD+8; *tonew++ = 0; FILLARG(tonew)
02134                 *tonew++ = 8; *tonew++ = SYMBOL; *tonew++ = 4;
02135                 *tonew++ = ff[ARGHEAD+3]; *tonew++ = ff[ARGHEAD+4];
02136                 *tonew++ = 1; *tonew++ = 1;
02137                 if ( ff[ARGHEAD+7] > 0 ) *tonew++ = 3;
02138                 else                     *tonew++ = -3;
02139             }
02140             ff += *ff;
02141         }
02142         if ( tonew > AT.WorkTop ) goto OverWork;
02143         iarg++;
02144     }
02145 /*
02146     Copy the tail, settle the size and copy the whole thing back.
02147 */
02148     while ( ff < tstop ) *tonew++ = *ff++;
02149     i = newfun[1] = tonew-newfun;
02150     NCOPY(fun,newfun,i)
02151     return(0);
02152 OverWork:;
02153     MLOCK(ErrorMessageLock);
02154     MesWork();
02155     MUNLOCK(ErrorMessageLock);
02156     return(-1);
02157 }
02158
02159 /*
02160         #] RunExplode :
02161         #[ RunPermute :
02162 */
02163
02164 int RunPermute(PHEAD WORD *fun, WORD *args, WORD *info)
02165 {
02166     WORD *tt, totarg, *tstop, arg1, arg2, n, num, i, *f, *f1, *f2, *infostop;
02167     WORD *in, *iw, withdollar;
02168     DOLLARS d;
02169     if ( *args != ARGRANGE ) {
02170         MLOCK(ErrorMessageLock);
02171         MesPrint("Illegal range encountered in RunPermute");
02172         MUNLOCK(ErrorMessageLock);
02173         Terminate(-1);
02174     }
02175     if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
02176       tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
02177       while ( tt < tstop ) { totarg++; NEXTARG(tt); }
02178       arg1 = 1; arg2 = totarg;
02179 /*
02180       We need to:
02181         1: get pointers to the arguments
02182         2: permute the pointers
02183         3: copy the arguments to safe territory in the new order
02184         4: copy this new order back in situ.
02185 */
02186       num = arg2-arg1+1;
02187       WantAddPointers(num); /* Guarantees the presence of enough pointers */
02188       f = fun+FUNHEAD; n = 1; i = 0;
02189       while ( n < arg1 ) { n++; NEXTARG(f) }
02190       f1 = f;
02191       while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; NEXTARG(f) }
02192 /*
02193       Now the permutations
02194 */
02195       info++;
02196       while ( *info ) {
02197         infostop = info + *info;
02198         info++;
02199         if ( *info > totarg ) return(0);
02200 /*
02201         Now we have a look whether there are dollar variables to be expanded
02202         We also shift out all values that are out of range.
```

```
02203 */
02204         withdollar = 0;  in = info;
02205         while ( in < infostop ) {
02206             if ( *in < 0 ) { /* Dollar variable -(number+1) */
02207                 d = Dollars - *in - 1;
02208 #ifdef WITHPTHREADS
02209                 {
02210                     int nummodopt, dtype = -1, numdollar = -*in-1;
02211                     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
02212                         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
02213                             if ( numdollar == ModOptdollars[nummodopt].number ) break;
02214                         }
02215                         if ( nummodopt < NumModOptdollars ) {
02216                             dtype = ModOptdollars[nummodopt].type;
02217                             if ( DollarLocalCopy(dtype) ) {
02218                                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
02219                             }
02220                             else {
02221                                 LOCK(d->pthreadslock);
02222                             }
02223                         }
02224                     }
02225                 }
02226 #endif
02227                 if ( ( d->type == DOLNUMBER || d->type == DOLTERMS )
02228                  && d->where[0] == 4 && d->where[4] == 0 ) {
02229                     if ( d->where[3] < 0 || d->where[2] != 1 || d->where[1] > totarg ) return(0);
02230                 }
02231                 else if ( d->type == DOLWILDARGS ) {
02232                     iw = d->where+1;
02233                     while ( *iw ) {
02234                         if ( *iw == -SNUMBER ) {
02235                             if ( iw[1] <= 0 || iw[1] > totarg ) return(0);
02236                         }
02237                         else goto IllType;
02238                         iw += 2;
02239                     }
02240                 }
02241                 else {
02242 IllType:
02243                     MLOCK(ErrorMessageLock);
02244                     MesPrint("Illegal type of $-variable in RunPermute");
02245                     MUNLOCK(ErrorMessageLock);
02246                     Terminate(-1);
02247                 }
02248                 withdollar++;
02249             }
02250             else if ( *in > totarg ) return(0);
02251             in++;
02252         }
02253         if ( withdollar ) { /* We need some space for a copy */
02254             WORD *incopy, *tocopy;
02255             incopy = TermMalloc("RunPermute");
02256             tocopy = incopy+1; in = info;
02257             while ( in < infostop ) {
02258                 if ( *in < 0 ) {
02259                     d = Dollars - *in - 1;
02260 #ifdef WITHPTHREADS
02261                 {
02262                     int nummodopt, dtype = -1, numdollar = -*in-1;
02263                     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
02264                         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
02265                             if ( numdollar == ModOptdollars[nummodopt].number ) break;
02266                         }
02267                         if ( nummodopt < NumModOptdollars ) {
02268                             dtype = ModOptdollars[nummodopt].type;
02269                             if ( DollarLocalCopy(dtype) ) {
02270                                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
02271                             }
02272                             else {
02273                                 LOCK(d->pthreadslock);
02274                             }
02275                         }
02276                     }
02277                 }
02278 #endif
02279                     if ( d->type == DOLNUMBER || d->type == DOLTERMS ) {
02280                         *tocopy++ = d->where[1] - 1;
02281                     }
02282                     else if ( d->type == DOLWILDARGS ) {
02283                         iw = d->where+1;
02284                         while ( *iw ) {
02285                             *tocopy++ = iw[1] - 1;
02286                             iw += 2;
02287                         }
02288                     }
02289                     in++;
```

```
02290                 }
02291                 else *tocopy++ = *in++;
02292             }
02293             *tocopy = 0;
02294             *incopy = tocopy - incopy;
02295             in = incopy+1;
02296             tt = AT.pWorkSpace[AT.pWorkPointer+*in];
02297             in++;
02298             while ( in < tocopy ) {
02299                 if ( *in > totarg ) return(0);
02300                 AT.pWorkSpace[AT.pWorkPointer+in[-1]] = AT.pWorkSpace[AT.pWorkPointer+*in];
02301                 in++;
02302             }
02303             AT.pWorkSpace[AT.pWorkPointer+in[-1]] = tt;
02304             TermFree(incopy,"RunPermute");
02305             info = infostop;
02306         }
02307         else {
02308             tt = AT.pWorkSpace[AT.pWorkPointer+*info];
02309             info++;
02310             while ( info < infostop ) {
02311                 if ( *info > totarg ) return(0);
02312                 AT.pWorkSpace[AT.pWorkPointer+info[-1]] = AT.pWorkSpace[AT.pWorkPointer+*info];
02313                 info++;
02314             }
02315             AT.pWorkSpace[AT.pWorkPointer+info[-1]] = tt;
02316         }
02317       }
02318 /*
02319       info++;
02320       while ( *info ) {
02321         infostop = info + *info;
02322         info++;
02323         if ( *info > totarg ) return(0);
02324         tt = AT.pWorkSpace[AT.pWorkPointer+*info];
02325         info++;
02326         while ( info < infostop ) {
02327             if ( *info > totarg ) return(0);
02328             AT.pWorkSpace[AT.pWorkPointer+info[-1]] = AT.pWorkSpace[AT.pWorkPointer+*info];
02329             info++;
02330         }
02331         AT.pWorkSpace[AT.pWorkPointer+info[-1]] = tt;
02332       }
02333 */
02334 /*
02335       And the final cleanup
02336 */
02337       if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
02338       f2 = tstop;
02339       for ( i = 0; i < num; i++ ) { f = AT.pWorkSpace[AT.pWorkPointer+i]; COPY1ARG(f2,f) }
02340       i = f2 - tstop;
02341       NCOPY(f1,tstop,i)
02342     }
02343     else {  /* tensors */
02344       tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = tstop-tt;
02345       arg1 = 1; arg2 = totarg;
02346       num = arg2-arg1+1;
02347       WantAddPointers(num); /* Guarantees the presence of enough pointers */
02348       f = fun+FUNHEAD; n = 1; i = 0;
02349       while ( n < arg1 ) { n++; f++; }
02350       f1 = f;
02351       while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; f++; }
02352 /*
02353       Now the permutations
02354 */
02355       info++;
02356       while ( *info ) {
02357         infostop = info + *info;
02358         info++;
02359         if ( *info > totarg ) return(0);
02360         tt = AT.pWorkSpace[AT.pWorkPointer+*info];
02361         info++;
02362         while ( info < infostop ) {
02363             if ( *info > totarg ) return(0);
02364             AT.pWorkSpace[AT.pWorkPointer+info[-1]] = AT.pWorkSpace[AT.pWorkPointer+*info];
02365             info++;
02366         }
02367         AT.pWorkSpace[AT.pWorkPointer+info[-1]] = tt;
02368       }
02369 /*
02370       And the final cleanup
02371 */
02372       if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
02373       f2 = tstop;
02374       for ( i = 0; i < num; i++ ) { f = AT.pWorkSpace[AT.pWorkPointer+i]; *f2++= *f++; }
02375       i = f2 - tstop;
02376       NCOPY(f1,tstop,i)
```

```
02377     }
02378     return(0);
02379 OverWork:;
02380     MLOCK(ErrorMessageLock);
02381     MesWork();
02382     MUNLOCK(ErrorMessageLock);
02383     return(-1);
02384 }
02385
02386 /*
02387         #] RunPermute :
02388         #[ RunReverse :
02389 */
02390
02391 int RunReverse(PHEAD WORD *fun, WORD *args)
02392 {
02393     WORD *tt, totarg, *tstop, arg1, arg2, n, num, i, *f, *f1, *f2, i1, i2;
02394     if ( *args != ARGRANGE ) {
02395         MLOCK(ErrorMessageLock);
02396         MesPrint("Illegal range encountered in RunReverse");
02397         MUNLOCK(ErrorMessageLock);
02398         Terminate(-1);
02399     }
02400     if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
02401       tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
02402       while ( tt < tstop ) { totarg++; NEXTARG(tt); }
02403       if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
02404 /*
02405       We need to:
02406         1: get pointers to the arguments
02407         2: reverse the order of the pointers
02408         3: copy the arguments to safe territory in the new order
02409         4: copy this new order back in situ.
02410 */
02411       if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
02412       if ( arg2 > totarg ) return(0);
02413
02414       num = arg2-arg1+1;
02415       WantAddPointers(num); /* Guarantees the presence of enough pointers */
02416       f = fun+FUNHEAD; n = 1; i = 0;
02417       while ( n < arg1 ) { n++; NEXTARG(f) }
02418       f1 = f;
02419       while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; NEXTARG(f) }
02420       i1 = i-1; i2 = 0;
02421       while ( i1 > i2 ) {
02422         tt = AT.pWorkSpace[AT.pWorkPointer+i1];
02423         AT.pWorkSpace[AT.pWorkPointer+i1] = AT.pWorkSpace[AT.pWorkPointer+i2];
02424         AT.pWorkSpace[AT.pWorkPointer+i2] = tt;
02425         i1--; i2++;
02426       }
02427       if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
02428       f2 = tstop;
02429       for ( i = 0; i < num; i++ ) { f = AT.pWorkSpace[AT.pWorkPointer+i]; COPY1ARG(f2,f) }
02430       i = f2 - tstop;
02431       NCOPY(f1,tstop,i)
02432     }
02433     else {  /* Tensors */
02434       tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = tstop - tt;
02435       if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
02436 /*
02437       We need to:
02438         1: get pointers to the arguments
02439         2: reverse the order of the pointers
02440         3: copy the arguments to safe territory in the new order
02441         4: copy this new order back in situ.
02442 */
02443       if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
02444       if ( arg2 > totarg ) return(0);
02445
02446       num = arg2-arg1+1;
02447       WantAddPointers(num); /* Guarantees the presence of enough pointers */
02448       f = fun+FUNHEAD; n = 1; i = 0;
02449       while ( n < arg1 ) { n++; f++; }
02450       f1 = f;
02451       while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; f++; }
02452       i1 = i-1; i2 = 0;
02453       while ( i1 > i2 ) {
02454         tt = AT.pWorkSpace[AT.pWorkPointer+i1];
02455         AT.pWorkSpace[AT.pWorkPointer+i1] = AT.pWorkSpace[AT.pWorkPointer+i2];
02456         AT.pWorkSpace[AT.pWorkPointer+i2] = tt;
02457         i1--; i2++;
02458       }
02459       if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
02460       f2 = tstop;
02461       for ( i = 0; i < num; i++ ) { f = AT.pWorkSpace[AT.pWorkPointer+i]; *f2++ = *f++; }
02462       i = f2 - tstop;
02463       NCOPY(f1,tstop,i)
```

```
02464     }
02465     return(0);
02466 OverWork:;
02467     MLOCK(ErrorMessageLock);
02468     MesWork();
02469     MUNLOCK(ErrorMessageLock);
02470     return(-1);
02471 }
02472
02473 /*
02474         #] RunReverse :
02475         #[ RunDedup :
02476 */
02477
02478 int RunDedup(PHEAD WORD *fun, WORD *args)
02479 {
02480     WORD *tt, totarg, *tstop, arg1, arg2, n, i, j,k, *f, *f1, *f2, *fd, *fstart;
02481     if ( *args != ARGRANGE ) {
02482         MLOCK(ErrorMessageLock);
02483         MesPrint("Illegal range encountered in RunDedup");
02484         MUNLOCK(ErrorMessageLock);
02485         Terminate(-1);
02486     }
02487     if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
02488       tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
02489       while ( tt < tstop ) { totarg++; NEXTARG(tt); }
02490       if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
02491
02492       if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
02493       if ( arg2 > totarg ) return(0);
02494
02495       f = fun+FUNHEAD; n = 1;
02496       while ( n < arg1 ) { n++; NEXTARG(f) }
02497       f1 = f; // fast forward to first element in range
02498       i = 0; // new argument count
02499       fstart = f1;
02500
02501       for (; n <= arg2; n++ ) {
02502         f2 = fstart;
02503         for ( j = 0; j < i; j++ ) { // check all previous terms
02504             fd = f2;
02505             NEXTARG(fd)
02506             for ( k = 0; k < fd-f2; k++ ) // byte comparison of args
02507                 if ( f2[k] != f[k] ) break;
02508
02509             if ( k == fd-f2 ) break; // duplicate arg
02510             f2 = fd;
02511         }
02512
02513         if ( j == i ) {
02514             // unique factor, copy in situ
02515             COPY1ARG(f1,f)
02516             i++;
02517         } else {
02518             NEXTARG(f)
02519         }
02520       }
02521
02522       // move the terms from after the range
02523       for (j = n; j <= totarg; j++) {
02524         COPY1ARG(f1,f)
02525       }
02526
02527       fun[1] = f1 - fun; // resize function
02528     }
02529     else {  /* Tensors */
02530       tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = tstop - tt;
02531       if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
02532
02533       if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
02534       if ( arg2 > totarg ) return(0);
02535
02536       f = fun+FUNHEAD;
02537       i = arg1; // new argument count
02538       n = i;
02539
02540       for (; n <= arg2; n++ ) {
02541         for ( j = arg1; j < i; j++ ) { // check all previous terms
02542             if ( f[n-1] == f[j-1] ) break; // duplicate arg
02543         }
02544
02545         if ( j == i ) {
02546             // unique factor, copy in situ
02547             f[i-1] = f[n-1];
02548             i++;
02549         }
02550       }
```

```
02551
02552       // move the terms from after the range
02553       for (j = n; j <= totarg; j++, i++) {
02554         f[i-1] = f[j-1];
02555       }
02556
02557       fun[1] = f + i - 1 - fun; // resize function
02558     }
02559     return(0);
02560 }
02561
02562 /*
02563         #] RunDedup :
02564         #[ RunCycle :
02565 */
02566
02567 int RunCycle(PHEAD WORD *fun, WORD *args, WORD *info)
02568 {
02569     WORD *tt, totarg, *tstop, arg1, arg2, n, num, i, j, *f, *f1, *f2, x, ncyc, cc;
02570     if ( *args != ARGRANGE ) {
02571         MLOCK(ErrorMessageLock);
02572         MesPrint("Illegal range encountered in RunCycle");
02573         MUNLOCK(ErrorMessageLock);
02574         Terminate(-1);
02575     }
02576     ncyc = info[1];
02577     if ( ncyc >= MAXPOSITIVE2 ) { /* $ variable */
02578         ncyc -= MAXPOSITIVE2;
02579         if ( ncyc >= MAXPOSITIVE4 ) {
02580             ncyc -= MAXPOSITIVE4; /* -$ */
02581             cc = -1;
02582         }
02583         else cc = 1;
02584         ncyc = DolToNumber(BHEAD ncyc);
02585         if ( AN.ErrorInDollar ) {
02586             MesPrint(" Error in Dollar variable in transform,cycle()=$");
02587             return(-1);
02588         }
02589         if ( ncyc >= MAXPOSITIVE4 || ncyc <= -MAXPOSITIVE4 ) {
02590             MesPrint(" Illegal value from Dollar variable in transform,cycle()=$");
02591             return(-1);
02592         }
02593         ncyc *= cc;
02594     }
02595     if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
02596       tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
02597       while ( tt < tstop ) { totarg++; NEXTARG(tt); }
02598       if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
02599       if ( arg1 > arg2 ) { n = arg1; arg1 = arg2; arg2 = n; }
02600       if ( arg2 > totarg ) return(0);
02601 /*
02602       We need to:
02603         1: get pointers to the arguments
02604         2: cycle the pointers
02605         3: copy the arguments to safe territory in the new order
02606         4: copy this new order back in situ.
02607 */
02608       num = arg2-arg1+1;
02609       WantAddPointers(num); /* Guarantees the presence of enough pointers */
02610       f = fun+FUNHEAD; n = 1; i = 0;
02611       while ( n < arg1 ) { n++; NEXTARG(f) }
02612       f1 = f;
02613       while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; NEXTARG(f) }
02614 /*
02615       Now the cycle(s). First minimize the number of cycles.
02616 */
02617       x = ncyc;
02618       if ( x >= i ) {
02619         x %= i;
02620         if ( x > i/2 ) x -= i;
02621       }
02622       else if ( x <= -i ) {
02623         x = -((-x) % i);
02624         if ( x <= -i/2 ) x += i;
02625       }
02626       while ( x ) {
02627         if ( x > 0 ) {
02628             tt = AT.pWorkSpace[AT.pWorkPointer+i-1];
02629             for ( j = i-1; j > 0; j-- )
02630                 AT.pWorkSpace[AT.pWorkPointer+j] = AT.pWorkSpace[AT.pWorkPointer+j-1];
02631             AT.pWorkSpace[AT.pWorkPointer] = tt;
02632             x--;
02633         }
02634         else {
02635             tt = AT.pWorkSpace[AT.pWorkPointer];
02636             for ( j = 1; j < i; j++ )
02637                 AT.pWorkSpace[AT.pWorkPointer+j-1] = AT.pWorkSpace[AT.pWorkPointer+j];
```

```
02638             AT.pWorkSpace[AT.pWorkPointer+j-1] = tt;
02639             x++;
02640           }
02641         }
02642 /*
02643         And the final cleanup
02644 */
02645         if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
02646         f2 = tstop;
02647         for ( i = 0; i < num; i++ ) { f = AT.pWorkSpace[AT.pWorkPointer+i]; COPY1ARG(f2,f) }
02648         i = f2 - tstop;
02649         NCOPY(f1,tstop,i)
02650       }
02651       else {  /* Tensors */
02652         tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = tstop - tt;
02653         if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
02654         if ( arg1 > arg2 ) { n = arg1; arg1 = arg2; arg2 = n; }
02655         if ( arg2 > totarg ) return(0);
02656 /*
02657         We need to:
02658           1: get pointers to the arguments
02659           2: cycle the pointers
02660           3: copy the arguments to safe territory in the new order
02661           4: copy this new order back in situ.
02662 */
02663         num = arg2-arg1+1;
02664         WantAddPointers(num); /* Guarantees the presence of enough pointers */
02665         f = fun+FUNHEAD; n = 1; i = 0;
02666         while ( n < arg1 ) { n++; f++; }
02667         f1 = f;
02668         while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; f++; }
02669 /*
02670         Now the cycle(s). First minimize the number of cycles.
02671 */
02672         x = ncyc;
02673         if ( x >= i ) {
02674           x %= i;
02675           if ( x > i/2 ) x -= i;
02676         }
02677         else if ( x <= -i ) {
02678           x = -((-x) % i);
02679           if ( x <= -i/2 ) x += i;
02680         }
02681         while ( x ) {
02682           if ( x > 0 ) {
02683               tt = AT.pWorkSpace[AT.pWorkPointer+i-1];
02684               for ( j = i-1; j > 0; j-- )
02685                   AT.pWorkSpace[AT.pWorkPointer+j] = AT.pWorkSpace[AT.pWorkPointer+j-1];
02686               AT.pWorkSpace[AT.pWorkPointer] = tt;
02687               x--;
02688           }
02689           else {
02690               tt = AT.pWorkSpace[AT.pWorkPointer];
02691               for ( j = 1; j < i; j++ )
02692                   AT.pWorkSpace[AT.pWorkPointer+j-1] = AT.pWorkSpace[AT.pWorkPointer+j];
02693               AT.pWorkSpace[AT.pWorkPointer+j-1] = tt;
02694               x++;
02695           }
02696         }
02697 /*
02698         And the final cleanup
02699 */
02700         if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
02701         f2 = tstop;
02702         for ( i = 0; i < num; i++ ) { f = AT.pWorkSpace[AT.pWorkPointer+i]; *f2++ = *f++; }
02703         i = f2 - tstop;
02704         NCOPY(f1,tstop,i)
02705       }
02706       return(0);
02707 OverWork:;
02708       MLOCK(ErrorMessageLock);
02709       MesWork();
02710       MUNLOCK(ErrorMessageLock);
02711       return(-1);
02712 }
02713
02714 /*
02715         #] RunCycle :
02716         #[ RunAddArg :
02717 */
02718
02719 int RunAddArg(PHEAD WORD *fun, WORD *args)
02720 {
02721       WORD *tt, totarg, *tstop, arg1, arg2, n, num, *f, *f1, *f2;
02722       WORD scribble[10+ARGHEAD];
02723       LONG space;
02724       if ( *args != ARGRANGE ) {
```

```
02725         MLOCK(ErrorMessageLock);
02726         MesPrint("Illegal range encountered in RunAddArg");
02727         MUNLOCK(ErrorMessageLock);
02728         Terminate(-1);
02729     }
02730     if ( functions[fun[0]-FUNCTION].spec == TENSORFUNCTION ) {
02731         MLOCK(ErrorMessageLock);
02732         MesPrint("Illegal attempt to add arguments of a tensor in AddArg");
02733         MUNLOCK(ErrorMessageLock);
02734         Terminate(-1);
02735     }
02736     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
02737     while ( tt < tstop ) { totarg++; NEXTARG(tt); }
02738     /* ignore functions with no arguments */
02739     if ( totarg == 0 ) return(0);
02740     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
02741 /*
02742     We need to:
02743         1: establish that we actually need to add something
02744         2: start a sort
02745         3: if needed, convert arguments to long arguments
02746         4: send (terms in) argument to StoreTerm
02747         5: EndSort and copy the result back into the function
02748     Note that the function is in the workspace, above the term and no
02749     relevant information is trailing it.
02750 */
02751     if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
02752     if ( arg2 > totarg ) return(0);
02753     num = arg2-arg1+1;
02754     if ( num == 1 ) return(0);
02755     f = fun+FUNHEAD; n = 1;
02756     while ( n < arg1 ) { n++; NEXTARG(f) }
02757     f1 = f;
02758     NewSort(BHEAD0);
02759     while ( n <= arg2 ) {
02760         if ( *f > 0 ) {
02761             f2 = f + *f; f += ARGHEAD;
02762             while ( f < f2 ) { StoreTerm(BHEAD f); f += *f; }
02763         }
02764         else if ( *f == -SNUMBER && f[1] == 0 ) {
02765             f+= 2;
02766         }
02767         else {
02768             ToGeneral(f,scribble,1);
02769             StoreTerm(BHEAD scribble);
02770             NEXTARG(f);
02771         }
02772         n++;
02773     }
02774     if ( EndSort(BHEAD tstop+ARGHEAD,1) < 0 ) return(-1);
02775     num = 0;
02776     f2 = tstop+ARGHEAD;
02777     while ( *f2 ) { f2 += *f2; num++; }
02778     *tstop = f2-tstop;
02779     for ( n = 1; n < ARGHEAD; n++ ) tstop[n] = 0;
02780     if ( num == 1 && ToFast(tstop,tstop) == 1 ) {
02781         f2 = tstop; NEXTARG(f2);
02782     }
02783     if ( *tstop == ARGHEAD ) {
02784         *tstop = -SNUMBER; tstop[1] = 0;
02785         f2 = tstop+2;
02786     }
02787 /*
02788     Copy the trailing arguments after the new argument, then copy the whole back.
02789 */
02790     while ( f < tstop ) *f2++ = *f++;
02791     while ( f < f2 ) *f1++ = *f++;
02792     space = f1 - fun;
02793     if ( (space+8)*sizeof(WORD) > (UWORD)AM.MaxTer ) {
02794         MLOCK(ErrorMessageLock);
02795         MesWork();
02796         MUNLOCK(ErrorMessageLock);
02797         return(-1);
02798     }
02799     fun[1] = (WORD)space;
02800     return(0);
02801 }
02802
02803 /*
02804         #] RunAddArg :
02805         #[ RunMulArg :
02806 */
02807
02808 int RunMulArg(PHEAD WORD *fun, WORD *args)
02809 {
02810     WORD *t, totarg, *tstop, arg1, arg2, n, *f, nb, *m, i, *w;
02811     WORD *scratch, argbuf[20], argsize, *where, *newterm;
```

```
02812     LONG oldcpointer_pos;
02813     CBUF *C = cbuf + AT.ebufnum;
02814     if ( *args != ARGRANGE ) {
02815         MLOCK(ErrorMessageLock);
02816         MesPrint("Illegal range encountered in RunMulArg");
02817         MUNLOCK(ErrorMessageLock);
02818         Terminate(-1);
02819     }
02820     if ( functions[fun[0]-FUNCTION].spec == TENSORFUNCTION ) {
02821         MLOCK(ErrorMessageLock);
02822         MesPrint("Illegal attempt to multiply arguments of a tensor in MulArg");
02823         MUNLOCK(ErrorMessageLock);
02824         Terminate(-1);
02825     }
02826     t = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
02827     while ( t < tstop ) { totarg++; NEXTARG(t); }
02828     /* ignore functions with no arguments */
02829     if ( totarg == 0 ) return(0);
02830     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
02831     if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
02832     if ( arg1 > totarg ) return(0);
02833     if ( arg2 < 1 ) return(0);
02834     if ( arg1 < 1 ) arg1 = 1;
02835     if ( arg2 > totarg ) arg2 = totarg;
02836     if ( arg1 == arg2 ) return(0);
02837 /*
02838     Now we move the arguments to a compiler buffer
02839     Then we create a term in the workspace that is the product of
02840     subexpression pointers to the objects in the compiler buffer.
02841     Next we let Generator work out that term.
02842     Finally we pick up the results from EndSort and put it in the function.
02843 */
02844     f = fun+FUNHEAD; n = 1;
02845     while ( n < arg1 ) { n++; NEXTARG(f) }
02846     t = f;
02847     if ( fun >= AT.WorkSpace && fun < AT.WorkTop ) {
02848         if ( AT.WorkPointer < fun+fun[1] ) AT.WorkPointer = fun+fun[1];
02849     }
02850     scratch = AT.WorkPointer;
02851     w = scratch+1;
02852     oldcpointer_pos = C->Pointer-C->Buffer;
02853     nb = C->numrhs;
02854     while ( n <= arg2 ) {
02855         if ( *t > 0 ) {
02856             argsize = *t - ARGHEAD; where = t + ARGHEAD; t += *t;
02857         }
02858         else if ( *t <= -FUNCTION ) {
02859             argbuf[0] = FUNHEAD+4; argbuf[1] = -*t++; argbuf[2] = FUNHEAD;
02860             for ( i = 2; i < FUNHEAD; i++ ) argbuf[i+1] = 0;
02861             argbuf[FUNHEAD+1] = 1;
02862             argbuf[FUNHEAD+2] = 1;
02863             argbuf[FUNHEAD+3] = 3;
02864             argsize = argbuf[0];
02865             where = argbuf;
02866         }
02867         else if ( *t == -SYMBOL ) {
02868             argbuf[0] = 8; argbuf[1] = SYMBOL; argbuf[2] = 4;
02869             argbuf[3] = t[1]; argbuf[4] = 1;
02870             argbuf[5] = 1; argbuf[6] = 1; argbuf[7] = 3;
02871             argsize = 8; t += 2;
02872             where = argbuf;
02873         }
02874         else if ( *t == -VECTOR || *t == -MINVECTOR ) {
02875             argbuf[0] = 7; argbuf[1] = INDEX; argbuf[2] = 3;
02876             argbuf[3] = t[1];
02877             argbuf[4] = 1; argbuf[5] = 1;
02878             if ( *t == -MINVECTOR ) argbuf[6] = -3;
02879             else argbuf[6] = 3;
02880             argsize = 7; t += 2;
02881             where = argbuf;
02882         }
02883         else if ( *t == -INDEX ) {
02884             argbuf[0] = 7; argbuf[1] = INDEX; argbuf[2] = 3;
02885             argbuf[3] = t[1];
02886             argbuf[4] = 1; argbuf[5] = 1; argbuf[6] = 3;
02887             argsize = 7; t += 2;
02888             where = argbuf;
02889         }
02890         else if ( *t == -SNUMBER ) {
02891             if ( t[1] < 0 ) {
02892                 argbuf[0] = 4; argbuf[1] = -t[1]; argbuf[2] = 1; argbuf[3] = -3;
02893             }
02894             else {
02895                 argbuf[0] = 4; argbuf[1] = t[1]; argbuf[2] = 1; argbuf[3] = 3;
02896             }
02897             argsize = 4; t += 2;
02898             where = argbuf;
```

```
02899         }
02900         else {
02901             /* unreachable */
02902             return(1);
02903         }
02904 /*
02905         Now add the argbuf to AT.ebufnum
02906 */
02907         m = AddRHS(AT.ebufnum,1);
02908         while ( (m + argsize + 10) > C->Top ) m = DoubleCbuffer(AT.ebufnum,m,17);
02909         for ( i = 0; i < argsize; i++ ) m[i] = where[i];
02910         m[i] = 0;
02911         C->Pointer = m + i + 1;
02912         n++;
02913         *w++ = SUBEXPRESSION; *w++ = SUBEXPSIZE; *w++ = C->numrhs; *w++ = 1;
02914         *w++ = AT.ebufnum; FILLSUB(w);
02915     }
02916     *w++ = 1; *w++ = 1; *w++ = 3;
02917     *scratch = w-scratch;
02918     AT.WorkPointer = w;
02919     NewSort(BHEAD0);
02920     Generator(BHEAD scratch,AR.Cnumlhs);
02921     newterm = AT.WorkPointer;
02922     EndSort(BHEAD newterm+ARGHEAD,1);
02923     C->Pointer = C->Buffer+oldcpointer_pos;
02924     C->numrhs = nb;
02925     w = newterm+ARGHEAD; while ( *w ) w += *w;
02926     *newterm = w-newterm; newterm[1] = 0;
02927     if ( ToFast(newterm,newterm) ) {
02928         if ( *newterm <= -FUNCTION ) w = newterm+1;
02929         else w = newterm+2;
02930     }
02931     while ( t < tstop ) *w++ = *t++;
02932     i = w - newterm;
02933     t = newterm; NCOPY(f,t,i);
02934     fun[1] = f-fun;
02935     AT.WorkPointer = scratch;
02936     if ( AT.WorkPointer > AT.WorkSpace && AT.WorkPointer < f ) AT.WorkPointer = f;
02937     return(0);
02938 }
02939
02940 /*
02941         #] RunMulArg :
02942         #[ RunIsLyndon :
02943
02944         Determines whether the range constitutes a Lyndon word.
02945         The two cases of ordering are distinguished by the order of
02946         the numbers of the arguments in the range.
02947 */
02948
02949 int RunIsLyndon(PHEAD WORD *fun, WORD *args, int par)
02950 {
02951     WORD *tt, totarg, *tstop, arg1, arg2, arg, num, *f, n, i;
02952 /*  WORD *f1; */
02953     WORD sign, i1, i2, retval;
02954     if ( fun[0] <= GAMMASEVEN && fun[0] >= GAMMA ) return(0);
02955     if ( *args != ARGRANGE ) {
02956         MLOCK(ErrorMessageLock);
02957         MesPrint("Illegal range encountered in RunIsLyndon");
02958         MUNLOCK(ErrorMessageLock);
02959         Terminate(-1);
02960     }
02961     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
02962     while ( tt < tstop ) { totarg++; NEXTARG(tt); }
02963     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
02964     if ( arg1 > totarg || arg2 > totarg ) return(-1);
02965 /*
02966     Now make a list of the relevant arguments.
02967 */
02968     if ( arg1 == arg2 ) return(1);
02969     if ( arg2 < arg1 ) {    /* greater, rather than smaller */
02970         arg = arg1; arg1 = arg2; arg2 = arg; sign = 1;
02971     }
02972     else sign = 0;
02973
02974     num = arg2-arg1+1;
02975     WantAddPointers(num);   /* Guarantees the presence of enough pointers */
02976     f = fun+FUNHEAD; n = 1; i = 0;
02977     while ( n < arg1 ) { n++; NEXTARG(f) }
02978 /*  f1 = f; */
02979     while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; NEXTARG(f) }
02980 /*
02981     If sign == 1 we should alter the order of the pointers first
02982 */
02983     if ( sign ) {
02984         i1 = i-1; i2 = 0;
02985         while ( i1 > i2 ) {
```

```
02986             tt = AT.pWorkSpace[AT.pWorkPointer+i1];
02987             AT.pWorkSpace[AT.pWorkPointer+i1] = AT.pWorkSpace[AT.pWorkPointer+i2];
02988             AT.pWorkSpace[AT.pWorkPointer+i2] = tt;
02989             i1--; i2++;
02990         }
02991     }
02992 /*
02993     The argument range is from f1 to f and the num pointers to the arguments
02994     are in AT.pWorkSpace[AT.pWorkPointer] to AT.pWorkSpace[AT.pWorkPointer+num-1]
02995 */
02996     for ( i1 = 1; i1 < num; i1++ ) {
02997         retval = par * CompArg(AT.pWorkSpace[AT.pWorkPointer+i1],
02998                                AT.pWorkSpace[AT.pWorkPointer]);
02999         if ( retval > 0 ) continue;
03000         if ( retval < 0 ) return(0);
03001         for ( i2 = 1; i2 < num; i2++ ) {
03002             retval = par * CompArg(AT.pWorkSpace[AT.pWorkPointer+(i1+i2)%num],
03003                                AT.pWorkSpace[AT.pWorkPointer+i2]);
03004             if ( retval < 0 ) return(0);
03005             if ( retval > 0 ) goto nexti1;
03006         }
03007 /*
03008         If we come here the sequence is not unique.
03009 */
03010         return(0);
03011 nexti1:;
03012     }
03013     return(1);
03014 }
03015
03016 /*
03017         #] RunIsLyndon :
03018         #[ RunToLyndon :
03019
03020         Determines whether the range constitutes a Lyndon word.
03021         If not, we rotate it to a Lyndon word. If this is not possible
03022         we return the noLyndon condition.
03023         The two cases of ordering are distinguished by the order of
03024         the numbers of the arguments in the range.
03025 */
03026
03027 WORD RunToLyndon(PHEAD WORD *fun, WORD *args, int par)
03028 {
03029     WORD *tt, totarg, *tstop, arg1, arg2, arg, num, *f, *f1, *f2, n, i;
03030     WORD sign, i1, i2, retval, unique;
03031     if ( fun[0] <= GAMMASEVEN && fun[0] >= GAMMA ) return(0);
03032     if ( *args != ARGRANGE ) {
03033         MLOCK(ErrorMessageLock);
03034         MesPrint("Illegal range encountered in RunToLyndon");
03035         MUNLOCK(ErrorMessageLock);
03036         Terminate(-1);
03037     }
03038     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
03039     while ( tt < tstop ) { totarg++; NEXTARG(tt); }
03040     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
03041     if ( arg1 > totarg || arg2 > totarg ) return(-1);
03042 /*
03043     Now make a list of the relevant arguments.
03044 */
03045     if ( arg1 == arg2 ) return(1);
03046     if ( arg2 < arg1 ) {    /* greater, rather than smaller */
03047         arg = arg1; arg1 = arg2; arg2 = arg; sign = 1;
03048     }
03049     else sign = 0;
03050
03051     num = arg2-arg1+1;
03052     WantAddPointers((2*num));   /* Guarantees the presence of enough pointers */
03053     f = fun+FUNHEAD; n = 1; i = 0;
03054     while ( n < arg1 ) { n++; NEXTARG(f) }
03055     f1 = f;
03056     while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; NEXTARG(f) }
03057 /*
03058     If sign == 1 we should alter the order of the pointers first
03059 */
03060     if ( sign ) {
03061         i1 = i-1; i2 = 0;
03062         while ( i1 > i2 ) {
03063             tt = AT.pWorkSpace[AT.pWorkPointer+i1];
03064             AT.pWorkSpace[AT.pWorkPointer+i1] = AT.pWorkSpace[AT.pWorkPointer+i2];
03065             AT.pWorkSpace[AT.pWorkPointer+i2] = tt;
03066             i1--; i2++;
03067         }
03068     }
03069 /*
03070     The argument range is from f1 to f and the num pointers to the arguments
03071     are in AT.pWorkSpace[AT.pWorkPointer] to AT.pWorkSpace[AT.pWorkPointer+num-1]
03072 */
```

```
03073     unique = 1;
03074     for ( i1 = 1; i1 < num; i1++ ) {
03075         retval = par * CompArg(AT.pWorkSpace[AT.pWorkPointer+i1],
03076                                AT.pWorkSpace[AT.pWorkPointer]);
03077         if ( retval > 0 ) continue;
03078         if ( retval < 0 ) {
03079 Rotate:;
03080 /*
03081             Rotate so that i1 becomes the zero element. Then start again.
03082 */
03083             for ( i2 = 0; i2 < num; i2++ ) {
03084                 AT.pWorkSpace[AT.pWorkPointer+num+i2] =
03085                     AT.pWorkSpace[AT.pWorkPointer+(i1+i2)%num];
03086             }
03087             for ( i2 = 0; i2 < num; i2++ ) {
03088                 AT.pWorkSpace[AT.pWorkPointer+i2] =
03089                     AT.pWorkSpace[AT.pWorkPointer+i2+num];
03090             }
03091             i1 = 0;
03092             goto nexti1;
03093         }
03094         for ( i2 = 1; i2 < num; i2++ ) {
03095             retval = par * CompArg(AT.pWorkSpace[AT.pWorkPointer+(i1+i2)%num],
03096                                    AT.pWorkSpace[AT.pWorkPointer+i2]);
03097             if ( retval < 0 ) goto Rotate;
03098             if ( retval > 0 ) goto nexti1;
03099         }
03100 /*
03101         If we come here the sequence is not unique.
03102 */
03103         unique = 0;
03104 nexti1:;
03105     }
03106     if ( sign ) {
03107         i1 = i-1; i2 = 0;
03108         while ( i1 > i2 ) {
03109             tt = AT.pWorkSpace[AT.pWorkPointer+i1];
03110             AT.pWorkSpace[AT.pWorkPointer+i1] = AT.pWorkSpace[AT.pWorkPointer+i2];
03111             AT.pWorkSpace[AT.pWorkPointer+i2] = tt;
03112             i1--; i2++;
03113         }
03114     }
03115 /*
03116     Now rewrite the arguments into the proper order
03117 */
03118     if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
03119     f2 = tstop;
03120     for ( i = 0; i < num; i++ ) { f = AT.pWorkSpace[AT.pWorkPointer+i]; COPY1ARG(f2,f) }
03121     i = f2 - tstop;
03122     NCOPY(f1,tstop,i)
03123 /*
03124     The return value indicates whether we have a Lyndon word
03125 */
03126     return(unique);
03127 OverWork:;
03128     MLOCK(ErrorMessageLock);
03129     MesWork();
03130     MUNLOCK(ErrorMessageLock);
03131     return(-2);
03132 }
03133
03134 /*
03135         #] RunToLyndon :
03136         #[ RunDropArg :
03137 */
03138
03139 int RunDropArg(PHEAD WORD *fun, WORD *args)
03140 {
03141     WORD *t, *tstop, *f, totarg, arg1, arg2, n;
03142
03143     t = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
03144     while ( t < tstop ) { totarg++; NEXTARG(t); }
03145     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
03146     if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
03147     if ( arg1 > totarg ) return(0);
03148     if ( arg2 < 1 ) return(0);
03149     if ( arg1 < 1 ) arg1 = 1;
03150     if ( arg2 > totarg ) arg2 = totarg;
03151     f = fun+FUNHEAD; n = 1;
03152     while ( n < arg1 ) { n++; NEXTARG(f) }
03153     t = f;
03154     while ( n <= arg2 ) { n++; NEXTARG(t) }
03155     while ( t < tstop ) *f++ = *t++;
03156     fun[1] = f-fun;
03157     return(0);
03158 }
03159
```

```
03160 /*
03161         #] RunDropArg :
03162         #[ RunSelectArg :
03163 */
03164
03165 int RunSelectArg(PHEAD WORD *fun, WORD *args)
03166 {
03167     WORD *t, *tstop, *f, *tt, totarg, arg1, arg2, n;
03168
03169     t = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
03170     while ( t < tstop ) { totarg++; NEXTARG(t); }
03171     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
03172     if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
03173     if ( arg1 > totarg ) return(0);
03174     if ( arg2 < 1 ) return(0);
03175     if ( arg1 < 1 ) arg1 = 1;
03176     if ( arg2 > totarg ) arg2 = totarg;
03177     f = fun+FUNHEAD; n = 1; t = f;
03178     while ( n < arg1 ) { n++; NEXTARG(t) }
03179     while ( n <= arg2 ) {
03180         tt = t; NEXTARG(tt)
03181         while ( t < tt ) *f++ = *t++;
03182         n++;
03183     }
03184     fun[1] = f-fun;
03185     return(0);
03186 }
03187
03188 /*
03189         #] RunSelectArg :
03190         #[ RunZtoHArg :
03191 */
03192
03193 int RunZtoHArg(PHEAD WORD *fun, WORD *args)
03194 {
03195     WORD *tt, totarg, *tstop, arg1, arg2, n, i, *f, *f1;
03196     int sign = 0;
03197     WORD *t, *t1, *t2, *t3;
03198     if ( *args != ARGRANGE ) {
03199         MLOCK(ErrorMessageLock);
03200         MesPrint("Illegal range encountered in RunZtoHArg.");
03201         MUNLOCK(ErrorMessageLock);
03202         Terminate(-1);
03203     }
03204     if ( functions[fun[0]-FUNCTION].spec != 0 ) {
03205         MLOCK(ErrorMessageLock);
03206         MesPrint("The ZtoH transformation can only be executed on regular functions with nonzero
      integer arguments.");
03207         MUNLOCK(ErrorMessageLock);
03208         Terminate(-1);
03209     }
03210     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
03211     while ( tt < tstop ) { totarg++; NEXTARG(tt); }
03212     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
03213 /*
03214     Check the arguments. Should be -SNUMBER x!=0
03215 */
03216     f = fun+FUNHEAD; n = 1;
03217     while ( n < arg1 ) { n++; NEXTARG(f) }
03218     f1 = f;
03219     for ( i = arg1; i <= arg2; i++, f += 2 ) {
03220         if ( *f != -SNUMBER || f[1] == 0 ) return(-1);
03221     }
03222 /*
03223     Now we need a copy.
03224 */
03225     t = f1; t1 = t2 = tt = TermMalloc("RunZtoHArg");
03226     while ( t < f ) { *t1++ = *t++; *t1++ = *t++; }
03227     t = f1;
03228     while ( t2 < t1 ) {
03229         t += 2;
03230         if ( t2[1] < 0 ) {
03231             t3 = t;
03232             while ( t3 < f ) { t3[1] = -t3[1]; t3 += 2; }
03233         }
03234         t2 += 2;
03235     }
03236     TermFree(tt,"RunZtoHArg");
03237 /*
03238     Now the overall sign.
03239 */
03240     while ( f1 < f ) { if ( f1[1] < 0 ) sign = 1-sign; f1 += 2; }
03241     return(sign);
03242 }
03243
03244 /*
03245         #] RunZtoHArg :
```

```
03246         #[ RunHtoZArg :
03247 */
03248
03249 int RunHtoZArg(PHEAD WORD *fun, WORD *args)
03250 {
03251     WORD *tt, totarg, *tstop, arg1, arg2, n, i, *f, *f1, *f2;
03252     int sign = 0;
03253     WORD *t, *t1, *t2;
03254     if ( *args != ARGRANGE ) {
03255         MLOCK(ErrorMessageLock);
03256         MesPrint("Illegal range encountered in RunZtoHArg.");
03257         MUNLOCK(ErrorMessageLock);
03258         Terminate(-1);
03259     }
03260     if ( functions[fun[0]-FUNCTION].spec != 0 ) {
03261         MLOCK(ErrorMessageLock);
03262         MesPrint("The HtoZ transformation can only be executed on regular functions with nonzero
      integer arguments.");
03263         MUNLOCK(ErrorMessageLock);
03264         Terminate(-1);
03265     }
03266     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
03267     while ( tt < tstop ) { totarg++; NEXTARG(tt); }
03268     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
03269 /*
03270     Check the arguments. Should be -SNUMBER x!=0
03271 */
03272     f = fun+FUNHEAD; n = 1;
03273     while ( n < arg1 ) { n++; NEXTARG(f) }
03274     f2 = f1 = f;
03275     for ( i = arg1; i <= arg2; i++, f += 2 ) {
03276         if ( *f != -SNUMBER || f[1] == 0 ) return(-1);
03277     }
03278 /*
03279     First the overall sign.
03280 */
03281     while ( f2 < f ) { if ( f2[1] < 0 ) sign = 1-sign; f2 += 2; }
03282 /*
03283     Now we need a copy.
03284 */
03285     t = f1; t1 = tt = TermMalloc("RunHtoZArg");
03286     while ( t < f ) { *t1++ = *t++; *t1++ = *t++; }
03287 /*
03288     Now the transformation.
03289 */
03290     t = f1; t2 = tt + 2;
03291     while ( t2 < t1 ) {
03292         t += 2;
03293         if ( t2[-1] < 0 ) t[1] = -t[1];
03294         t2 += 2;
03295     }
03296     TermFree(tt,"RunHtoZArg");
03297     return(sign);
03298 }
03299
03300 /*
03301         #] RunHtoZArg :
03302         #[ TestArgNum :
03303
03304         Looks whether argument n is contained in any of the ranges
03305         specified in args. Args contains objects of the types
03306             ALLARGS
03307             NUMARG,num
03308             ARGRANGE,num1,num2
03309         The object MAKEARGS,num1,num2 is skipped
03310         Any other object terminates the range specifications.
03311
03312         Currently only ARGRANGE is used (10-may-2016)
03313 */
03314
03315 int TestArgNum(int n, int totarg, WORD *args)
03316 {
03317     GETIDENTITY
03318     WORD x1, x2;
03319     for(;;) {
03320         switch ( *args ) {
03321             case ALLARGS:
03322                 return(1);
03323             case NUMARG:
03324                 if ( n == args[1] ) return(1);
03325                 if ( args[1] >= MAXPOSITIVE4 ) {
03326                     x1 = args[1]-MAXPOSITIVE4;
03327                     if ( totarg-x1 == n ) return(1);
03328                 }
03329                 args += 2;
03330                 break;
03331             case ARGRANGE:
```

```
03332                 if ( args[1] >= MAXPOSITIVE2 ) {
03333                     x1 = args[1] - MAXPOSITIVE2;
03334                     if ( x1 > MAXPOSITIVE4 ) {
03335                         x1 = x1 - MAXPOSITIVE4;
03336                         x1 = DolToNumber(BHEAD x1);
03337                         x1 = totarg - x1;
03338                     }
03339                     else {
03340                         x1 = DolToNumber(BHEAD x1);
03341                     }
03342                 }
03343                 else if ( args[1] >= MAXPOSITIVE4 ) {
03344                     x1 = totarg-(args[1]-MAXPOSITIVE4);
03345                 }
03346                 else x1 = args[1];
03347                 if ( args[2] >= MAXPOSITIVE2 ) {
03348                     x2 = args[2] - MAXPOSITIVE2;
03349                     if ( x2 > MAXPOSITIVE4 ) {
03350                         x2 = x2 - MAXPOSITIVE4;
03351                         x2 = DolToNumber(BHEAD x2);
03352                         x2 = totarg - x2;
03353                     }
03354                     else {
03355                         x2 = DolToNumber(BHEAD x2);
03356                     }
03357                 }
03358                 else if ( args[2] >= MAXPOSITIVE4 ) {
03359                     x2 = totarg-(args[2]-MAXPOSITIVE4);
03360                 }
03361                 else x2 = args[2];
03362                 if ( x1 >= x2 ) {
03363                     if ( n >= x2 && n <= x1 ) return(1);
03364                 }
03365                 else {
03366                     if ( n >= x1 && n <= x2 ) return(1);
03367                 }
03368                 args += 3;
03369                 break;
03370             case MAKEARGS:
03371                 args += 3;
03372                 break;
03373             default:
03374                 return(0);
03375         }
03376     }
03377 }
03378
03379 /*
03380         #] TestArgNum :
03381         #[ PutArgInScratch :
03382 */
03383
03384 WORD PutArgInScratch(WORD *arg,UWORD *scrat)
03385 {
03386     WORD size, *t, i;
03387     if ( *arg == -SNUMBER ) {
03388         scrat[0] = ABS(arg[1]);
03389         if ( arg[1] < 0 ) size = -1;
03390         else              size =  1;
03391     }
03392     else {
03393         t = arg+*arg-1;
03394         if ( *t < 0 ) { i = ((-*t)-1)/2; size = -i; }
03395         else          { i = (  *t -1)/2; size =  i; }
03396         t = arg+ARGHEAD+1;
03397         NCOPY(scrat,t,i);
03398     }
03399     return(size);
03400 }
03401
03402 /*
03403         #] PutArgInScratch :
03404         #[ ReadRange :
03405
03406         Comes in at the bracket and leaves at the = sign
03407         Ranges can be:
03408             #1,#2  with # numbers. If the second is smaller than the
03409                    first we work it backwards.
03410             first,#2 or #2,first
03411             #1,last  or last,#1
03412             first,last or last,first
03413         First is represented by 1. Last is represented by MAXPOSITIVE4.
03414
03415         par = 0: we need the = after.
03416         par = 1: we need a , or '\0' after.
03417         par = 2: we need a :
03418 */
```

```
03419
03420 UBYTE *ReadRange(UBYTE *s, WORD *out, int par)
03421 {
03422     UBYTE *in = s, *ss, c;
03423     LONG x1, x2;
03424
03425     SKIPBRA3(in)
03426     if ( par == 0 && in[1] != '=' ) {
03427         MesPrint("&A range in this type of transform statement should be followed by an = sign");
03428         return(0);
03429     }
03430     else if ( par == 1 && in[1] != ',' && in[1] != '\0' ) {
03431         MesPrint("&A range in this type of transform statement should be followed by a comma or
      end-of-statement");
03432         return(0);
03433     }
03434     else if ( par == 2 && in[1] != ':' ) {
03435         MesPrint("&A range in this type of transform statement should be followed by a :");
03436         return(0);
03437     }
03438     s++;
03439     if ( FG.cTable[*s] == 0 ) {
03440         ss = s; while ( FG.cTable[*s] == 0 ) s++;
03441         c = *s; *s = 0;
03442         if ( StrICmp(ss,(UBYTE *)"first") == 0 ) {
03443             *s = c;
03444             x1 = 1;
03445         }
03446         else if ( StrICmp(ss,(UBYTE *)"last") == 0 ) {
03447             *s = c;
03448             if ( c == '-' ) {
03449                 s++;
03450                 if ( *s == '$' ) {
03451                     s++; ss = s;
03452                     while ( FG.cTable[*s] == 0 || FG.cTable[*s] == 1 ) s++;
03453                     c = *s; *s = 0;
03454                     if ( ( x1 = GetDollar(ss) ) < 0 ) goto Error;
03455                     *s = c;
03456                     x1 += MAXPOSITIVE2;
03457                 }
03458                 else {
03459                     x1 = 0;
03460                     while ( *s >= '0' && *s <= '9' ) {
03461                         x1 = 10*x1 + *s++ - '0';
03462                         if ( x1 >= MAXPOSITIVE4 ) {
03463                             MesPrint("&Fixed range indicator bigger than %l",(LONG)MAXPOSITIVE4);
03464                             return(0);
03465                         }
03466                     }
03467                 }
03468                 x1 += MAXPOSITIVE4;
03469             }
03470             else x1 = MAXPOSITIVE4;
03471         }
03472         else {
03473             MesPrint("&Illegal keyword inside range specification");
03474             return(0);
03475         }
03476     }
03477     else if ( FG.cTable[*s] == 1 ) {
03478         x1 = 0;
03479         while ( *s >= '0' && *s <= '9' ) {
03480             x1 = x1*10 + *s++ - '0';
03481             if ( x1 >= MAXPOSITIVE4 ) {
03482                 MesPrint("&Fixed range indicator bigger than %l",(LONG)MAXPOSITIVE4);
03483                 return(0);
03484             }
03485         }
03486     }
03487     else if ( *s == '$' ) {
03488         s++; ss = s;
03489         while ( FG.cTable[*s] == 0 || FG.cTable[*s] == 1 ) s++;
03490         c = *s; *s = 0;
03491         if ( ( x1 = GetDollar(ss) ) < 0 ) goto Error;
03492         *s = c;
03493         x1 += MAXPOSITIVE2;
03494     }
03495     else {
03496         MesPrint("&Illegal character in range specification");
03497         return(0);
03498     }
03499     if ( *s != ',' ) {
03500         MesPrint("&A range is two indicators, separated by a comma or blank");
03501         return(0);
03502     }
03503     s++;
03504     if ( FG.cTable[*s] == 0 ) {
```

```
03505         ss = s; while ( FG.cTable[*s] == 0 ) s++;
03506         c = *s; *s = 0;
03507         if ( StrICmp(ss,(UBYTE *)"first") == 0 ) {
03508             *s = c;
03509             x2 = 1;
03510         }
03511         else if ( StrICmp(ss,(UBYTE *)"last") == 0 ) {
03512             *s = c;
03513             if ( c == '-' ) {
03514                 s++;
03515                 if ( *s == '$' ) {
03516                     s++; ss = s;
03517                     while ( FG.cTable[*s] == 0 || FG.cTable[*s] == 1 ) s++;
03518                     c = *s; *s = 0;
03519                     if ( ( x2 = GetDollar(ss) ) < 0 ) goto Error;
03520                     *s = c;
03521                     x2 += MAXPOSITIVE2;
03522                 }
03523                 else {
03524                     x2 = 0;
03525                     while ( *s >= '0' && *s <= '9' ) {
03526                         x2 = 10*x2 + *s++ - '0';
03527                         if ( x2 >= MAXPOSITIVE4 ) {
03528                             MesPrint("&Fixed range indicator bigger than %l",(LONG)MAXPOSITIVE4);
03529                             return(0);
03530                         }
03531                     }
03532                 }
03533                 x2 += MAXPOSITIVE4;
03534             }
03535             else x2 = MAXPOSITIVE4;
03536         }
03537         else {
03538             MesPrint("&Illegal keyword inside range specification");
03539             return(0);
03540         }
03541     }
03542     else if ( FG.cTable[*s] == 1 ) {
03543         x2 = 0;
03544         while ( *s >= '0' && *s <= '9' ) {
03545             x2 = x2*10 + *s++ - '0';
03546             if ( x2 >= MAXPOSITIVE4 ) {
03547                 MesPrint("&Fixed range indicator bigger than %l",(LONG)MAXPOSITIVE4);
03548                 return(0);
03549             }
03550         }
03551     }
03552     else if ( *s == '$' ) {
03553         s++; ss = s;
03554         while ( FG.cTable[*s] == 0 || FG.cTable[*s] == 1 ) s++;
03555         c = *s; *s = 0;
03556         if ( ( x2 = GetDollar(ss) ) < 0 ) goto Error;
03557         *s = c;
03558         x2 += MAXPOSITIVE2;
03559     }
03560     else {
03561         MesPrint("&Illegal character in range specification");
03562         return(0);
03563     }
03564     if ( s < in ) {
03565         MesPrint("&A range is two indicators, separated by a comma or blank between parentheses");
03566         return(0);
03567     }
03568     out[0] = x1; out[1] = x2;
03569     return(in+1);
03570 Error:
03571     MesPrint("&Undefined variable $%s in range",ss);
03572     return(0);
03573 }
03574
03575 /*
03576         #] ReadRange :
03577         #[ FindRange :
03578 */
03579
03580 int FindRange(PHEAD WORD *args, WORD *arg1, WORD *arg2, WORD totarg)
03581 {
03582     WORD n[2], fromlast, i;
03583     for ( i = 0; i < 2; i++ ) {
03584         n[i] = args[i+1];
03585         fromlast = 0;
03586         if ( n[i] >= MAXPOSITIVE2 ) { /* This is a dollar variable */
03587             n[i] -= MAXPOSITIVE2;
03588             if ( n[i] >= MAXPOSITIVE4 ) {
03589                 fromlast = 1;
03590                 n[i] -= MAXPOSITIVE4; /* Now we have the number of the dollar variable   */
03591             }
```

```
03592             n[i] = DolToNumber(BHEAD n[i]);
03593             if ( AN.ErrorInDollar ) {
03594                 MLOCK(ErrorMessageLock);
03595                 MesPrint("Illegal $ value in range while executing transform statement.");
03596                 MUNLOCK(ErrorMessageLock);
03597                 return(-1);
03598             }
03599             if ( fromlast ) n[i] = totarg-n[i];
03600         }
03601         else if ( n[i] >= MAXPOSITIVE4 ) { n[i] = totarg-(n[i]-MAXPOSITIVE4); }
03602         if ( n[i] <= 0 ) {
03603             MLOCK(ErrorMessageLock);
03604             MesPrint("Illegal non-positive value in range (%d) while executing transform statement.",
      i+1);
03605             MUNLOCK(ErrorMessageLock);
03606             return(-1);
03607         }
03608     }
03609     *arg1 = n[0];
03610     *arg2 = n[1];
03611     return(0);
03612 }
03613
03614 /*
03615         #] FindRange :
03616     #] Transform :
03617 */
```

4.153 unix.h File Reference

Macros

- #define LINEFEED '\n'
- #define CARRIAGERETURN 0x0D
- #define WITHPIPE
- #define WITHSYSTEM
- #define WITHEXTERNALCHANNEL
- #define TRAPSIGNALS
- #define P_term(code) exit((int)(code<0?-code:code))
- #define SEPARATOR '/'
- #define ALTSEPARATOR '/'
- #define PATHSEPARATOR ':'
- #define WITH_ENV
- #define WITH_ALARM

4.153.1 Detailed Description

Settings for Unix-like systems.

Definition in file unix.h.

4.153.2 Macro Definition Documentation

4.153.2.1 LINEFEED

```
#define LINEFEED '\n'
```

Definition at line 33 of file unix.h.

4.153.2.2 CARRIAGERETURN

```
#define CARRIAGERETURN 0x0D
```

Definition at line 34 of file unix.h.

4.153.2.3 WITHPIPE

```
#define WITHPIPE
```

Definition at line 36 of file unix.h.

4.153.2.4 WITHSYSTEM

```
#define WITHSYSTEM
```

Definition at line 37 of file unix.h.

4.153.2.5 WITHEXTERNALCHANNEL

```
#define WITHEXTERNALCHANNEL
```

Definition at line 46 of file unix.h.

4.153.2.6 TRAPSIGNALS

```
#define TRAPSIGNALS
```

Definition at line 49 of file unix.h.

4.153.2.7 P_term

```
#define P_term(
              code ) exit((int)(code<0?-code:code))
```

Definition at line 52 of file unix.h.

4.153.2.8 SEPARATOR

```
#define SEPARATOR '/'
```

Definition at line 54 of file unix.h.

4.153.2.9 ALTSEPARATOR

```
#define ALTSEPARATOR '/'
```

Definition at line 55 of file unix.h.

4.153.2.10 PATHSEPARATOR

```
#define PATHSEPARATOR ':'
```

Definition at line 56 of file unix.h.

4.153.2.11 WITH_ENV

```
#define WITH_ENV
```

Definition at line 57 of file unix.h.

4.153.2.12 WITH_ALARM

```
#define WITH_ALARM
```

Definition at line 58 of file unix.h.

4.154 unix.h

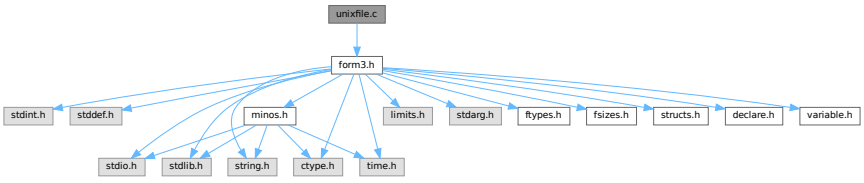
Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032
00033 #define LINEFEED '\n'
00034 #define CARRIAGERETURN 0x0D
00035
00036 #define WITHPIPE
00037 #define WITHSYSTEM
00038
00039 /*[13jul2005 mt]:*/
```

```
00040 /*With SAFESIGNAL defined, write() and read() syscalls are wrapped by
00041 the errno checkup*/
00042 /*#define SAFESIGNAL*/
00043 /*:[13jul2005 mt]*/
00044
00045 /*[29apr2004 mt]:*/
00046 #define WITHEXTERNALCHANNEL
00047 /*
00048 */
00049 #define TRAPSIGNALS
00050 /*:[29apr2004 mt]*/
00051
00052 #define P_term(code)    exit((int)(code<0?-code:code))
00053
00054 #define SEPARATOR '/'
00055 #define ALTSEPARATOR '/'
00056 #define PATHSEPARATOR ':'
00057 #define WITH_ENV
00058 #define WITH_ALARM
```

4.155 unixfile.c File Reference

`#include "form3.h"`
Include dependency graph for unixfile.c:



4.155.1 Detailed Description

The interface to a fast variety of file routines in the unix system.

Definition in file unixfile.c.

4.156 unixfile.c

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
```

```
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
00026  *
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
00031 /*
00032     #[ Includes :
00033
00034     File with direct interface to the UNIX functions.
00035     This makes a big difference in speed!
00036 */
00037 #include "form3.h"
00038
00039
00040 /*[13jul2005 mt]:*/
00041 #ifdef SAFESIGNAL
00042 /*[15oct2004 mt]:*/
00043 /*To access errno variable and EINTR constant:*/
00044 #include <errno.h>
00045 /*:[15oct2004 mt]*/
00046 #endif
00047 /*:[13jul2005 mt]*/
00048
00049 #ifdef UFILES
00050
00051 FILES TheStdout = { 1 };
00052 FILES *Ustdout = &TheStdout;
00053
00054 #ifdef GPP
00055 extern "C" open();
00056 extern "C" close();
00057 extern "C" read();
00058 extern "C" write();
00059 extern "C" lseek();
00060 #endif
00061
00062 /*
00063     #] Includes :
00064     #[ Uopen :
00065 */
00066
00067 FILES *Uopen(char *filename, char *mode)
00068 {
00069     FILES *f = (FILES *)Malloc1(sizeof(FILES),"Uopen");
00070     int flags = 0, rights = 0644;
00071     while ( *mode ) {
00072         if      ( *mode == 'r' ) { flags |= O_RDONLY; }
00073         else if ( *mode == 'w' ) { flags |= O_CREAT | O_TRUNC; }
00074         else if ( *mode == 'a' ) { flags |= O_APPEND; }
00075         else if ( *mode == 'b' ) { }
00076         else if ( *mode == '+' ) { flags |= O_RDWR; }
00077         mode++;
00078     }
00079     f->descriptor = open(filename,flags,rights);
00080     if ( f->descriptor >= 0 ) return(f);
00081     if ( ( flags & O_APPEND ) != 0 ) {
00082         flags |= O_CREAT;
00083         f->descriptor = open(filename,flags,rights);
00084         if ( f->descriptor >= 0 ) return(f);
00085     }
00086     M_free(f,"Uopen");
00087     return(0);
00088 }
00089
00090 /*
00091     #] Uopen :
00092     #[ Uclose :
00093 */
00094
00095 int Uclose(FILES *f)
00096 {
00097     int retval;
00098     if ( f ) {
00099         retval = close(f->descriptor);
00100         M_free(f,"Uclose");
00101         return(retval);
00102     }
00103     return(EOF);
00104 }
00105
00106 /*
00107     #] Uclose :
00108     #[ Uread :
00109 */
00110
```

```
00111
00112 size_t Uread(char *ptr, size_t size, size_t nobj, FILES *f)
00113 {
00114 /*[13jul2005 mt]:*/
00115 #ifdef SAFESIGNAL
00116
00117 /*[15oct2004 mt]:*/
00118 /*Operation read() can be interrupted by a signal. Note, this is rather unlikely,
00119 so we do not save size*nobj for future attempts*/
00120 size_t ret;
00121     /*If the syscall is interrupted by a signal before it
00122         succeeded in getting any progress, it must be repeated:*/
00123     while( ( (ret=read(f->descriptor,ptr,size*nobj))<1)&&(errno == EINTR) );
00124     return(ret);
00125 #else
00126 #ifdef DEEPDEBUG
00127     {
00128         POSITION pos;
00129         SETBASEPOSITION(pos,lseek(f->descriptor,0L,SEEK_CUR));
00130         MesPrint("handle %d: reading %ld bytes from position %p\n",f->descriptor,size*nobj,pos);
00131     }
00132 #endif
00133
00134     return(read(f->descriptor,ptr,size*nobj));
00135
00136 #endif
00137 /*:[15oct2004 mt]*/
00138 /*:[13jul2005 mt]*/
00139 }
00140
00141 /*
00142     #] Uread :
00143     #[ Uwrite :
00144 */
00145
00146 size_t Uwrite(char *ptr, size_t size, size_t nobj, FILES *f)
00147 {
00148 /*[13jul2005 mt]:*/
00149 #ifdef SAFESIGNAL
00150 /*[15oct2004 mt]:*/
00151 /*Operation write() can be interrupted by a signal. */
00152 size_t ret;
00153 size_t thesize=size*nobj;
00154     /*If the syscall is interrupted by a signal before it
00155         succeeded in getting any progress, it must be repeated:*/
00156     while( ( (ret=write(f->descriptor,ptr,thesize))<1 )&&(errno == EINTR) );
00157
00158     return(ret);
00159 #else
00160 #ifdef DEEPDEBUG
00161     {
00162         POSITION pos;
00163         SETBASEPOSITION(pos,lseek(f->descriptor,0L,SEEK_CUR));
00164         MesPrint("handle %d: writing %ld bytes to position %p\n",f->descriptor,size*nobj,pos);
00165     }
00166 #endif
00167     return(write(f->descriptor,ptr,size*nobj));
00168
00169 /*:[15oct2004 mt]*/
00170 #endif
00171 /*:[13jul2005 mt]*/
00172 }
00173
00174 /*
00175     #] Uwrite :
00176     #[ Useek :
00177 */
00178
00179 int Useek(FILES *f, off_t offset, int origin)
00180 {
00181     if ( f && ( lseek(f->descriptor,offset,origin) >= 0 ) ) return(0);
00182     return(-1);
00183 }
00184
00185 /*
00186     #] Useek :
00187     #[ Utell :
00188 */
00189
00190 off_t Utell(FILES *f)
00191 {
00192     if ( f ) return((off_t)lseek(f->descriptor,0L,SEEK_CUR));
00193     else return(-1);
00194 }
00195
00196 /*
00197     #] Utell :
```

```
00198     #[ Uflush :
00199 */
00200
00201 void Uflush(FILES *f) { DUMMYUSE(f); }
00202
00203 /*
00204     #] Uflush :
00205     #[ Ugetpos :
00206 */
00207
00208 int Ugetpos(FILES *f, fpos_t *ptr)
00209 {
00210     DUMMYUSE(f); DUMMYUSE(ptr);
00211     return(-1);
00212 }
00213
00214 /*
00215     #] Ugetpos :
00216     #[ Usetpos :
00217 */
00218
00219 int Usetpos(FILES *f,fpos_t *ptr)
00220 {
00221     DUMMYUSE(f); DUMMYUSE(ptr);
00222     return(-1);
00223 }
00224
00225 /*
00226     #] Usetpos :
00227     #[ Usetbuf :
00228 */
00229
00230 void Usetbuf(FILES *f, char *ptr) { DUMMYUSE(f); DUMMYUSE(ptr); }
00231
00232 /*
00233     #] Usetbuf :
00234 */
00235 #endif
```

4.157 variable.h File Reference

This graph shows which files directly or indirectly include this file:

Macros

- #define chartype FG.cTable
- #define Procedures ((PROCEDURE *)(AP.ProcList.lijst))
- #define NumProcedures AP.ProcList.num
- #define MaxProcedures AP.ProcList.maxnum
- #define DoLoops ((DOLOOP *)(AP.LoopList.lijst))
- #define NumDoLoops AP.LoopList.num
- #define MaxDoLoops AP.LoopList.maxnum
- #define PreVar ((PREVAR *)(AP.PreVarList.lijst))
- #define NumPre AP.PreVarList.num
- #define MaxNumPre AP.PreVarList.maxnum
- #define SetElements ((WORD *)(AC.SetElementList.lijst))
- #define Sets ((SETS)(AC.SetList.lijst))
- #define functions ((FUNCTIONS)(AC.FunctionList.lijst))
- #define indices ((INDICES)(AC.IndexList.lijst))
- #define symbols ((SYMBOLS)(AC.SymbolList.lijst))
- #define vectors ((VECTORS)(AC.VectorList.lijst))

- #define tablebases ((DBASE *)(AC.TableBaseList.lijst))
- #define NumFunctions AC.FunctionList.num
- #define NumIndices AC.IndexList.num
- #define NumSymbols AC.SymbolList.num
- #define NumVectors AC.VectorList.num
- #define NumSets AC.SetList.num
- #define NumSetElements AC.SetElementList.num
- #define NumTableBases AC.TableBaseList.num
- #define GlobalFunctions AC.FunctionList.numglobal
- #define GlobalIndices AC.IndexList.numglobal
- #define GlobalSymbols AC.SymbolList.numglobal
- #define GlobalVectors AC.VectorList.numglobal
- #define GlobalSets AC.SetList.numglobal
- #define GlobalSetElements AC.SetElementList.numglobal
- #define cbuf ((CBUF *)(AC.cbufList.lijst))
- #define channels ((CHANNEL *)(AC.ChannelList.lijst))
- #define NumOutputChannels AC.ChannelList.num
- #define Dollars ((DOLLARS)(AP.DollarList.lijst))
- #define NumDollars AP.DollarList.num
- #define Dubious ((DUBIOUSV)(AC.DubiousList.lijst))
- #define NumDubious AC.DubiousList.num
- #define Expressions ((EXPRESSIONS)(AC.ExpressionList.lijst))
- #define NumExpressions AC.ExpressionList.num
- #define autofunctions ((FUNCTIONS)(AC.AutoFunctionList.lijst))
- #define autoindices ((INDICES)(AC.AutoIndexList.lijst))
- #define autosymbols ((SYMBOLS)(AC.AutoSymbolList.lijst))
- #define autovectors ((VECTORS)(AC.AutoVectorList.lijst))
- #define xsymbol (cbuf[AM.sbufnum].rhs)
- #define numxsymbol (cbuf[AM.sbufnum].numrhs)
- #define PotModdollars ((WORD *)(AC.PotModDolList.lijst))
- #define NumPotModdollars AC.PotModDolList.num
- #define ModOptdollars ((MODOPTDOLLAR *)(AC.ModOptDolList.lijst))
- #define NumModOptdollars AC.ModOptDolList.num
- #define BUG A.bug;
- #define AC A.Cc
- #define AM A.M
- #define AN A.N
- #define AO A.O
- #define AP A.P
- #define AR A.R
- #define AS A.S
- #define AT A.T
- #define AX A.X

Variables

- WRITEBUFTOEXTCHANNEL **writeBufToExtChannel**
- GETCFROMEXTCHANNEL **getcFromExtChannel**
- SETTERMINATORFOREXTERNALCHANNEL **setTerminatorForExternalChannel**
- SETKILLMODEFOREXTERNALCHANNEL **setKillModeForExternalChannel**
- WRITEFILE WriteFile
- ALLGLOBALS A
- FIXEDGLOBALS FG
- FIXEDSET fixedsets [ ]
- char * setupfilename

4.157.1 Detailed Description

Contains a number of defines to make the coding easier. Especially the defines for the use of the lists are very nice. And of course the AC for A.Cc and AT for either A.T of B->T are indispensable to keep FORM and TFORM in one set of sources.

Definition in file variable.h.

4.157.2 Macro Definition Documentation

4.157.2.1 chartype

```
#define chartype FG.cTable
```

Definition at line 77 of file variable.h.

4.157.2.2 Procedures

```
#define Procedures ((PROCEDURE *)(AP.ProcList.lijst))
```

Definition at line 79 of file variable.h.

4.157.2.3 NumProcedures

```
#define NumProcedures AP.ProcList.num
```

Definition at line 80 of file variable.h.

4.157.2.4 MaxProcedures

```
#define MaxProcedures AP.ProcList.maxnum
```

Definition at line 81 of file variable.h.

4.157.2.5 DoLoops

```
#define DoLoops ((DOLOOP *)(AP.LoopList.lijst))
```

Definition at line 82 of file variable.h.

4.157.2.6 NumDoLoops

```
#define NumDoLoops AP.LoopList.num
```

Definition at line 83 of file variable.h.

4.157.2.7 MaxDoLoops

```
#define MaxDoLoops AP.LoopList.maxnum
```

Definition at line 84 of file variable.h.

4.157.2.8 PreVar

```
#define PreVar ((PREVAR *)(AP.PreVarList.lijst))
```

Definition at line 85 of file variable.h.

4.157.2.9 NumPre

```
#define NumPre AP.PreVarList.num
```

Definition at line 86 of file variable.h.

4.157.2.10 MaxNumPre

```
#define MaxNumPre AP.PreVarList.maxnum
```

Definition at line 87 of file variable.h.

4.157.2.11 SetElements

```
#define SetElements ((WORD *)(AC.SetElementList.lijst))
```

Definition at line 88 of file variable.h.

4.157.2.12 Sets

```
#define Sets ((SETS)(AC.SetList.lijst))
```

Definition at line 89 of file variable.h.

4.157.2.13 functions

```
#define functions ((FUNCTIONS)(AC.FunctionList.lijst))
```

Definition at line 90 of file variable.h.

4.157.2.14 indices

```
#define indices ((INDICES)(AC.IndexList.lijst))
```

Definition at line 91 of file variable.h.

4.157.2.15 symbols

```
#define symbols ((SYMBOLS)(AC.SymbolList.lijst))
```

Definition at line 92 of file variable.h.

4.157.2.16 vectors

```
#define vectors ((VECTORS)(AC.VectorList.lijst))
```

Definition at line 93 of file variable.h.

4.157.2.17 tablebases

```
#define tablebases ((DBASE *)(AC.TableBaseList.lijst))
```

Definition at line 94 of file variable.h.

4.157.2.18 NumFunctions

```
#define NumFunctions AC.FunctionList.num
```

Definition at line 95 of file variable.h.

4.157.2.19 NumIndices

```
#define NumIndices AC.IndexList.num
```

Definition at line 96 of file variable.h.

4.157.2.20 NumSymbols

```
#define NumSymbols AC.SymbolList.num
```

Definition at line 97 of file variable.h.

4.157.2.21 NumVectors

```
#define NumVectors AC.VectorList.num
```

Definition at line 98 of file variable.h.

4.157.2.22 NumSets

```
#define NumSets AC.SetList.num
```

Definition at line 99 of file variable.h.

4.157.2.23 NumSetElements

```
#define NumSetElements AC.SetElementList.num
```

Definition at line 100 of file variable.h.

4.157.2.24 NumTableBases

```
#define NumTableBases AC.TableBaseList.num
```

Definition at line 101 of file variable.h.

4.157.2.25 GlobalFunctions

```
#define GlobalFunctions AC.FunctionList.numglobal
```

Definition at line 102 of file variable.h.

4.157.2.26 GlobalIndices

```
#define GlobalIndices AC.IndexList.numglobal
```

Definition at line 103 of file variable.h.

4.157.2.27 GlobalSymbols

```
#define GlobalSymbols AC.SymbolList.numglobal
```

Definition at line 104 of file variable.h.

4.157.2.28 GlobalVectors

```
#define GlobalVectors AC.VectorList.numglobal
```

Definition at line 105 of file variable.h.

4.157.2.29 GlobalSets

```
#define GlobalSets AC.SetList.numglobal
```

Definition at line 106 of file variable.h.

4.157.2.30 GlobalSetElements

```
#define GlobalSetElements AC.SetElementList.numglobal
```

Definition at line 107 of file variable.h.

4.157.2.31 cbuf

```
#define cbuf ((CBUF *)(AC.cbufList.lijst))
```

Definition at line 108 of file variable.h.

4.157.2.32 channels

```
#define channels ((CHANNEL *)(AC.ChannelList.lijst))
```

Definition at line 109 of file variable.h.

4.157.2.33 NumOutputChannels

```
#define NumOutputChannels AC.ChannelList.num
```

Definition at line 110 of file variable.h.

4.157.2.34 Dollars

```
#define Dollars ((DOLLARS)(AP.DollarList.lijst))
```

Definition at line 111 of file variable.h.

4.157.2.35 NumDollars

```
#define NumDollars AP.DollarList.num
```

Definition at line 112 of file variable.h.

4.157.2.36 Dubious

```
#define Dubious ((DUBIOUSV)(AC.DubiousList.lijst))
```

Definition at line 113 of file variable.h.

4.157.2.37 NumDubious

```
#define NumDubious AC.DubiousList.num
```

Definition at line 114 of file variable.h.

4.157.2.38 Expressions

```
#define Expressions ((EXPRESSIONS)(AC.ExpressionList.lijst))
```

Definition at line 115 of file variable.h.

4.157.2.39 NumExpressions

```
#define NumExpressions AC.ExpressionList.num
```

Definition at line 116 of file variable.h.

4.157.2.40 autofunctions

```
#define autofunctions ((FUNCTIONS)(AC.AutoFunctionList.lijst))
```

Definition at line 117 of file variable.h.

4.157.2.41 autoindices

```
#define autoindices ((INDICES)(AC.AutoIndexList.lijst))
```

Definition at line 118 of file variable.h.

4.157.2.42 autosymbols

```
#define autosymbols ((SYMBOLS)(AC.AutoSymbolList.lijst))
```

Definition at line 119 of file variable.h.

4.157.2.43 autovectors

```
#define autovectors ((VECTORS)(AC.AutoVectorList.lijst))
```

Definition at line 120 of file variable.h.

4.157.2.44 xsymbol

```
#define xsymbol (cbuf[AM.sbufnum].rhs)
```

Definition at line 121 of file variable.h.

4.157.2.45 numxsymbol

```
#define numxsymbol (cbuf[AM.sbufnum].numrhs)
```

Definition at line 122 of file variable.h.

4.157.2.46 PotModdollars

```
#define PotModdollars ((WORD *)(AC.PotModDolList.lijst))
```

Definition at line 124 of file variable.h.

4.157.2.47 NumPotModdollars

```
#define NumPotModdollars AC.PotModDolList.num
```

Definition at line 125 of file variable.h.

4.157.2.48 ModOptdollars

```
#define ModOptdollars ((MODOPTDOLLAR *)(AC.ModOptDolList.lijst))
```

Definition at line 126 of file variable.h.

4.157.2.49 NumModOptdollars

```
#define NumModOptdollars AC.ModOptDolList.num
```

Definition at line 127 of file variable.h.

4.157.2.50 BUG

```
#define BUG A.bug;
```

Definition at line 129 of file variable.h.

4.157.2.51 AC

```
#define AC A.Cc
```

Definition at line 145 of file variable.h.

4.157.2.52 AM

```
#define AM A.M
```

Definition at line 146 of file variable.h.

4.157.2.53 AN

```
#define AN A.N
```

Definition at line 147 of file variable.h.

4.157.2.54 AO

```
#define AO A.O
```

Definition at line 148 of file variable.h.

4.157.2.55 AP

```
#define AP A.P
```

Definition at line 149 of file variable.h.

4.157.2.56 AR

```
#define AR A.R
```

Definition at line 150 of file variable.h.

4.157.2.57 AS

```
#define AS A.S
```

Definition at line 151 of file variable.h.

4.157.2.58 AT

```
#define AT A.T
```

Definition at line 152 of file variable.h.

4.157.2.59 AX

```
#define AX A.X
```

Definition at line 153 of file variable.h.

4.157.3 Variable Documentation

4.157.3.1 WriteFile

```
WRITEFILE WriteFile  [extern]
```

Definition at line 1361 of file tools.c.

4.157.3.2 A

```
ALLGLOBALS A  [extern]
```

Definition at line 133 of file inivar.h.

4.157.3.3 FG

```
FIXEDGLOBALS FG  [extern]
```

Definition at line 34 of file inivar.h.

4.157.3.4 fixedsets

```
FIXEDSET fixedsets[]  [extern]
```

Definition at line 259 of file inivar.h.

4.157.3.5 setupfilename

```
char* setupfilename  [extern]
```

Definition at line 277 of file inivar.h.

4.158 variable.h

Go to the documentation of this file.

```
00001 #ifndef __VARIABLE__
00002
00003 #define __VARIABLE__
00004
00013 /* #[ License : */
00014 /*
00015  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00016  *   When using this file you are requested to refer to the publication
00017  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00018  *   This is considered a matter of courtesy as the development was paid
00019  *   for by FOM the Dutch physics granting agency and we would like to
00020  *   be able to track its scientific use to convince FOM of its value
00021  *   for the community.
00022  *
00023  *   This file is part of FORM.
00024  *
00025  *   FORM is free software: you can redistribute it and/or modify it under the
00026  *   terms of the GNU General Public License as published by the Free Software
00027  *   Foundation, either version 3 of the License, or (at your option) any later
00028  *   version.
00029  *
00030  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00031  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00032  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00033  *   details.
00034  *
00035  *   You should have received a copy of the GNU General Public License along
00036  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00037  */
00038 /* #] License : */
00039
00040 /*See the file extcmd.c*/
00041
00042 #ifdef REMOVEDBY_MT
00043 extern int (*writeBufToExtChannel)(char *buf, size_t n);
00044 extern int (*getcFromExtChannel)();
00045 extern int (*setTerminatorForExternalChannel)(char *newterminator);
00046 #endif
00047 extern WRITEBUFTOEXTCHANNEL writeBufToExtChannel;
00048 extern GETCFROMEXTCHANNEL getcFromExtChannel;
00049 extern SETTERMINATORFOREXTERNALCHANNEL setTerminatorForExternalChannel;
00050 extern SETKILLMODEFOREXTERNALCHANNEL setKillModeForExternalChannel;
00051
00052 /*
00053 extern LONG (*WriteFile)(int handle, UBYTE *buffer, LONG number);
00054 */
```

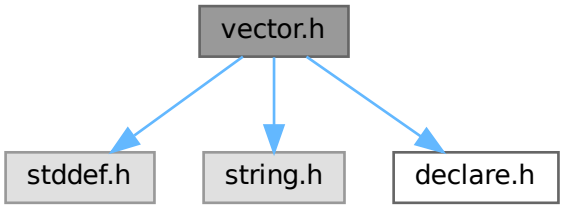
```
00055 extern WRITEFILE WriteFile;
00056 /*:[17nov2005 mt]*/
00057
00058 extern ALLGLOBALS A;
00059 #ifdef WITHPTHREADS
00060 extern ALLPRIVATES **AB;
00061 #endif
00062
00063 extern FIXEDGLOBALS FG;
00064 extern FIXEDSET fixedsets[];
00065
00066 extern char *setupfilename;
00067
00068 EXTERNLOCK(ErrorMessageLock)
00069 EXTERNLOCK(FileReadLock)
00070 EXTERNLOCK(dummylock)
00071
00072 #ifdef VMS
00073 #include <stdio.h>
00074 extern FILE **FileStructs;
00075 #endif
00076
00077 #define chartype FG.cTable
00078
00079 #define Procedures ((PROCEDURE *)(AP.ProcList.lijst))
00080 #define NumProcedures AP.ProcList.num
00081 #define MaxProcedures AP.ProcList.maxnum
00082 #define DoLoops ((DOLOOP *)(AP.LoopList.lijst))
00083 #define NumDoLoops AP.LoopList.num
00084 #define MaxDoLoops AP.LoopList.maxnum
00085 #define PreVar ((PREVAR *)(AP.PreVarList.lijst))
00086 #define NumPre AP.PreVarList.num
00087 #define MaxNumPre AP.PreVarList.maxnum
00088 #define SetElements ((WORD *)(AC.SetElementList.lijst))
00089 #define Sets ((SETS)(AC.SetList.lijst))
00090 #define functions ((FUNCTIONS)(AC.FunctionList.lijst))
00091 #define indices ((INDICES)(AC.IndexList.lijst))
00092 #define symbols ((SYMBOLS)(AC.SymbolList.lijst))
00093 #define vectors ((VECTORS)(AC.VectorList.lijst))
00094 #define tablebases ((DBASE *)(AC.TableBaseList.lijst))
00095 #define NumFunctions AC.FunctionList.num
00096 #define NumIndices AC.IndexList.num
00097 #define NumSymbols AC.SymbolList.num
00098 #define NumVectors AC.VectorList.num
00099 #define NumSets AC.SetList.num
00100 #define NumSetElements AC.SetElementList.num
00101 #define NumTableBases AC.TableBaseList.num
00102 #define GlobalFunctions AC.FunctionList.numglobal
00103 #define GlobalIndices AC.IndexList.numglobal
00104 #define GlobalSymbols AC.SymbolList.numglobal
00105 #define GlobalVectors AC.VectorList.numglobal
00106 #define GlobalSets AC.SetList.numglobal
00107 #define GlobalSetElements AC.SetElementList.numglobal
00108 #define cbuf ((CBUF *)(AC.cbufList.lijst))
00109 #define channels ((CHANNEL *)(AC.ChannelList.lijst))
00110 #define NumOutputChannels AC.ChannelList.num
00111 #define Dollars ((DOLLARS)(AP.DollarList.lijst))
00112 #define NumDollars AP.DollarList.num
00113 #define Dubious ((DUBIOUSV)(AC.DubiousList.lijst))
00114 #define NumDubious AC.DubiousList.num
00115 #define Expressions ((EXPRESSIONS)(AC.ExpressionList.lijst))
00116 #define NumExpressions AC.ExpressionList.num
00117 #define autofunctions ((FUNCTIONS)(AC.AutoFunctionList.lijst))
00118 #define autoindices ((INDICES)(AC.AutoIndexList.lijst))
00119 #define autosymbols ((SYMBOLS)(AC.AutoSymbolList.lijst))
00120 #define autovectors ((VECTORS)(AC.AutoVectorList.lijst))
00121 #define xsymbol (cbuf[AM.sbufnum].rhs)
00122 #define numxsymbol (cbuf[AM.sbufnum].numrhs)
00123
00124 #define PotModdollars ((WORD *)(AC.PotModDolList.lijst))
00125 #define NumPotModdollars AC.PotModDolList.num
00126 #define ModOptdollars ((MODOPTDOLLAR *)(AC.ModOptDolList.lijst))
00127 #define NumModOptdollars AC.ModOptDolList.num
00128
00129 #define BUG A.bug;
00130
00131 #ifdef WITHPTHREADS
00132 #define AC A.Cc
00133 #define AM A.M
00134 #define AO A.O
00135 #define AP A.P
00136 #define AS A.S
00137 #define AX A.X
00138 #define AN B->N
00139 #define AR B->R
00140 #define AT B->T
00141 #define AN0 B0->N
```

```
00142 #define AR0 B0->R
00143 #define AT0 B0->T
00144 #else
00145 #define AC A.Cc
00146 #define AM A.M
00147 #define AN A.N
00148 #define AO A.O
00149 #define AP A.P
00150 #define AR A.R
00151 #define AS A.S
00152 #define AT A.T
00153 #define AX A.X
00154 #endif
00155
00156 #endif
```

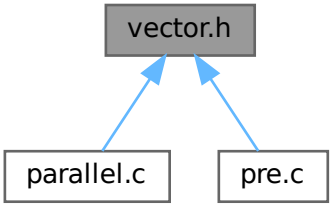
4.159 vector.h File Reference

```
#include <stddef.h>
#include <string.h>
#include "declare.h"
```

Include dependency graph for vector.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define VectorStruct(T)

- #define Vector(T, X) VectorStruct(T) X = { NULL, 0, 0 }
- #define DeclareVector(T, X) VectorStruct(T) X
- #define VectorInit(X)
- #define VectorFree(X)
- #define VectorPtr(X) ((X).ptr)
- #define VectorFront(X) ((X).ptr[0])
- #define VectorBack(X) ((X).ptr[(X).size - 1])
- #define VectorSize(X) ((X).size)
- #define VectorCapacity(X) ((X).capacity)
- #define VectorEmpty(X) ((X).size == 0)
- #define VectorClear(X) do { (X).size = 0; } while (0)
- #define VectorReserve(X, newcapacity)
- #define VectorPushBack(X, x)
- #define VectorPushBacks(X, src, n)
- #define VectorPopBack(X) do { (X).size --; } while (0)
- #define VectorInsert(X, index, x)
- #define VectorInserts(X, index, src, n)
- #define VectorErase(X, index)
- #define VectorErases(X, index, n)

4.159.1 Detailed Description

An implementation of dynamic array.

Example:
```
size_t i;
Vector(int, vec);
VectorPushBack(vec, 1);
VectorPushBack(vec, 2);
VectorPushBack(vec, 3);
for ( i = 0; i < VectorSize(vec); i++ )
  printf("%d\n", VectorPtr(vec)[i]);
VectorFree(vec);
```

Definition in file vector.h.

4.159.2 Macro Definition Documentation

4.159.2.1 VectorStruct

```
#define VectorStruct(
                T )
```

Value:
```
    struct { \
        T *ptr; \
        size_t size; \
        size_t capacity; \
    }
```

A struct for vector objects.

Parameters

T	the type of elements.

Definition at line 65 of file vector.h.

4.159.2.2 Vector

```
#define Vector(
              T,
              X )  VectorStruct(T) X = { NULL, 0, 0 }
```

Defines and initialises a vector X of the type T. The user must call VectorFree(X) after the use of X.

Parameters

T	the type of elements.
X	the name of vector object

Definition at line 84 of file vector.h.

4.159.2.3 DeclareVector

```
#define DeclareVector(
              T,
              X )  VectorStruct(T) X
```

Declares a vector X of the type T. The user must call VectorInit(X) before the use of X.

Parameters

T	the type of elements.
X	the name of vector object

Definition at line 99 of file vector.h.

4.159.2.4 VectorInit

```
#define VectorInit(
              X )
```

Value:

```
    do { \
        (X).ptr = NULL; \
        (X).size = 0; \
        (X).capacity = 0; \
    } while (0)
```

Initialises a vector X of the type T. The user must call VectorFree(X) after the use of X.

Parameters

X	the vector object.

Definition at line 113 of file vector.h.

4.159.2.5 VectorFree

```
#define VectorFree(
                X )
```

Value:

```
    do { \
        M_free((X).ptr, "VectorFree:" #X); \
        (X).ptr = NULL; \
        (X).size = 0; \
        (X).capacity = 0; \
    } while (0)
```

Frees the buffer allocated by the vector X.

Parameters

X	the vector object.

Definition at line 130 of file vector.h.

4.159.2.6 VectorPtr

```
#define VectorPtr(
                X )   ((X).ptr)
```

Returns the pointer to the buffer for the vector X. NULL when VectorCapacity(X) == 0.

Parameters

X	the vector object.

Returns

the pointer to the allocated buffer for the vector.

Definition at line 150 of file vector.h.

4.159.2.7 VectorFront

```
#define VectorFront(
                X )   ((X).ptr[0])
```

Returns the first element of the vector X. Undefined when VectorSize(X) == 0.

Parameters

X	the vector object.

Returns

the first element of the vector.

Definition at line 165 of file vector.h.

4.159.2.8 VectorBack

```
#define VectorBack(
              X )   ((X).ptr[(X).size - 1])
```

Returns the last element of the vector X. Undefined when VectorSize(X) == 0.

Parameters

X	the vector object.

Returns

the last element of the vector.

Definition at line 180 of file vector.h.

4.159.2.9 VectorSize

```
#define VectorSize(
              X )   ((X).size)
```

Returns the size of the vector X.

Parameters

X	the vector object.

Returns

the size of the vector.

Definition at line 194 of file vector.h.

4.159.2.10 VectorCapacity

```
#define VectorCapacity(
              X )   ((X).capacity)
```

Returns the capacity (the number of the already allocated elements) of the vector X.

Parameters

X	the vector object.

Returns

the capacity of the vector.

Definition at line 208 of file vector.h.

4.159.2.11 VectorEmpty

```
#define VectorEmpty(
              X )  ((X).size == 0)
```

Returns true the size of the vector X is zero.

Parameters

X	the vector object.

Returns

true if the vector has no elements, false otherwise.

Definition at line 222 of file vector.h.

4.159.2.12 VectorClear

```
#define VectorClear(
              X )  do { (X).size = 0; } while (0)
```

Sets the size of the vector X to zero.

Parameters

X	the vector object.

Definition at line 235 of file vector.h.

4.159.2.13 VectorReserve

```
#define VectorReserve(
              X,
              newcapacity )
```

Value:

```
    do { \
        size_t v_tmp_newcapacity_ = (newcapacity); \
        if ( (X).capacity < v_tmp_newcapacity_ ) { \
            void *v_tmp_newptr_; \
            v_tmp_newcapacity_ = (v_tmp_newcapacity_ * 3) / 2; \
            if ( v_tmp_newcapacity_ < 4 ) v_tmp_newcapacity_ = 4; \
            v_tmp_newptr_ = Mallocl(sizeof((X).ptr[0]) * v_tmp_newcapacity_, "VectorReserve:" #X); \
            if ( (X).ptr != NULL ) { \
                memcpy(v_tmp_newptr_, (X).ptr, (X).size * sizeof((X).ptr[0])); \
                M_free((X).ptr, "VectorReserve:" #X); \
            } \
            (X).ptr = v_tmp_newptr_; \
            (X).capacity = v_tmp_newcapacity_; \
        } \
    } while (0)
```

Requires that the capacity of the vector X is equal to or lager than newcapacity.

Parameters

X	the vector object.
newcapacity	the capacity to be reserved.

Definition at line 249 of file vector.h.

4.159.2.14 VectorPushBack

```
#define VectorPushBack(
                X,
                x )
```

Value:

```
    do { \
        VectorReserve((X), (X).size + 1); \
        (X).ptr[(X).size++] = (x); \
    } while (0)
```

Adds an element x at the end of the vector X.

Parameters

X	the vector object.
x	the element to be added.

Definition at line 277 of file vector.h.

4.159.2.15 VectorPushBacks

```
#define VectorPushBacks(
                X,
                src,
                n )
```

Value:

```
    do { \
        size_t v_tmp_n_ = (n); \
        VectorReserve((X), (X).size + v_tmp_n_); \
        memcpy((X).ptr + (X).size, (src), v_tmp_n_ * sizeof((X).ptr[0])); \
        (X).size += v_tmp_n_; \
    } while (0)
```

Adds an n elements of src at the end of the vector X.

Parameters

X	the vector object.
src	the starting address of the buffer storing elements to be added.
n	the number of elements to be added.

Definition at line 295 of file vector.h.

4.159.2.16 VectorPopBack

```
#define VectorPopBack(
              X )   do { (X).size --; } while (0)
```

Removes the last element of the vector X. VectorSize(X) must be > 0.

Parameters

X	the vector object.

Definition at line 314 of file vector.h.

4.159.2.17 VectorInsert

```
#define VectorInsert(
              X,
              index,
              x )
```

Value:

```
    do { \
        size_t v_tmp_index_ = (index); \
        VectorReserve((X), (X).size + 1); \
        memmove((X).ptr + v_tmp_index_ + 1, (X).ptr + v_tmp_index_, ((X).size - v_tmp_index_) *
      sizeof((X).ptr[0])); \
        (X).ptr[v_tmp_index_] = (x); \
        (X).size++; \
    } while (0)
```

Inserts an element x at the specified index of the vector X. The index must be 0 <= index < VectorSize(X).

Parameters

X	the vector object.
index	the position at which the element will be inserted.
x	the element to be inserted.

Definition at line 330 of file vector.h.

4.159.2.18 VectorInserts

```
#define VectorInserts(
              X,
```

```
                index,
                src,
                n )
```

Value:

```
do { \
    size_t v_tmp_index_ = (index), v_tmp_n_ = (n); \
    VectorReserve((X), (X).size + v_tmp_n_); \
    memmove((X).ptr + v_tmp_index_ + v_tmp_n_, (X).ptr + v_tmp_index_, ((X).size - v_tmp_index_) *
  sizeof((X).ptr[0])); \
    memcpy((X).ptr + v_tmp_index_, (src), v_tmp_n_ * sizeof((X).ptr[0])); \
    (X).size += v_tmp_n_; \
} while (0)
```

Inserts an n elements of src at the specified index of the vector X. The index must be 0 <= index < VectorSize(X).

Parameters

X	the vector object.
index	the position at which the elements will be inserted.
src	the starting address of the buffer storing elements to be inserted.
n	the number of elements to be inserted.

Definition at line 353 of file vector.h.

4.159.2.19 VectorErase

```
#define VectorErase(
                X,
                index )
```

Value:

```
do { \
    size_t v_tmp_index_ = (index); \
    memmove((X).ptr + v_tmp_index_, (X).ptr + v_tmp_index_ + 1, ((X).size - v_tmp_index_ - 1) *
  sizeof((X).ptr[0])); \
    (X).size--; \
} while (0)
```

Removes an element at the specified index of the vector X. The index must be 0 <= index < VectorSize(X).

Parameters

X	the vector object.
index	the position of the element to be removed.

Definition at line 374 of file vector.h.

4.159.2.20 VectorErases

```
#define VectorErases(
                X,
                index,
                n )
```

Value:

```
do { \
    size_t v_tmp_index_ = (index), v_tmp_n_ = (n); \
    memmove((X).ptr + v_tmp_index_, (X).ptr + v_tmp_index_ + v_tmp_n_, ((X).size - v_tmp_index_ - 1) *
  sizeof((X).ptr[0])); \
    (X).size -= v_tmp_n_; \
} while (0)
```

Removes an n elements at the specified index of the vector X. The index must be 0 <= index < VectorSize(X) - n + 1.

Parameters

X	the vector object.
index	the starting position of the elements to be removed.
n	the number of elements to be removed.

Definition at line 394 of file vector.h.

4.160 vector.h

Go to the documentation of this file.

```
00001 #ifndef VECTOR_H_
00002 #define VECTOR_H_
00003
00020 /* #[ License : */
00021 /*
00022  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00023  *   When using this file you are requested to refer to the publication
00024  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00025  *   This is considered a matter of courtesy as the development was paid
00026  *   for by FOM the Dutch physics granting agency and we would like to
00027  *   be able to track its scientific use to convince FOM of its value
00028  *   for the community.
00029  *
00030  *   This file is part of FORM.
00031  *
00032  *   FORM is free software: you can redistribute it and/or modify it under the
00033  *   terms of the GNU General Public License as published by the Free Software
00034  *   Foundation, either version 3 of the License, or (at your option) any later
00035  *   version.
00036  *
00037  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00038  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00039  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00040  *   details.
00041  *
00042  *   You should have received a copy of the GNU General Public License along
00043  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00044  */
00045 /* #] License : */
00046 /*
00047     #[ Includes :
00048 */
00049
00050 #include <stddef.h>
00051 #include <string.h>
00052 #include "declare.h"
00053
00054 /*
00055     #] Includes :
00056     #[ Vector :
00057         #[ VectorStruct :
00058 */
00059
00065 #define VectorStruct(T) \
00066     struct { \
00067         T *ptr; \
00068         size_t size; \
00069         size_t capacity; \
00070     }
00071
00072 /*
00073         #] VectorStruct :
```

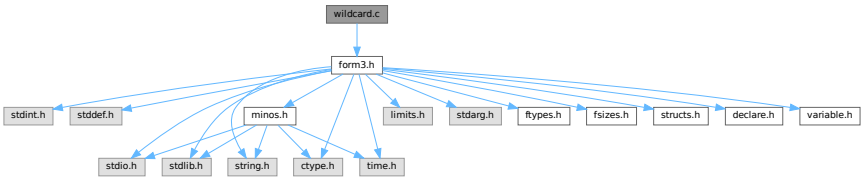
```
00074         #[ Vector :
00075 */
00076
00084 #define Vector(T, X) \
00085     VectorStruct(T) X = { NULL, 0, 0 }
00086
00087 /*
00088         #] Vector :
00089         #[ DeclareVector :
00090 */
00091
00099 #define DeclareVector(T, X) \
00100     VectorStruct(T) X
00101
00102 /*
00103         #] DeclareVector :
00104         #[ VectorInit :
00105 */
00106
00113 #define VectorInit(X) \
00114     do { \
00115         (X).ptr = NULL; \
00116         (X).size = 0; \
00117         (X).capacity = 0; \
00118     } while (0)
00119
00120 /*
00121         #] VectorInit :
00122         #[ VectorFree :
00123 */
00124
00130 #define VectorFree(X) \
00131     do { \
00132         M_free((X).ptr, "VectorFree:" #X); \
00133         (X).ptr = NULL; \
00134         (X).size = 0; \
00135         (X).capacity = 0; \
00136     } while (0)
00137
00138 /*
00139         #] VectorFree :
00140         #[ VectorPtr :
00141 */
00142
00150 #define VectorPtr(X) \
00151     ((X).ptr)
00152
00153 /*
00154         #] VectorPtr :
00155         #[ VectorFront :
00156 */
00157
00165 #define VectorFront(X) \
00166     ((X).ptr[0])
00167
00168 /*
00169         #] VectorFront :
00170         #[ VectorBack :
00171 */
00172
00180 #define VectorBack(X) \
00181     ((X).ptr[(X).size - 1])
00182
00183 /*
00184         #] VectorBack :
00185         #[ VectorSize :
00186 */
00187
00194 #define VectorSize(X) \
00195     ((X).size)
00196
00197 /*
00198         #] VectorSize :
00199         #[ VectorCapacity :
00200 */
00201
00208 #define VectorCapacity(X) \
00209     ((X).capacity)
00210
00211 /*
00212         #] VectorCapacity :
00213         #[ VectorEmpty :
00214 */
00215
00222 #define VectorEmpty(X) \
00223     ((X).size == 0)
00224
```

```
00225 /*
00226         #] VectorEmpty :
00227         #[ VectorClear :
00228 */
00229
00235 #define VectorClear(X) \
00236     do { (X).size = 0; } while (0)
00237
00238 /*
00239         #] VectorClear :
00240         #[ VectorReserve :
00241 */
00242
00249 #define VectorReserve(X, newcapacity) \
00250     do { \
00251         size_t v_tmp_newcapacity_ = (newcapacity); \
00252         if ( (X).capacity < v_tmp_newcapacity_ ) { \
00253             void *v_tmp_newptr_; \
00254             v_tmp_newcapacity_ = (v_tmp_newcapacity_ * 3) / 2; \
00255             if ( v_tmp_newcapacity_ < 4 ) v_tmp_newcapacity_ = 4; \
00256             v_tmp_newptr_ = Malloc1(sizeof((X).ptr[0]) * v_tmp_newcapacity_, "VectorReserve:" #X); \
00257             if ( (X).ptr != NULL ) { \
00258                 memcpy(v_tmp_newptr_, (X).ptr, (X).size * sizeof((X).ptr[0])); \
00259                 M_free((X).ptr, "VectorReserve:" #X); \
00260             } \
00261             (X).ptr = v_tmp_newptr_; \
00262             (X).capacity = v_tmp_newcapacity_; \
00263         } \
00264     } while (0)
00265
00266 /*
00267         #] VectorReserve :
00268         #[ VectorPushBack :
00269 */
00270
00277 #define VectorPushBack(X, x) \
00278     do { \
00279         VectorReserve((X), (X).size + 1); \
00280         (X).ptr[(X).size++] = (x); \
00281     } while (0)
00282
00283 /*
00284         #] VectorPushBack :
00285         #[ VectorPushBacks :
00286 */
00287
00295 #define VectorPushBacks(X, src, n) \
00296     do { \
00297         size_t v_tmp_n_ = (n); \
00298         VectorReserve((X), (X).size + v_tmp_n_); \
00299         memcpy((X).ptr + (X).size, (src), v_tmp_n_ * sizeof((X).ptr[0])); \
00300         (X).size += v_tmp_n_; \
00301     } while (0)
00302
00303 /*
00304         #] VectorPushBacks :
00305         #[ VectorPopBack :
00306 */
00307
00314 #define VectorPopBack(X) \
00315     do { (X).size --; } while (0)
00316
00317 /*
00318         #] VectorPopBack :
00319         #[ VectorInsert :
00320 */
00321
00330 #define VectorInsert(X, index, x) \
00331     do { \
00332         size_t v_tmp_index_ = (index); \
00333         VectorReserve((X), (X).size + 1); \
00334         memmove((X).ptr + v_tmp_index_ + 1, (X).ptr + v_tmp_index_, ((X).size - v_tmp_index_) *
      sizeof((X).ptr[0])); \
00335         (X).ptr[v_tmp_index_] = (x); \
00336         (X).size++; \
00337     } while (0)
00338
00339 /*
00340         #] VectorInsert :
00341         #[ VectorInserts :
00342 */
00343
00353 #define VectorInserts(X, index, src, n) \
00354     do { \
00355         size_t v_tmp_index_ = (index), v_tmp_n_ = (n); \
00356         VectorReserve((X), (X).size + v_tmp_n_); \
00357         memmove((X).ptr + v_tmp_index_ + v_tmp_n_, (X).ptr + v_tmp_index_, ((X).size - v_tmp_index_) *
```

```
      sizeof((X).ptr[0])); \
00358         memcpy((X).ptr + v_tmp_index_, (src), v_tmp_n_ * sizeof((X).ptr[0])); \
00359         (X).size += v_tmp_n_; \
00360     } while (0)
00361
00362 /*
00363         #] VectorInserts :
00364         #[ VectorErase :
00365 */
00366
00374 #define VectorErase(X, index) \
00375     do { \
00376         size_t v_tmp_index_ = (index); \
00377         memmove((X).ptr + v_tmp_index_, (X).ptr + v_tmp_index_ + 1, ((X).size - v_tmp_index_ - 1) *
      sizeof((X).ptr[0])); \
00378         (X).size--; \
00379     } while (0)
00380
00381 /*
00382         #] VectorErase :
00383         #[ VectorErases :
00384 */
00385
00394 #define VectorErases(X, index, n) \
00395     do { \
00396         size_t v_tmp_index_ = (index), v_tmp_n_ = (n); \
00397         memmove((X).ptr + v_tmp_index_, (X).ptr + v_tmp_index_ + v_tmp_n_, ((X).size - v_tmp_index_ -
      1) * sizeof((X).ptr[0])); \
00398         (X).size -= v_tmp_n_; \
00399     } while (0)
00400
00401 /*
00402         #] VectorErases :
00403     #] Vector :
00404 */
00405 #endif  /* VECTOR_H_ */
```

4.161 wildcard.c File Reference

`#include "form3.h"`
Include dependency graph for wildcard.c:



Macros

- #define DEBUG(x)

Functions

- WORD WildFill (PHEAD WORD *to, WORD *from, WORD *sub)
- int ResolveSet (PHEAD WORD *from, WORD *to, WORD *subs)
- void ClearWild (PHEAD0)
- int AddWild (PHEAD WORD oldnumber, WORD type, WORD newnumber)
- int CheckWild (PHEAD WORD oldnumber, WORD type, WORD newnumber, WORD *newval)
- int DenToFunction (WORD *term, WORD numfun)

4.161.1 Detailed Description

Contains the functions that deal with the wildcards. During the pattern matching there are two steps: 1: check that a wildcard substitution is correct (if there was already an assignment for this variable, it is the same; it is part of the proper set; it is the proper type of variables, etc.) 2: make the assignment In addition we have to be able to clear assignments. During execution we have to make the actual replacements (WildFill)

Definition in file wildcard.c.

4.161.2 Macro Definition Documentation

4.161.2.1 DEBUG

```
#define DEBUG(
              x )
```

Definition at line 44 of file wildcard.c.

4.161.3 Function Documentation

4.161.3.1 WildFill()

```
WORD WildFill (
             PHEAD WORD * to,
             WORD * from,
             WORD * sub )
```

Definition at line 65 of file wildcard.c.

4.161.3.2 ResolveSet()

```
int ResolveSet (
             PHEAD WORD * from,
             WORD * to,
             WORD * subs )
```

Definition at line 1361 of file wildcard.c.

4.161.3.3 ClearWild()

```
void ClearWild (
             PHEAD0  )
```

Definition at line 1518 of file wildcard.c.

4.161.3.4 AddWild()

```
int AddWild (
              PHEAD WORD oldnumber,
              WORD type,
              WORD newnumber )
```

Definition at line 1544 of file wildcard.c.

4.161.3.5 CheckWild()

```
int CheckWild (
              PHEAD WORD oldnumber,
              WORD type,
              WORD newnumber,
              WORD * newval )
```

Definition at line 1797 of file wildcard.c.

4.161.3.6 DenToFunction()

```
int DenToFunction (
              WORD * term,
              WORD numfun )
```

Definition at line 2565 of file wildcard.c.

4.162 wildcard.c

Go to the documentation of this file.

```
00001
00012 /* #[ License : */
00013 /*
00014  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00015  *   When using this file you are requested to refer to the publication
00016  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00017  *   This is considered a matter of courtesy as the development was paid
00018  *   for by FOM the Dutch physics granting agency and we would like to
00019  *   be able to track its scientific use to convince FOM of its value
00020  *   for the community.
00021  *
00022  *   This file is part of FORM.
00023  *
00024  *   FORM is free software: you can redistribute it and/or modify it under the
00025  *   terms of the GNU General Public License as published by the Free Software
00026  *   Foundation, either version 3 of the License, or (at your option) any later
00027  *   version.
00028  *
00029  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00030  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00031  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00032  *   details.
00033  *
00034  *   You should have received a copy of the GNU General Public License along
00035  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00036  */
00037 /* #] License : */
00038 /*
00039     #[ Includes : wildcard.c
00040 */
00041
00042 #include "form3.h"
00043
```

```
00044 #define DEBUG(x)
00045
00046 /*
00047 #define DEBUG(x) x
00048
00049     #] Includes :
00050     #[ Wildcards :
00051         #[ WildFill :           WORD WildFill(to,from,sub)
00052
00053         Takes the term in from and puts it into to while
00054         making wildcard substitutions.
00055         The return value is the number of words put in to.
00056         The length as the first word of from is not copied.
00057
00058         There are two possible algorithms:
00059         1:  For each element in `from': scan sub.
00060         2:  For each wildcard in sub replace elements in term.
00061         The original algorithm used 1:
00062
00063 */
00064
00065 WORD WildFill(PHEAD WORD *to, WORD *from, WORD *sub)
00066 {
00067     GETBIDENTITY
00068     WORD i, j, *s, *t, *m, len, dflag, odirt, adirt;
00069     WORD *r, *u, *v, *w, *z, *zst, *zz, *subs, *accu, na, dirty = 0, *tstop;
00070     WORD *temp = 0, *uu, *oldcpointer, sgn;
00071     WORD subcount, setflag, *setlist = 0, si;
00072     accu = oldcpointer = AR.CompressPointer;
00073     t = sub;
00074     t += sub[1];
00075     s = sub + SUBEXPSIZE;
00076     i = 0;
00077     while ( s < t && *s != FROMBRAC ) {
00078         i++; s += s[1];
00079     }
00080     if ( !i ) {         /* No wildcards -> done quickly */
00081         j = i = *from;
00082         NCOPY(to,from,i);
00083         if ( dirty ) AN.WildDirt = dirty;
00084         return(j);
00085     }
00086     sgn = 0;
00087     subs = sub + SUBEXPSIZE;
00088     t = from;
00089     GETSTOP(t,r);
00090     t++;
00091     m = to + 1;
00092     if ( t < r ) do {
00093         uu = u = t + t[1];
00094         setflag = 0;
00095 ReSwitch:
00096         switch ( *t ) {
00097             case SYMBOL:
00098 /*
00099             #[ SYMBOLS :
00100 */
00101                 z = accu;
00102                 *m++ = *t++;
00103                 *m++ = *t++;
00104                 v = m;
00105                 while ( t < u ) {
00106                     *m = *t;
00107                     for ( si = 0; si < setflag; si += 2 ) {
00108                         if ( t == temp + setlist[si] ) goto sspow;
00109                     }
00110                     s = subs;
00111                     for ( j = 0; j < i; j++ ) {
00112                         if ( *t == s[2] ) {
00113                             if ( *s == SYMTOSYM ) {
00114                                 *m = s[3]; dirty = 1;
00115                                 break;
00116                             }
00117                             else if ( *s == SYMTONUM ) {
00118                                 dirty = 1;
00119                                 zst = z;
00120                                 *z++ = SNUMBER;
00121                                 *z++ = 4;
00122                                 *z++ = s[3];
00123                                 w = z;
00124                                 *z++ = *++t;
00125                                 if ( ABS(*t) >= 2*MAXPOWER) {
00126 DoPow:                           s = subs;
00127                                 for ( j = 0; j < i; j++ ) {
00128                                     if ( ( *s == SYMTONUM ) &&
00129                                     ( ABS(*t) - 2*MAXPOWER ) == s[2] ) {
00130                                         dirty = 1;
```

```
00131                                       *w = s[3];
00132                                       if ( *t < 0 ) *w = -*w;
00133                                       break;
00134                                   }
00135                                   if ( ( *s == SYMTOSYM ) &&
00136                                   ( ABS(*t) - 2*MAXPOWER ) == s[2] ) {
00137                                       dirty = 1;
00138                                       zz = z;
00139                                       while ( --zz >= zst ) {
00140                                           zz[1+FUNHEAD+ARGHEAD] = *zz;
00141                                       }
00142                                       w += 1+FUNHEAD+ARGHEAD;
00143                                       *zst = EXPONENT;
00144                                       zst[2] = DIRTYFLAG;
00145                                       zst[FUNHEAD+ARGHEAD] = WORDDIF(z,zst)+4;
00146                                       zst[1+FUNHEAD] = 1;
00147                                       zst[FUNHEAD] = WORDDIF(z,zst)+4+ARGHEAD;
00148                                       z += FUNHEAD+ARGHEAD+1;
00149                                       *w = 1; /* exponent -> 1 */
00150                                       *z++ = 1;
00151                                       *z++ = 1;
00152                                       *z++ = 3;
00153                                       if ( *t > 0 ) {
00154                                           *z++ = -SYMBOL;
00155                                           *z++ = s[3];
00156                                       }
00157                                       else {
00158                                           *z++ = ARGHEAD+8;
00159                                           *z++ = 1;
00160                                           *z++ = 8;
00161                                           *z++ = SYMBOL;
00162                                           *z++ = 4;
00163                                           *z++ = s[3];
00164                                           *z++ = 1;
00165                                           *z++ = 1;
00166                                           *z++ = 1;
00167                                           *z++ = -3;
00168                                       }
00169                                       zst[1] = WORDDIF(z,zst);
00170                                       break;
00171                                   }
00172                                   if ( *s == SYMTOSUB &&
00173                                   ( ABS(*t) - 2*MAXPOWER ) == s[2] ) {
00174 MakeExp:                            dirty = 1;
00175                                       zz = z;
00176                                       while ( --zz >= zst ) {
00177                                           zz[1+FUNHEAD+ARGHEAD] = *zz;
00178                                       }
00179                                       w += 1+FUNHEAD+ARGHEAD;
00180                                       *zst = EXPONENT;
00181                                       zst[2] = DIRTYFLAG;
00182                                       zst[FUNHEAD+ARGHEAD] = WORDDIF(z,zst)+4;
00183                                       zst[1+FUNHEAD] = 1;
00184                                       zst[FUNHEAD] = WORDDIF(z,zst)+4+ARGHEAD;
00185                                       z += FUNHEAD+ARGHEAD+1;
00186                                       *w = 1; /* exponent -> 1 */
00187                                       *z++ = 1;
00188                                       *z++ = 1;
00189                                       *z++ = 3;
00190                                       *z++ = 4+SUBEXPSIZE+ARGHEAD;
00191                                       *z++ = 1;
00192                                       *z++ = 4+SUBEXPSIZE;
00193                                       *z++ = SUBEXPRESSION;
00194                                       *z++ = SUBEXPSIZE;
00195                                       *z++ = s[3];
00196                                       *z++ = 1;
00197                                       *z++ = AT.ebufnum;
00198                                       FILLSUB(z)
00199                                       *z++ = 1;
00200                                       *z++ = 1;
00201                                       *z++ = *t > 0 ? 3: -3;
00202                                       zst[1] = WORDDIF(z,zst);
00203                                       break;
00204                                   }
00205                                   s += s[1];
00206                               }
00207                               }
00208                               if ( !*w ) z = w - 3;
00209                               t++;
00210                               goto Seven;
00211                           }
00212                           else if ( *s == SYMTOSUB ) {
00213                               dirty = 1;
00214                               zst = z;
00215                               *z++ = SUBEXPRESSION;
00216                               *z++ = SUBEXPSIZE;
00217                               *z++ = s[3];
```

```
00218                                   w = z;
00219                                   *z++ = *++t;
00220                                   *z++ = AT.ebufnum;
00221                                   FILLSUB(z)
00222                                   goto DoPow;
00223                               }
00224                           }
00225                           s += s[1];
00226                       }
00227 sspow:
00228                       s = subs;
00229                       *++m = *++t;
00230                       for ( si = 0; si < setflag; si += 2 ) {
00231                           if ( t == temp + setlist[si] ) {
00232                               t++; m++;
00233                               goto Seven;
00234                           }
00235                       }
00236                       for ( j = 0; j < i; j++ ) {
00237                           if ( ( ABS(*t) - 2*MAXPOWER ) == s[2] ) {
00238                               if ( *s == SYMTONUM ) {
00239                                   dirty = 1;
00240                                   *m = s[3];
00241                                   if ( *t < 0 ) *m = -*m;
00242                                   break;
00243                               }
00244                               else if ( *s == SYMTOSYM ) {
00245                                   dirty = 1;
00246                                   *z++ = EXPONENT;
00247                                   if ( *t < 0 ) *z++ = FUNHEAD+ARGHEAD+10;
00248                                   else *z++ = 4+FUNHEAD;
00249                                   *z++ = 0;
00250                                   FILLFUN3(z)
00251                                   *z++ = -SYMBOL;
00252                                   *z++ = m[-1];
00253                                   if ( *t < 0 ) {
00254                                       *z++ = ARGHEAD+8;
00255                                       *z++ = 0;
00256                                       *z++ = 8;
00257                                       *z++ = SYMBOL;
00258                                       *z++ = 4;
00259                                       *z++ = s[3];
00260                                       *z++ = 1;
00261                                       *z++ = 1;
00262                                       *z++ = 1;
00263                                       *z = -3;
00264                                   }
00265                                   else {
00266                                       *z++ = -SYMBOL;
00267                                       *z++ = s[3];
00268                                   }
00269                                   m -= 2;
00270                                   break;
00271                               }
00272                               else if ( *s == SYMTOSUB ) {
00273                                   zst = z;
00274                                   *z++ = SYMBOL;
00275                                   *z++ = 4;
00276                                   *z++ = *--m;
00277                                   w = z;
00278                                   *z++ = *t;
00279                                   goto MakeExp;
00280                               }
00281                           }
00282                           s += s[1];
00283                       }
00284                       t++;
00285                       if ( *m ) m++;
00286                       else m--;
00287 Seven:;
00288                   }
00289                   j = WORDDIF(m,v);
00290                   if ( !j ) m -= 2;
00291                   else v[-1] = j + 2;
00292                   s = accu;
00293                   while ( s < z ) *m++ = *s++;
00294                   break;
00295 /*
00296               #] SYMBOLS :
00297 */
00298               case DOTPRODUCT:
00299 /*
00300               #[ DOTPRODUCTS :
00301 */
00302                   *m++ = *t++;
00303                   *m++ = *t++;
00304                   v = m;
```

```
00305                 z = accu;
00306                 while ( t < u ) {
00307                     *m = *t;
00308                     subcount = 0;
00309                     /* Process the first vector of the DOTPRODUCT */
00310                     for ( si = 0; si < setflag; si += 2 ) {
00311                         if ( t == temp + setlist[si] ) goto ss2;
00312                     }
00313                     s = subs;
00314                     for ( j = 0; j < i; j++ ) {
00315                         if ( *t == s[2] ) {
00316                             if ( *s == VECTOVEC ) {
00317                                 *m = s[3]; dirty = 1; break;
00318                             }
00319                             if ( *s == VECTOMIN ) {
00320                                 *m = s[3];
00321                                 dirty = 1;
00322                                 if ( ( ABS(t[2]) - 2*MAXPOWER ) < 0 ) {
00323                                     /* The power is a number */
00324                                     sgn += t[2];
00325                                 }
00326                                 else {
00327                                     /* The power is a wildcard. Put a -, resolve later. */
00328                                     sgn++;
00329                                 }
00330                                 break;
00331                             }
00332                             if ( *s == VECTOSUB ) {
00333                                 *m = s[3]; dirty = 1; subcount = 1; break;
00334                             }
00335                         }
00336                         s += s[1];
00337                     }
00338 ss2:
00339                     *++m = *++t;
00340                     s = subs;
00341                     /* Process the second vector of the DOTPRODUCT */
00342                     for ( si = 0; si < setflag; si += 2 ) {
00343                         if ( t == temp + setlist[si] ) goto ss3;
00344                     }
00345                     for ( j = 0; j < i; j++ ) {
00346                         if ( *t == s[2] ) {
00347                             if ( *s == VECTOVEC ) {
00348                                 *m = s[3]; dirty = 1; break;
00349                             }
00350                             if ( *s == VECTOMIN ) {
00351                                 *m = s[3];
00352                                 dirty = 1;
00353                                 if ( ( ABS(t[1]) - 2*MAXPOWER ) < 0 ) {
00354                                     /* The power is a number */
00355                                     sgn += t[1];
00356                                 }
00357                                 else {
00358                                     /* The power is a wildcard. Put a -, resolve later. */
00359                                     sgn++;
00360                                 }
00361                                 break;
00362                             }
00363                             if ( *s == VECTOSUB ) {
00364                                 *m = s[3]; dirty = 1; subcount += 2; break;
00365                             }
00366                         }
00367                         s += s[1];
00368                     }
00369 ss3:                *++m = *++t;
00370                     /* Process the power */
00371                     if ( ( ABS(*t) - 2*MAXPOWER ) < 0 ) goto RegPow;
00372                     s = subs;
00373                     for ( j = 0; j < i; j++ ) {
00374                         if ( ( ABS(*t) - 2*MAXPOWER ) == s[2] ) {
00375                             if ( *s == SYMTONUM ) {
00376                                 *m = s[3];
00377                                 /* Since the power is a wildcard, sgn is 0,1,2 depending whether
00378                                    there were 0,1,2 VECTOMIN in the DOTPRODUCT. Multiply by the
00379                                    power, which is positive currently. */
00380                                 sgn *= *m;
00381                                 /* Now flip the sign of the power, if the wildcard came with a - */
00382                                 if ( *t < 0 ) *m = -*m;
00383                                 dirty = 1;
00384                                 break;
00385                             }
00386                             if ( *s <= SYMTOSUB ) {
00387 /*
00388                 Here we put together a power function with the proper
00389                 arguments. Note that a p?.q? resolves to a single power.
00390 */
00391                                 m -= 2;
```

```
00392                           *z++ = EXPONENT;
00393                           w = z;
00394                           if ( subcount == 0 ) {
00395                               *z++ = 17+FUNHEAD+2*ARGHEAD;
00396                               *z++ = DIRTYFLAG;
00397                               FILLFUN3(z)
00398                               *z++ = 9+ARGHEAD;
00399                               *z++ = 0;
00400                               FILLARG(z)
00401                               *z++ = 9;
00402                               *z++ = DOTPRODUCT;
00403                               *z++ = 5;
00404                               *z++ = *m;
00405                               *z++ = m[1];
00406                               *z++ = 1;
00407                               *z++ = 1;
00408                               *z++ = 1;
00409                               *z++ = 3;
00410                               if ( *s == SYMTOSYM ) {
00411                                   *z++ = 8+ARGHEAD;
00412                                   *z++ = 0;
00413                                   FILLARG(z)
00414                                   *z++ = 8;
00415                                   *z++ = SYMBOL;
00416                                   *z++ = 4;
00417                                   *z++ = s[3];
00418                                   *z++ = 1;
00419                               }
00420                               else {
00421                                   *z++ = 4+SUBEXPSIZE+ARGHEAD;
00422                                   *z++ = 1;
00423                                   FILLARG(z)
00424                                   *z++ = 4+SUBEXPSIZE;
00425                                   *z++ = SUBEXPRESSION;
00426                                   *z++ = SUBEXPSIZE;
00427                                   *z++ = s[3];
00428                                   *z++ = 1;
00429                                   *z++ = AT.ebufnum;
00430                                   FILLSUB(z)
00431                               }
00432                               *z++ = 1; *z++ = 1;
00433                               *z++ = ( s[2] > 0 ) ? 3: -3;
00434                           }
00435                           else if ( subcount == 3 ) {
00436                               *z++ = 20+2*SUBEXPSIZE+FUNHEAD+2*ARGHEAD;
00437                               *z++ = DIRTYFLAG;
00438                               FILLFUN3(z)
00439                               *z++ = 12+2*SUBEXPSIZE+ARGHEAD;
00440                               *z++ = 1;
00441                               *z++ = 12+2*SUBEXPSIZE;
00442                               *z++ = SUBEXPRESSION;
00443                               *z++ = 4+SUBEXPSIZE;
00444                               *z++ = *m + 1;
00445                               *z++ = 1;
00446                               *z++ = AT.ebufnum;
00447                               FILLSUB(z)
00448                               *z++ = INDTOIND;
00449                               *z++ = 4;
00450                               *z++ = FUNNYVEC;
00451                               *z++ = ++AR.CurDum;
00452
00453                               *z++ = SUBEXPRESSION;
00454                               *z++ = 4+SUBEXPSIZE;
00455                               *z++ = m[1] + 1;
00456                               *z++ = 1;
00457                               *z++ = AT.ebufnum;
00458                               FILLSUB(z)
00459                               *z++ = INDTOIND;
00460                               *z++ = 4;
00461                               *z++ = FUNNYVEC;
00462                               *z++ = AR.CurDum;
00463                               *z++ = 1; *z++ = 1; *z++ = 3;
00464                           }
00465                           else {
00466                               if ( subcount == 2 ) {
00467                                   j = *m; *m = m[1]; m[1] = j;
00468                               }
00469                               *z++ = 16+SUBEXPSIZE+FUNHEAD+2*ARGHEAD;
00470                               *z++ = DIRTYFLAG;
00471                               FILLFUN3(z)
00472                               *z++ = 8+SUBEXPSIZE+ARGHEAD;
00473                               *z++ = 1;
00474                               *z++ = 8+SUBEXPSIZE;
00475                               *z++ = SUBEXPRESSION;
00476                               *z++ = 4+SUBEXPSIZE;
00477                               *z++ = *m + 1;
00478                               *z++ = 1;
```

```
00479                                   *z++ = AT.ebufnum;
00480                                   FILLSUB(z)
00481                                   *z++ = INDTOIND;
00482                                   *z++ = 4;
00483                                   *z++ = FUNNYVEC;
00484                                   *z++ = m[1];
00485                                   *z++ = 1; *z++ = 1; *z++ = 3;
00486                               }
00487                               if ( *s == SYMTOSYM ) {
00488                                   if ( s[2] > 0 ) {
00489                                       *z++ = -SYMBOL;
00490                                       *z++ = s[3];
00491                                       t++;
00492                                       *w = z-w+1;
00493                                       goto NextDot;
00494                                   }
00495                                   *z++ = 8+ARGHEAD;
00496                                   *z++ = 0;
00497                                   *z++ = 8;
00498                                   *z++ = SYMBOL;
00499                                   *z++ = 4;
00500                                   *z++ = s[3];
00501                                   *z++ = 1;
00502                               }
00503                               else {
00504                                   *z++ = 4+SUBEXPSIZE+ARGHEAD;
00505                                   *z++ = 1;
00506                                   *z++ = 4+SUBEXPSIZE;
00507                                   *z++ = SUBEXPRESSION;
00508                                   *z++ = SUBEXPSIZE;
00509                                   *z++ = s[3];
00510                                   *z++ = 1;
00511                                   *z++ = AT.ebufnum;
00512                                   FILLSUB(z)
00513                               }
00514                               *z++ = 1; *z++ = 1;
00515                               *z++ = ( s[2] > 0 ) ? 3: -3;
00516                               t++;
00517                               *w = z-w+1;
00518                               goto NextDot;
00519                           }
00520                       }
00521                       s += s[1];
00522                   }
00523 RegPow:           if ( *m ) m++;
00524                   else { m -= 2; subcount = 0; }
00525                   t++;
00526                   if ( subcount ) {
00527                       m -= 3;
00528                       if ( subcount == 3 ) {
00529                           if ( m[2] < 0 ) {
00530                               j = (-m[2]) * (2*SUBEXPSIZE+8);
00531                               *z++ = DENOMINATOR;
00532                               *z++ = j + 8 + FUNHEAD + ARGHEAD;
00533                               *z++ = DIRTYFLAG;
00534                               FILLFUN3(z)
00535                               *z++ = j + 8 + ARGHEAD;
00536                               *z++ = 1;
00537                               *z++ = j + 8;
00538                               while ( m[2] < 0 ) {
00539                                   (m[2])++;
00540                                   *z++ = SUBEXPRESSION;
00541                                   *z++ = 4+SUBEXPSIZE;
00542                                   *z++ = *m + 1;
00543                                   *z++ = 1;
00544                                   *z++ = AT.ebufnum;
00545                                   FILLSUB(z)
00546                                   *z++ = INDTOIND;
00547                                   *z++ = 4;
00548                                   *z++ = FUNNYVEC;
00549                                   *z++ = ++AR.CurDum;
00550                                   *z++ = SUBEXPRESSION;
00551                                   *z++ = 8+SUBEXPSIZE;
00552                                   *z++ = m[1] + 1;
00553                                   *z++ = 1;
00554                                   *z++ = AT.ebufnum;
00555                                   FILLSUB(z)
00556                                   *z++ = INDTOIND;
00557                                   *z++ = 4;
00558                                   *z++ = FUNNYVEC;
00559                                   *z++ = AR.CurDum;
00560                                   *z++ = SYMTOSYM;    /* Needed to avoid */
00561                                   *z++ = 4;           /* problems with   */
00562                                   *z++ = 1000;        /* conversion to   */
00563                                   *z++ = 1000;        /* square of subexp*/
00564                               }
00565                               *z++ = 1; *z++ = 1; *z++ = 3;
```

```
00566                           }
00567                           else {
00568                               while ( m[2] > 0 ) {
00569                                   (m[2])--;
00570                                   *z++ = SUBEXPRESSION;
00571                                   *z++ = 4+SUBEXPSIZE;
00572                                   *z++ = *m + 1;
00573                                   *z++ = 1;
00574                                   *z++ = AT.ebufnum;
00575                                   FILLSUB(z)
00576                                   *z++ = INDTOIND;
00577                                   *z++ = 4;
00578                                   *z++ = FUNNYVEC;
00579                                   *z++ = ++AR.CurDum;
00580                                   *z++ = SUBEXPRESSION;
00581                                   *z++ = 4+SUBEXPSIZE;
00582                                   *z++ = m[1] + 1;
00583                                   *z++ = 1;
00584                                   *z++ = AT.ebufnum;
00585                                   FILLSUB(z)
00586                                   *z++ = INDTOIND;
00587                                   *z++ = 4;
00588                                   *z++ = FUNNYVEC;
00589                                   *z++ = AR.CurDum;
00590                               }
00591                           }
00592                       }
00593                       else {
00594                           if ( subcount == 2 ) {
00595                               j = *m; *m = m[1]; m[1] = j;
00596                           }
00597                           if ( m[2] < 0 ) {
00598                               *z++ = DENOMINATOR;
00599                               *z++ = 8+SUBEXPSIZE+FUNHEAD+ARGHEAD;
00600                               *z++ = DIRTYFLAG;
00601                               FILLFUN3(z)
00602                               *z++ = 8+SUBEXPSIZE+ARGHEAD;
00603                               *z++ = 1;
00604                               *z++ = 8+SUBEXPSIZE;
00605                           }
00606                           *z++ = SUBEXPRESSION;
00607                           *z++ = 4+SUBEXPSIZE;
00608                           *z++ = *m + 1;
00609                           *z++ = ABS(m[2]);
00610                           *z++ = AT.ebufnum;
00611                           FILLSUB(z)
00612                           *z++ = INDTOIND;
00613                           *z++ = 4;
00614                           *z++ = FUNNYVEC;
00615                           *z++ = m[1];
00616                           if ( m[2] < 0 ) {
00617                               *z++ = 1; *z++ = 1; *z++ = 3;
00618                           }
00619                       }
00620                   }
00621 NextDot:;
00622               }
00623               if ( m <= v ) m = v - 2;
00624               else v[-1] = WORDDIF(m,v) + 2;
00625               if ( z > accu ) {
00626                   j = WORDDIF(z,accu);
00627                   z = accu;
00628                   NCOPY(m,z,j);
00629               }
00630               break;
00631 /*
00632           #] DOTPRODUCTS :
00633 */
00634           case SETSET:
00635 /*
00636           #[ SETS :
00637 */
00638               temp = accu + (((AR.ComprTop - accu)>>1)&(-2));
00639               if ( ResolveSet(BHEAD t,temp,sub) ) {
00640                   Terminate(-1);
00641               }
00642               setlist = t + 2 + t[3];
00643               setflag = t[1] - 2 - t[3];  /* Number of elements * 2 */
00644               t = temp; u = t + t[1];
00645               goto ReSwitch;
00646 /*
00647           #] SETS :
00648 */
00649           case VECTOR:
00650 /*
00651           #[ VECTORS :
00652 */
```

```
00653                 *m++ = *t++;
00654                 *m++ = *t++;
00655                 v = m;
00656                 z = accu;
00657                 while ( t < u ) {
00658                     *m = *t;
00659                     for ( si = 0; si < setflag; si += 2 ) {
00660                         if ( t == temp + setlist[si] ) goto ss4;
00661                     }
00662                     s = subs;
00663                     for ( j = 0; j < i; j++ ) {
00664                         if ( *t == s[2] ) {
00665                             if ( *s == INDTOIND || *s == VECTOVEC ) {
00666                                 *m = s[3]; dirty = 1; break;
00667                             }
00668                             if ( *s == VECTOMIN ) {
00669                                 *m = s[3]; dirty = 1; sgn++; break;
00670                             }
00671                             else if ( *s == VECTOSUB ) {
00672                                 *z++ = SUBEXPRESSION;
00673                                 *z++ = 4+SUBEXPSIZE;
00674                                 *z++ = s[3]+1;
00675                                 *z++ = 1;
00676                                 *z++ = AT.ebufnum;
00677                                 FILLSUB(z)
00678                                 *z++ = VECTOVEC;
00679                                 *z++ = 4;
00680                                 *z++ = FUNNYVEC;
00681                                 *z++ = *++t;
00682                                 m--;
00683                                 s = subs;
00684                                 for ( j = 0; j < i; j++ ) {
00685                                     if ( z[-1] == s[2] ) {
00686                                         if ( *s == INDTOIND || *s == VECTOVEC ) {
00687                                             z[-1] = s[3];
00688                                             break;
00689                                         }
00690                                         if ( *s == INDTOSUB || *s == VECTOSUB ) {
00691                                             z[-1] = ++AR.CurDum;
00692                                             *z++ = SUBEXPRESSION;
00693                                             *z++ = 4+SUBEXPSIZE;
00694                                             *z++ = s[3]+1;
00695                                             *z++ = 1;
00696                                             *z++ = AT.ebufnum;
00697                                             FILLSUB(z)
00698                                             if ( *s == INDTOSUB ) *z++ = INDTOIND;
00699                                             else *z++ = VECTOSUB;
00700                                             *z++ = 4;
00701                                             *z++ = FUNNYVEC;
00702                                             *z++ = AR.CurDum;
00703                                             break;
00704                                         }
00705                                     }
00706                                     s += s[1];
00707                                 }
00708                                 dirty = 1;
00709                                 break;
00710                             }
00711                             else if ( *s == INDTOSUB ) {
00712                                 *z++ = SUBEXPRESSION;
00713                                 *z++ = 4+SUBEXPSIZE;
00714                                 *z++ = s[3]+1;
00715                                 *z++ = 1;
00716                                 *z++ = AT.ebufnum;
00717                                 FILLSUB(z)
00718                                 *z++ = INDTOIND;
00719                                 *z++ = 4;
00720                                 *z++ = FUNNYVEC;
00721                                 m -= 2;
00722                                 *z++ = m[1];
00723                                 dirty = 1;
00724                                 t++;
00725                                 break;
00726                             }
00727                         }
00728                         s += s[1];
00729                     }
00730 ss4:                 m++; t++;
00731                 }
00732                 if ( m <= v ) m = v-2;
00733                 else v[-1] = WORDDIF(m,v)+2;
00734                 if ( z > accu ) {
00735                     j = WORDDIF(z,accu); z = accu;
00736                     NCOPY(m,z,j);
00737                 }
00738                 break;
00739 /*
```

```
00740             #] VECTORS :
00741 */
00742             case INDEX:
00743 /*
00744             #[ INDEX :
00745 */
00746                 *m++ = *t++;
00747                 *m++ = *t++;
00748                 v = m;
00749                 z = accu;
00750                 while ( t < u ) {
00751                     *m = *t;
00752                     for ( si = 0; si < setflag; si += 2 ) {
00753                         if ( t == temp + setlist[si] ) goto ss5;
00754                     }
00755                     s = subs;
00756                     for ( j = 0; j < i; j++ ) {
00757                         if ( *t == s[2] ) {
00758                             if ( *s == INDTOIND || *s == VECTOVEC )
00759                                 { *m = s[3]; dirty = 1; break; }
00760                             if ( *s == VECTOMIN )
00761                                 { *m = s[3]; dirty = 1; sgn++; break; }
00762                             else if ( *s == VECTOSUB || *s == INDTOSUB ) {
00763                                 *z++ = SUBEXPRESSION;
00764                                 *z++ = SUBEXPSIZE;
00765                                 *z++ = s[3];
00766                                 *z++ = 1;
00767                                 *z++ = AT.ebufnum;
00768                                 FILLSUB(z)
00769                                 m--;
00770                                 dirty = 1;
00771                                 break;
00772                             }
00773                         }
00774                         s += s[1];
00775                     }
00776 ss5:                m++; t++;
00777                 }
00778                 if ( m <= v ) m = v-2;
00779                 else v[-1] = WORDDIF(m,v)+2;
00780                 if ( z > accu ) {
00781                     j = WORDDIF(z,accu); z = accu;
00782                     NCOPY(m,z,j);
00783                 }
00784                 break;
00785 /*
00786             #] INDEX :
00787 */
00788             case DELTA:
00789             case LEVICIVITA:
00790             case GAMMA:
00791 /*
00792             #[ SPECIALS :
00793 */
00794                 v = m;
00795                 *m++ = *t++;
00796                 *m++ = *t++;
00797 #if FUNHEAD > 2
00798                 if ( t[-2] != DELTA ) *m++ = *t++;
00799 #endif
00800 Tensors:
00801                 COPYFUN3(m,t)
00802                 z = accu;
00803                 while ( t < u ) {
00804                     *m = *t;
00805                     for ( si = 0; si < setflag; si += 2 ) {
00806                         if ( t == temp + setlist[si] ) goto ss6;
00807                     }
00808                     s = subs;
00809                     if ( *m == FUNNYWILD ) {
00810                         CBUF *C = cbuf+AT.ebufnum;
00811                         t++;
00812                         for ( j = 0; j < i; j++ ) {
00813                             if ( *s == ARGTOARG && *t == s[2] ) {
00814                                 v[2] |= DIRTYFLAG;
00815                                 if ( s[3] < 0 ) { /* empty */
00816                                     t++; break;
00817                                 }
00818                                 w = C->rhs[s[3]];
00819 DEBUG(MesPrint("Thread %w(a): s[3] = %d, w=(%d,%d,%d,%d)",s[3],w[0],w[1],w[2],w[3]);)
00820                                 j = *w++;
00821                                 if ( j > 0 ) {
00822                                     NCOPY(m,w,j);
00823                                 }
00824                                 else {
00825                                     while ( *w ) {
00826                                         if ( *w == -INDEX || *w == -VECTOR
```

```
00827                                       || *w == -MINVECTOR
00828                                       || ( *w == -SNUMBER && w[1] >= 0
00829                                       && w[1] < AM.OffsetIndex ) ) {
00830                                           if ( *w == -MINVECTOR ) sgn++;
00831                                           w++;
00832                                           *m++ = *w++;
00833                                       }
00834                                       else {
00835                                           MLOCK(ErrorMessageLock);
00836 DEBUG(MesPrint("Thread %w(aa): *w = %d",*w);)
00837                                           MesPrint("Illegal substitution of argument field in
      tensor");
00838                                           MUNLOCK(ErrorMessageLock);
00839                                           SETERROR(-1)
00840                                       }
00841                                   }
00842                               }
00843                               t++;
00844                               break;
00845                           }
00846                           s += s[1];
00847                       }
00848                   }
00849                   else {
00850                       for ( j = 0; j < i; j++ ) {
00851                           if ( *t == s[2] ) {
00852                               if ( *s == INDTOIND || *s == VECTOVEC )
00853                                   { *m = s[3]; dirty = 1; break; }
00854                               if ( *s == VECTOMIN )
00855                                   { *m = s[3]; dirty = 1; sgn++; break; }
00856                               else if ( *s == VECTOSUB || *s == INDTOSUB ) {
00857                                   *m = ++AR.CurDum;
00858                                   *z++ = SUBEXPRESSION;
00859                                   *z++ = 4+SUBEXPSIZE;
00860                                   *z++ = s[3]+1;
00861                                   *z++ = 1;
00862                                   *z++ = AT.ebufnum;
00863                                   FILLSUB(z)
00864                                   *z++ = INDTOIND;
00865                                   *z++ = 4;
00866                                   *z++ = FUNNYVEC;
00867                                   *z++ = AR.CurDum;
00868                                   dirty = 1;
00869                                   break;
00870                               }
00871                           }
00872                           s += s[1];
00873                       }
00874                       if ( j < i && *v != DELTA ) v[2] |= DIRTYFLAG;
00875 ss6:                   m++; t++;
00876                   }
00877               }
00878               v[1] = WORDDIF(m,v);
00879               if ( z > accu ) {
00880                   j = WORDDIF(z,accu); z = accu;
00881                   NCOPY(m,z,j);
00882               }
00883               break;
00884 /*
00885           #] SPECIALS :
00886 */
00887           case SUBEXPRESSION:
00888 /*
00889           #[ SUBEXPRESSION :
00890 */
00891               dirty = 1;
00892               tstop = t + t[1];
00893               *m++ = *t++;
00894               *m++ = *t++;
00895               *m++ = *t++;
00896               *m++ = *t++;
00897               if ( t[-1] >= 2*MAXPOWER || t[-1] <= -2*MAXPOWER ) {
00898                   s = subs;
00899                   for ( j = 0; j < i; j++ ) {
00900                       if ( *s == SYMTONUM &&
00901                               ( ABS(t[-1]) - 2*MAXPOWER ) == s[2] ) {
00902                           m[-1] = s[3];
00903                           if ( t[-1] < 0 ) m[-1] = -m[-1];
00904                           break;
00905                       }
00906                       s += s[1];
00907                   }
00908               }
00909               *m++ = *t++;
00910               COPYSUB(m,t)
00911               while ( t < tstop ) {
00912                   for ( si = 0; si < setflag; si += 2 ) {
```

```
00913                       if ( t == temp + setlist[si] - 2 ) goto ss7;
00914                   }
00915                   s = subs;
00916                   for ( j = 0; j < i; j++ ) {
00917                       if ( s[2] == t[2] ) {
00918                           if ( ( *s <= SYMTOSUB && *t <= SYMTOSUB )
00919                           || ( *s == *t && *s < FROMBRAC )
00920                           || ( *s == VECTOVEC && ( *t == VECTOSUB || *t == VECTOMIN ) )
00921                           || ( *s == VECTOSUB && ( *t == VECTOVEC || *t == VECTOMIN ) )
00922                           || ( *s == VECTOMIN && ( *t == VECTOSUB || *t == VECTOVEC ) )
00923                           || ( *s == INDTOIND && *t == INDTOSUB )
00924                           || ( *s == INDTOSUB && *t == INDTOIND ) ) {
00925                               WORD *vv = m;
00926 /*                            *t = *s;  Wrong!!! Overwrites compiler buffer  */
00927                               j = t[1];
00928                               NCOPY(m,t,j);
00929                               vv[0] = s[0];
00930                               vv[3] = s[3];
00931                               goto sr7;
00932                           }
00933                       }
00934                       s += s[1];
00935                   }
00936 ss7:               j = t[1];
00937                   NCOPY(m,t,j);
00938 sr7:;
00939               }
00940               break;
00941 /*
00942           #] SUBEXPRESSION :
00943 */
00944           case EXPRESSION:
00945 /*
00946           #[ EXPRESSION :
00947 */
00948               dirty = 1;
00949               tstop = t + t[1];
00950               v = m;
00951               *m++ = *t++;
00952               *m++ = *t++;
00953               *m++ = *t++;
00954               *m++ = *t++;
00955               s = subs;
00956               for ( j = 0; j < i; j++ ) {
00957                   if ( ( ABS(t[-1]) - 2*MAXPOWER ) == s[2] ) {
00958                       if ( *s == SYMTONUM ) {
00959                           m[-1] = s[3];
00960                           if ( t[-1] < 0 ) m[-1] = -m[-1];
00961                           break;
00962                       }
00963                       else if ( *s <= SYMTOSUB ) {
00964                           MLOCK(ErrorMessageLock);
00965                           MesPrint("Wildcard power of expression should be a number");
00966                           MUNLOCK(ErrorMessageLock);
00967                           SETERROR(-1)
00968                       }
00969                   }
00970                   s += s[1];
00971               }
00972               *m++ = *t++;
00973               COPYSUB(m,t)
00974               while ( t < tstop && *t != WILDCARDS ) {
00975                   j = t[1];
00976                   NCOPY(m,t,j);
00977               }
00978               if ( t < tstop && *t == WILDCARDS ) {
00979                   *m++ = *t;
00980                   s = sub;
00981                   j = s[1];
00982                   *m++ = j+2;
00983                   NCOPY(m,s,j);
00984                   t += t[1];
00985               }
00986               if ( t < tstop && *t == FROMBRAC ) {
00987                   w = m;
00988                   *m++ = *t;
00989                   *m++ = t[1];
00990                   if ( WildFill(BHEAD m,t+2,sub) < 0 ) {
00991                       MLOCK(ErrorMessageLock);
00992                       MesCall("WildFill");
00993                       MUNLOCK(ErrorMessageLock);
00994                       SETERROR(-1)
00995                   }
00996                   m += *m;
00997                   w[1] = m - w;
00998                   t += t[1];
00999               }
```

```
01000                 while ( t < tstop ) {
01001                     j = t[1];
01002                     NCOPY(m,t,j);
01003                 }
01004                 v[1] = m-v;
01005                 break;
01006 /*
01007             #] EXPRESSION :
01008 */
01009             default:
01010 /*
01011             #[ FUNCTIONS :
01012 */
01013                 if ( *t >= FUNCTION ) {
01014                     dflag = 0;
01015                     na = 0;
01016                     *m = *t;
01017                     for ( si = 0; si < setflag; si += 2 ) {
01018                         if ( t == temp + setlist[si] ) {
01019                             dflag = DIRTYFLAG;  goto ss8;
01020                         }
01021                     }
01022                     s = subs;
01023                     for ( j = 0; j < i; j++ ) {
01024                         if ( *s == FUNTOFUN && *t == s[2] )
01025                             { *m = s[3]; dirty = 1; dflag = DIRTYFLAG; break; }
01026                         s += s[1];
01027                     }
01028 ss8:                v = m;
01029                     if ( *t >= FUNCTION && functions[*t-FUNCTION].spec
01030                     >= TENSORFUNCTION ) {
01031                         if ( *m < FUNCTION || functions[*m-FUNCTION].spec
01032                         < TENSORFUNCTION ) {
01033                             MLOCK(ErrorMessageLock);
01034                             MesPrint("Illegal wildcarding of regular function to tensorfunction");
01035                             MUNLOCK(ErrorMessageLock);
01036                             SETERROR(-1)
01037                         }
01038                         m++; t++;
01039                         *m++ = *t++;
01040                         *m++ = *t++ | dflag;
01041                         goto Tensors;
01042                     }
01043                     m++; t++;
01044                     *m++ = *t++;
01045                     *m++ = *t++ | dflag;
01046                     COPYFUN3(m,t)
01047                     z = accu;
01048                     while ( t < u ) {       /* do an argument */
01049                         if ( *t < 0 ) {
01050 /*
01051             #[ Simple arguments :
01052 */
01053                             CBUF *C = cbuf+AT.ebufnum;
01054                             for ( si = 0; si < setflag; si += 2 ) {
01055                                 if ( *t <= -FUNCTION ) {
01056                                     if ( t == temp + setlist[si] ) {
01057                                         v[2] |= DIRTYFLAG; goto ss10; }
01058                                 }
01059                                 else {
01060                                     if ( t == temp + setlist[si]-1 ) {
01061                                         v[2] |= DIRTYFLAG; goto ss9; }
01062                                 }
01063                             }
01064                             if ( *t == -ARGWILD ) {
01065                                 s = subs;
01066                                 for ( j = 0; j < i; j++ ) {
01067                                     if ( *s == ARGTOARG && s[2] == t[1] ) break;
01068                                     s += s[1];
01069                                 }
01070                                 v[2] |= DIRTYFLAG;
01071                                 w = C->rhs[s[3]];
01072 DEBUG(MesPrint("Thread %w(b): s[3] = %d, w=(%d,%d,%d,%d)",s[3],w[0],w[1],w[2],w[3]);)
01073                                 if ( *w == 0 ) {
01074                                     w++;
01075                                     while ( *w ) {
01076                                         if ( *w > 0 ) j = *w;
01077                                         else if ( *w <= -FUNCTION ) j = 1;
01078                                         else j = 2;
01079                                         NCOPY(m,w,j);
01080                                     }
01081                                 }
01082                                 else {
01083                                     j = *w++;
01084                                     while ( --j >= 0 ) {
01085                                         if ( *w < MINSPEC ) *m++ = -VECTOR;
01086                                         else if ( *w >= 0 && *w < AM.OffsetIndex )
```

```
01087                                           *m++ = -SNUMBER;
01088                                       else *m++ = -INDEX;
01089                                       *m++ = *w++;
01090                                   }
01091                               }
01092                               t += 2;
01093                               dirty = 1;
01094                               if ( ( *v == NUMARGSFUN || *v == NUMTERMSFUN )
01095                               && t >= u && m == v + FUNHEAD ) {
01096                                   m = v;
01097                                   *m++ = SNUMBER; *m++ = 3; *m++ = 0;
01098                                   break;
01099                               }
01100                           }
01101                           else if ( *t <= -FUNCTION ) {
01102                               *m = *t;
01103                               s = subs;
01104                               for ( j = 0; j < i; j++ ) {
01105                                   if ( -*t == s[2] ) {
01106                                       if ( *s == FUNTOFUN )
01107                                           { *m = -s[3]; dirty = 1; v[2] |= DIRTYFLAG; break; }
01108                                   }
01109                                   s += s[1];
01110                               }
01111                               m++; t++;
01112                           }
01113                           else if ( *t == -SYMBOL ) {
01114                               *m++ = *t++;
01115                               *m = *t;
01116                               s = subs;
01117                               for ( j = 0; j < i; j++ ) {
01118                                   if ( *t == s[2] && *s <= SYMTOSUB ) {
01119                                       dirty = 1; v[2] |= DIRTYFLAG;
01120                                       if ( AR.PolyFunType == 2 && v[0] == AR.PolyFun )
01121                                           v[2] |= MUSTCLEANPRF;
01122                                       if ( *s == SYMTOSYM ) *m = s[3];
01123                                       else if ( *s == SYMTONUM ) {
01124                                           m[-1] = -SNUMBER;
01125                                           *m = s[3];
01126                                       }
01127                                       else if ( *s == SYMTOSUB ) {
01128 ToSub:                                      m--;
01129                                           w = C->rhs[s[3]];
01130 DEBUG(MesPrint("Thread %w(c): s[3] = %d, w=(%d,%d,%d,%d)",s[3],w[0],w[1],w[2],w[3]);)
01131                                           s = m;
01132                                           m += 2;
01133                                           while ( *w ) {
01134                                               j = *w;
01135                                               NCOPY(m,w,j);
01136                                           }
01137                                           *s = WORDDIF(m,s);
01138                                           s[1] = 0;
01139                                           *m = 0;
01140                                           if ( t[-1] == -MINVECTOR ) {
01141                                               w = s+2;
01142                                               while ( *w ) {
01143                                                   w += *w;
01144                                                   w[-1] = -w[-1];
01145                                               }
01146                                           }
01147                                           if ( ToFast(s,s) ) {
01148                                               if ( *s <= -FUNCTION ) m = s;
01149                                               else m = s + 1;
01150                                           }
01151                                           else m--;
01152                                       }
01153                                       break;
01154                                   }
01155                                   s += s[1];
01156                               }
01157                               m++; t++;
01158                           }
01159                           else if ( *t == -INDEX ) {
01160                               *m++ = *t++;
01161                               *m = *t;
01162                               s = subs;
01163                               for ( j = 0; j < i; j++ ) {
01164                                   if ( *t == s[2] ) {
01165                                       if ( *s == INDTOIND || *s == VECTOVEC ) {
01166                                           *m = s[3];
01167                                           if ( *m < MINSPEC ) m[-1] = -VECTOR;
01168                                           else if ( *m >= 0 && *m < AM.OffsetIndex )
01169                                               m[-1] = -SNUMBER;
01170                                           else m[-1] = -INDEX;
01171                                       }
01172                                       else if ( *s == VECTOSUB || *s == INDTOSUB ) {
01173                                           m[-1] = -INDEX;
```

```
01174                                           *m = ++AR.CurDum;
01175                                           *z++ = SUBEXPRESSION;
01176                                           *z++ = 4+SUBEXPSIZE;
01177                                           *z++ = s[3]+1;
01178                                           *z++ = 1;
01179                                           *z++ = AT.ebufnum;
01180                                           FILLSUB(z)
01181                                           *z++ = INDTOIND;
01182                                           *z++ = 4;
01183                                           *z++ = FUNNYVEC;
01184                                           *z++ = AR.CurDum;
01185                                       }
01186                                       v[2] |= DIRTYFLAG; dirty = 1;
01187                                       break;
01188                                   }
01189                                   s += s[1];
01190                               }
01191                               m++; t++;
01192                           }
01193                           else if ( *t == -VECTOR || *t == -MINVECTOR ) {
01194                               *m++ = *t++;
01195                               *m = *t;
01196                               s = subs;
01197                               for ( j = 0; j < i; j++ ) {
01198                                   if ( *t == s[2] ) {
01199                                       if ( *s == VECTOVEC ) *m = s[3];
01200                                       else if ( *s == VECTOMIN ) {
01201                                           *m = s[3];
01202                                           if ( t[-1] == -VECTOR )
01203                                               m[-1] = -MINVECTOR;
01204                                           else
01205                                               m[-1] = -VECTOR;
01206                                       }
01207                                       else if ( *s == VECTOSUB ) goto ToSub;
01208                                       dirty = 1; v[2] |= DIRTYFLAG;
01209                                       break;
01210                                   }
01211                                   s += s[1];
01212                               }
01213                               m++; t++;
01214                           }
01215                           else if ( *t == -SNUMBER ) {
01216                               *m++ = *t++;
01217                               *m = *t;
01218                               s = subs;
01219                               for ( j = 0; j < i; j++ ) {
01220                                   if ( *t == s[2] && *s >= NUMTONUM && *s <= NUMTOSUB ) {
01221                                       dirty = 1; v[2] |= DIRTYFLAG;
01222                                       if ( *s == NUMTONUM ) *m = s[3];
01223                                       else if ( *s == NUMTOSYM ) {
01224                                           m[-1] = -SYMBOL;
01225                                           *m = s[3];
01226                                       }
01227                                       else if ( *s == NUMTOIND ) {
01228                                           m[-1] = -INDEX;
01229                                           *m = s[3];
01230                                       }
01231                                       else if ( *s == NUMTOSUB ) goto ToSub;
01232                                       break;
01233                                   }
01234                                   s += s[1];
01235                               }
01236                               m++; t++;
01237                           }
01238                           else {
01239 ss9:                            *m++ = *t++;
01240 ss10:                           *m++ = *t++;
01241                           }
01242                           na = WORDDIF(z,accu);
01243 /*
01244             #] Simple arguments :
01245 */
01246                       }
01247                       else {
01248                           w = m;
01249                           zz = t;
01250                           NEXTARG(zz)
01251                           odirt = AN.WildDirt; AN.WildDirt = 0;
01252                           AR.CompressPointer = accu + na;
01253                           for ( j = 0; j < ARGHEAD; j++ ) *m++ = *t++;
01254                           j = 0;
01255                           adirt = 0;
01256                           while ( t < zz ) {  /* do a term */
01257                               if ( ( len = WildFill(BHEAD m,t,sub) ) < 0 ) {
01258                                   MLOCK(ErrorMessageLock);
01259                                   MesCall("WildFill");
01260                                   MUNLOCK(ErrorMessageLock);
```

```
01261                                   SETERROR(-1)
01262                               }
01263                               if ( AN.WildDirt ) {
01264                                   adirt = AN.WildDirt;
01265                                   AN.WildDirt = 0;
01266                               }
01267                               m += len;
01268                               t += *t;
01269                           }
01270                           *w = WORDDIF(m,w);  /* Fill parameter length */
01271                           if ( adirt ) {
01272                               dirty = w[1] = 1; v[2] |= DIRTYFLAG;
01273                               if ( AR.PolyFunType == 2 && v[0] == AR.PolyFun )
01274                                   v[2] |= MUSTCLEANPRF;
01275                               AN.WildDirt = adirt;
01276                           }
01277                           else {
01278                               AN.WildDirt = odirt;
01279                           }
01280                           if ( ToFast(w,w) ) {
01281                               if ( *w <= -FUNCTION ) {
01282                                   if ( *w == NUMARGSFUN || *w == NUMTERMSFUN ) {
01283                                       *w = -SNUMBER; w[1] = 0; m = w + 2;
01284                                   }
01285                                   else m = w+1;
01286                               }
01287                               else m = w+2;
01288                           }
01289                           AR.CompressPointer = oldcpointer;
01290                       }
01291                   }
01292                   v[1] = WORDDIF(m,v);        /* Fill function length */
01293                   s = accu;
01294                   NCOPY(m,s,na);
01295 /*
01296                   Now some code to speed up a few special cases
01297 */
01298                   if ( v[0] == EXPONENT ) {
01299                       if ( v[1] == FUNHEAD+4 && v[FUNHEAD] == -SYMBOL &&
01300                       v[FUNHEAD+2] == -SNUMBER && v[FUNHEAD+3] < MAXPOWER
01301                       && v[FUNHEAD+3] > -MAXPOWER ) {
01302                           v[0] = SYMBOL;
01303                           v[1] = 4;
01304                           v[2] = v[FUNHEAD+1];
01305                           v[3] = v[FUNHEAD+3];
01306                           m = v+4;
01307                       }
01308                       else if ( v[1] == FUNHEAD+ARGHEAD+11
01309                        && v[FUNHEAD] == ARGHEAD+9
01310                        && v[FUNHEAD+ARGHEAD] == 9
01311                        && v[FUNHEAD+ARGHEAD+1] == DOTPRODUCT
01312                        && v[FUNHEAD+ARGHEAD+8] == 3
01313                        && v[FUNHEAD+ARGHEAD+7] == 1
01314                        && v[FUNHEAD+ARGHEAD+6] == 1
01315                        && v[FUNHEAD+ARGHEAD+5] == 1
01316                        && v[FUNHEAD+ARGHEAD+9] == -SNUMBER
01317                        && v[FUNHEAD+ARGHEAD+10] < MAXPOWER
01318                        && v[FUNHEAD+ARGHEAD+10] > -MAXPOWER ) {
01319                           v[0] = DOTPRODUCT;
01320                           v[1] = 5;
01321                           v[2] = v[FUNHEAD+ARGHEAD+3];
01322                           v[3] = v[FUNHEAD+ARGHEAD+4];
01323                           v[4] = v[FUNHEAD+ARGHEAD+10];
01324                           m = v+5;
01325                       }
01326                   }
01327               }
01328               else { while ( t < u ) *m++ = *t++; }
01329 /*
01330           #] FUNCTIONS :
01331 */
01332       }
01333       t = uu;
01334   } while ( t < r );
01335   t = from;           /* Copy coefficient */
01336   t += *t;
01337   if ( r < t ) do { *m++ = *r++; } while ( r < t );
01338   if ( ( sgn & 1 ) != 0 ) m[-1] = -m[-1];
01339   *to = WORDDIF(m,to);
01340   if ( dirty ) AN.WildDirt = dirty;
01341   return(*to);
01342 }
01343
01344 /*
01345       #] WildFill :
01346       #[ ResolveSet :         WORD ResolveSet(from,to,subs)
01347
```

```
01348         The set syntax is:
01349         SET,length,subterm,where,whichmember[,where,whichmember]
01350
01351         setlength is 2*n+1 with n the number of set substitutions.
01352         length = setlength + subtermlength + 2
01353
01354         At `where' is the number of the set and `whichmember' is the
01355         number of the element. This is still a symbol/dollar and we
01356         have to find the substitution in the wildcards.
01357         The output is the subterm in which the setelements have been
01358         substituted. This is ready for further wildcard substitutions.
01359 */
01360
01361 int ResolveSet(PHEAD WORD *from, WORD *to, WORD *subs)
01362 {
01363     GETBIDENTITY
01364     WORD *m, *s, *w, j, i, ii, i3, flag, num;
01365     DOLLARS d = 0;
01366 #ifdef WITHPTHREADS
01367     int nummodopt, dtype = -1;
01368 #endif
01369     m = to;                     /* pointer in output */
01370     s = from + 2;
01371     w = s + s[1];
01372     while ( s < w ) *m++ = *s++;
01373     j = (from[1] - WORDDIF(w,from) ) >> 1;
01374     m = subs + subs[1];
01375     subs += SUBEXPSIZE;
01376     s = subs;
01377     i = 0;
01378     while ( s < m ) { i++; s += s[1]; }
01379     m = to;
01380     if ( *m >= FUNCTION && functions[*m-FUNCTION].spec
01381     >= TENSORFUNCTION ) flag = 0;
01382     else flag = 1;
01383     while ( --j >= 0 ) {
01384         if ( w[1] >= 0 ) {
01385             s = subs;
01386             for ( ii = 0; ii < i; ii++ ) {
01387                 if ( *s == SYMTONUM && s[2] == w[1] ) { num = s[3]; goto GotOne; }
01388                 s += s[1];
01389             }
01390             MLOCK(ErrorMessageLock);
01391             MesPrint(" Unresolved setelement during substitution");
01392             MUNLOCK(ErrorMessageLock);
01393             return(-1);
01394         }
01395         else {  /* Dollar ! */
01396             d = Dollars - w[1];
01397 #ifdef WITHPTHREADS
01398             if ( AS.MultiThreaded ) {
01399                 for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01400                     if ( -w[1] == ModOptdollars[nummodopt].number ) break;
01401                 }
01402                 if ( nummodopt < NumModOptdollars ) {
01403                     dtype = ModOptdollars[nummodopt].type;
01404                     if ( DollarLocalCopy(dtype) ) {
01405                         d = ModOptdollars[nummodopt].dstruct+AT.identity;
01406                     }
01407                     else {
01408                         LOCK(d->pthreadslock);
01409                     }
01410                 }
01411             }
01412 #endif
01413             if ( d->type == DOLNUMBER || d->type == DOLTERMS ) {
01414                 if ( d->where[0] == 4 && d->where[3] == 3 && d->where[2] == 1
01415                 && d->where[1] > 0 && d->where[4] == 0 ) {
01416                     num = d->where[1]; goto GotOne;
01417                 }
01418             }
01419             else if ( d->type == DOLINDEX ) {
01420                 if ( d->index > 0 && d->index < AM.OffsetIndex ) {
01421                     num = d->index; goto GotOne;
01422                 }
01423             }
01424             else if ( d->type == DOLARGUMENT ) {
01425                 if ( d->where[0] == -SNUMBER && d->where[1] > 0 ) {
01426                     num = d->where[1]; goto GotOne;
01427                 }
01428             }
01429             else if ( d->type == DOLWILDARGS ) {
01430                 if ( d->where[0] == 1 &&
01431                 d->where[1] > 0 && d->where[1] < AM.OffsetIndex ) {
01432                     num = d->where[1]; goto GotOne;
01433                 }
01434                 if ( d->where[0] == 0 && d->where[1] < 0 && d->where[3] == 0 ) {
```

```
01435                     if ( ( d->where[1] == -SNUMBER && d->where[2] > 0 )
01436                     || ( d->where[1] == -INDEX && d->where[2] > 0
01437                     && d->where[2] < AM.OffsetIndex ) ) {
01438                         num = d->where[2]; goto GotOne;
01439                     }
01440                 }
01441             }
01442 #ifdef WITHPTHREADS
01443             if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
01444 #endif
01445             MLOCK(ErrorMessageLock);
01446             MesPrint("Unusable type of variable $%s in set substitution",
01447                 AC.dollarnames->namebuffer+d->name);
01448             MUNLOCK(ErrorMessageLock);
01449             return(-1);
01450         }
01451 GotOne:;
01452 #ifdef WITHPTHREADS
01453         if ( dtype > 0 && ! DollarLocalCopy(dtype) ) { UNLOCK(d->pthreadslock); }
01454 #endif
01455         ii = m[*w];
01456         if ( ii >= 2*MAXPOWER ) i3 = ii - 2*MAXPOWER;
01457         else if ( ii <= -2*MAXPOWER ) i3 = -ii - 2*MAXPOWER;
01458         else i3 = ( ii >= 0 ) ? ii: -ii - 1;
01459 
01460         if ( num > ( Sets[i3].last - Sets[i3].first ) || num <= 0 ) {
01461             MLOCK(ErrorMessageLock);
01462             MesPrint("Array bound check during set substitution");
01463             MesPrint("    value is %d",num);
01464             MUNLOCK(ErrorMessageLock);
01465             return(-1);
01466         }
01467         m[*w] = (SetElements+Sets[i3].first)[num-1];
01468         if ( Sets[i3].type == CSYMBOL && m[*w] > MAXPOWER ) {
01469             if ( ii >= 2*MAXPOWER ) m[*w] -= 2*MAXPOWER;
01470             else if ( ii <= -2*MAXPOWER ) m[*w] = -(m[*w] - 2*MAXPOWER);
01471             else {
01472                 m[*w] -= MAXPOWER;
01473                 if ( m[*w] < MAXPOWER ) m[*w] -= 2*MAXPOWER;
01474                 if ( flag ) MakeDirty(m,m+*w,1);
01475             }
01476         }
01477         else if ( Sets[i3].type == CSYMBOL ) {
01478             if ( ii >= 2*MAXPOWER ) m[*w] += 2*MAXPOWER;
01479             else if ( ii <= -2*MAXPOWER ) m[*w] = -m[*w] - 2*MAXPOWER;
01480             else if ( ii < 0 ) m[*w] = - m[*w];
01481         }
01482         else if ( ii < 0 ) m[*w] = - m[*w];
01483         w += 2;
01484     }
01485     m = to;
01486     if ( *m >= FUNCTION && functions[*m-FUNCTION].spec
01487     >= TENSORFUNCTION ) {
01488         w = from + 2 + from[3];
01489         if ( *w == 0 ) {    /* We had function -> tensor */
01490             m = from + 2 + FUNHEAD; s = to + FUNHEAD;
01491             while ( m < w ) {
01492                 if ( *m == -INDEX || *m == -VECTOR ) {}
01493                 else if ( *m == -ARGWILD ) { *s++ = FUNNYWILD; }
01494                 else {
01495                     MLOCK(ErrorMessageLock);
01496                     MesPrint("Illegal argument in tensor after set substitution");
01497                     MUNLOCK(ErrorMessageLock);
01498                     SETERROR(-1)
01499                 }
01500                 *s++ = m[1];
01501                 m += 2;
01502             }
01503             to[1] = WORDDIF(s,to);
01504         }
01505     }
01506     return(0);
01507 }
01508 
01509 /*
01510         #] ResolveSet :
01511         #[ ClearWild :          void ClearWild()
01512 
01513     Clears the current wildcard settings and makes them ready for
01514     CheckWild and AddWild.
01515 
01516 */
01517 
01518 void ClearWild(PHEAD0)
01519 {
01520     GETBIDENTITY
01521     WORD n, nn, *w;
```

```
01522     n = (AN.WildValue[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;     /* Number of wildcards */
01523     AN.NumWild = nn = n;
01524     if ( n > 0 ) {
01525         w = AT.WildMask;
01526         do { *w++ = 0; } while ( --n > 0 );
01527         w = AN.WildValue;
01528         do {
01529             if ( *w == SYMTONUM ) *w = SYMTOSYM;
01530             w += w[1];
01531         } while ( --nn > 0 );
01532     }
01533 }
01534
01535 /*
01536         #] ClearWild :
01537         #[ AddWild :            WORD AddWild(oldnumber,type,newnumber)
01538
01539         Adds a wildcard assignment.
01540         Extra parameter in AN.argaddress;
01541
01542 */
01543
01544 int AddWild(PHEAD WORD oldnumber, WORD type, WORD newnumber)
01545 {
01546     GETBIDENTITY
01547     WORD *w, *m, n, k, i = -1;
01548     CBUF *C = cbuf+AT.ebufnum;
01549     WORD eattensor = type & EATTENSOR;
01550     type = type & ~EATTENSOR;
01551 DEBUG(WORD *mm;)
01552     AN.WildReserve = 0;
01553     m = AT.WildMask;
01554     w = AN.WildValue;
01555     n = AN.NumWild;
01556     if ( n <= 0 ) { return(-1); }
01557     if ( type <= SYMTOSUB ) {
01558         do {
01559             if ( w[2] == oldnumber && *w <= SYMTOSUB ) {
01560                 if ( n > 1 && w[4] == SETTONUM ) i = w[7];
01561                 *w = type;
01562                 if ( *m != 2 ) *m = 1;
01563                 if ( type != SYMTOSUB ) {
01564                     if ( type == SYMTONUM ) AN.MaskPointer = m;
01565                     w[3] = newnumber;
01566                     goto FlipOn;
01567                 }
01568                 m = AddRHS(AT.ebufnum,1);
01569                 w[3] = C->numrhs;
01570                 w = AN.argaddress;
01571 DEBUG(mm = m;)
01572                 n = *w - ARGHEAD;
01573                 w += ARGHEAD;
01574                 while ( (m + n + 10) > C->Top ) m = DoubleCbuffer(AT.ebufnum,m,4);
01575                 while ( --n >= 0 ) *m++ = *w++;
01576                 *m++ = 0;
01577                 C->rhs[C->numrhs+1] = m;
01578 DEBUG(MesPrint("Thread %w(d): m=(%d,%d,%d,%d)(%d)",mm[0],mm[1],mm[2],mm[3],C->numrhs);)
01579                 C->Pointer = m;
01580                 goto FlipOn;
01581             }
01582             m++; w += w[1];
01583         } while ( --n > 0 );
01584     }
01585     else if ( type == ARGTOARG ) {
01586         do {
01587             if ( w[2] == oldnumber && *w == ARGTOARG ) {
01588                 *m = 1;
01589                 m = AddRHS(AT.ebufnum,1);
01590                 w[3] = C->numrhs;
01591                 w = AN.argaddress;
01592 DEBUG(mm=m;)
01593                 if ( eattensor ) {
01594                     n = newnumber;
01595                     *m++ = n;
01596                     w = AN.argaddress;
01597                 }
01598                 else {
01599                     while ( --newnumber >= 0 ) { NEXTARG(w) }
01600                     n = WORDDIF(w,AN.argaddress);
01601                     w = AN.argaddress;
01602                     *m++ = 0;
01603                 }
01604                 while ( (m + n + 10) > C->Top ) m = DoubleCbuffer(AT.ebufnum,m,5);
01605 DEBUG(if ( mm != m-1 ) MesPrint("Thread %w(e): Alarm!"); mm = m-1;)
01606                 while ( --n >= 0 ) *m++ = *w++;
01607                 *m++ = 0;
01608                 C->rhs[C->numrhs+1] = m;
```

```
01609                 C->Pointer = m;
01610 DEBUG(MesPrint("Thread %w(e): w=(%d,%d,%d,%d)(%d)",mm[0],mm[1],mm[2],mm[3],C->numrhs);)
01611                 return(0);
01612             }
01613             m++; w += w[1];
01614         } while ( --n > 0 );
01615     }
01616     else if ( type == ARLTOARL ) {
01617         do {
01618             if ( w[2] == oldnumber && *w == ARGTOARG ) {
01619                 WORD **a;
01620                 *m = 1;
01621                 m = AddRHS(AT.ebufnum,1);
01622                 w[3] = C->numrhs;
01623 DEBUG(mm=m;)
01624                 a = (WORD **)(AN.argaddress); n = 0; k = newnumber;
01625                 while ( --newnumber >= 0 ) {
01626                     w = *a++;
01627                     if ( *w > 0 ) n += *w;
01628                     else if ( *w <= -FUNCTION ) n++;
01629                     else n += 2;
01630                 }
01631                 *m++ = 0;
01632                 while ( (m + n + 10) > C->Top ) m = DoubleCbuffer(AT.ebufnum,m,6);
01633 DEBUG(if ( mm != m-1 ) MesPrint("Thread %w(f): Alarm!"); mm = m-1;)
01634                 a = (WORD **)(AN.argaddress);
01635                 while ( --k >= 0 ) {
01636                     w = *a++;
01637                     if ( *w > 0 ) { n = *w; NCOPY(m,w,n); }
01638                     else if ( *w <= -FUNCTION ) *m++ = *w++;
01639                     else { *m++ = *w++; *m++ = *w++; }
01640                 }
01641                 *m++ = 0;
01642                 C->rhs[C->numrhs+1] = m;
01643 DEBUG(MesPrint("Thread %w(f): w=(%d,%d,%d,%d)(%d)",mm[0],mm[1],mm[2],mm[3],C->numrhs);)
01644                 C->Pointer = m;
01645                 return(0);
01646             }
01647             m++; w += w[1];
01648         } while ( --n > 0 );
01649     }
01650     else if ( type == VECTOSUB || type == INDTOSUB ) {
01651         WORD *ss, *sstop, *tt, *ttstop, j, *v1, *v2 = 0;
01652         do {
01653             if ( w[2] == oldnumber && ( *w == type ||
01654             ( type == VECTOSUB && ( *w == VECTOVEC || *w == VECTOMIN ) )
01655             || ( type == INDTOSUB && *w == INDTOIND ) ) ) {
01656                 if ( n > 1 && w[4] == SETTONUM ) i = w[7];
01657                 *w = type;
01658                 *m = 1;
01659                 m = AddRHS(AT.ebufnum,1);
01660                 w[3] = C->numrhs;
01661                 w = AN.argaddress;
01662                 n = *w - ARGHEAD;
01663                 w += ARGHEAD;
01664                 while ( (m + n + 10) > C->Top ) m = DoubleCbuffer(AT.ebufnum,m,7);
01665                 while ( --n >= 0 ) *m++ = *w++;
01666                 *m++ = 0;
01667                 C->rhs[C->numrhs+1] = m;
01668                 C->Pointer = m;
01669                 m = AddRHS(AT.ebufnum,1);
01670                 w = AN.argaddress;
01671                 n = *w - ARGHEAD;
01672                 w += ARGHEAD;
01673                 while ( (m + n + 10) > C->Top ) m = DoubleCbuffer(AT.ebufnum,m,8);
01674                 sstop = w + n;
01675                 while ( w < sstop ) {           /* Run over terms */
01676                     tt = w + *w; ttstop = tt - ABS(tt[-1]);
01677                     ss = m; m++; w++;
01678                     while ( w < ttstop ) {      /* Subterms */
01679                         if ( *w != INDEX ) {
01680                             j = w[1];
01681                             NCOPY(m,w,j);
01682                         }
01683                         else {
01684                             v1 = m;
01685                             *m++ = *w++;
01686                             *m++ = j = *w++;
01687                             j -= 2;
01688                             while ( --j >= 0 ) {
01689                                 if ( *w >= MINSPEC ) *m++ = *w++;
01690                                 else v2 = w++;
01691                             }
01692                             j = WORDDIF(m,v1);
01693                             if ( j != v1[1] ) {
01694                                 if ( j <= 2 ) m -= 2;
01695                                 else v1[1] = j;
```

```
01696                                   *m++ = VECTOR;
01697                                   *m++ = 4;
01698                                   *m++ = *v2;
01699                                   *m++ = FUNNYVEC;
01700                               }
01701                           }
01702                       }
01703                       while ( w < tt ) *m++ = *w++;
01704                       *ss = WORDDIF(m,ss);
01705                   }
01706                   *m++ = 0;
01707                   C->rhs[C->numrhs+1] = m;
01708                   C->Pointer = m;
01709                   if ( m > C->Top ) {
01710 /* INTERNAL_ERROR_EXCL_START */
01711                       MLOCK(ErrorMessageLock);
01712                       MesPrint("!>Internal problems with extra compiler buffer");
01713                       MUNLOCK(ErrorMessageLock);
01714                       Terminate(-1);
01715 /* INTERNAL_ERROR_EXCL_STOP */
01716                   }
01717                   goto FlipOn;
01718               }
01719               m++; w += w[1];
01720           } while ( --n > 0 );
01721       }
01722       else {
01723           do {
01724               if ( w[2] == oldnumber && ( *w == type || ( type == VECTOVEC
01725               && ( *w == VECTOMIN || *w == VECTOSUB ) ) || ( type == VECTOMIN
01726               && ( *w == VECTOVEC || *w == VECTOSUB ) )
01727               || ( type == INDTOIND && *w == INDTOSUB ) ) ) {
01728                   if ( n > 1 && w[4] == SETTONUM ) i = w[7];
01729                   *w = type;
01730                   w[3] = newnumber;
01731                   *m = 1;
01732                   goto FlipOn;
01733               }
01734               m++; w += w[1];
01735           } while ( --n > 0 );
01736       }
01737 /* INTERNAL_ERROR_EXCL_START */
01738       MLOCK(ErrorMessageLock);
01739       MesPrint("!>Bug in AddWild.");
01740       MUNLOCK(ErrorMessageLock);
01741       return(-1);
01742 /* INTERNAL_ERROR_EXCL_STOP */
01743 FlipOn:
01744       if ( i >= 0 ) {
01745           m = AT.WildMask;
01746           w = AN.WildValue;
01747           n = AN.NumWild;
01748           while ( --n >= 0 ) {
01749               if ( w[2] == i && *w == SYMTONUM ) {
01750                   *m = 2;
01751                   return(0);
01752               }
01753               m++; w += w[1];
01754           }
01755 /* INTERNAL_ERROR_EXCL_START */
01756           MLOCK(ErrorMessageLock);
01757           MesPrint("!>Bug in AddWild with passing set[i]");
01758           MUNLOCK(ErrorMessageLock);
01759 /* INTERNAL_ERROR_EXCL_STOP */
01760 /*
01761           For the moment we want to crash here. That is easier with debugging.
01762 */
01763 #ifdef WITHPTHREADS
01764           { WORD *s = 0;
01765               *s++ = 1;
01766           }
01767 #endif
01768           Terminate(-1);
01769       }
01770       return(0);
01771 }
01772
01773 /*
01774           #] AddWild :
01775           #[ CheckWild :          WORD CheckWild(oldnumber,type,newnumber,newval)
01776
01777           Tests whether a wildcard assignment is allowed.
01778           A return value of zero means that it is allowed (nihil obstat).
01779           If the variable has been assigned already its existing
01780           assignment is returned in AN.oldvalue and AN.oldtype, which are
01781           global variables.
01782
```

```
01783         Note the special problem with name?set[i]. Here we have to pass
01784         an extra assignment. This cannot be done via globals as we
01785         call CheckWild sometimes twice before calling AddWild.
01786         Trick: Check the assignment of the number and if OK put it
01787         in place, but don't alter the used flag (if needed).
01788         Then AddWild can alter the used flag but the value is there.
01789         As long as this trick is `hanging' we turn on the flag:
01790         `AN.WildReserve' which is either turned off by AddWild or by
01791         a failing call to CheckWild.
01792
01793         With ARGTOARG the tensors give the number of arguments
01794         or-ed with EATTENSOR which is at least 8192.
01795 */
01796
01797 int CheckWild(PHEAD WORD oldnumber, WORD type, WORD newnumber, WORD *newval)
01798 {
01799     GETBIDENTITY
01800     WORD *w, *m, *s, n, old2, inset;
01801     WORD n2, oldval, dirty, i, j;
01802     int notflag = 0, retblock = 0;
01803     CBUF *C = cbuf+AT.ebufnum;
01804     WORD eattensor = type & EATTENSOR;
01805     type = type & ~EATTENSOR;
01806     m = AT.WildMask;
01807     w = AN.WildValue;
01808     n = AN.NumWild;
01809     if ( n <= 0 ) { AN.oldtype = -1; AN.WildReserve = 0; return(-1); }
01810     switch ( type ) {
01811         case SYMTONUM :
01812             *newval = newnumber;
01813             do {
01814                 if ( w[2] == oldnumber && *w <= SYMTOSUB ) {
01815                     old2 = *w;
01816                     if ( !*m ) goto TestSet;
01817                     AN.MaskPointer = m;
01818                     if ( *w == SYMTONUM  && w[3] == newnumber ) {
01819                         return(0);
01820                     }
01821                     AN.oldtype = old2; AN.oldvalue = w[3]; goto NoMatch;
01822                 }
01823                 m++; w += w[1];
01824             } while ( --n > 0 );
01825             break;
01826         case SYMTOSYM :
01827             *newval = newnumber;
01828             do {
01829                 if ( w[2] == oldnumber && *w <= SYMTOSUB ) {
01830                     old2 = *w;
01831                     if ( *w == SYMTOSYM ) {
01832                         if ( !*m ) goto TestSet;
01833                         if ( newnumber >= 0 && (w+4) < AN.WildStop
01834                         && ( w[4] == FROMSET || w[4] == SETTONUM )
01835                         && w[7] >= 0 ) goto TestSet;
01836                         if ( w[3] == newnumber ) return(0);
01837                     }
01838                     else {
01839                         if ( !*m ) goto TestSet;
01840                     }
01841                     goto NoM;
01842                 }
01843                 m++; w += w[1];
01844             } while ( --n > 0 );
01845             break;
01846         case SYMTOSUB :
01847 /*
01848             Now newval contains the pointer to the argument.
01849 */
01850             {
01851 /*
01852                 Search for vector or index nature. If so: reject.
01853 */
01854                 WORD *ss, *sstop, *tt, *ttstop;
01855                 ss = newval;
01856                 sstop = ss + *ss;
01857                 ss += ARGHEAD;
01858                 while ( ss < sstop ) {
01859                     tt = ss + *ss;
01860                     ttstop = tt - ABS(tt[-1]);
01861                     ss++;
01862                     while ( ss < ttstop ) {
01863                         if ( *ss == INDEX ) goto NoMatch;
01864                         ss += ss[1];
01865                     }
01866                     ss = tt;
01867                 }
01868             }
01869             do {
```

```
01870                 if ( w[2] == oldnumber && *w <= SYMTOSUB ) {
01871                     old2 = *w;
01872                     if ( *w == SYMTONUM || *w == SYMTOSYM ) {
01873                         if ( !*m ) {
01874                             s = w + w[1];
01875                             if ( s >= AN.WildStop || *s != SETTONUM )
01876                                     goto TestSet;
01877                         }
01878                     }
01879                     else if ( *w == SYMTOSUB ) {
01880                         if ( !*m ) {
01881                             s = w + w[1];
01882                             if ( s >= AN.WildStop || *s != SETTONUM )
01883                                     goto TestSet;
01884                         }
01885                         n = *newval - 2;
01886                         newval += 2;
01887                         m = C->rhs[w[3]];
01888                         if ( (C->rhs[w[3]+1] - m - 1) == n ) {
01889                             while ( n > 0 ) {
01890                                 if ( *m != *newval ) {
01891                                     m++; newval++; break;
01892                                 }
01893                                 m++; newval++;
01894                                 n--;
01895                             }
01896                             if ( n <= 0 ) return(0);
01897                         }
01898                     }
01899                     AN.oldtype = old2; AN.oldvalue = w[3]; goto NoMatch;
01900                 }
01901                 m++; w += w[1];
01902             } while ( --n > 0 );
01903             break;
01904         case ARGTOARG :
01905             do {
01906                 if ( w[2] == oldnumber && *w == ARGTOARG ) {
01907                     if ( !*m ) return(0);       /* nihil obstat */
01908                     m = C->rhs[w[3]];
01909                     if ( eattensor ) {
01910                         n = newnumber;
01911                         if ( *m != 0 ) {
01912                             if ( n == *m ) {
01913                                 m++;
01914                                 while ( --n >= 0 ) {
01915                                     if ( *m != *newval ) {
01916                                         m++; newval++; break;
01917                                     }
01918                                     m++; newval++;
01919                                 }
01920                                 if ( n < 0 ) return(0);
01921                             }
01922                         }
01923                         else {
01924                             m++;
01925                             while ( --n >= 0 ) {
01926                                 if ( *newval != m[1] || ( *m != -INDEX
01927                                 && *m != -VECTOR && *m != -SNUMBER ) ) break;
01928                                 m += 2;
01929                                 newval++;
01930                             }
01931                             if ( n < 0 && *m == 0 ) return(0);
01932                         }
01933                     }
01934                     else {
01935                         i = newnumber;
01936                         if ( *m != 0 ) {    /* Tensor field */
01937                             if ( *m == i ) {
01938                                 m++;
01939                                 while ( --i >= 0 ) {
01940                                     if ( *m != newval[1]
01941                                     || ( *newval != -VECTOR
01942                                       && *newval != -INDEX
01943                                       && *newval != -SNUMBER ) ) break;
01944                                     newval += 2;
01945                                     m++;
01946                                 }
01947                                 if ( i < 0 ) return(0);
01948                             }
01949                         }
01950                         else {
01951                             m++;
01952                             s = newval;
01953                             while ( --i >= 0 ) { NEXTARG(s) }
01954                             n = WORDDIF(s,newval);
01955                             while ( --n >= 0 ) {
01956                                 if ( *m != *newval ) {
```

```
01957                                   m++; newval++; break;
01958                               }
01959                               m++; newval++;
01960                           }
01961                           if ( n < 0 && *m == 0 ) return(0);
01962                       }
01963                   }
01964                   AN.oldtype = *w; AN.oldvalue = w[3]; goto NoMatch;
01965               }
01966               m++; w += w[1];
01967           } while ( --n > 0 );
01968           break;
01969       case ARLTOARL :
01970           do {
01971               if ( w[2] == oldnumber && *w == ARGTOARG ) {
01972                   WORD **a;
01973                   if ( !*m ) return(0);       /* nihil obstat */
01974                   m = C->rhs[w[3]];
01975                   i = newnumber;
01976                   a = (WORD **)newval;
01977                   if ( *m != 0 ) {    /* Tensor field */
01978                       if ( *m == i ) {
01979                           m++;
01980                           while ( --i >= 0 ) {
01981                               s = *a++;
01982                               if ( *m != s[1]
01983                               || ( *s != -VECTOR
01984                                 && *s != -INDEX
01985                                 && *s != -SNUMBER ) ) break;
01986                               m++;
01987                           }
01988                           if ( i < 0 ) return(0);
01989                       }
01990                   }
01991                   else {
01992                       m++;
01993                       while ( --i >= 0 ) {
01994                           s = *a++;
01995                           if ( *s > 0 ) {
01996                               n = *s;
01997                               while ( --n >= 0 ) {
01998                                   if ( *s != *m ) {
01999                                       s++; m++; break;
02000                                   }
02001                                   s++; m++;
02002                               }
02003                               if ( n >= 0 ) break;
02004                           }
02005                           else if ( *s <= -FUNCTION ) {
02006                               if ( *s != *m ) {
02007                                   s++; m++; break;
02008                               }
02009                               s++; m++;
02010                           }
02011                           else {
02012                               if ( *s != *m ) {
02013                                   s++; m++; break;
02014                               }
02015                               s++; m++;
02016                               if ( *s != *m ) {
02017                                   s++; m++; break;
02018                               }
02019                               s++; m++;
02020                           }
02021                       }
02022                       if ( i < 0 && *m == 0 ) return(0);
02023                   }
02024                   AN.oldtype = *w; AN.oldvalue = w[3]; goto NoMatch;
02025               }
02026               m++; w += w[1];
02027           } while ( --n > 0 );
02028           break;
02029       case VECTOSUB :
02030       case INDTOSUB :
02031 /*
02032           Now newval contains the pointer to the argument(s).
02033 */
02034           {
02035 /*
02036               Search for vector or index nature. If not so: reject.
02037 */
02038               WORD *ss, *sstop, *tt, *ttstop, count, jt;
02039               ss = newval;
02040               sstop = ss + *ss;
02041               ss += ARGHEAD;
02042               while ( ss < sstop ) {
02043                   tt = ss + *ss;
```

```
02044                   ttstop = tt - ABS(tt[-1]);
02045                   ss++;
02046                   count = 0;
02047                   while ( ss < ttstop ) {
02048                       if ( *ss == INDEX ) {
02049                           jt = ss[1] - 2; ss += 2;
02050                           while ( --jt >= 0 ) {
02051                               if ( *ss < MINSPEC ) count++;
02052                               ss++;
02053                           }
02054                       }
02055                       else ss += ss[1];
02056                   }
02057                   if ( count != 1 ) goto NoMatch;
02058                   ss = tt;
02059               }
02060           }
02061           do {
02062               if ( w[2] == oldnumber ) {
02063                   old2 = *w;
02064                   if ( ( type == VECTOSUB && ( *w == VECTOVEC || *w == VECTOMIN ) )
02065                   || ( type == INDTOSUB && *w == INDTOIND ) ) {
02066                       if ( !*m ) goto TestSet;
02067                       AN.oldtype = old2; AN.oldvalue = w[3]; goto NoMatch;
02068                   }
02069                   else if ( *w == type ) {
02070                       if ( !*m ) goto TestSet;
02071                       if ( type != INDTOIND && type != INDTOSUB ) {   /* Prevent double index */
02072                           n = *newval - 2;
02073                           newval += 2;
02074                           m = C->rhs[w[3]];
02075                           if ( (C->rhs[w[3]+1] - m - 1) == n ) {
02076                               while ( n > 0 ) {
02077                                   if ( *m != *newval ) {
02078                                       m++; newval++; break;
02079                                   }
02080                                   m++; newval++;
02081                                   n--;
02082                               }
02083                               if ( n <= 0 ) return(0);
02084                           }
02085                       }
02086                       AN.oldtype = old2; AN.oldvalue = w[3]; goto NoMatch;
02087                   }
02088               }
02089               m++; w += w[1];
02090           } while ( --n > 0 );
02091           break;
02092       default :
02093           *newval = newnumber;
02094           do {
02095               if ( w[2] == oldnumber ) {
02096                   if ( *w == type ) {
02097                       old2 = *w;
02098                       if ( !*m ) goto TestSet;
02099                       if ( newnumber >= 0 && (w+4) < AN.WildStop &&
02100                           ( w[4] == FROMSET || w[4] == SETTONUM )
02101                           && w[7] >= 0 ) goto TestSet;
02102                       if ( newnumber < 0 && *w == VECTOVEC
02103                           && (w+4) < AN.WildStop && ( w[4] == FROMSET
02104                           || w[4] == SETTONUM ) && w[7] >= 0 ) goto TestSet;
02105 /*
02106                       The next statement kills multiple indices -> vector
02107 */
02108                       if ( *w == INDTOIND && w[3] < 0 ) goto NoMatch;
02109                       if ( w[3] == newnumber ) {
02110                           if ( *w != FUNTOFUN || newnumber < FUNCTION
02111                           || functions[newnumber-FUNCTION].spec ==
02112                           functions[oldnumber-FUNCTION].spec )
02113                               return(0);
02114                       }
02115                       AN.oldtype = old2; AN.oldvalue = w[3]; goto NoMatch;
02116                   }
02117                   else if ( ( type == VECTOVEC &&
02118                           ( *w == VECTOSUB || *w == VECTOMIN ) )
02119                          || ( type == INDTOIND && *w == INDTOSUB ) ) {
02120                       if ( *m ) goto NoMatch;
02121                       old2 = *w;
02122                       goto TestSet;
02123                   }
02124                   else if ( type == VECTOMIN &&
02125                           ( *w == VECTOSUB || *w == VECTOVEC ) ) {
02126                       if ( *m ) goto NoMatch;
02127                       old2 = *w;
02128                       goto TestSet;
02129                   }
02130               }
```

```
02131                 m++; w += w[1];
02132                 if ( n > 1 && ( *w == FROMSET
02133                 || *w == SETTONUM ) ) { n--; m++; w += w[1]; }
02134             } while ( --n > 0 );
02135             break;
02136     }
02137 /* INTERNAL_ERROR_EXCL_START */
02138     AN.oldtype = -1;
02139     AN.oldvalue = -1;
02140     AN.WildReserve = 0;
02141     MLOCK(ErrorMessageLock);
02142     MesPrint("!>Inconsistency in Wildcard prototype.");
02143     MUNLOCK(ErrorMessageLock);
02144     return(-1);
02145 /* INTERNAL_ERROR_EXCL_STOP */
02146 NoMatch:
02147     AN.WildReserve = 0;
02148     return(1+retblock);
02149 /*
02150     Here we test the compatibility with a set specification.
02151 */
02152 TestSet:
02153     dirty = *m;
02154     oldval = w[3];
02155     w += w[1];
02156     if ( w < AN.WildStop && ( *w == FROMSET || *w == SETTONUM ) ) {
02157         WORD k;
02158         s = w;
02159         j = w[2]; n2 = w[3];
02160 /*
02161             if SETTONUM:  x?j[n2]
02162             if FROMSET:   x?j?n2    or x?j and n2 = -WOLDOFFSET.
02163 */
02164         if ( j > WILDOFFSET ) {
02165             j -= 2*WILDOFFSET;
02166             notflag = 1;
02167 /*
02168         ???????
02169 */
02170             AN.oldtype = -1;
02171             AN.oldvalue = -1;
02172         }
02173         if ( j < AM.NumFixedSets ) {    /* special set */
02174             retblock = 1;
02175             switch ( j ) {
02176                 case POS_:
02177                     if ( type != SYMTONUM ||
02178                     newnumber <= 0 ) goto NoMnot;
02179                     break;
02180                 case POS0_:
02181                     if ( type != SYMTONUM ||
02182                     newnumber < 0 ) goto NoMnot;
02183                     break;
02184                 case NEG_:
02185                     if ( type != SYMTONUM ||
02186                     newnumber >= 0 ) goto NoMnot;
02187                     break;
02188                 case NEG0_:
02189                     if ( type != SYMTONUM ||
02190                     newnumber > 0 ) goto NoMnot;
02191                     break;
02192                 case EVEN_:
02193                     if ( type != SYMTONUM ||
02194                     ( newnumber & 1 ) != 0 ) goto NoMnot;
02195                     break;
02196                 case ODD_:
02197                     if ( type != SYMTONUM ||
02198                     ( newnumber & 1 ) == 0 ) goto NoMnot;
02199                     break;
02200                 case Z_:
02201                     if ( type != SYMTONUM ) goto NoMnot;
02202                     break;
02203                 case SYMBOL_:
02204                     if ( type != SYMTOSYM ) goto NoMnot;
02205                     break;
02206                 case FIXED_:
02207                     if ( type != INDTOIND ||
02208                     newnumber >= AM.OffsetIndex ||
02209                     newnumber < 0 ) goto NoMnot;
02210                     break;
02211                 case INDEX_:
02212                     if ( type != INDTOIND ||
02213                     newnumber < 0 ) goto NoMnot;
02214                     break;
02215                 case Q_:
02216                     if ( type == SYMTONUM ) break;
02217                     if ( type == SYMTOSUB ) {
```

```
02218                           WORD *ss, *sstop;
02219                           ss = newval;
02220                           sstop = ss + *ss;
02221                           ss += ARGHEAD;
02222                           if ( ss >= sstop ) break;
02223                           if ( ss + *ss < sstop ) goto NoMnot;
02224                           if ( ABS(sstop[-1]) == ss[0]-1 ) break;
02225                       }
02226                       goto NoMnot;
02227                   case DUMMYINDEX_:
02228                       if ( type != INDTOIND ||
02229                       newnumber < AM.IndDum || newnumber >= AM.IndDum+MAXDUMMIES ) goto NoMnot;
02230                       break;
02231                   case VECTOR_:
02232                       if ( type != VECTOVEC ) goto NoMnot;
02233                       break;
02234                   default:
02235                       goto NoMnot;
02236               }
02237 Mnot:
02238               if ( notflag ) goto NoM;
02239               return(0);
02240 NoMnot:
02241               if ( !notflag ) goto NoM;
02242               return(0);
02243           }
02244           else if ( Sets[j].type == CRANGE ) {
02245               if ( ( type == SYMTONUM )
02246               || ( type == INDTOIND && ( newnumber > 0
02247                && newnumber <= AM.OffsetIndex ) ) ) {
02248                   if ( Sets[j].first < MAXPOWER ) {
02249                       if ( newnumber >= Sets[j].first ) goto NoMnot;
02250                   }
02251                   else if ( Sets[j].first < 3*MAXPOWER ) {
02252                       if ( newnumber+2*MAXPOWER > Sets[j].first ) goto NoMnot;
02253                   }
02254                   if ( Sets[j].last > -MAXPOWER ) {
02255                       if ( newnumber <= Sets[j].last ) goto NoMnot;
02256                   }
02257                   else if ( Sets[j].last > -3*MAXPOWER ) {
02258                       if ( newnumber-2*MAXPOWER < Sets[j].last ) goto NoMnot;
02259                   }
02260                   goto Mnot;
02261               }
02262               goto NoMnot;
02263           }
02264 /*
02265           Now we have to determine which set element
02266 */
02267           w = SetElements + Sets[j].first;
02268           m = SetElements + Sets[j].last;
02269           if ( w == m ) {
02270 /*
02271               The set is empty! This is not a match unless we have !{}, in
02272               which case it is.
02273 */
02274               if ( notflag ) return 0;
02275               AN.oldtype = old2;
02276               AN.oldvalue = oldval;
02277               goto NoMatch;
02278           }
02279
02280           if ( ( Sets[j].flags & ORDEREDSET ) == ORDEREDSET ) {
02281 /*
02282               We search first and ask questions later
02283 */
02284               i = BinarySearch(w,Sets[j].last-Sets[j].first,newnumber);
02285               if ( i < 0 ) {  /* no matter what, it is not in the set. */
02286                   goto NoMnot;
02287               }
02288               else {
02289 /*
02290                   We can set the proper parameters now to make only the
02291                   checks for the given set element.
02292                   After that we jump into the appropriate loop.
02293 */
02294                   w = m = SetElements + i;
02295                   i++;
02296                   if ( Sets[j].type == -1 || Sets[j].type == CNUMBER ) {
02297                       goto insideloop1;
02298                   }
02299                   else {
02300                       goto insideloop2;
02301                   }
02302               }
02303           }
02304           i = 1;
```

```
02305         if ( Sets[j].type == -1 || Sets[j].type == CNUMBER ) {
02306           do {
02307             insideloop1:
02308             if ( notflag ) {
02309                 switch ( type ) {
02310                     case SYMTOSYM:
02311                         if ( Sets[j].type == CNUMBER ) {}
02312                         else {
02313                             if ( *w == newnumber ) goto NoMatch;
02314                         }
02315                         break;
02316                     case SYMTONUM:
02317                     case INDTOIND:
02318                         if ( *w == newnumber ) goto NoMatch;
02319                         break;
02320                     default:
02321                         break;
02322                 }
02323             }
02324             else if ( type != SYMTONUM && type != INDTOIND
02325             && type != SYMTOSYM ) goto NoMatch;
02326             else if ( type == SYMTOSYM && Sets[j].type == CNUMBER ) goto NoMatch;
02327             else if ( *w == newnumber ) {
02328                 if ( *s == SETTONUM ) {
02329                     if ( n2 == oldnumber && type
02330                     <= SYMTOSUB ) goto NoMatch;
02331                     m = AT.WildMask;
02332                     w = AN.WildValue;
02333                     n = AN.NumWild;
02334                     while ( --n >= 0 ) {
02335                         if ( w[2] == n2 && *w <= SYMTOSUB ) {
02336                             if ( !*m ) {
02337                                 *w = SYMTONUM;
02338                                 w[3] = i;
02339                                 AN.WildReserve = 1;
02340                                 return(0);
02341                             }
02342                             if ( *w != SYMTONUM )
02343                                 goto NoMatch;
02344                             if ( w[3] == i ) return(0);
02345                             i = w[3];
02346                             j = (SetElements + Sets[j].first)[i];
02347                             if ( j == n2 ) return(0);
02348                             goto NoMatch;
02349                         }
02350                         m++; w += w[1];
02351                     }
02352                 }
02353                 else if ( n2 >= 0 ) {
02354                     *newval = *(w - Sets[j].first + Sets[n2].first);
02355                     if ( *newval > MAXPOWER ) *newval -= 2*MAXPOWER;
02356                     if ( dirty && *newval != oldval ) {
02357                         *newval = oldval; goto NoMatch;
02358                     }
02359                 }
02360                 return(0);
02361             }
02362             i++;
02363           } while ( ++w < m );
02364         }
02365         else {
02366           do {
02367             insideloop2:
02368             inset = *w;
02369             if ( notflag ) {
02370             switch ( type ) {
02371                 case SYMTONUM:
02372                 case SYMTOSYM:
02373                     if ( ( type == SYMTOSYM && *w == newnumber )
02374                     || ( type == SYMTONUM && *w-2*MAXPOWER == newnumber ) ) {
02375                         goto NoMatch;
02376                     }
02377                     /* fall through */
02378                 case SYMTOSUB:
02379                     if ( *w < 0 ) {
02380                         WORD *mm = AT.WildMask, *mmm, *part;
02381                         WORD *ww = AN.WildValue;
02382                         WORD nn = AN.NumWild;
02383                         k = -*w;
02384                         while ( --nn >= 0 ) {
02385                             if ( *mm && ww[2] == k && ww[0] == type ) {
02386                                 if ( type != SYMTOSUB ) {
02387                                     if ( ww[3] == newnumber ) goto NoMatch;
02388                                 }
02389                                 else {
02390                                     mmm = C->rhs[ww[3]];
02391                                     nn = *newval-2;
```

```
02392                                   part = newval+2;
02393                                   if ( (C->rhs[ww[3]+1]-mmm-1) == nn ) {
02394                                       while ( --nn >= 0 ) {
02395                                           if ( *mmm != *part ) {
02396                                               mmm++; part++; break;
02397                                           }
02398                                           mmm++; part++;
02399                                       }
02400                                       if ( nn < 0 ) goto NoMatch;
02401                                   }
02402                               }
02403                               break;
02404                           }
02405                           mm++; ww += ww[1];
02406                       }
02407                   }
02408                   break;
02409               case VECTOMIN:
02410                   if ( type == VECTOMIN ) {
02411                       if ( inset >= AM.OffsetVector ) { i++; continue; }
02412                       inset += WILDMASK;
02413                   }
02414                   /* fall through */
02415               case VECTOVEC:
02416                   if ( inset == newnumber ) goto NoMatch;
02417                   /* fall through */
02418               case VECTOSUB:
02419                   if ( inset - WILDOFFSET >= AM.OffsetVector ) {
02420                       WORD *mm = AT.WildMask, *mmm, *part;
02421                       WORD *ww = AN.WildValue;
02422                       WORD nn = AN.NumWild;
02423                       k = inset - WILDOFFSET;
02424                       while ( --nn >= 0 ) {
02425                           if ( *mm && ww[2] == k && ww[0] == type ) {
02426                               if ( type == VECTOVEC ) {
02427                                   if ( ww[3] == newnumber ) goto NoMatch;
02428                               }
02429                               else {
02430                                   mmm = C->rhs[ww[3]];
02431                                   nn = *newval-2;
02432                                   part = newval+2;
02433                                   if ( (C->rhs[ww[3]+1]-mmm-1) == nn ) {
02434                                       while ( --nn >= 0 ) {
02435                                           if ( *mmm != *part ) {
02436                                               mmm++; part++; break;
02437                                           }
02438                                           mmm++; part++;
02439                                       }
02440                                       if ( nn < 0 ) goto NoMatch;
02441                                   }
02442                               }
02443                               break;
02444                           }
02445                           mm++; ww += ww[1];
02446                       }
02447                   }
02448                   break;
02449               case INDTOIND:
02450                   if ( *w == newnumber ) goto NoMatch;
02451                   /* fall through */
02452               case INDTOSUB:
02453                   if ( *w - (WORD)WILDMASK >= AM.OffsetIndex ) {
02454                       WORD *mm = AT.WildMask, *mmm, *part;
02455                       WORD *ww = AN.WildValue;
02456                       WORD nn = AN.NumWild;
02457                       k = *w - WILDMASK;
02458                       while ( --nn >= 0 ) {
02459                           if ( *mm && ww[2] == k && ww[0] == type ) {
02460                               if ( type == INDTOIND ) {
02461                                   if ( ww[3] == newnumber ) goto NoMatch;
02462                               }
02463                               else {
02464                                   mmm = C->rhs[ww[3]];
02465                                   nn = *newval-2;
02466                                   part = newval+2;
02467                                   if ( (C->rhs[ww[3]+1]-mmm-1) == nn ) {
02468                                       while ( --nn >= 0 ) {
02469                                           if ( *mmm != *part ) {
02470                                               mmm++; part++; break;
02471                                           }
02472                                           mmm++; part++;
02473                                       }
02474                                       if ( nn < 0 ) goto NoMatch;
02475                                   }
02476                               }
02477                               break;
02478                           }
```

```
02479                               mm++; ww += ww[1];
02480                           }
02481                       }
02482                       break;
02483                   case FUNTOFUN:
02484                       if ( *w == newnumber ) goto NoMatch;
02485                       if ( ( type == FUNTOFUN &&
02486                       ( k = *w - WILDMASK ) > FUNCTION ) ) {
02487                           WORD *mm = AT.WildMask;
02488                           WORD *ww = AN.WildValue;
02489                           WORD nn = AN.NumWild;
02490                           while ( --nn >= 0 ) {
02491                               if ( *mm && ww[2] == k && ww[0] == type ) {
02492                                   if ( ww[3] == newnumber ) goto NoMatch;
02493                                   break;
02494                               }
02495                               mm++; ww += ww[1];
02496                           }
02497                       }
02498                   default:
02499                       break;
02500               }
02501               }
02502               else {
02503                 if ( type == VECTOMIN ) {
02504                   if ( inset >= AM.OffsetVector ) { i++; continue; }
02505                   inset += WILDMASK;
02506                 }
02507                 if ( ( inset == newnumber && type != SYMTONUM ) ||
02508                 ( type == SYMTONUM && inset-2*MAXPOWER == newnumber ) ) {
02509                   if ( *s == SETTONUM ) {
02510                       if ( n2 == oldnumber && type
02511                       <= SYMTOSUB ) goto NoMatch;
02512                       m = AT.WildMask;
02513                       w = AN.WildValue;
02514                       n = AN.NumWild;
02515                       while ( --n >= 0 ) {
02516                           if ( w[2] == n2 && *w <= SYMTOSUB ) {
02517                               if ( !*m ) {
02518                                   *w = SYMTONUM;
02519                                   w[3] = i;
02520                                   AN.WildReserve = 1;
02521                                   return(0);
02522                               }
02523                               if ( *w != SYMTONUM )
02524                                   goto NoMatch;
02525                               if ( w[3] == i ) return(0);
02526                               i = w[3];
02527                               j = (SetElements + Sets[j].first)[i];
02528                               if ( j == n2 ) return(0);
02529                               goto NoMatch;
02530                           }
02531                           m++; w += w[1];
02532                       }
02533                   }
02534                   else if ( n2 >= 0 ) {
02535                       *newval = *(w - Sets[j].first + Sets[n2].first);
02536                       if ( *newval > MAXPOWER ) *newval -= 2*MAXPOWER;
02537                       if ( dirty && *newval != oldval ) {
02538                           *newval = oldval; goto NoMatch;
02539                       }
02540                   }
02541                   return(0);
02542                 }
02543               }
02544               i++;
02545             } while ( ++w < m );
02546         }
02547         if ( notflag ) return(0);
02548         AN.oldtype = old2; AN.oldvalue = oldval; goto NoMatch;
02549     }
02550     else { return(0); }
02551
02552 NoM:
02553     AN.oldtype = old2; AN.oldvalue = w[3]; goto NoMatch;
02554 }
02555
02556 /*
02557         #] CheckWild :
02558     #] Wildcards :
02559     #[ DenToFunction :
02560
02561     Renames the denominator function into a function with the given number.
02562     For the syntax see   Denominators,function;
02563 */
02564
02565 int DenToFunction(WORD *term, WORD numfun)
```

```
02566 {
02567     int action = 0;
02568     WORD *t, *tstop, *tnext, *arg, *argstop, *targ;
02569     t = term+1;
02570     tstop = term + *term; tstop -= ABS(tstop[-1]);
02571     while ( t < tstop ) {
02572         if ( *t == DENOMINATOR ) {
02573             *t = numfun; t[2] |= DIRTYFLAG; action = 1;
02574         }
02575         tnext = t + t[1];
02576         if ( *t >= FUNCTION && functions[*t-FUNCTION].spec <= 0 ) {
02577             arg = t + FUNHEAD;
02578             while ( arg < tnext ) {
02579                 if ( *arg > 0 ) {
02580                     targ = arg + ARGHEAD; argstop = arg + *arg;
02581                     while ( targ < argstop ) {
02582                         if ( DenToFunction(targ,numfun) ) {
02583                             arg[1] |= DIRTYFLAG; t[2] |= DIRTYFLAG; action = 1;
02584                         }
02585                         targ += *targ;
02586                     }
02587                     arg = argstop;
02588                 }
02589                 else if ( *arg <= -FUNCTION ) arg++;
02590                 else arg += 2;
02591             }
02592         }
02593         t = tnext;
02594     }
02595     return(action);
02596 }
02597
02598 /*
02599     #] DenToFunction :
02600 */
```

Index

T